Artificial Intelligence 255 (2018) 43–70

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Fixpoint semantics for active integrity constraints

Bart Bogaerts [a,*], Luís Cruz-Filipe [b]

[a] KU Leuven, Department of Computer Science, Celestijnenlaan 200A, Leuven, Belgium
[b] University of Southern Denmark, Department of Mathematics and Computer Science, Campusvej 55, Odense, Denmark

ARTICLE INFO

Article history:
Received 4 July 2017
Received in revised form 2 October 2017
Accepted 18 November 2017
Available online 23 November 2017

Keywords:
Active integrity constraints
Approximation fixpoint theory

ABSTRACT

Active integrity constraints (AICs) constitute a formalism to associate with a database not just the constraints it should adhere to, but also how to fix the database in case one or more of these constraints are violated. The intuitions regarding which repairs are "good" given such a description are closely related to intuitions that live in various areas of non-monotonic reasoning, such as logic programming and autoepistemic logic.

In this paper, we apply *approximation fixpoint theory*, an abstract, algebraic framework designed to unify semantics of non-monotonic logics, to the field of AICs. This results in a new family of semantics for AICs. We study properties of our new semantics and relationships to existing semantics. In particular, we argue that two of the newly defined semantics stand out. *Grounded repairs* have a simple definition that is purely based on semantic principles that semantics for AICs should adhere to. And, as we show, they coincide with the intended interpretation of AICs on many examples. The second semantics of interest is the AFT-well-founded semantics: it is a computationally cheap semantics that provides upper and lower bounds for many other classes of repairs.

1. Introduction

One of the key components of modern-day databases are integrity constraints: logical formulas that specify semantic relationships between the data being modeled that have to be satisfied at all times. When the database is changed (typically due to updating), it is necessary to check if its integrity constraints still hold; in the negative case, the database must be repaired.

The problem of database repair has been an important topic of research for more than thirty years [1]. There are two major problems when deciding how to repair an inconsistent database: *finding* possible repairs and *choosing* which one to apply. Indeed, there are typically several ways to fix an inconsistent database, and several criteria to choose the "best" one have been proposed over the years. Among the most widely accepted criteria are minimality of change [45,25] – change as little as possible – and the common-sense law of inertia (discussed in, e.g., [33]) – do not change anything unless there is a reason for the change.

A typical implementation of integrity constraints in database systems is by means of event–condition–action (ECA) rules [38,44], which specify update actions to be performed when a particular event (a trigger) occurs and specific conditions hold. ECA rules are widely used in practice, as they are simple to implement and their individual semantics is easy

* Corresponding author.
E-mail addresses: bart.bogaerts@cs.kuleuven.be (B. Bogaerts), lcfilipe@gmail.com (L. Cruz-Filipe).

https://doi.org/10.1016/j.artint.2017.11.003
0004-3702/

to understand. However, the lack of declarative semantics for ECA rules makes their interaction complex to analyze and their joint behavior hard to understand.

The formalism of *active integrity constraints (AICs)* [27] was inspired by a similar idea. AICs express database dependencies through logic programming-style rules that include update actions in their heads. They come with a set of declarative semantics that identifies several progressively more restricted classes of repairs, which can be used as criteria to select a preferred repair [13]. These repairs can be computed directly by means of tree algorithms [17], which have been implemented as a prototype [16].

Example 1.1. We motivate the use of AICs in practice by means of a simple example. Consider a company's database, including tables employee and dept (relating employees to the department where they work). In particular, each employee is assigned to a unique department; if an employee is listed as working in two different departments, then the database is inconsistent, and this inconsistency must be fixed by removing one of those entries.

We can write this requirement as the following AIC.

$$\forall x, y, z : \text{employee}(x), \text{dept}(x, y), \text{dept}(x, z), y \neq z \supset -\text{dept}(x, y)$$

The intended meaning of this rule is: if all the literals in the lefthandside (body) of the rule are true in some state of the database, for particular values of x, y and z, then the database is inconsistent, and this inconsistency can be solved by performing the action on the right.

Suppose that the database is

$$DB = \{\text{employee}(\text{john}), \text{dept}(\text{john}, \text{finance}), \text{dept}(\text{john}, \text{hr})\}.$$

This database is inconsistent, and applying our AIC with $x = \text{john}$, $y = \text{finance}$ and $z = \text{hr}$ gives us a possible fix consisting of the action "remove dept(john, finance)". Observe, however, that the instantiation $x = \text{john}$, $y = \text{hr}$ and $z = \text{finance}$ detects the same inconsistency, but proposes instead the fix "remove dept(john, hr)": in general, there can be several different ways to repair inconsistencies.

AICs may also interact with each other. Suppose that we add the constraint

$$\forall x, y, z : \text{supervisor}(x, y), \text{dept}(x, z), \neg\text{dept}(y, z) \supset +\text{dept}(y, z) \tag{1}$$

stating that employees can only supervise people from their own department, and that whenever this constraint is violated, the department of the supervisee needs to be updated (i.e., the supervisor table and the department of the supervisor are deemed correct). If the database is now

$$DB = \{\text{employee}(\text{john}), \text{employee}(\text{ann}), \text{dept}(\text{john}, \text{finance}), \text{dept}(\text{ann}, \text{hr}), \text{supervisor}(\text{ann}, \text{john})\}$$

then this AIC detects an inconsistency, and suggests that it be fixed by adding the entry dept(john, hr). The database is still inconsistent, though, since there are now two entries for John in the dept table; restoring inconsistency would also require removing the entry dept(john, finance).

An alternative repair of the integrity constraint that the supervisee and supervisor should belong to the same department would be to change the department information associated with ann. By using *active* integrity constraints, we discard this solution: rule (1) only allows to insert a new department for the supervisee. If we additionally also want to allow changing ann's department, we need an extra constraint. ▲

It is striking that many intuitions about what "good" repairs are, such as minimality of change, are similar to intuitions that surfaced in other domains of non-monotonic reasoning, such as logic programming [39] and default logic [34]. Still, it has been hard to find satisfying semantics for AICs. As shown by Cruz-Filipe et al. [17], the semantics of so-called *founded repairs* [12] unexpectedly fails to respect the common-sense law of inertia, while the more restricted semantics of justified repairs [13] forbids natural repairs in some cases. That work proposed the operational semantics of well-founded repairs, which however is not modular [14] and is therefore severely restricted in its practical applicability.

In this work, we begin by defining a new semantics for AICs that avoids these problems: *grounded repairs*. Grounded repairs are natural counterparts to existing semantics in various non-monotonic reasoning domains such as logic programming; we discuss how they relate to other semantics for AICs. We also argue that grounded repairs match our intuitions regarding AICs on a broad set of examples.

We then give a more abstract characterization of the different semantics for AICs by associating with each set of AICs η a semantic operator $\mathcal{T}_\eta$. This operator immediately induces several semantics:

(i) *weak repairs* are *fixpoints* of $\mathcal{T}_\eta$;
(ii) *repairs* are *minimal fixpoints* of $\mathcal{T}_\eta$;
(iii) *grounded repairs* are *grounded fixpoints* [7] of $\mathcal{T}_\eta$.

The first two semantics are pre-existing semantics for AICs that we recover in an operator-based fashion.