Reasoning Formally about Database Queries and Updates

Jon Haël Brenas[1], Rachid Echahed[2][0000−0002−8535−8057], and Martin Strecker[3][0000−0001−9953−9871]

[1] UTHSC - ORNL, Memphis, Tennessee, USA
[2] CNRS and University of Grenoble, France
[3] http://martin-strecker.org/

Abstract. This paper explores formal verification in the area of database technology, in particular how to reason about queries and updates in a database system. The formalism is sufficiently general to be applicable to relational and graph databases. We first define a domain-specific language consisting of nested query and update primitives, and give its operational semantics. Queries are in full first-order logic. The problem we try to solve is whether a database satisfying a given pre-condition will satisfy a given post-condition after execution of a given sequence of queries and updates. We propose a weakest-precondition calculus and prove its correctness. We finally examine a restriction of our framework that produces formulas in the guarded fragment of predicate logic and thus leads to a decidable proof problem.

Keywords: Automated Theorem Proving, Modal Logic, Graph Transformations, Program Verification

1 Introduction

1.1 Context and Contributions

The work reported here has initially grown out of an effort to verify graph-manipulating programs that owe much to a traditional imperative programming style. The transformation language presented in this paper is inspired by query and update primitives found in graph databases such as Cypher [27], but we do not try to mimic a specific DB language, and our language is sufficiently general that it is also interesting for relational DBs. The structure of the language is in principle very simple, consisting of nested `match` constructs (however with queries that are full first-order logic formulas) and addition and deletion of relations. We are here interested in structural aspects, dealing only with uninterpreted relations. The transformation language (syntax and well-formedness constraints and semantics) will be defined in Section 2.

The transformation language has a clearly imperative (as opposed to functional) flavour, with a notion of DB state that coincides with a non-standard notion of interpretation of formulas. The main focus of the paper is on verifying whether a DB satisfying a given pre-condition will satisfy a given post-condition after the transformation. These conditions are again full first-order formulas. It is important to emphasise that we are dealing with the verification of the correctness of transformations as such, and not the validation of the satisfaction of constraints for particular instances of a DB (thus a kind of model checking problem). The program correctness calculus (a particular form of weakest precondition calculus) is described in Section 3. The resulting proof obligations are undecidable in general. However, in Section 4, we restrict our attention to the Guarded Fragment of predicate logic. By imposing suitable restrictions on the formulas occurring in assertions and selection statements, we identify a natural class of transformations that give rise to decidable proof obligations.

A particular challenge of our formalism is to take into account contextual information stemming from nested `match` statements, and to deal with relational update (an essentially second-order construct) in a first-order framework.

Related Work The view of a database transformation as an imperative program, with pre- and post-conditions, seems to be new.

Work in the context of deductive DBs ([8,26], also see [23] for an overview) mainly seems to address the problem of maintaining the consistency of DB *w.r.t.* specific constraints after individual updates, and not deductive verification. Consistency maintenance is then often enforced by Prolog-like inference rules. The more general question of DB updates as theory updates, for example in [11], has triggered an extensive amount of work, including investigations in non-monotonic logics. This line of research is not at all related to our approach that is situated in classical logic, with the credo that updates modify models and not theories.

A notable exception to the above is the work by Benedikt, Griffin and Libkin [5] that considers the problem of definability of database transactions for a very abstract notion of transformation language, leading mainly to negative decidability results. Contrary to this, we start with a specific (and, in particular in Section 4, restricted) language, to arrive at a proposal for a practically useful verification framework.

XML transformations [22,19] are transformations of particular tree-like structures, and the powerful type systems developed for them can be assimilated with program correctness assertions. However, XML transducers have a functional flavour, the verification method is not comparable to ours.

As mentioned before, our work has its origin in the verification of graph transformations. The landscape is heterogeneous, ranging from approaches based on category theory [16] to work in Monadic Second Order logic [10,20]. The graph decompositions inherent to this latter approach are often not compatible with updates performed naturally in graph structures (insertion or deletion of arcs, updates of attributes).

Our own work [9,7,6] has so far concentrated on particular decidable logics (modal or description logics). We have evoked the problem of the procedural transformation language; we mention in particular the difficulty with loops whose verification requires an annotation with invariants, so the verification approach is not fully automatic. Work that is very similar in spirit, also based on description logics and consistency management in ontologies, is [3,1,2]. The limitation of expressiveness of description logics leads to unpleasant circumlocutions: the logics are often not closed under simple operations like substitutions of relational expressions, with the consequence that extraction of proof obligations and proof procedures are intertwined. In order to have a clearer picture of the underlying mechanisms, we choose a plain first-order setting in this paper.

The modification of databases in conjunction with an imperative programming language is described in [21], with a verification procedure based on two-variable first order logic. To obtain decidability, severe restrictions on the domains (bounded domains and only one unbounded domain) have to be imposed.

As mentioned in the outset, we want to capture the spirit of DB languages like Cypher, without reproducing these languages in detail; our nested `match` statements seem to go beyond what is currently available in Cypher, and there are a huge number of features we do not cover, in particular paths. We are aware of a formal definition of the semantics of Cypher [12] and hope that a merger of this semantics and our language might make it possible to formally reason about integrity constraints in languages like Cypher.

1.2 Introductory Example

Before starting with the technical development, we present an example that informally introduces the principal notions and gives an overview of the verification methodology.

We consider the scenario of a database of a service provider for subscription of potential clients to its services. The database maintains some integrity constraints:

- *ValidClient*: All clients C registered in the database have to have their subscription approved ($Valid$) by an employee (E) of the company: $(\forall c. C(c) \longrightarrow \exists e. E(e) \wedge Valid(e, c))$
- *ActiveIfSubscr*: A service is activated for a client only if the client has previously subscribed to it: $(\forall s\, c.\ Active(s, c) \longrightarrow Subscr(s, c))$. The provider may suspend a service, so the inverse is not necessarily the case.

After registering at the service provider and subscribing to some services, the potential clients first get the status of applicants (A). At regular intervals, the database runs the program of Figure 1 to integrate applicants into its standard client base. This program proceeds as follows: it first retrieves all the applicants a that have their subscription approved (outer `match`) and adds these applicants as clients (first `add` statement). It then retrieves all the services s a given applicant a has subscribed to and activates these services (inner `match`). Finally, it removes the selected applicants from the set A (`del` statement).

The program is annotated with a pre-condition (the integrity constraint mentioned before) and a post-condition: the integrity constraint and the knowledge that all applicants remaining in A have not had their demand validated so far.

Let us run the program on a particular instance of a DB (the operational semantics in full generality is defined in Section 2.3). The extensions of the unary

```
Pre: $ValidClient \ \wedge \ ActiveIfSubscr$
    match  $a$  where  $A(a) \ \wedge \ \exists \ e. \ E(e) \ \wedge \ Valid(e, \ a)$  {
       add($C(a)$);
       match  $s$  where  $S(s) \ \wedge \ Subscr(s, \ a)$  {
          add($Active(s, \ a)$)
       };
       del($A(a)$)
    }
Post: $ValidClient \wedge \ ActiveIfSubscr \wedge \ (\forall \ a. \ A(a) \ \longrightarrow \ \neg\exists \ e. \ E(e) \ \wedge \ Valid(e,a))$
```

Fig. 1: An example program

predicates are sets of elements; and of the binary relations are sets of pairs. At the start of the program, we assume:

$$\begin{array}{llll}
A & = \{a_1, a_2, a_3\} & & E = \{e_1, e_2\} \\
C & = \{c_1, c_2\} & & S = \{s_1, s_2\} \\
Valid & = \{(e_1, c_1), (e_1, c_2), (e_1, a_1), (e_2, a_2)\} & & \\
Subscr & = \{(s_1, c_1), (s_2, c_2), (s_1, a_1), (s_2, a_1), (s_2, a_2)\} & & \\
Active & = \{(s_1, c_1)\} & &
\end{array}$$

Before looking in more detail at the execution of the program, it is important to understand the notion of a *state* of a program, which coincides with our non-standard notion of *interpretation* of a formula, which is set-based and not instance-based, as explained in the following. An interpretation is made up of three components: a domain (in this case, the set $\{a_1, a_2, \dots s_1, s_2\}$) and an interpretation of the predicate symbols (as above); all this is standard. The difference is in the way individual variables are interpreted: instead of having a single function mapping variables into the domain, we take a set of such functions. We are in particular interested in *maximal* interpretations that contain all the individual interpretation functions satisfying certain requirements.

As the precondition contains no free variables, the maximal interpretation set is initially the set of all functions mapping the set of variables to elements of the domain. The first `match` restricts the set of variable interpretations to those that map variable a to a_1 or a_2, as only these satisfy the condition of the match (the relation interpretations are not modified by `match`, and the domain remains invariant for all operations). The first `add` operation has an effect on the interpretation of relation C, adding the elements a_1, a_2 so that it will then become $\{c_1, c_2, a_1, a_2\}$ (here, the individual interpretations are not modified). The inner `match` limits the set of admissible individual interpretations still further to the set $\{(a \mapsto a_1, s \mapsto s_1), (a \mapsto a_1, s \mapsto s_2), (a \mapsto a_2, s \mapsto s_2)\}$. These pairs are then added to *Active*, whose extension is $\{(s_1, c_1), (s_1, a_1), (s_2, a_1), (s_2, a_2)\}$ at the end of the inner `match` statement. Note in particular that we do not simply take the cross-product of the elements $\{s_1, s_2\}$ bound to s and the elements $\{a_1, a_2\}$ bound to a: the pair (s_1, a_2) is not added to *Active*. We finally execute the `del` statement, which sets A to $\{a_3\}$. The net effect of the program is therefore an update of the relations A, C and *Active* in the DB.

Reasoning about these programs proceeds by backwards propagation of post-conditions, by computing weakest pre-conditions (wp). There are in particular two challenges for wp reasoning: taking into account contextual information (given by the conditions in the `match` clauses), and reasoning about sets and relations, instead of individuals.

When reasoning backwards, we first have to take the effect of `del(`$A(a)$`)` into account. We look up the contextual information about variable a. Its defining clause is $A(a) \wedge \exists e. E(e) \wedge Valid(e, a)$, so we symbolically remove from A in the post-condition all the elements satisfying this predicate. The subformula $(\forall a. A(a) \longrightarrow \neg \exists e. E(e) \wedge Valid(e, a))$ then becomes $(\forall a. (A(a) \wedge \neg (A(a) \wedge \exists e. E(e) \wedge Valid(e, a))) \longrightarrow \neg \exists e. E(e) \wedge Valid(e, a))$, which reduces to true. The subformula $\mathit{ValidClient} \wedge \mathit{ActiveIfSubscr}$ is not affected by the delete statement.

We next examine the effective of `add(`$\mathit{Active}(s, a)$`)` on the remaining post-condition. The contextual information for variable a is as before, and for variable s is $S(s) \wedge \mathit{Subscr}(s, a)$. In *ActiveIfSubscr*, we replace $\mathit{Active}(s, c)$ by a formula describing the union of *Active* and the conjunction of the characterising formulas of a and s, which yields $\forall s\, c.\ (\mathit{Active}(s, c) \vee A(c) \wedge (\exists e. E(e) \wedge Valid(e, c)) \wedge S(s) \wedge \mathit{Subscr}(s, c)) \longrightarrow \mathit{Subscr}(s, c)$. It is easy to see that this formula is implied by *ActiveIfSubscr* in the precondition of the program. In a similar spirit, we reason about `add(`$C(a)$`)`, replacing $C(c)$ in the precondition *ValidClient* by $C(c) \vee A(c) \wedge \exists e. E(e) \wedge Valid(e, c)$.

2 Transformation Language

This section defines the syntax of the transformation language (Section 2.1); it presents two notions of interpretation of formulas that are also instrumental for the concept of program state (Section 2.2); and it gives the operational semantics of programs (Section 2.3). The rest of this paper uses a semi-formal, mathematical style. A fully formal development in the Isabelle proof assistant is under way.[4]

2.1 Syntax

The syntax of statements *stmt* and programs *prog* is defined by the following grammar, where boldface $\mathbf{v}$ stands for a list of variables $v_1, \ldots, v_n$:

$$
\begin{array}{rcl}
\mathit{stmt} & ::= & \texttt{Skip} \\
& | & \texttt{add}(R(\mathbf{v})) \\
& | & \texttt{del}(R(\mathbf{v})) \\
& | & \texttt{match}\ \mathbf{v}\ \texttt{where}\ \mathit{form}\ \{\ \mathit{stmt}\ \} \\
& | & \mathit{stmt};\mathit{stmt} \\
\mathit{prog} & ::= & \texttt{Pre} : \mathit{form}\ \ \mathit{stmt}\ \ \texttt{Post} : \mathit{form}
\end{array}
$$

[4] Parts of the development can be found in the repository https://bitbucket.org/Martin_Strecker/db_queries_updates/

Formulas $form$ are occurring in `match` clauses and the pre- and post-conditions. They are formulas of standard first-order logic, defined by

$form ::= \bot \mid R(\mathbf{v}) \mid x = y \mid \neg form \mid form \wedge form \mid \forall v.form$

featuring constant symbol $\bot$, relational application $R(\mathbf{v})$, equality $x = y$ between individual variables, negation, binary connectors, first-order quantification over individual variables v. Other connectors and quantifiers than those shown are defined as usual.

Renaming individual variable x by y in formula ϕ is written $\phi[x := y]$. In formula manipulations like these, we assume that bound variables are correctly renamed to avoid clashes.

We assume that relation symbols have a fixed arity which can be enforced by typing or a naming convention; we do not describe the details here. Well-typing of a statement c in a context (list of variables) Γ, written $\Gamma \vdash c$, is defined by:

- $\Gamma \vdash$ `add`$(R(\mathbf{v}))$ if $\mathbf{v} \subseteq \Gamma$ and similarly for `del`
- $\Gamma \vdash$ `match` $\mathbf{v}$ `where` b $\{c\}$ if $\Gamma \cap \mathbf{v} = \{\}$ and $\Gamma @ \mathbf{v} \vdash c$ and $fv(b) \subseteq \Gamma @ \mathbf{v}$, where @ is list concatenation and $fv(b)$ is the set of free individual variables of b. In particular, `match` binds the variables $\mathbf{v}$ in b and c, and these variables should not occur in the context.
- $\Gamma \vdash c_1; c_2$ if $\Gamma \vdash c_1$ and $\Gamma \vdash c_2$

Pre- and post-conditions and statements may contain free individual variables, whose declaration constitutes the initial context for type checking. Since the programs we present in the examples are all closed, we have omitted the variable declaration clauses.

Apart from typing, we have to impose another restriction on the programs we analyse: There are no modifications of defining relations before use.

Example 1. Before defining this notion, we will look at a counter-example:

```
match  $a$ where  $A(a)$  {
   match  $b$ where  $B(b)$  {
      add($A(b)$)
   };
   del($C(a)$)
}
```

When reasoning about relation updates (`add` or `del`), we describe the changes induced *w.r.t.* the defining properties of the variables. Before the `add` in Example 1, the defining properties of a and b are $A(a)$ and $B(b)$ respectively. Intuitively and using a set-theoretic notation, the effect of the `add` is that the new A becomes $A^0 \cup B$, where A^0 is the original value of A. Computing this effect is not difficult.

The problem is the following `del`$(C(a))$, where we cannot proceed in a similar fashion. We cannot say that new C is $C^0 - A$ by looking up how a was defined in the corresponding `match` statement, because relation A has been modified between definition and use of a, but the variable a is still bound to the original values: before the `del` statement, $A(a)$ is not true any more. In fact, it should be that $C = C^0 - A^0$. Intuitively speaking, it seems that our analysis would become considerably more complex if it were necessary to precisely track which property was true for a variable in the execution history of the program, instead of taking its defining value.

We give a series of definitions that are reminiscent of the notion of definition-use chains in compiler technology [24], whence the name of *DU-stability* introduced below.

Definition 1 (DU-stability). *For a statement* `match` $\mathbf{v}$ `where` P*, we say that the* $v \in \mathbf{v}$ *are* defined *by this statement, and we say that* P *is their* defining property*. Note that in a well-typed program, a variable occurs in at most one* `match`*, so this notion is well-defined. The set of* defining relations *of a variable* v*,* $def\_rels(v)$*, is the set of relation symbols* R *that occur in the defining property of* v*. We say that a variable is* used *in the predicate of a* `match` *or in an* `add` *or* `del` *statement if it is among the free variables of the respective predicates. We say that a relation* R *is* modified *by an update if this update is* `add`$(R(\mathbf{v}))$ *or* `del`$(R(\mathbf{v}))$*. We say that a variable* v *defined in a* `match` *is* DU-stable *if in none of the execution paths leading from the definition to a use of* v*, any of the defining relations of* v *is modified. We say that a program is DU-stable if all its variables are.*

In order to avoid clutter, these definitions have been kept semi-formal in the sense that they are not defined inductively over the syntax and some parameters (such as the underlying program) remain implicit. Some related, more formal definitions are provided in Section 3.2.

The program of Example 1 is not DU-stable because the defining relation A is modified between the use of a in `del`$(C(a))$ and its definition. The program in Figure 1 is DU-stable, but it would not be if swapping `add`$(C(a))$ and `del`$(A(a))$, because then, the defining relation A of variable a would be modified before the uses of a.

Note that the restriction to property-preserving bindings is not a limitation, at least in principle and disregarding questions of efficiency of execution. Indeed, any breach of DU-stability can be avoided by storing values in an auxiliary relation and then retrieving this copy instead of referring to the modified relation.

2.2 Interpretations

We will introduce two kinds of semantics:

- an individual semantics that is the traditional logical semantics;
- a set-based semantics allowing to reason about sets of assignments of individual variables.

The *individual semantics* is given by interpretations $\iota = (\iota_d, \iota_r, \iota_i)$ where ι_d is a domain, ι_r is a function that assigns to each n-ary relation symbol of the language a subset of ι_d^n, and ι_i a function that assigns to each individual variable an element of ι_d. The relation $\iota \models \phi$ (interpretation ι is a model of formula ϕ) is defined as usual:

- $\iota \not\models \bot$
- $\iota \models R(\mathbf{v})$ if $\iota_i(\mathbf{v}) \in \iota_r(R)$, where $\iota_i(\mathbf{v})$ is the obvious mapping of ι_i on a vector of variables.
- $\iota \models x = y$ if $\iota_i(x) = \iota_i(y)$
- $\iota \models \neg\psi$ if $\iota \not\models \psi$
- $\iota \models \psi \wedge \phi$ if $\iota \models \psi$ and $\iota \models \phi$
- $\iota \models \forall v.\psi$ (first-order quantification) if for all $vi \in \iota_d$, we have $\iota^{v:=vi} \models \psi$. Here, if $\iota = (\iota_d, \iota_r, \iota_i)$, then $\iota^{v:=vi} = (\iota_d, \iota_r, \iota_i(v := vi))$ and $\iota_i(v := vi)$ is the update of function ι_i at variable v with value vi.

The *set-based semantics* is given by interpretations $\sigma = (\sigma_d, \sigma_r, \sigma_i)$ where σ_d and σ_r are as for the individual semantics, and σ_i is a set of individual assignments. We write $\iota \in \sigma$ if $\iota = (\iota_d, \iota_r, \iota_i)$ with $\iota_d = \sigma_d$ and $\iota_r = \sigma_r$ and $\iota_i \in \sigma_i$.

For instance, in the example of Section 1.2, we considered a set-based interpretation σ with a domain σ_d and relational assignment σ_r as defined there, and $\sigma_i = \{(a \mapsto a_1, s \mapsto s_1), (a \mapsto a_1, s \mapsto s_2), (a \mapsto a_2, s \mapsto s_2)\}$. One of the individual interpretations $\iota \in \sigma$ has the same domain and relational assignment, and individual variable assignment $\iota_i = (a \mapsto a_1, s \mapsto s_1)$.

The model relation[5] for the set-based semantics is defined by $\sigma \models \phi$ iff for all $\iota \in \sigma$, $\iota \models \phi$. The intuitive meaning of $\sigma \models \phi$ is that σ is a result for a query ϕ, where σ_i is the (not necessarily maximal) set of solutions, *i.e.* assignments to the free variables that satisfy ϕ, given the extension of the database as defined by σ_r.

A possibly bewildering consequence of this definition is that also formulas that are inconsistent (according to the individual semantics) have a model in the set-based semantics. Indeed, $(\sigma_d, \sigma_r, \{\}) \models \bot$. This choice is motivated by the intended behaviour of the operational semantics, which should be non-blocking: execution can always proceed after a `match` statement, even for an inconsistent match condition, but then with an empty solution set.

As usual, a formula is called *valid* if it is true under every interpretation. The notions coincide for the two semantics:

Lemma 1. *A formula is valid under the individual semantics iff it is valid under the set-based semantics.*

2.3 Operational Semantics

The operational semantics defines how the program state evolves when executing the instructions of a program. In our case, a program state is precisely a set-based interpretation in the sense of Section 2.2. Intuitively, in an interpretation $\sigma = (\sigma_d, \sigma_r, \sigma_i)$, the component σ_r corresponds to the extension of the database that is manipulated by `add` and `del` statements, and the component σ_i corresponds

[5] We use the same relation symbol $\models$ and disambiguate individual and set-based semantics with the designation of the model (ι resp. σ).

$$(\textsc{Add})\ \frac{\sigma' = add\_rel\_si\ R\ \mathbf{v}\ \sigma}{(\mathtt{add}(R(\mathbf{v})), \sigma) \Rightarrow \sigma'} \qquad (\textsc{Del})\ \frac{\sigma' = del\_rel\_si\ R\ \mathbf{v}\ \sigma}{(\mathtt{del}(R(\mathbf{v})), \sigma) \Rightarrow \sigma'}$$

$$(\textsc{Match})\ \frac{(c, max\_model\ b\ \sigma) \Rightarrow \sigma'' \quad \sigma' = \sigma''\llparenthesis i := \sigma.i \rrparenthesis}{(\mathtt{match}\ \mathbf{v}\ \mathtt{where}\ b\ \{c\}, \sigma) \Rightarrow \sigma'}$$

$$(\textsc{Skip})\ (\mathtt{Skip}, \sigma) \Rightarrow \sigma \qquad (\textsc{Seq})\ \frac{(c_1, \sigma) \Rightarrow \sigma'' \quad (c_2, \sigma'') \Rightarrow \sigma'}{(c_1; c_2, \sigma) \Rightarrow \sigma'}$$

Fig. 2: Big-step semantics rules

to variable bindings established by the `match` clauses. The domain σ_d remains unchanged throughout the program.

The rules of the operational semantics have the form $(c, \sigma) \Rightarrow \sigma'$, meaning that execution of statement c transforms state σ to state σ'. The inductive definition of the transition relation is given in Figure 2.

Before commenting on these rules, we introduce some more notation for manipulating interpretations. In an interpretation $\sigma = (\sigma_d, \sigma_r, \sigma_i)$, we retrieve the component σ_d, σ_r, resp. σ_i with $\sigma.d$, $\sigma.r$, resp. $\sigma.i$. Component update is written in banana brackets. Thus, $\sigma''\llparenthesis i := \sigma.i \rrparenthesis$ (as in rule Match) is the interpretation $(\sigma''_d, \sigma''_r, \sigma_i)$.

The rule for the `add` statement is defined with the aid of an auxiliary function that adds to relation R the values bound to variables $\mathbf{v}$ in state σ. The precise definition of $add\_rel\_si\ R\ \mathbf{v}\ \sigma$ is: $\sigma\llparenthesis r := \sigma.r(R := (\sigma.r(R) \cup ((\lambda ii.\mathtt{map}\ ii\ \mathbf{v}) \triangleright \sigma.i))) \rrparenthesis$.

In a similar spirit, the definition of $del\_rel\_si\ R\ \mathbf{v}\ \sigma$ is: $\sigma\llparenthesis r := \sigma.r(R := (\sigma.r(R) - ((\lambda ii.\mathtt{map}\ ii\ \mathbf{v}) \triangleright \sigma.i))) \rrparenthesis$.

Let us decipher the definition of add_rel_si: We update the relational interpretation of σ for relation R, so that the new interpretation of R becomes $(\sigma.r(R) \cup ((\lambda ii.\mathtt{map}\ ii\ \mathbf{v}) \triangleright \sigma.i))$. This is the old interpretation of relation R, plus new elements resulting from mapping the individual interpretations on the variable vector $\mathbf{v}$. Here, `map` is the mapping of a function on a list, and $\triangleright$ is the image of a set under a function. For example, if the relation to be updated is $Active$ with interpretation $\sigma.r(Active) = \{(s_1, c_1)\}$ and the individual variable interpretation $\sigma.i = \{(a \mapsto a_1, s \mapsto s_1), (a \mapsto a_1, s \mapsto s_2), (a \mapsto a_2, s \mapsto s_2)\}$, the expression $((\lambda ii.\mathtt{map}\ ii\ (s, a)) \triangleright \sigma.i)$ yields $\{(s_1, a_1), (s_2, a_1), (s_2, a_2)\}$ which are added to $\sigma.r(Active)$ (cf. example of Section 1.2).

For executing the `match` statement, we first compute the maximal model satisfying condition b in σ. Note that σ already incorporates the cumulative effect of surrounding `match` statements. The auxiliary function is defined as $max\_model\ b\ \sigma := fusion\ \sigma\ \{\iota \in \sigma \mid \iota \models b\}$, where $fusion\ \sigma\ I := (\sigma_d, \sigma_r, \bigcup \iota \in I.\{\iota.i\})$. "Maximality" of a set-based interpretation is here understood as "containing the maximum of individual interpretations". If σ is the maximal interpretation satisfying the surrounding `match` conditions, then $max\_model\ b\ \sigma$ is the maximal model satisfying in addition the current condition. Note that for a condition b that is inconsistent with the surrounding conditions, $max\_model\ b\ \sigma = (\sigma_d, \sigma_r, \{\})$.

Starting from this model, we execute the body c of the `match` statement, to reach a state σ''. We finally obtain the result state by restoring the individual variable bindings of the outer scope; of course, we keep the modifications induced by c on the relational assignment σ_r. The rules for `Skip` (no-op) and sequential composition are standard.

3 Program Logic

In this section, we show how to reason about the programs introduced in Section 2. For the programming language, we introduce extended Hoare triples (Section 3.1) that take contextual information into account, and establish a correspondence with the operational semantics, in the form of a soundness result (Section 3.2). We then show how to derive weakest pre-conditions (Section 3.3).

3.1 Hoare Triples: Definition

As is common practice in program logics, we reason about programs with Hoare triples $\{P\}\ c\ \{Q\}$ which express that when started in a program state that satisfies condition P, execution of statement c ends up in a program state satisfying condition Q. The programs of our language always terminate, never get stuck, and the language is deterministic, so there is no need to distinguish between partial and total correctness of programs.

To this triple, we add a context β that is the list of conditions accumulated while diving into nested `match` statements. This list dynamically grows or shrinks as we move into or out of a `match` statement. The conjunction of these formulas can be assumed to hold at the given point of the program. Indeed, in formulas (such as $R(\mathbf{v}) \vee \beta$), β does not stand for a list of formulas, but for the conjunction of the elements of the list. The inductive definition of the relation $\beta \vdash \{P\}\ c\ \{Q\}$ is given in Figure 3. At the start of a program, the context is assumed to be empty: $\beta = []$. In spite of its four components, we continue speaking about Hoare triples.

Again, the rules Skip and Seq are standard, and so is Conseq that permits to weaken pre- respectively post-conditions and that is provided to ensure completeness of the calculus.

The Match rule adds the match condition b to the list of bindings β (list concatenation $\beta @ [b]$) and then computes the pre-condition P for the body of the `match` statement. Whereas Q is outside the scope of the variables $\mathbf{v}$ bound by `match`, these variables could appear in P. The pre-condition of `match` therefore discharges the local condition b and abstracts over the local variables $\mathbf{v}$.

In rules Add and Del, we use relation update:

$$(\textsc{Add})\ \beta \vdash \{Q[R := \lambda \mathbf{v}.(R(\mathbf{v}) \lor \beta)]\}\ \texttt{add}(R(\mathbf{v}))\ \{Q\}$$

$$(\textsc{Del})\ \beta \vdash \{Q[R := \lambda \mathbf{v}.(R(\mathbf{v}) \land \neg\beta)]\}\ \texttt{del}(R(\mathbf{v}))\ \{Q\}$$

$$(\textsc{Match})\ \frac{\beta @ [b] \vdash \{P\}\ c\ \{Q\}}{\beta \vdash \{\forall \mathbf{v}.(b \longrightarrow P)\}\ (\texttt{match}\ \mathbf{v}\ \texttt{where}\ b\ \{c\})\ \{Q\}}$$

$$(\textsc{Skip})\ \beta \vdash \{P\}\ \texttt{Skip}\ \{P\} \qquad (\textsc{Seq})\ \frac{\beta \vdash \{P\}\ c_1\ \{R\} \quad \beta \vdash \{R\}\ c_2\ \{Q\}}{\beta \vdash \{P\}\ c_1; c_2\ \{Q\}}$$

$$(\textsc{Conseq})\ \frac{P \longrightarrow P' \quad \beta \vdash \{P'\}\ c\ \{Q'\} \quad Q' \longrightarrow Q}{\beta \vdash \{P\}\ c\ \{Q\}}$$

Fig. 3: Hoare Triples

Definition 2. *The* update *of relation R by relation S in formula Q is written as $Q[R := \lambda \mathbf{v}.S]$, where the variables $\mathbf{v}$ may occur in S. It is defined recursively with base case $R(a_1, \dots, a_n)[R := \lambda v_1, \dots, v_n.S] = (\lambda v_1, \dots, v_n.S)(a_1, \dots, a_n) = S[v_1 := a_1, \dots v_n := a_n]$ and $R'(a_1, \dots, a_n)[R := \lambda v_1, \dots, v_n.S] = R'(a_1, \dots, a_n)$ for $R \neq R'$. The propagation of update $[R := \lambda \mathbf{v}.S]$ through Boolean connectives is standard, with variable renaming in $(\forall v.\psi)[R := \lambda \mathbf{v}.S] = (\forall v'.\psi[v := v'][R := \lambda \mathbf{v}.S])$ to avoid free variable capture.*

Please refer back to Section 1.2 for an illustration: For example, for statement $\texttt{add}(\mathit{Active}(s, a))$, the context β is the conjunction of $A(a) \land \exists e.E(e) \land \mathit{Valid}(e, a)$ and $S(s) \land \mathit{Subscr}(s, a)$, and relation update $(\forall s\, c.\ \mathit{Active}(s, c) \longrightarrow \mathit{Subscr}(s, c))[\mathit{Active} := \lambda s\, a.\mathit{Active}(s, a) \lor \beta]$ yields $\forall s\, c.\ (\mathit{Active}(s, c) \lor A(c) \land (\exists e.E(e) \land \mathit{Valid}(e, c)) \land S(s) \land \mathit{Subscr}(s, c)) \longrightarrow \mathit{Subscr}(s, c)$.

3.2 Hoare Triples: Soundness

The proof of soundness follows a general approach that is relatively standard, see for example [25]. We first define a semantic notion of validity of a Hoare triple and then show that the inductively defined relation of Figure 3 implies semantic validity. We first define a simplified variant of validity (Definition 3), from which soundness is not directly provable. For the induction to go through and to take into account the notion of DU-stability, we have to define a more complex notion of validity (Definition 6) with a more involved soundness lemma (Lemma 2) of which the desired theorem (Theorem 1) is an instance.

Definition 3 (Validity of Hoare Triples). *For formulas P and Q and statement c, we define the relation $\models \{P\}\ c\ \{Q\}$ as: For all states σ, σ', if $(c, \sigma) \Rightarrow \sigma'$ and $\sigma \models P$, then $\sigma' \models Q$.*

Theorem 1 (Soundness). *Let c be a well-typed and DU-stable program. If $[] \vdash \{P\}\ c\ \{Q\}$, then $\models \{P\}\ c\ \{Q\}$.*

We prove this theorem later and first introduce additional notation.

An *exclusion set* X is a set of variables, with the intended meaning that if $v \in X$ at a particular point in program execution, then there exists an R that is a defining relation of v (see Definition 1) and R has been modified since the definition of v. Intuitively, this has as a consequence that if $P(v)$ is the defining property of v, then there is a risk that $P(v)$ is not true at this point any more.

To keep track of how exclusion sets evolve during execution of a program, we define a relation of exclusion propagation.

Definition 4 (Exclusion Propagation). *For statement c and exclusion sets X, X', we inductively define the relation of exclusion propagation $(c, X) \xrightarrow{\times} X'$ by:*

- $(\texttt{Skip}, X) \xrightarrow{\times} X$
- $(\texttt{add}(R(\mathbf{v})), X) \xrightarrow{\times} X \cup D(R)$ *where $D(R)$ is the set of variables v such that R is a defining relation of v*
- $(\texttt{del}(R(\mathbf{v})), X) \xrightarrow{\times} X \cup D(R)$
- $(\texttt{match}\ \mathbf{v}\ \texttt{where}\ b\ \{c\}, X) \xrightarrow{\times} (X' - \mathbf{v})$ *if* $(c, X) \xrightarrow{\times} X'$
  Note that the local variables $\mathbf{v}$ are not visible outside of c and can therefore be removed after the `match`.
- $((c_1; c_2), X) \xrightarrow{\times} X'$ *if* $(c_1, X) \xrightarrow{\times} X''$ *and* $(c_2, X'') \xrightarrow{\times} X'$

Example 2. Let us look back at the introductory example in Figure 1. When starting exclusion propagation with an empty set at the beginning of the program, it remains empty most of the time, until after the `del` statement, when it becomes $\{a\}$, so the defining property of a is not usable in the following, but this is not problematic as there are no further statements (*a fortiori*, statements where a is used).

Now please refer back to the program of Example 1. When starting exclusion propagation with an empty set, after the `add` statement, the exclusion set is $\{a\}$, and it remains so until the `del` statement. The problem is that variable a is still used at this point.

Definition 5 (Admissible Predicates). *For a list of formulas β and an exclusion set X, the set of admissible predicates is $adm(\beta, X) = \{b \in \beta \mid fv(b) \cap X = \{\}\}$. Taken as a formula, $adm(\beta, X)$ is understood to be the conjunction of the formulas contained in the set.*

Consider an exclusion propagation of a program that starts with an empty exclusion set. Assume that at a point before a statement $\texttt{add}(R(\mathbf{v}))$ (or similarly `del`), there is a $v \in \mathbf{v}$ that is also contained in the current exclusion set. Then this would contradict DU-stability of v and thus of the whole program. Differently said, in a DU-stable program, the variables of an `add` or `del` do not occur in an exclusion set.

Definition 6 (Validity of Hoare Triples with Exclusion Sets). *For a list of formulas β, exclusion set X, formulas P and Q and statement c, we define the relation $\beta, X \models \{P\}\ c\ \{Q\}$ as: For all states σ, σ', if $(c, \sigma) \Rightarrow \sigma'$ and $\sigma \models adm(\beta, X) \wedge P$, for all X', if $(c, X) \xrightarrow{\times} X'$, then $\sigma' \models adm(\beta, X') \wedge Q$.*

Lemma 2 (Soundness with Exclusion Sets). *Let c be a sub-statement of a well-typed and DU-stable program. If $\beta \vdash \{P\}\ c\ \{Q\}$, then $\beta, X \models \{P\}\ c\ \{Q\}$ for all X.*

A proof of this lemma is given in the formal Isabelle development.

Proof. (of Theorem 1): The theorem is an instance of Lemma 2, for $\beta = []$ and $X = \{\}$.

3.3 Weakest Pre-Conditions

The weakest pre-condition *wp* for a given post-condition Q and statement c is a pre-condition that is implied by any other pre-condition. We compute the *wp* with function $wp(\beta, c, Q)$ that also takes into account the local bindings. The recursive definition of *wp* is given in Figure 4.

$$\begin{aligned}
&wp(\beta, \texttt{Skip},\ Q) = Q\\
&wp(\beta, \texttt{add}(R(\mathbf{v})),\ Q) = Q[R := \lambda \mathbf{v}.(R(\mathbf{v}) \vee \beta)]\\
&wp(\beta, \texttt{del}(R(\mathbf{v})),\ Q) = Q[R := \lambda \mathbf{v}.(R(\mathbf{v}) \wedge \neg\beta)]\\
&wp(\beta, \texttt{match}\ \mathbf{v}\ \texttt{where}\ b\ \{c\},\ Q) = \forall \mathbf{v}.b \longrightarrow wp(\beta @ [b], c, Q)\\
&wp(\beta, c_1; c_2,\ Q) = wp(\beta, c_1,\ wp(\beta, c_2,\ Q))
\end{aligned}$$

Fig. 4: Weakest pre-conditions

The correspondence between the weakest pre-conditions and the program calculus of Section 3.1 is established by the following lemma, whose proof is by an easy induction over c.

Lemma 3. $\beta \vdash \{wp(\beta, c, Q)\}\ c\ \{Q\}$.

Initially, β is assumed to be empty. Proving the correctness of a program $\{Pre\}\ prog\ \{Post\}$ therefore amounts to showing that $Pre \longrightarrow wp([], prog, Post)$ is valid, by an application of rule Conseq.

Let us emphasise one point: in Section 2.2, we have defined two semantics. Because the notion of validity of Hoare triples is defined with reference to the set-based semantics, the whole soundness argument is carried out in this semantics. Showing that $Pre \longrightarrow wp([], prog, Post)$ is valid can be done *w.r.t.* the set-based semantics, but according to Lemma 1, it is equivalent to the standard individual semantics, so it is more convenient to switch to this semantics here to be able to use standard proof procedures of predicate logic.

4 Guarded Fragment

The results established in the previous section are sound for programs containing full first-order formulas, but application of the *wp* calculus to such programs will in general produce proof problems that are undecidable. The Guarded Fragment (GF) is a fragment of first-order predicate logic that has been introduced by Andréka, Németi and van Benthem [4] and studied in depth [14,15]. The aspect of interest for us is that GF is decidable; several decision procedures have been described [13,17] and implemented [18].

We summarise the essential features of GF: An *atomic formula* or *atom* is defined as an equality $x = y$ or the application of a relation symbol to a tuple of variables, $R(\mathbf{v})$. On this basis, we define GF:

Definition 7 (Guarded Fragment, GF).

- *All quantifier-free first-order formulas are formulas of GF.*
- *If* ψ *and* ϕ *are formulas of GF, then so are* $\neg\psi$ *and* $(\psi \wedge \phi)$*.*
- *If* $\psi(\mathbf{x}, \mathbf{y})$ *is a formula of GF and* $\alpha(\mathbf{x}, \mathbf{y})$ *is an atom and* $fv(\psi(\mathbf{x}, \mathbf{y})) \subseteq fv(\alpha(\mathbf{x}, \mathbf{y}))$*, then* $\exists \mathbf{y}.\alpha(\mathbf{x}, \mathbf{y}) \wedge \psi(\mathbf{x}, \mathbf{y})$ *and* $\forall \mathbf{y}.\alpha(\mathbf{x}, \mathbf{y}) \longrightarrow \psi(\mathbf{x}, \mathbf{y})$ *are formulas of GF. Here, we call* $\alpha(\mathbf{x}, \mathbf{y})$ *the* guard *and* $\psi(\mathbf{x}, \mathbf{y})$ *the* body *of a quantified formula.*

We say that a formula is guarded if it belongs to the guarded fragment of first-order logic. The definitions of *ValidClient* and *ActiveIfSubscr* of Section 1.2 are examples of guarded formulas.

Definition 8 (Guarded statement and program). *We say that a formula* b *is a* quasi-guard *if it can be written as* $\alpha_1(\mathbf{v_1}) \wedge \ldots \wedge \alpha_n(\mathbf{v_n}) \wedge \psi$*, where* ψ *is a guarded formula and the* α_i *are atoms, where different* $\mathbf{v_i}, \mathbf{v_j}$ *are disjoint.*

We say that a match clause `match` $\mathbf{v}$ `where` b *is guarded if* b *is a quasi-guard.*

We say that a statement is guarded if all its `match` *clauses are guarded. We say that a program is guarded if its pre- and post-conditions and its constituting statement are guarded.*

For example, the program of Figure 1 is guarded. A program with a clause `match` v_1, v_2 `where` $(\exists x.R(x, v_1)) \wedge (\exists y.R(y, v_2))$ is not guarded.

Theorem 2. *If* c *is a guarded statement,* Q *a guarded formula and* β *a list of quasi-guards, then* $wp(\beta, c, Q)$ *is a guarded formula.*

Proof. The proof is by induction on the structure of the statement. The proposition is evident for `Skip`. For a sequence $c_1; c_2$ of instructions and guarded Q, by induction hypothesis, we obtain a guarded formula for $wp(\beta, c_2, Q)$. Similarly, for a `match` statement, $wp(\beta@[b], c, Q)$ is a guarded formula G. If `match` $\mathbf{v}$ `where` b is guarded and b a quasi-guard, we can write $\forall \mathbf{v}.b \longrightarrow G$ as $\forall \mathbf{v_1}.\alpha_1(\mathbf{v_1}) \longrightarrow \ldots \longrightarrow \forall \mathbf{v_n}.\alpha_n(\mathbf{v_n}) \longrightarrow \psi \longrightarrow G$, which is again guarded.

The main concern is therefore preservation of guardedness in relation update; we first discuss the case $Q[R := \lambda \mathbf{v}.(R(\mathbf{v}) \vee \beta)]$. We reason by induction on Q. The only critical cases are existential and universal quantification; we only look at the latter, the former is similar. Thus, assume Q is of the form $\forall \mathbf{y}.\alpha(\mathbf{x}, \mathbf{y}) \longrightarrow \psi(\mathbf{x}, \mathbf{y})$. The case where $\alpha \neq R$ poses no problem, so assume Q of the form $\forall \mathbf{y}.R(\mathbf{x}, \mathbf{y}) \longrightarrow \psi(\mathbf{x}, \mathbf{y})$, with $Q[R := \lambda \mathbf{v}.(R(\mathbf{v}) \vee \beta)] = \forall \mathbf{y}.(R(\mathbf{x}, \mathbf{y}) \vee \beta[\mathbf{v} := \mathbf{x}, \mathbf{y}]) \longrightarrow \psi'$, where ψ' is the result of the relation update in $\psi(\mathbf{x}, \mathbf{y})$. This formula is not guarded any longer, but we can rewrite it to a conjunction of $\forall \mathbf{y}.R(\mathbf{x}, \mathbf{y}) \longrightarrow \psi'$ (which is guarded) and $\forall \mathbf{y}.\beta[\mathbf{v} := \mathbf{x}, \mathbf{y}] \longrightarrow \psi'$, with β a list of quasi-guards, which can be turned into a guarded formula in a similar form as seen for the `match` statement.

The reasoning for a relation update $Q[R := \lambda \mathbf{v}.(R(\mathbf{v}) \wedge \neg\beta)]$ for a delete statement proceeds along the same line, but is slightly simpler: the intermediate formula $\forall \mathbf{y}.(R(\mathbf{x}, \mathbf{y}) \wedge \neg\beta[\mathbf{v} := \mathbf{x}, \mathbf{y}]) \longrightarrow \psi'$ can directly be rewritten to the guarded $\forall \mathbf{y}.R(\mathbf{x}, \mathbf{y}) \longrightarrow (\neg\beta[\mathbf{v} := \mathbf{x}, \mathbf{y}]) \longrightarrow \psi'$.

From this theorem, the fact that a program $\{Pre\}\ c\ \{Post\}$ yields a proof obligation $Pre \longrightarrow wp([], c, Post)$, and the decidability of GF, we obtain:

Corollary 1.

- *Application of the weakest pre-condition calculus of Section 3.3 to a guarded program produces a guarded proof obligation.*
- *The correctness problem of guarded programs is decidable.*

5 Conclusions

This paper has presented a language combining queries and updates that can be used for graph and relational databases. The focus of the paper is on verifying assertions in the form of pre- and post-conditions, the operational aspect of the language was secondary. It might nevertheless be interesting to make this language executable, which is not possible when bluntly taking the operational semantics as it stands, because the semantics is manipulating possibly infinite sets of individual interpretations. We are however convinced that it is easy to derive a realistic operational semantics, by a restriction to relevant variables (the variables occurring in the program).

Our current efforts concentrate on formally verifying the theory developed in this paper in the Isabelle proof assistant, in order to obtain a fully verified proof obligation generator. Completeness of the calculus presented here is an open question. Further steps in the theory are extensions of the logic permitting to reason about paths in graphs, leading us to consider logics with transitive closure.

Acknowledgements. We are grateful to Lison Kardassevitch for implementing a prototype of the verification framework.

References

1. Ahmetaj, S., Calvanese, D., Ortiz, M., Simkus, M.: Managing change in graph-structured data using description logics. In: Proc. of the 28th AAAI Conf. on Artificial Intelligence (AAAI 2014). pp. 966–973. AAAI Press (2014), http://www.inf.unibz.it/~calvanese/papers-html/AAAI-2014-graph-dbs.html
2. Ahmetaj, S., Calvanese, D., Ortiz, M., Simkus, M.: Managing change in graph-structured data using description logics. ACM Trans. Comput. Log. **18**(4), 27:1–27:35 (2017). https://doi.org/10.1145/3143803, http://doi.acm.org/10.1145/3143803
3. Ahmeti, A., Calvanese, D., Polleres, A.: Updating RDFS aboxes and tboxes in SPARQL. In: The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I. pp. 441–456 (2014), http://www.inf.unibz.it/~calvanese/papers/ahme-calv-poll-ISWC-2014.pdf
4. Andréka, H., Németi, I., van Benthem, J.: Modal languages and bounded fragments of predicate logic. Journal of Philosophical Logic **27**(3), 217–274 (1998), http://www.fenrong.net/teaching/Andreka.pdf
5. Benedikt, M., Griffin, T., Libkin, L.: Verifiable properties of database transactions. Information and Computation **147**(1), 57–88 (1998), https://core.ac.uk/download/pdf/82337092.pdf
6. Brenas, J.H., Echahed, R., Strecker, M.: Ensuring correctness of model transformations while remaining decidable. In: Sampaio, A., Wang, F. (eds.) Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9965, pp. 315–332 (2016)
7. Brenas, J.H., Echahed, R., Strecker, M.: A Hoare-like calculus using the SROIQ$^\sigma$ logic on transformations of graphs. In: Diaz, J., Lanese, I., Sangiorgi, D. (eds.) Theoretical Computer Science, Lecture Notes in Computer Science, vol. 8705, pp. 164–178. Springer Berlin Heidelberg (2014), http://www.irit.fr/~Martin.Strecker/Publications/tcs2014.html
8. Bry, F., Decker, H., Manthey, R.: A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In: International Conference on Extending Database Technology. pp. 488–505. Springer (1988)
9. Chaabani, M., Echahed, R., Strecker, M.: Logical foundations for reasoning about transformations of knowledge bases. In: Eiter, T., Glimm, B., Kazakov, Y., Krötzsch, M. (eds.) DL – Description Logics. CEUR Workshop Proceedings, vol. 1014, pp. 616–627. CEUR-WS.org (2013)
10. Courcelle, B., Engelfriet, J.: Graph structure and monadic second-order logic, a language theoretic approach. Cambridge University Press (2011), http://www.labri.fr/perso/courcell/Book/TheBook.pdf
11. Fagin, R., Ullman, J.D., Vardi, M.Y.: On the semantics of updates in databases. In: Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems. pp. 352–365. ACM (1983)
12. Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An evolving query language for property graphs. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018. pp. 1433–1445 (2018). https://doi.org/10.1145/3183713.3190657, http://doi.acm.org/10.1145/3183713.3190657

13. Grädel, E.: Decision procedures for guarded logics. In: Automated Deduction – CADE16. Proceedings of 16th International Conference on Automated Deduction, Trento, 1999. LNCS, vol. 1632. Springer (1999), http://www.logic.rwth-aachen.de/pub/graedel/Gr-cade99.ps
14. Grädel, E.: On the restraining power of guards. Journal of Symbolic Logic **64**, 1719–1742 (1999), http://www.logic.rwth-aachen.de/pub/graedel/Gr-jsl99.ps
15. Grädel, E.: Decidable fragments of first-order and fixed-point logic. From prefix-vocabulary classes to guarded logics. In: Proceedings of Kalmár Workshop on Logic and Computer Science, Szeged (2003), http://www.logic.rwth-aachen.de/pub/graedel/Gr-kalmar03.ps
16. Habel, A., Pennemann, K.H., Rensink, A.: Weakest preconditions for high-level programs. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) Graph Transformations (ICGT), Natal, Brazil. Lecture Notes in Computer Science, vol. 4178, pp. 445–460. Springer Verlag, Berlin (September 2006), http://eprints.eemcs.utwente.nl/1937/01/icgt2006-preconditions.pdf
17. Hirsch, C.: Guarded Logics: Algorithms and Bisimulation. Ph.D. thesis, RWTH Aachen (2002), http://www.logic.rwth-aachen.de/pub/hirsch/hirsch.pdf
18. Hladik, J.: Implementation and optimisation of a tableau algorithm for the guarded fragment. In: International Conference on Automated Reasoning with Analytic Tableaux and Related Methods. pp. 145–159. Springer (2002)
19. Hosoya, H.: XML processing - The Tree-Automata Approach. Cambridge University Press (2011)
20. Inaba, K., Hidaka, S., Hu, Z., Kato, H., Nakano, K.: Graph-transformation verification using monadic second-order logic. In: International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP). pp. 17–28 (Jul 2011), http://dl.acm.org/authorize?442117
21. Itzhaky, S., Kotek, T., Rinetzky, N., Sagiv, M., Tamir, O., Veith, H., Zuleger, F.: On the Automated Verification of Web Applications with Embedded SQL. In: Benedikt, M., Orsi, G. (eds.) 20th International Conference on Database Theory (ICDT 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 68, pp. 16:1–16:18. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017). https://doi.org/10.4230/LIPIcs.ICDT.2017.16, http://drops.dagstuhl.de/opus/volltexte/2017/7050
22. Martens, W., Neven, F.: Frontiers of tractability for typechecking simple xml transformations. Journal of Computer and System Sciences **73**(3), 362–390 (2007)
23. Martinenghi, D., Christiansen, H., Decker, H.: Integrity checking and maintenance in relational and deductive database and beyond. In: Intelligent Databases: Technologies and Applications, pp. 238–285. IGI Global (2007)
24. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
25. Nipkow, T., Klein, G.: Concrete Semantics. http://concrete-semantics.org/ (2014)
26. Olivé, A.: Integrity constraints checking in deductive databases. In: VLDB. pp. 513–523. Citeseer (1991)
27. openCypher Project: Cypher Query Language Reference, version 9 edn. (2018), http://www.opencypher.org/