A Hoare-Like Calculus Using the SROIQσ Logic on Transformations of Graphs

Jon Brenas, Rachid Echahed, Martin Strecker

A Hoare-like calculus using the SROIQ$^\sigma$ logic on transformations of graphs *

Jon Haël Brenas[1] Rachid Echahed[1] Martin Strecker[2]

[1] CNRS and University of Grenoble
[2] Université de Toulouse / IRIT

Abstract. We tackle the problem of partial correctness of programs processing structures defined as graphs. We introduce a kernel imperative programming language endowed with atomic actions that participate in the transformation of graph structures and provide a decidable logic for reasoning about these transformations in a Hoare-style calculus. The logic for reasoning about the transformations (baptized $\mathcal{SROIQ}^\sigma$) is an extension of the Description Logic (DL) $\mathcal{SROIQ}$, and the graph structures manipulated by the programs are models of this logic. The programming language is non-standard in that it has an instruction set targeted at graph manipulations (such as insertion and deletion of arcs), and its conditional statements (in loops and selections) are $\mathcal{SROIQ}^\sigma$ formulas. The main challenge solved in this paper is to show that the resulting proof problems are decidable.

Keywords: Description Logic; Graph Transformation; Programming Language Semantics; Tableau Calculus

1 Introduction

1.1 Problem Statement and Contribution

The work presented here has arisen out of the authors' effort to prove properties about graph transformations. These transformations are ubiquitous, among others, in traditional imperative programs that modify pointer structures. The obstacle to satisfactory solutions in this area is that traditional programming languages are too expressive and interesting problems often need to be stated in non-decidable logics.

In this paper, we focus on a class of decidable Description Logics (DLs). The spectrum of DLs [1] is well explored, there are numerous application areas, such as capturing the static semantics of modeling languages (in the style of UML) or graph database schemas (in the style of RDF).

To be effective, the transformation is defined in a programming language. We propose an imperative language annotated with pre- and postconditions and

* This work has been funded by projects CLIMT (ANR-11-BS02-016) and TGV (CNRS-INRIA-FAPERGS/156779 and 12/0997-7).

loop invariants. Peculiarities of the language are conditions in *if* and *while* statements that are Boolean queries, and a non-deterministic assignment statement. The language constructs are restricted to structural transformations and have been chosen carefully so that the resulting program verification problem becomes decidable.

Here, program verification means *a priori* verification: Given a program with its pre- and postcondition, can we ascertain that every input structure satisfying the precondition is transformed into a structure satisfying the postcondition? This is in contrast to *a posteriori* verification where satisfaction of the postcondition is checked individually for each graph, once the transformation has been performed. The latter has the disadvantage that the verification has to be done for each single instance (whereas our verification ensures correctness once and for all), and the approach becomes impractical for very large structures.

Technically speaking, we present a programming language, a logic and a Hoare-style *program calculus* relating them. We only consider partial correctness, *i.e.* correctness of a program under the condition that it terminates. We establish that the program calculus is sound *wrt.* the programming language semantics (if a pre-post-relation is established by the calculus, a graph is transformed as required).

The program calculus is related to, but has to be distinguished from a *logic calculus* which is used for establishing the validity of the correctness conditions extracted with the aid of the program calculus. We show that, for the fragment of correctness conditions, there is a logic calculus that is sound, complete and terminates.

Outline of the paper After an introductory example in Sect. 1.2 and a review of related work in Sect. 1.3, we define the logical framework used for expressing program properties and conditions in statements (Sect. 2), before presenting the syntax and semantics of the programming language (Sect. 3). We then turn to more technical issues: intuitively, the extraction of weakest preconditions in Sect. 4 takes a program and its correctness condition and derives a formula whose validity ensures correctness. In Sect. 5, we show how to prove that such formulae are valid.

1.2 Example of program

To get an intuition of the kind of transformation we are aiming at, let's consider the ontology Friend of a friend (FOAF)[1]. It is used to describe persons, their activities and their relationships with other people and objects. Its components are individuals, sets of individuals (called concepts here) and binary relations on individuals (called roles here).

The program whose correctness we want to prove modifies a graph representing this ontology. It is shown in Fig. 2. We consider the problem of moving a researcher R to a laboratory L. As the Friend of a friend ontology is much too

[1] The website of the project can be found at `www.foaf-project.org`

big to be efficiently reproduced in an introduction, we adapt it to our needs, as shown in Fig. 1.

$$
\begin{aligned}
\mathit{onto} \;:=\; & \textsc{Researcher} \subseteq \textsc{Agent} \wedge \textsc{Lab} \subseteq \textsc{Agent} \wedge \textsc{Distinguished} \subseteq \textsc{Agent} \\
& \wedge\ \textsc{Topic\_interest} :: \textsc{Agent} \times \textsc{Thing} \\
& \wedge\ \textsc{Topic} :: \textsc{Document} \times \textsc{Thing} \\
& \wedge\ \textsc{Publication} :: \textsc{Researcher} \times \textsc{Document} \\
& \wedge\ \textsc{Member} :: \textsc{Lab} \times \textsc{Agent}
\end{aligned}
$$

Fig. 1. An ontology example

The concept Researcher is used to represent researchers, the concept Lab represents laboratories, Agent is the concept of those that can "act", Thing is the representation of topic and finally Distinguished singles out those that have received a distinction. The role Topic_interest is used to represent the topics of interest of an agent while Topic represents the subjects of a document. The role Publication links a person to her publications. The role Member lists the members of a group.

The ontology provides the formal definition of the relationships between concepts and roles. It states, for instance, that Researchers are Agents and that the role Member relates Labs with their own Agents.

The precondition of the program stipulates (see Fig. 2) that the ontology is respected before starting the program, that L is a Lab, and that R is a Researcher.

The if statement then checks if the researcher is listed as being a member of a laboratory. If it is the case, we select that laboratory PL and we remove the researcher from their roster. If, in addition, there is no researcher left with a distinction, PL loses its Distinguished quality.

Now that the researcher is available, we add the fact that he is a member of L. In case R was Distinguished, L becomes Distinguished. L may well have been Distinguished before the arrival of R but that is not relevant.

The while loop adds all the topics that R has written articles about to the subjects that interest L. This is done by going through the set of Things that are not a topic of interest for the laboratory ($\neg(L\ \textsc{Topic\_interest}\ t)$) but that are the topic of a publication by R ($R : \mathbf{Ex}\ \textsc{Publication}\ (\mathbf{Ex}\ \textsc{Topic}\ \{t\})$). While this set is not empty, we select one of its elements and we add it to the topics of interest of L. The size of the set thus decreases, which is encouraging but of no great importance as our framework does not check termination. The invariant of the loop is the same as the precondition.

The postcondition states that the ontology structure is satisfied again, that L is still a Lab, that R is still a person, that all the subjects of the publications

```
vars R, L, To, PL;
concepts Researcher, Lab, Document, Agent, Thing, Distinguished;
roles Topic_interest, Topic, Publication, Member;

pre : onto ∧ R : Researcher ∧ L : Lab;

if ∃ l. l Member R then {
   select PL with PL Member R;
   delR (PL Member R);
   if PL : All Member (¬ Distinguished) then {
      delC (PL : Distinguished)
   };
};
addR(L Member R);
if R : Distinguished then {
   addC(L : Distinguished)
};
while (∃ t. ¬(L Topic_interest t) ∧ R : Ex Publication (Ex Topic {t})) {
inv : onto ∧ R : Researcher ∧ L : Lab
  select To
     with ¬(L Topic_interest To) ∧ R : Ex Publication (Ex Topic {To});
  addR(L Topic_interest To)
};

post : onto ∧ R : Researcher ∧ L : Lab
∧ R : All Publication (All Topic (Ex Topic_interest⁻ {L}))
∧ L Member R
∧ R : Distinguished ⇒ L : Distinguished;
```

Fig. 2. An example program

of R are TOPICS OF INTERESTS of L, that R is now a MEMBER of L and that if R is DISTINGUISHED so is L.

A small graph illustrating the transformation is shown in Fig. 3. A distinguished researcher R which belongs to lab LB moves to lab L. The arrow relating R to LB is removed and a new one relating R to L is added. Moreover, LB no longer is distinguished. New arrows are created relating L to the topics of interest of R, say T_0 and T_1.

1.3 Related work

Reasoning about graph transformations in full generality is hard [9]. A first step towards the verification of programs operating on graphs has been made in [7] where the authors follow Dijkstra's approach to program verification by

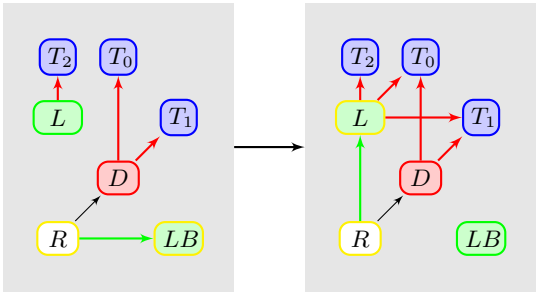


Fig. 3. Resulting transformation

constructing the weakest preconditions for so-called high-level programs. Pre- and post-conditions are expressed as *nested graph conditions*. These conditions have also been used recently in [12] where a Hoare-style program verification on graph programs has been proposed. Unfortunately the verification problem in these two proposals is undecidable in general.

Some decidable logics for graph transductions are known, such as MSO [6], but these are descriptive, applicable to a limited number of graphs and often do not match with an algorithmic notion of transformation. Some implementations of verification environments for pointer manipulating programs exist [11], but they often impose severe restrictions on the kind of graphs that can be manipulated, such as having a clearly identified spanning tree.

In [4], the authors investigated the static verification of the evolution of graph databases where integrity constraints are expressed in a description logic called $\mathcal{ALCHOIQ}br$. This work is very close to our proposal. However, the authors did consider only programs consisting of finite sequences of atomic actions. These actions may compute the union or the difference of roles and concepts. Their verification procedure is based on a transformation TR [4, Definition 5] which mimics the computation of weakest preconditions in Hoare's like calculi.

Work on Knowledge Bases (KB) updates [10] seems to approach the problem from the opposite direction: Add facts to a KB and transform the KB at the same time such that certain formulas remain satisfied. In our approach, the modification of the KB is exclusively specified by the program.

The present paper is a follow-up of a previous one by the authors [5] working on a simpler description logic (ALCQ) and a simpler programming language. In order to obtain decidable verification conditions, the logic $\mathcal{SROIQ}^\sigma$ requires more subtle restrictions on the form of assertions occurring in programs. The decision procedure (a tableau algorithm) differs from the one presented in [5].

2 The Logic $\mathcal{SROIQ}^\sigma$

In this section, we introduce a new description logic we call $\mathcal{SROIQ}^\sigma$. It is an extension of the description logic $\mathcal{SROIQ}$ [8] augmented with a notion of sub-

stitution. We show that the satisfiability problem in $\mathcal{SROIQ}^{\sigma}$ is decidable. The decision procedure is intended to be as general as possible and to be adaptable to a wide variety of logics of the description logic family, under certain assumptions.

We start by some basic definitions.

Definition 1 (Concept and role names; nominals). *Let* $\boldsymbol{C}$ *be a set of* ***concept names*** *including a subset* $\boldsymbol{N}$ *of* ***nominals***, $\boldsymbol{R}$ *a set of* ***role names*** *including the universal role* U *and* $\boldsymbol{I}$ *a set of* ***individuals***. *The set of roles is* $\boldsymbol{R} \cup \{R^- | R \in \boldsymbol{R}\}$, *where a role* R^- *is called the* ***inverse role*** *of* R.

Example 1. In our example, $\mathbf{C} = \{$Researcher, Lab, Agent, Thing, Distinguished, R, L, PL, To$\}$, $\mathbf{N} = \{$ R, L, PL, To$\}$ and $\mathbf{R} = \{$Topic_interest, Topic, Publication, Member, $U\}$.

As usual, an **interpretation** $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ consists of a set $\Delta^{\mathcal{I}}$, called the **domain** of $\mathcal{I}$, and a **valuation** $.^{\mathcal{I}}$ which associates with every concept name C a subset $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, which is a singleton for each nominal, with each role name R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, with the universal role U the universal relation $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ and with each individual name a an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. The technical definition of interpretations could be consulted in *e.g.* [8,3].

The considered logic allows one to provide so-called *role axioms*. A role axiom can either be a role inclusion axiom or a role assertion. We will deal with the role inclusion axioms first. For that, we need to define an ordering on roles.

Definition 2. *A strict partial order* $\prec$ *on a set* A *is an irreflexive and transitive relation on* A. *A strict partial order* $\prec$ *on the set of roles is called a* ***regular order*** *if* $\prec$ *satisfies, additionally,* $S \prec R \Leftrightarrow S^- \prec R$ *for all roles* R *and* S.

Definition 3. *A* ***role inclusion axiom*** *is an expression of the form* $w \subseteq R$ *where* w *is a finite string of roles not containing the universal role* U *and* R *is a role name, with* $R \neq U$. *A* ***role hierarchy*** $\mathcal{R}_h$ *is a finite set of role inclusion axioms. A role inclusion axiom* $w \subseteq R$ *is* $\prec$***-regular*** *if* R *is a role name and* w *is defined by the following grammar:*
$w = RR \mid R^- \mid S_1 \dots S_n \mid RS_1 \dots S_n \mid S_1 \dots S_n R$ *with* $S_i \prec R$ *for all* $1 \leq i \leq n$.
Finally, a role hierarchy $\mathcal{R}_h$ *is* ***regular*** *if there exists a regular order* $\prec$ *such that each role inclusion axiom in* $\mathcal{R}_h$ *is* $\prec$*-regular. An interpretation* ***satisfies*** *a role inclusion axiom* $w \subseteq R$ *if the interpretation of* w *is included in the interpretation of* R. *An interpretation is a* ***model*** *of a role hierarchy* $\mathcal{R}_h$ *if it satisfies all role inclusion axioms in* $\mathcal{R}_h$.

Example 2. Let us consider the roles Brother and Sibling with their intuitive meanings, it seems correct that Brother $\subseteq$ Sibling.

The second possible kind of role axiom is the role assertion.

Definition 4 (Role assertions). *For role names R,S, we call the assertions Ref(R) (role reflexivity), Irr(R) (role irreflexivity), Sym(R) (role symmetry), Asy(R) (role asymmetry), Tra(R) (role transitivity) and Dis(R,S) (role disjunction)* ***role assertions***.

Example 3. To keep with the roles previously defined, Sym(SIBLING) and Tra(SIBLING) are a correct set of role assertions.

One can observe that some of the role assertions (namely transitivity and symmetry) are simply a rewriting of some role axioms: $Sym(R)$ is equivalent to $R^- \subseteq R$, and $Tra(R)$ is equivalent to $RR \subseteq R$. For these reasons, we will henceforth only consider role assertions without Sym and Tra.

Finally, when introducing complex concepts, we will need simple roles to avoid undecidability. Intuitively, a simple role is a role that does not appear as the right-hand side of a role inclusion axiom whose left-hand side is a string composed of at least two roles.

Definition 5 (Simple role). *Given a role hierarchy $\mathcal{R}_h$ and a set of role assertions $\mathcal{R}_a$, a simple role is inductively defined as either a role name that does not occur in the right-hand side of any role inclusion axiom, or R^- for R simple, or the right-hand side of a role inclusion axiom $w \subseteq R$ where w is a simple role. $\mathcal{R}_a$ is called **simple** if all roles appearing in role assertions are simple.*

Starting from now, the only role hierarchies that we consider are regular and the only sets of role assertions that we consider are finite and simple.

Definition 6 (Concept). *A concept is defined as:*
$C ::= \bot \mid c \mid \neg\, C \mid C \sqcap D \mid C \sqcup D \mid (\geq n\ S\ C) \mid (< n\ S\ C) \mid$ **Ex** $R\ C \mid$ **All** $R\ C \mid \{o\} \mid$ **Ex** S **Self** $\mid C\ subst$
where c is a concept name, R is a role, S is a simple role, o is a nominal, C, D are concepts and $subst$ is a substitution.

Intuitively, $\neg C$ stands for the complement of C with respect to the domain of interpretation. $C \sqcap D$ (respectively $C \sqcup D$) stands for the intersection (respectively the union) of concepts. $(\geq n\ S\ C)$ (respectively $(< n\ S\ C)$) stands for the set of elements related via role S to at least n (respectively at most n-1) distinct individuals of concept C. **Ex** $R\ C$ stands for the set of elements related via role R to at least one individual of concept C and **All** $R\ C$ stands for the set of elements related via role R only to elements of concept C. $\{o\}$ stands for the singleton associated to nominal o. **Ex** S **Self** stands for the set of elements related to themselves via role S. $C\ subst$ stands for the set of elements of C updated according to the substitution $subst$. Missing definitions can be found in [3].

Substitutions, that appear in the last constructor, allow one to modify roles and concepts by adding or removing individuals. Substitutions, being the difference between $\mathcal{SROIQ}^\sigma$ and $\mathcal{SROIQ}$, are defined next. As will be shown later on, the computation of weakest preconditions and verification conditions may generate substitutions of the following form.

Definition 7 (Substitution). *Given a concept name c and a role name R, a substitution is:*

$subst ::= \epsilon$ *(empty substitution)*
$\quad\quad | \quad [RS]$ *(role substitution)*
$\quad\quad | \quad [CS]$ *(concept substitution)*
A role substitution is defined as follows:
$RS ::= R - (i, j)$ *(deletion of relation instance)*
$\quad\quad | \quad R + (i, j)$ *(insertion of relation instance)*
while a concept substitution is defined as follows:
$CS ::= c - i$ *(deletion of a concept instance)*
$\quad\quad | \quad c + i$ *(insertion of a concept instance)*

Example 4. All elements of **C** are examples of concepts. Another example of concept is **All** MEMBER ¬DISTINGUISHED. It can be translated into "none of the members is distinguished".

Theorem 1. *If Φ_0 is a concept and $\mathcal{R}_h$ is a regular role hierarchy and $\mathcal{R}_a$ is a finite simple set of role assertions, the satisfiability of $\mathcal{R}_h \wedge \mathcal{R}_a \wedge \Phi_0$ is decidable.*

The proof of Theorem 1 can be found in [3].

In the following, we introduce the notions of assertions and conditions used in the rest of the paper.

Definition 8 (Assertion). *An assertion is defined as either:*
$assert ::= i : C \mid i\ R\ j \mid i\ (\neg\ R)\ j \mid i = j \mid i \neq j \mid role\_axiom \mid \neg\ assert \mid assert \wedge assert \mid assert \vee assert \mid$ **All** $U\ C \mid$ **Ex** $U\ C \mid \forall\ i.assert \mid \exists\ i.assert \mid assert\ subst$
where C is a concept, role_axiom is either a role inclusion axiom or a role assertion, i, j are individuals, R is a role, U is the universal role defined previously and subst is a substitution.

Example 5. Assertions without substitutions are the building blocks of ontologies. Our simplified example of FOAF (cf. Fig. 1) contains two main kinds of assertions. The first deals with the hierarchy of concepts and the second one with the concepts of the elements linked by a role.

Among others, RESEARCHER $\subseteq$ AGENT is a short way of writing the assertion **All** U ¬RESEARCHER $\sqcup$ AGENT which can be translated into "researchers are agents".

In the next definition, we introduce the notion of *conditions*, which is used in Sect. 3 in while-loops, if-statements and select-statements.

Definition 9 (Condition). *A condition is an assertion without role axioms and without quantification on individuals, that is no sub-expression of the form $\forall$ i.assert or $\exists$ i.assert.*

3 Programming Language

3.1 Syntax

In this section, we introduce the programming language for performing transformations (see the example of Fig. 2). The programming language is an imperative

language manipulating relational structures. Its distinctive features are conditions (in conditional statements and loops) in the sense of Sect. 2. These formulas can be understood as Boolean queries on a database. The language also has a non-deterministic assignment statement allowing to select an element satisfying a condition. This corresponds to a database query retrieving an element satisfying a condition. Traditional types (numbers, inductive types) are not provided in the language.

In this paper, we only consider a core language with traditional control flow constructs, but without procedures. The language has primitives for adding an individual element to a concept, or for removing it. Similarly, there are primitives for the insertion or removal of edges. Thus, it is only possible to modify a relational structure, but not to allocate or deallocate objects, in a strict sense.

The abstract syntax of statements is defined by:

stmt ::=	`Skip`	(empty stmt)
	`addC(`i : c`)`	(insert element)
|	`delC(`i : c`)`	(delete element)
|	`addR(`i R j`)`	(insert edge)
|	`delR(`i R j`)`	(delete edge)
|	`select` i `with` *cond*	(assignment)
|	*stmt* `;` *stmt*	(sequence)
|	`if` *cond* `then` { *stmt* } `else` { *stmt* }	
|	`while` *cond* { `inv:` *assert* *stmt*}	

The non-terminals *cond* and *assert* corresponds, respectively, to conditions (defined in Def 9) and assertions (defined in Def 8). i and j stand for individuals, c stands for a concept name and R stands for a role name. There are two variants of insertion and deletion operations (for individuals and a concept name (`addC` and `delC`) and for two individuals and a relation name (`addR` and `delR`)).

A program is a statement embedded in declarations of variables, concepts and roles and a pre- and a postcondition.

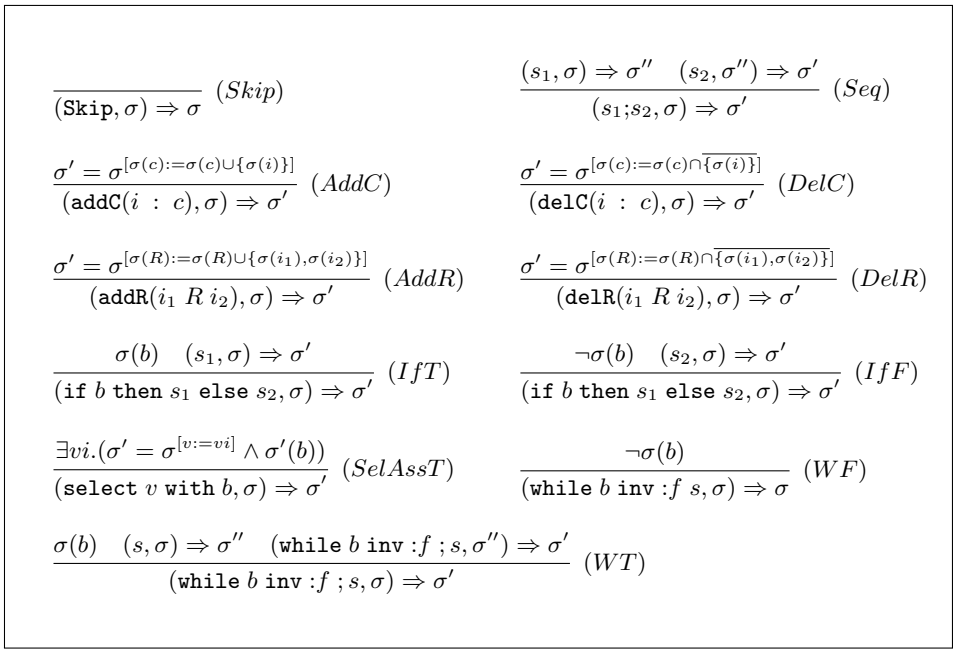
prog ::= `vars` $\overrightarrow{i}$ `; concepts` $\overrightarrow{c}$ `; roles` $\overrightarrow{R}$ `;`
`pre:` *assert*`;` *stmt*`;` `post:` *assert*`;`

3.2 Semantics

The semantics is a big-step semantics describing how a state evolves during the execution of a statement. The state is a relational structure, and the state space is just the type of interpretations. In accordance with traditional notation in semantics, we use the symbol σ to denote a state. We may therefore write $\sigma(b)$ to evaluate the condition b in state σ.

The rules have the form $(s, \sigma) \Rightarrow \sigma'$ expressing that executing statement s in state σ produces a new state σ'. The rules of the semantics are given below. Beware that we overload logical symbols such as $\exists$, $\wedge$ and $\neg$ for use in the meta-syntax and as constructors of *assert*.

The rules of the traditional control constructs are standard, apart from the fact that we do not use expressions as conditions. The invariant in the *while*-



$$\frac{}{(\texttt{Skip}, \sigma) \Rightarrow \sigma}\ (Skip) \qquad \frac{(s_1, \sigma) \Rightarrow \sigma'' \quad (s_2, \sigma'') \Rightarrow \sigma'}{(s_1;s_2, \sigma) \Rightarrow \sigma'}\ (Seq)$$

$$\frac{\sigma' = \sigma^{[\sigma(c):=\sigma(c)\cup\{\sigma(i)\}]}}{(\texttt{addC}(i\ :\ c), \sigma) \Rightarrow \sigma'}\ (AddC) \qquad \frac{\sigma' = \sigma^{[\sigma(c):=\sigma(c)\cap\overline{\{\sigma(i)\}}]}}{(\texttt{delC}(i\ :\ c), \sigma) \Rightarrow \sigma'}\ (DelC)$$

$$\frac{\sigma' = \sigma^{[\sigma(R):=\sigma(R)\cup\{\sigma(i_1),\sigma(i_2)\}]}}{(\texttt{addR}(i_1\ R\ i_2), \sigma) \Rightarrow \sigma'}\ (AddR) \qquad \frac{\sigma' = \sigma^{[\sigma(R):=\sigma(R)\cap\overline{\{\sigma(i_1),\sigma(i_2)\}}]}}{(\texttt{delR}(i_1\ R\ i_2), \sigma) \Rightarrow \sigma'}\ (DelR)$$

$$\frac{\sigma(b) \quad (s_1, \sigma) \Rightarrow \sigma'}{(\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \sigma) \Rightarrow \sigma'}\ (IfT) \qquad \frac{\neg\sigma(b) \quad (s_2, \sigma) \Rightarrow \sigma'}{(\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \sigma) \Rightarrow \sigma'}\ (IfF)$$

$$\frac{\exists vi.(\sigma' = \sigma^{[v:=vi]} \wedge \sigma'(b))}{(\texttt{select}\ v\ \texttt{with}\ b, \sigma) \Rightarrow \sigma'}\ (SelAssT) \qquad \frac{\neg\sigma(b)}{(\texttt{while}\ b\ \texttt{inv}:f\ s, \sigma) \Rightarrow \sigma}\ (WF)$$

$$\frac{\sigma(b) \quad (s, \sigma) \Rightarrow \sigma'' \quad (\texttt{while}\ b\ \texttt{inv}:f\ ; s, \sigma'') \Rightarrow \sigma'}{(\texttt{while}\ b\ \texttt{inv}:f\ ; s, \sigma) \Rightarrow \sigma'}\ (WT)$$

Fig. 4. Big-step semantics rules

loop is without operational significance. It is only used for calculating weakest preconditions (Sect. 4).

For lack of space, we do not detail all the rules here as they are quite intuitive. We roughly explain rules `addC` and `select`:

- `addC`(i : c) adds a node to a concept. Adding an already existing element has no effect (*i.e.* , is not perceived as an error). $[\sigma(c) := \sigma(c) \cup \{\sigma(i)\}]$ modifies the interpretation for c to include the element that i denotes.

$$\frac{\sigma' = \sigma^{[c:=\sigma(c)\cup\{\sigma(i)\}]}}{(\texttt{addC}(i\ :\ c), \sigma) \Rightarrow \sigma'}\ (AddC)$$

- `select` i `with` b selects an element vi from the semantic domain that satisfies condition b (note that i typically occurs in b), and assigns it to i. The subsequent statements are then executed with i bound to vi. For example, `select` a `with` $a : A \wedge (a\, R\, d)$ selects an element which is an instance of concept A and is R-related to a given element referred to by d, and assigns it to a. More formally, we pick an instance $vi \in \Delta$, check whether the condition b would be satisfied under this choice, and if this is the case, keep this assignment:

$$\frac{\exists vi.(\sigma' = \sigma^{[i:=vi]} \wedge \sigma'(b))}{(\texttt{select}\ i\ \texttt{with}\ b, \sigma) \Rightarrow \sigma'}\ (SelAssT)$$

Note that the semantics blocks (*i.e.* , there is no successor state) in case no instance satisfying the condition exists.

4 Verification Conditions

4.1 Generating Verification Conditions

We follow the standard approach for verifying that a program satisfies its specification: If the program has precondition pre, statement s and postcondition $post$, we compute the weakest precondition $wp(s, post)$ and then show that it is implied by the precondition. Using the terminology of Sect. 1.1, this section is thus concerned with a program calculus.

The definition of wp is given in Fig. 5. Let us insist on one point: in traditional expositions of Hoare calculi, substitution is a meta-operation which syntactically replaces a symbol by an expression. This works as long as the syntax of the logic is closed under meta-substitutions, which is not the case we consider. For example, a replacement of R by $R - (v_1, v_2)$ in $(< n\ R\ C)$ would yield a syntactically ill-formed concept expression. This motivates our introduction of explicit substitutions as a constructor.

$$
\begin{array}{l}
wp(\texttt{Skip},\ Q) = Q \\
wp(\texttt{addC}(i\ :\ c)\ Q) = Q[c := c + i] \\
wp(\texttt{delC}(i\ :\ c),\ Q) = Q[c := c - i] \\
wp(\texttt{addR}(i_1\ R\ i_2),\ Q) = Q[R := R + (i_1, i_2)] \\
wp(\texttt{delR}(i_1\ R\ i_2),\ Q) = Q[R := R - (i_1, i_2)] \\
wp(\texttt{select}\ i\ \texttt{with}\ b,\ Q) = \forall i.(b \longrightarrow Q) \\
wp(s_1; s_2,\ Q) = wp(s_1, wp(s_2,\ Q)) \\
wp(\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2,\ Q) = (b \rightarrow wp(s_1, Q)) \wedge (\neg b \rightarrow wp(s_2, Q))) \\
wp(\texttt{while}\ b\ \texttt{inv}\ :f\ s,\ Q) = f
\end{array}
$$

Fig. 5. Weakest preconditions

Also, our *while*-loops are supposed to be annotated with invariants. Whether these invariants necessarily have to be supplied by the human end-user or whether they could be inferred automatically in a pre-processing step is not subject of concern here. In any case, program verification also has to ascertain that the given loop annotation has the desired properties of an invariant: being preserved during execution of the loop body, and ensuring the postcondition when the loop terminates. Recursively collecting these verification conditions is done by function $vc(s, post)$ for a statement s and postcondition $post$ (Fig. 6).

$$
\begin{aligned}
&vc(\texttt{Skip},\ Q) = \top \\
&vc(\texttt{add}(i\ :\ c),\ Q) = \top \\
&vc(\texttt{delete}(i\ :\ c),\ Q) = \top \\
&vc(\texttt{add}(i_1\ R\ i_2),\ Q) = \top \\
&vc(\texttt{delete}(i_1\ R\ i_2),\ Q) = \top \\
&vc(\texttt{select}\ i\ \texttt{with}\ b,\ Q) = \top \\
&vc(s_1; s_2,\ Q) = vc(s_1, wp(s_2, Q)) \wedge vc(s_2, Q) \\
&vc(\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2,\ Q) = vc(s_1, Q) \wedge vc(s_2, Q) \\
&vc(\texttt{while}\ b\ \texttt{inv}: f\ s,\ Q) = (f \wedge \neg b \longrightarrow Q) \\
&\qquad\qquad\qquad\qquad \wedge (f \wedge b \longrightarrow wp(s, f)) \wedge vc(s, f)
\end{aligned}
$$

Fig. 6. Verification conditions

4.2 Correctness

The two aforementioned criteria are used to define the *correctness condition* of a program *prog* with precondition *pre*, statement s and postcondition *post*:

$$correct(pre,\ s,\ post) =_{def} vc(s, post) \wedge (pre \longrightarrow wp(s, post))$$

We now have the necessary notions to state the soundness of our program calculus:

Theorem 2 (Soundness). *Let prog be a program with precondition pre, statement s and postcondition. post If correct(pre, s, post) is valid, then for all states σ and σ', if $(s, \sigma) \Rightarrow \sigma'$, then $\sigma(pre)$ implies $\sigma'(post)$.*

The proof of this theorem is straightforward and is done by induction on the structure of the statements.

5 Proving Verification Conditions

Let us recapitulate the development so far: In Sect. 3, we have presented a programming language annotated with formulas specifying the correctness of programs. In Sect. 4, we have given a program calculus (embodied by function *correct*) that takes an annotated program, removes all computational contents and returns a formula, say Φ. For sake of decidability of the verification program, we focus in this section on assertions which generate a particular formula Φ we call *essentially universally quantified*.

Definition 10 (Essentially quantified). *We say that an assertion Φ is **essentially universally quantified** (respectively **essentially existentially quantified**) if the occurrences of $\forall$ in Φ are only below an even (respectively odd) number of negations and the occurrences of $\exists$ in Φ are only below an odd (respectively even) number of negations.*

Lemma 1 (Universally quantified).

1. *Let Q be essentially universally quantified. Assume that the invariants in statement s do not include negated role axioms. Then $wp(s, Q)$ and $vc(s, Q)$ are essentially universally quantified.*
2. *If pre (respectively post) is essentially existentially (respectively universally) quantified and the invariants in statement s do not include negated role axioms , then correct(pre, s, post) is essentially universally quantified and does not contain substitutions over negated role axioms.*

We now discuss briefly a decision procedure for verifying the validity of essentially universally quantified formulae. For more details see [2].

Actually, what we have to do is to prove that we can apply Theorem 1 to prove the validity of *correct*(*pre*, *s*, *post*) whenever it is essentially universally quantified.

The first thing to do is to make sure that substitutions only affect the basic components of the assertion (such as role axioms and concepts). This can be done by pushing substitutions by using the following rules.

- $(\neg\, assert)\ subst \rightsquigarrow \neg\, (assert\ subst)$
- $(assert_1\ \wedge\ assert_2)\ subst \rightsquigarrow (assert_1\ subst)\ \wedge\ (assert_2\ subst)$
- $(assert_1\ \vee\ assert_2)\ subst \rightsquigarrow (assert_1\ subst)\ \vee\ (assert_2\ subst)$
- $(\forall i.\ assert)\ subst \rightsquigarrow \forall i.\ (assert\ subst)$
- $(\exists i.\ assert)\ subst \rightsquigarrow \exists i.\ (assert\ subst)$

It happens that the formulae generated by *correct*, after pushing the substitutions, may include substitutions over role axioms as well as quantifiers which prevent the direct use of Theorem 1. To overcome this drawback we actually show that we can get rid of those substitutions by means of a set of transformation rules. Unfortunately, there is not enough room here to give all the transformations. We give below two examples of such rules. The first rule shows how to get rid of a particular substitution $[R := R + (i_1, i_2)]$ when applied to the role axiom $Asym(R)$. The second rule shows how to get rid of a particular substitution $[R := R - (i_1, i_2)]$ when applied to the role axiom $s_1 \ldots s_n R \subseteq R$.

- $Asym(R)[R := R + (i_1, i_2)] \rightsquigarrow i_2\ \neg R\ i_1 \wedge Asym(R)$ that is R will be asymmetric after adding the edge (v_1, v_2) to R if R was asymmetric before and (i_2, i_1) is not already part of R.
- $(s_1 \ldots s_n R \subseteq R)[R := R - (i_1, i_2)] \rightsquigarrow \forall x.\forall y.\forall z.x : \mathbf{All}\ s_1 \ldots \mathbf{All}\ s_n\ \neg\{y\} \vee y\ (\neg r)\ z \vee (x\ =\ i_1 \wedge z\ =\ i_2) \vee x\ R\ z$. That can be rewritten as $\forall x.\forall y.\forall z.\ (x : \mathbf{Ex}\ s_1 \ldots \mathbf{Ex}\ s_n\ \{y\} \wedge y\ R\ z \wedge (x\ \neq\ i_1 \vee z\ \neq\ i_2)) \Longrightarrow x\ R\ z$ that is $s_1 \ldots s_n R \subseteq R$ after removing (i_1, i_2) from R if for each couple (x, z) different from (i_1, i_2), for each element y such that there is a path $s_1 \ldots s_n$ from x to y and $y\ R\ z$, then $x\ R\ z$.

Lemma 2. *For every essentially universally quantified formula not containing substitutions over negated role axioms, there is an equivalent universally quantified formula without substitutions on role axioms.*

Now that substitutions only occur over concepts, we get a formula $\Phi_1$which is essentially universally quantified. The last step before using Theorem 1 consists in eliminating the quantifiers of Φ_1. The rough lines of the procedure for determining whether $\Phi_1$is valid are spelled out in the following.

1. Convert $\Phi_1$to an equivalent prenex normal form p, which will consist of a prefix of universal quantifiers, and a quantifier-free body: $\forall x_1 \ldots x_n.b$
2. p is valid iff its universal closure $ucl(p)$ (universal abstraction over all free variables of p) is.
3. Show the validity of $ucl(p)$ by showing the unsatisfiability of $\neg ucl(p)$.
4. $\neg ucl(p)$ has the form $\neg\forall v_1 \ldots v_k, x_1 \ldots x_n.b$. Pull negation inside the universal quantifier prefix, remove the resulting existential quantifier prefix, and show unsatisfiability of $\neg b$ by using Theorem 1.

Computation of prenex normal forms is standard. Care has to be taken to avoid capture of free variables, by renaming bound variables. Free variables are defined as usual; the free variables of a substitution $f[R := R-(i_1, i_2)]$ are those of f and in addition i_1 and i_2 (similarly for edge insertion). We illustrate the problem with the following statement prg:
select a **with** $a : A$; **select** b **with** $b\ R\ a$;
select a **with** $a\ \neg R\ b$; **addR**$(b\ R\ a)$

Assume the post-condition is $Asym(R)$, we obtain $wp(prg, Q) =$
$\forall a.a : A \longrightarrow \forall b.(b\ R\ a) \longrightarrow \forall a.(a\ \neg R\ b) \longrightarrow Asym(R)[R := R + (b, a)]$.
Removing the substitution yields
$wp(prg, Q) = \forall a.a : A \longrightarrow \forall b.(b\ R\ a) \longrightarrow \forall a.(a\ \neg R\ b) \longrightarrow (a\ \neg R\ b \wedge Asym(R))$
whose prenex normal form
$\forall a_1, b, a_2.\ (a_1 : A \longrightarrow (b\ R\ a_1) \longrightarrow (a_2\ \neg R\ b) \longrightarrow (a_2\ \neg R\ b \wedge Asym(R)))$
contains more logical variables than prg contains program variables.

After removing the quantifiers and taking the negation, we obtain
$\neg(a_1 : A \longrightarrow (b\ R\ a_1) \longrightarrow (a_2\ \neg R\ b) \longrightarrow (a_2\ \neg R\ b \wedge Asym(R)))$ an assertion without substitutions over role axioms and without quantifiers on individuals whose unsatisfiability is equivalent to the validity of *correct*. This assertion fits the conditions of Theorem 1 and thus the validity of *correct* is decidable.

6 Conclusions

This paper proposes a language for rewriting graphs, and methods for reasoning about the correctness of these programs, by means of a Hoare-style calculus. DL formulas are directly integrated into the statements of the programming language. The verification conditions extracted from these programs have been shown to be decidable.

The work described here is still not entirely finished, and the following points indicate directions for future investigations:

– We are in the process of coding the theory in the Isabelle proof assistant. Most proofs concerning the elimination of substitutions and the tableau algorithm still have to be done. The purpose is to obtain a framework that will allow us to experiment more easily with variations of the logic.

– We have currently focused on the logic $\mathcal{SROIQ}^\sigma$, which is one of the most expressive description logics. It might be interesting to consider less expressive logics which offer more space for optimizations. The process described in Sect. 5 is rather generic, but it remains to be seen which other DLs can be accommodated.
– In a similar vein, it would be interesting to implement a transformation engine on the basis of the language described here, also with the purpose of evaluating the practical expressiveness of the language on larger examples.

References

1. Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003.
2. Jon Haël Brenas, Rachid Echahed, and Martin Strecker. A hoare-like calculus using the $\mathcal{SROIQ}^\sigma$ logic on transformation of graphs (extended version). *CoRR, arxiv.org*, 2014.
3. Jon Haël Brenas, Rachid Echahed, and Martin Strecker. $\mathcal{SROIQ}^\sigma$ is decidable. *CoRR, arxiv.org*, 2014.
4. Diego Calvanese, Magdalena Ortiz, and Mantas Simkus. Evolving graph databases under description logic constraints. In *Proc. of the 26th Int. Workshop on Description Logics (DL 2013)*, volume 1014 of *CEUR Electronic Workshop Proceedings*, pages 120–131, 2013.
5. Mohamed Chaabani, Rachid Echahed, and Martin Strecker. Logical foundations for reasoning about transformations of knowledge bases. In Thomas Eiter, Birte Glimm, Yevgeny Kazakov, and Markus Krötzsch, editors, *DL Description Logics*, volume 1014 of *CEUR Workshop Proceedings*, pages 616–627. CEUR-WS.org, 2013.
6. Bruno Courcelle and Joost Engelfriet. *Graph structure and monadic second-order logic, a language theoretic approach*. Cambridge University Press, 2011.
7. Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In *Third International Conference on Graph Transformations (ICGT)*, volume 4178 of *Lecture Notes in Computer Science*, pages 445–460. Springer, 2006.
8. Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible SROIQ. In *Proc. of the 10th International Conference of Knowledge Representation and Reasoning (KR-2006, Lake District UK)*, 2006.
9. Neil Immerman, Alex Rabinovich, Tom Reps, Mooly Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic*, volume 3210 of *Lecture Notes in Computer Science*, pages 160–174. Springer Berlin / Heidelberg, 2004.
10. Hongkai Liu, Carsten Lutz, Maja Milicic, and Frank Wolter. Foundations of instance level updates in expressive description logics. *Artificial Intelligence*, 175(18):2170–2197, 2011.
11. Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, pages 221–231, 2001.
12. Christopher M. Poskitt and Detlef Plump. Hoare-style verification of graph programs. *Fundam. Inform.*, 118(1-2):135–175, 2012.