Provably correct graph transformations with small-tALC

Nadezhda Baklanova, Jon Haël Brenas, Rachid Echahed, Christian Percebois, Martin Strecker, Hanh Nhi Tran

Provably correct graph transformations with small-t$\mathcal{ALLC}^\star$

Nadezhda Baklanova[2], Jon Haël Brenas[1], Rachid Echahed[1], Christian Percebois[2], Martin Strecker[2], Hanh Nhi Tran[2]

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

Abstract. We present a prototype for executing and verifying graph transformations. The transformations are written in a simple imperative programming language, and pre- and post-conditions as well as loop invariants are specified in the Description Logic $\mathcal{ALC}$ (whence the name of the tool). The programming language has a precisely defined operational semantics and a sound Hoare-style calculus. The tool consists of the following sub-components: a compiler to Java for executing the transformations; a verification condition generator; and a tableau prover for an extension of $\mathcal{ALC}$ capable of deciding the generated verification conditions. A description of these components and their interaction is the main purpose of this paper.

Keywords: Graph Transformations, Programming Language Semantics, Tableau Calculus, Description Logic
Key Terms: ModelBasedSoftwareDevelopmentMethodology, FormalMethod, MathematicalModel, VerificationProcess

1 Introduction

Provably correct transformations of graph structures become increasingly important, for example for pointer manipulating programs, model driven engineering (such as EMF [1]) or the Semantic Web (with representation formats such as RDF [2]).

Contributions: This paper presents a new language, called small-t$\mathcal{ALC}$, and accompanying programming environment for executing graph transformations and reasoning about them. Let us characterize in a few words what our work is about and what it is not about:

– The primary aim of our development is to be able to *reason about graph transformations* in a pre- / post-condition style: can we ensure that any graph satisfying the pre-condition is transformed into a graph satisfying the post-condition? Essential ingredients of such a setup are a language for describing the transformations, and an assertional formalism for specifying the pre- and post-conditions.

⋆ Part of this research has been supported by the *Climt* project (ANR-11-BS02-016).

– The *transformation language* is an imperative programming language with special operations for manipulating graphs. This language is endowed with traditional control flow constructs (selection and loops) and elementary statements for adding and deleting arcs of a graph. There is a `select` statement that can be understood as a generalized, non-deterministic assignment operation and whose purpose is to perform matchings of rules in a target graph. After a high-level overview of small-t$\mathcal{ALC}$ (Section 2), we will give a more detailed account of the program logic (in Section 3.1) and transformation language (in Section 3.2). Our transformation language is by no means a full-fledged programming language: for example, arithmetic operations are excluded.
– The transformation language is *not graphical*, but textual. We do not question the utility and appeal of a graphical notation, but this issue is orthogonal to our concerns. We can imagine to couple small-t$\mathcal{ALC}$ with existing graphical editors, such as Henshin [3], in the sense of translating a graphical description of a rule to our textual format. The usefulness of the inverse direction is less evident, because the textual format is more expressive (offering, among others, nested loops and branching statements).
– The transformation language is *executable*, by a translation to Java (see Section 4): a code generator translates small-t$\mathcal{ALC}$ to Java code, which can then transform graphs specified in an appropriate format.

Altogether, we are thus primarily interested in *proofs of correctness* of graph transformations, for which two major approaches have emerged:

1. Model checking of graph transformations: given an initial graph and a set of transformation rules, check whether the graph can eventually evolve into a graph having certain properties, or whether specific properties can be ascertained to be always satisfied. This kind of reasoning is possible in principle (the initial graph can be specified by a pre-condition, invariants can be specified as loop conditions, eventuality properties as post-conditions), but our approach is clearly not geared towards this activity.
2. Full correctness proofs: given an arbitrary graph satisfying the pre-condition, verify that it evolves into a graph satisfying the post-condition. This is the kind of verification we are aiming at.

Full correctness proofs are hard, and undecidability of the generated proof obligations is a major concern for rich logics [4]. We propose to use a relatively simple logic, $\mathcal{ALC}$, belonging to the family of Description Logics (DLs). We summarize the logic in Section 3.1, and the fine-tuned interplay of the logic and the transformation language (among others: branching and loop conditions are formulas of this logic) brings it about that the proof obligations extracted from programs are decidable, as argued in Section 5. We are currently working on extending this approach to more expressive description logics, with the purpose of being able to tackle realistic problems in the areas of UML-style model transformations and RDF graph database transformations.

The work described here has reached the state of a sound prototype. In the corresponding sections, we will make precise which parts of the development are completed to which degree, and indicate which missing parts still have to be filled in. The small-t$\mathcal{ALC}$ environment is available from the following web page, where it will be regularly updated: http://www.irit.fr/~Martin.Strecker/CLIMT/Software/smalltalc.html.

Related work: Hoare-like logics have already been used to reason on graph transformations (see, e.g. [5]) but, as far as we are aware, no tool has been implemented. small-t$\mathcal{ALC}$, which is also based on a Hoare-like calculus, allows one to decide the verification problem, of programs operating on graphs, when the properties are expressed in the $\mathcal{ALC}$ logic. Some implementations of verification environments for pointer manipulating programs exist [6], however they often impose severe restrictions on the kind of graphs that can be manipulated, such as having a clearly identified spanning tree.

Other tools dedicated to reasoning on graph transformations have been proposed. For example, the GROOVE [7] system implements model-checking techniques using LTL or CTL formulas and thus departs from small-t$\mathcal{ALC}$ techniques.

The computation of weakest preconditions from a graph rewriting system is described by Habel, Pennemann and Rensink [8,9]. This work is concerned with extraction of weakest preconditions, but no proof system for the formulas is given. Pennemann [10] then describes a method of translating the extracted formulas to a resolution theorem prover. Radke [11] uses a more expressive logic: MSO. The spirit of the work described in this paper is similar, but we explicitly restrict the expressiveness of the logical framework to obtain decidable proof problems.

In a similar vein, Asztalos *et al.* [12] describe the verification of graph transformations based on category-theoretic notions and by translation to a logic for which no complete calculus is provided.

Raven[3] is a tool suite designed to handle and manipulate graph automata. In some sense Raven tends to generalize model-checking techniques from word to graph processing. Therefore techniques behind Raven tool are not directly comparable to small-t$\mathcal{ALC}$.

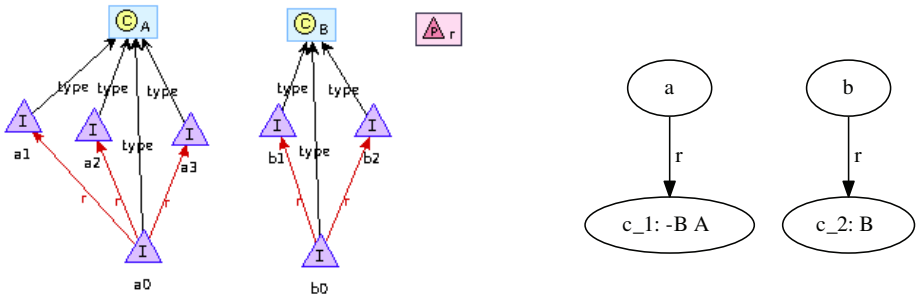
Alloy [13] is a popular framework for specifying and exploring relational structures, and it has been used to analyze graph transformations [14] written in the AGG transformation engine. Alloy interfaces with model checkers and can display counter models in case a transformation does not satisfy its specification. For verification, Alloy uses bounded model checking: errors for graphs of a certain size are systematically detected, but has the disadvantage that graphs beyond that size are not covered. As opposed to this, the proof method presented here is exhaustive, being based on a complete, decidable calculus.

[3] http://www.ti.inf.uni-due.de/research/tools/raven

2 System Description

2.1 User's View

To explore the perspective of a user of small-t$\mathcal{ALC}$, we will walk through processing a simple program, but before, let us take a look at the kind of graphs we will be transforming, such as the example graph in Figure 1a (displayed with RDF-Gravity[4]). We will be processing graphs in RDF [2] format. These graphs consist of nodes and typed edges. The graphs are simple: there cannot be multiple edges of the same type between two nodes, but several edges, each of different type. In the example, there is only one type of relation (also called *role*): `r`. Here, instance node `a0` is linked with nodes `a1, a2` and `a3`; similarly `b0` with `b1` and `b2`. Nodes can be typed. In our example, we have two types (also called *concepts*) `A` and `B`. Nodes $\mathtt{a}_i$ are of type `A`, and nodes $\mathtt{b}_j$ of type `B`. It is a matter of display to represent concepts as (meta-)nodes in Figure 1a, and also the (meta-)relation `type` as arc linking a node to its type, but these meta-entities are subject to a different treatment than object nodes and relations.



(a) Input graph

(b) Counter model after failed proof

Fig. 1: Graph structures

Let us now turn to transformation programs, as the one depicted in Figure 2. A program is composed of one or several parameterized rules; and a parameterless `main` rule whose purpose is to specify the input- and output graph to be transformed and to identify the root nodes of the input graph. Rules can be assimilated to non-recursive procedures or macros. Procedural abstraction is so far not fully developed in our framework, so the analysis presented in the following concentrates on rule bodies.

The rule `ex_rule` has a precondition (`pre`) saying that node `a` is only connected (via arcs of type `r`) to nodes of type `A`, and that `b` is only connected to

[4] http://semweb.salzburgresearch.at/apps/rdf-gravity/

```
concepts A, B; roles r;

rule ex_rule (a, b) {
vars c;
pre: (a : ([!] r A)) && (b : ([!] r B));
select c with (b r c);
add(a r c);
post: (a : ([?] r B));
}
```

```
rule main () {
vars a, b;
ingraph "input_graph.rdf";
outgraph "output_graph.rdf";
a := node("a0");
b := node("b0");
ex_rule(a, b);
}
```

Fig. 2: An example program

nodes of type B. The program now does the following: among the nodes that b is connected to, we non-deterministically pick a node c and introduce an arc r between a and c. For example, the program might introduce an arc between a0 and b1 in the graph of Figure 1a (or between a0 and b2). We can now assert that after running this program, the node that variable a points to is connected via r to at least one element of type B, as expressed in the postcondition.

Suppose the example program is in file example.trans. Running the verifier as follows confirms that the program is correct, *i.e.* that any graph satisfying the precondition is transformed into a graph satisfying the postcondition.

```
> graphprover example
starting proof ...formula valid
```

Let us modify the post-condition, claiming that a is exclusively connected to elements of type B: `post: (a : ([!] r B));`

When running the verifier again, we see that the property is incorrect, and that a counter-model has been created (see Figure 1b, here displayed with Graphviz[5]). This counter-model describes the state at the beginning of the program, namely a graph with four nodes, where c_1 is of type A and not of type B, and c_2 of type B. Clearly, when connecting a with c_2, the post-condition is violated.

We correct the post-condition, saying that a is only connected to elements of type A or B: `post: (a : ([!] r (A [||] B)));` Running the verifier again convinces us that this property is satisfied.

How does the verifier validate or invalidate a program? The approach is classic: from the annotated program, we extract a proof obligation by computing weakest pre-conditions (see Section 3.2). This is an $\mathcal{ALC}$ formula that is sent to a tableau decision procedure (described in Section 5.2). A failed proof attempt produces a saturated tableau from which a counter-model can always be extracted.

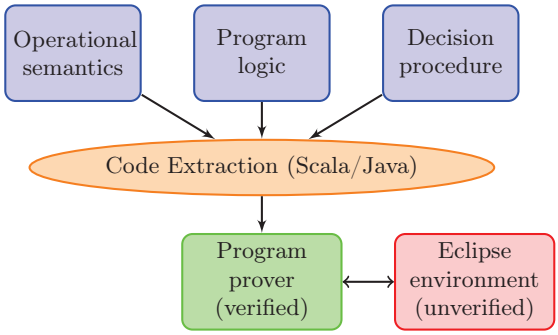


Fig. 3: Schema of formal development

2.2 Developer's View

Major parts of small-t$\mathcal{ALC}$ have a strong formal basis and are being developed in a proof assistant. We use Isabelle [15], but the formalization is easily adaptable to related proof assistants. Essential ingredients (see Figure 3) are the formalization of the program logic, the semantics of the programming language and a decision procedure of the extension of $\mathcal{ALC}$ we use (the latter has currently not been completely verified yet). This formalization (written in Isabelle's own functional and proof language) is automatically extracted to a general-purpose programming language, which is Scala in our case. We therefore obtain a highly reliable program prover, which is coupled with interface functionality (such as parsers and viewers) provided by Eclipse / Xtext to obtain the verifier described in Section 2.1. The transformation engine, described more in detail in Section 4, is so far unverified, but at least the Java code generator (Section 4) could be formally verified with by now standard compiler verification techniques.

3 Foundations

3.1 Logic

Our logic is a three-tier framework, the first level being Description Logic (DL) *concepts*, the second level *facts*, the third level *formulas* (Boolean combinations of facts and a simple form of quantification). Formulas occur not only in assertions (such as pre- and postconditions), but also in statements (Boolean conditions and `select` statement).

Concepts: In this paper, we concentrate on the description logic $\mathcal{ALC}$ [16]. For a being atomic concept names and r role (or relation) names, the abstract syntax of concepts C can be defined by the grammar:

[5] http://www.graphviz.org/

$$
\begin{array}{lll}
C ::= \bot & \text{(empty concept)} & \mid a \quad \text{(atomic concept)} \\
\mid \neg\, C & \text{(complement)} & \\
\mid C \sqcap C & \text{(intersection)} & \mid C \sqcup C \quad \text{(union)} \\
\mid ([?]\ r\ C) & \text{(some)} & \mid ([!]\ r\ C) \quad \text{(all)} \\
\mid C\tau & \text{(explicit substitution)} &
\end{array}
$$

The semantics of DLs is given by Kripke structures or, differently speaking, by typed graphs. Under this interpretation, *concepts* represent sets of individuals. The constructors $\neg, \sqcap, \sqcup$ (in Ascii notation: `!`, `[&&]`, `[||]`) then have the obvious meaning. $([?]\ r\ C)$ is the set of individuals x such that there is at least one r-typed edge $(x\ r\ y)$ between x and y, where y belongs to C. Dually, $([!]\ r\ C)$ is the set of individuals x all of whose r-edges go to individuals of type C.

The last constructor, *explicit substitution* [17], is a particularity of our framework, required for a gradual elimination of substitutions, as further described in Section 5.5. We have three kinds of substitutions τ:

- Replacement of a variable by another variable, of the form $[x := y]$,
- Adding a node v to / removing a node from an atomic concept a, of the form $[a := a + \{v\}]$ respectively $[a := a - \{v\}]$,
- Adding an edge (v_1, v_2) to / removing an edge from a role r, of the form $[r := r + \{(v_1, v_2)\}]$ respectively $[r := r - \{(v_1, v_2)\}]$.

Facts: Facts make assertions about an instance being an element of a concept, and about being in a relation. The grammar of facts is defined as follows:

$$
\begin{array}{lll}
fact ::= i : C & \text{(instance of concept)} \\
\mid i\ r\ i & \text{(instance of role)} \\
\mid i\ (\neg r)\ i & \text{(instance of role complement)} \\
\mid i \equiv i & \text{(equality of instances)} \\
\mid i \not\equiv i & \text{(inequality of instances)}
\end{array}
$$

Please note that since concepts are closed by complement, facts are closed by negation (the negation of a fact is again representable as a fact), and this is the main motivation for introducing the constructors "instance of role complement" and "inequality of instances".

Formulas: A formula is a Boolean combination of facts. We also allow quantification over individuals i (but not over relations or concepts), and, again, have a constructor for explicit substitution. We overload the notation $\bot$ for empty concepts and the Falsum.

$$
\begin{array}{lll}
form ::= \bot & \mid fact & \mid \neg form \\
\mid form \wedge form & \mid form \vee form & \\
\mid \forall i.form & \mid \exists i.form & \\
\mid form\ \tau & &
\end{array}
$$

In Figure 2, we use the Ascii notation `!`, `&&`, `||` for negation, conjunction and disjunction. The extension of interpretations from facts to formulas is standard. As usual, a formula that is true under all interpretations is called *valid*.

When calculating weakest preconditions (in Section 5.1), we obtain formulas which essentially contain no existential quantifiers; we keep them as constructor

because they can occur as intermediate result of computations. We say that a formula is *essentially universally quantified* if $\forall$ only occurs below an even and $\exists$ only below an odd number of negations. For example, $\neg(\exists x.\, x : C \wedge \neg(\forall y.\, y : D))$ is essentially universally quantified.

3.2 Programming Language

The programming language is an imperative language manipulating relational structures. Its distinctive features are conditions (in conditional statements and loops) that are restricted formulas of the logic $\mathcal{ALC}$, in the sense of Section 3.1. It has a non-deterministic assignment statement `select ... with` allowing to select an element satisfying a fact. Traditional types (numbers, arrays, inductive types) and accompanying operations are not provided; the language is thus only targeted at transformations of graphs.

Statements of our language are defined by the following grammar:

stmt ::=	`Skip`	(empty statement)
	`select` i `with` $form$	(assignment)
	`delete(`i `:` C`)`	(delete element from concept)
	`add(`i `:` C`)`	(add element to concept)
	`delete(`i r i`)`	(delete edge from relation)
	`add(`i r i`)`	(insert edge in relation)
	stmt `;` *stmt*	(sequence)
	`if` $form$ `then` *stmt* `else` *stmt*	
	`while` $form$ `do` *stmt*	

Please note that the keywords `add` and `delete` are overloaded for nodes and for edges. There is no direct support for creating or deleting nodes in a graph, only for "moving" them between concepts. We intend to simulate node creation and deletion by providing a predefined concept `heap` such that `add(n: heap)` corresponds to creating node `n` and `delete(n: heap)` to deallocating node `n`. Details still have to be worked out.

The semantics is a big-step semantics with rules of the form $(st, \sigma) \Rightarrow \sigma'$ expressing that executing statement st in state σ produces a new state σ'.

The rules of the semantics are given in the Figure 4. Beware that we overload logical symbols such as $\exists$, $\wedge$ and $\neg$ for use in the meta-syntax and as constructors of $form$.

We do not enter into the details (also see the Isabelle formalization). Intuitively, the states σ manipulated by the operational semantics are the same as the interpretations of formulas, and they describe the current structure of a graph: which nodes are contained in each concept; which pair of nodes are contained in a role; and which variables are bound to which nodes. We write $\sigma(b)$ to evaluate the condition b (a formula) in state σ.

Most of the rules are standard, apart from the fact that we do not use expressions, but formulas as conditions. The auxiliary function *delete_edge* modifies the state σ by removing an r-edge between the elements represented by v_1 and v_2, and similarly for *generate_edge*. There are analogous functions for adding / deleting in concepts.

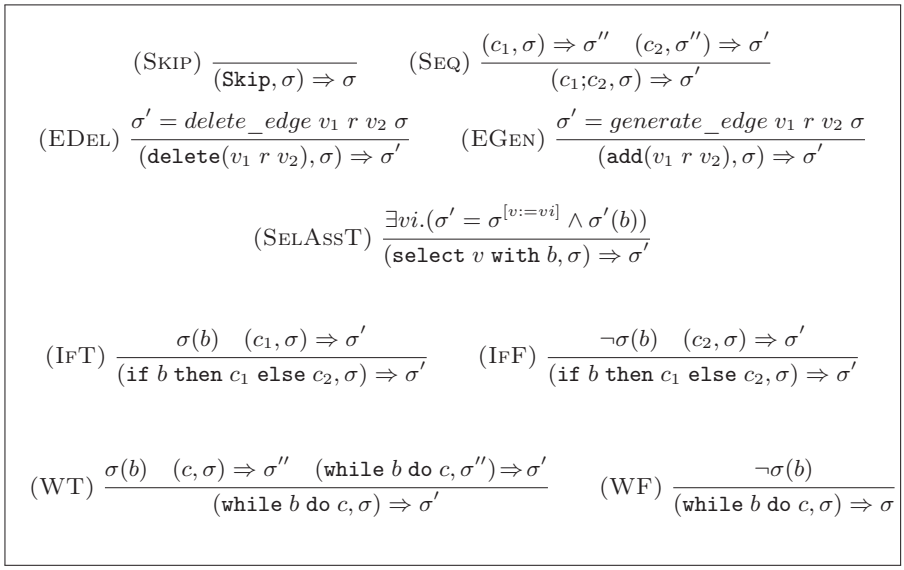


Fig. 4: Big-step semantics rules

The statement `select` v `with` $F(v)$ selects an element vi that satisfies formula F, and assigns it to v. For example, `select` a `with` $a : A \wedge (a\ r\ b)$ selects an element a which is and instance of concept A and being r-related with a given element b.

`select` is a generalization of a traditional assignment statement. There may be several instances that satisfy F, and the expressiveness of the logic might not suffice to distinguish them. In this case, any such element is selected, non-deterministically. Let us spell out the precondition of (SelAssT): Here, $\sigma^{[v:=vi]}$ is an interpretation update for individuals, modifying σ for variable v and assigning it a value vi in the semantic domain. We check whether the formula b would be satisfied under this choice, and if it is the case, keep this assignment. In case no satisfying instance exists, the semantics blocks, *i.e.* the given state does not have a successor state, which can be considered as an error situation.

4 Executing Graph Transformations

Generating Java Code: For processing small-t$\mathcal{ALC}$ programs such as the one in Figure 2 and generating Java code, we use the Eclipse environment and, in particular, the Xtext[6] facilities for parsing, syntax highlighting and context-dependent help. The program prover is currently not fully integrated in this framework, so

[6] http://www.eclipse.org/Xtext/

that the interaction with the prover is performed via shell commands as described in Section 2.1.

In order to generate Java code for small-t$\mathcal{ALC}$ programs, we parse the program and then traverse the syntax tree with Xtext/Xtend, issuing calls to appropriate Java functions that manipulate a graph (which is initially the input graph provided in the program's `main` rule). Here is a glimpse at the Xtend code snippet that translates statements, in particular the `add` statement for roles:

```
def statement(Stmt s){
      switch s{
        Add_stmt: add(s.lvar,s.role,s.rvar)
        ...
      }
}
def add(String lvar,String role,String rvar)'''
       «graph».insertEdge(«lvar»,«role»,«rvar»);'''
```

Thus, a small-t$\mathcal{ALC}$ program fragment `add(a r b);` is translated to a Java call `g.insertEdge(a, r, b);`, where the graph `g` is the current graph.

Transforming Graphs: Once a Java program has been generated for a given small-t$\mathcal{ALC}$ program, it can be compiled and linked with a library that provides graph manipulating functions such as the above-mentioned `insertEdge`. When executing this program, it remains to read an input file containing a graph description, to perform the transformation and to output the new graph. We represent graphs in the RDF [2] format. Parsing and printing of RDF files is based on the Apache Jena framework[7].

5 Reasoning about Graph Transformations

5.1 Weakest Preconditions

For proving program correctness, we use a standard approach in program verification. For proving that a program *prog* establishes the postcondition Q if started in a state satisfying the precondition P, we calculate the weakest precondition of *prog* with respect to Q and then show that P implies this weakest precondition.

The details are inspired by the description in [18]: we compute weakest preconditions *wp* (propagating post-conditions over statements and taking loop invariants for granted) and verification conditions *vc* that aim at verifying loop invariants. Both take a statement and a DL formula as argument and produce a DL formula. For this purpose, while loops have to be annotated with loop invariants, and the `while` constructor becomes: `while` $\{form\}$ $form$ `do` *stmt*. Here, the first formula (in braces) is the invariant, the second formula the termination condition. The two functions are defined by primitive recursion over statements, see Figure 5 for the definition of *wp* (and the Isabelle sources for *vc*).

[7] http://jena.apache.org/

$$
\begin{array}{l}
wp(\texttt{Skip},\ Q) = Q \\
wp(\texttt{delete}(v :\ C),\ Q) = Q[C := C - \{v\}] \\
wp(\texttt{add}(v :\ C),\ Q) = Q[C := C + \{v\}] \\
wp(\texttt{delete}(v_1\ r\ v_2),\ Q) = Q[r := r - (v_1, v_2)] \\
wp(\texttt{add}(v_1\ r\ v_2),\ Q) = Q[r := r + (v_1, v_2)] \\
wp(\texttt{select}\ v\ \texttt{with}\ b,\ Q) = \forall v.(b \longrightarrow Q) \\
wp(c_1; c_2,\ Q) = wp(c_1, wp(c_2,\ Q)) \\
wp(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\ Q) = ite(b, wp(c_1, Q), wp(c_2, Q)) \\
wp(\texttt{while}\{iv\}\ b\ \texttt{do}\ c,\ Q) = iv
\end{array}
$$

Fig. 5: Weakest preconditions and verification conditions

Without going further into program semantics issues, let us only state the following soundness result that relates the operational semantics and the functions wp and vc:

Theorem 1 (Soundness). *If $vc(c, Q)$ is valid and $(c, \sigma) \Rightarrow \sigma'$, then $\sigma(wp(c, Q))$ implies $\sigma'(Q)$.*

What is more relevant for our purposes is the structure of the formulas generated by wp and vc, because it has an impact on the decision procedure for the DL fragment under consideration here. Besides the notion of "essentially universally quantified" introduced in Section 3.1, we need the notion of *quantifier-free* formula: A formula not containing a quantifier. In extension, we say that a statement is quantifier-free if all of its formulas are quantifier-free.

By induction on c, one shows:

Lemma 1 (Universally quantified). *Let Q be essentially universally quantified and c be a quantifier-free statement. Then $wp(c, Q)$ and $vc(c, Q)$ are essentially universally quantified.*

There is one major problem with the definition of function wp: the substitutions, such as $C := C - \{v\}$ or $r := r - (v_1, v_2)$. When conceiving them as a meta-operations, as is usually done, we see that substitutions would yield syntactically ill-formed formulas. For example, reducing $([?]\ r\ C)[C := C - \{v\}]$ would give $([?]\ r\ (C - \{v\}))$, which is not a valid concept expression. There are two ways out of this difficulty: we could either relax our syntax and accept expressions of the form $([?]\ r\ (C - \{v\}))$. This would induce a rather heavy change on the logic. Alternatively, we can treat substitution as a constructor of our language. This is the approach we have adopted, and therefore, substitutions appear as syntactic elements in the definitions of Section 3.1. It remains to be seen (in Section 5.2) how substitutions can be dealt with by proof methods of $\mathcal{ALC}$.

5.2 Tableau Method

The core of the decision procedure for proving the verification conditions that are obtained as described in Section 5.1 is a tableau calculus which combines

the traditional logical rules of a tableau calculus [19] with rules for progressively eliminating the substitutions which are not part of the logic $\mathcal{ALC}$.

As a consequence, and departing again from common practice in the DL literature, our tableau procedure does not manipulate facts (in the sense of Section 3.1), but formulas, *i.e.* Boolean combinations of facts. This extension becomes necessary because elimination of substitutions generates complex formulas. These could in principle be directly decomposed into sub-tableaux, but such a procedure obscures both the presentation and the implementation.

Preprocessing: The tableau manipulates quantifier-free formulas in negation normal form (nnf).

The formulas obtained from function vc do possibly contain quantifiers, but as mentioned before, the formulas are essentially universally quantified. To get rid of these quantifiers, we therefore perform the following steps:

- We convert the entry formula f to a prenex normal form, *i.e.* a form $\forall x_1 \ldots x_n.b$ with quantifier-free body b.
- We drop the quantifier prefix; more precisely, we replace the bound variables $x_1 \ldots x_n$ in b by free variables. This transformation preserves validity.
- We start the tableau with $nnf(\neg b)$. The procedure is a satisfiability check that either produces an empty tableau (meaning that f is valid) or a model of $\neg b$ that is a counter-example of f.

In negation normal form, negations only occur in front of atomic concepts (of the form $\neg a$, where a is an atomic concept). This invariant is maintained throughout the tableau procedure.

5.3 Tableau Rules

In the following, we present a high-level description of the tableau procedure. (The reader consulting the Isabelle theories will notice that the formalization is on two levels: a set-based, relational version, aiming at proving essential properties such as soundness and completeness of the rules; and a list-based implementation. The formal proofs of these theories are not yet finalized.)

A tableau manipulates sets of branches (also called *abox*es - "assertional boxes" in DL terminology). Each branch Γ is a set of formulas. We first concentrate on a set of rules aiming at decomposing formulas on a single branch. They have the form $\Gamma \hookrightarrow \Gamma'$, expressing that branch Γ is rewriten to Γ'. We write Γ, f instead of $\Gamma \cup \{f\}$ for adding formula f to Γ. The rules are displayed in Figure 6.

Let us comment on the rules: The structural rules conjC, disjCr, disjCl (for concepts) and conjF, disjFr, disjFl (for formulas) should be clear. The rule all allows to conclude $y : C$ if x is only r-connected to elements of type C, and there is an arc $(x\ r\ y)$. The rule some inserts an arc $(x\ r\ z)$ and a membership $z : C$ for an arbitrary z if it is known that x is r-connected to at least one element of type C. The rule eq propagates an equality $x \equiv y$ in the branch, provided the equality is not $x \equiv x$.

$$\textsc{conjC}\ \frac{(x : (C_1 \sqcap C_2)) \in \Gamma \quad \text{not}((x : C_1) \in \Gamma \text{ and } (x : C_2) \in \Gamma)}{\Gamma \hookrightarrow \Gamma, (x : C_1), (x : C_2)}$$

$$\textsc{disjCr}\ \frac{(x : (C_1 \sqcup C_2)) \in \Gamma \quad (x : C_1) \notin \Gamma \quad (x : C_2) \notin \Gamma}{\Gamma \hookrightarrow \Gamma, (x : C_1)}$$

$$\textsc{disjCl}\ \frac{(x : (C_1 \sqcup C_2)) \in \Gamma \quad (x : C_1) \notin \Gamma \quad (x : C_2) \notin \Gamma}{\Gamma \hookrightarrow \Gamma, (x : C_2)}$$

$$\textsc{all}\ \frac{(x : ([!]\ r\ C)) \in \Gamma \qquad (x\ r\ y) \in \Gamma \qquad (y : C) \notin \Gamma}{\Gamma \hookrightarrow \Gamma, (y : C)}$$

$$\textsc{some}\ \frac{(x : ([?]\ r\ C)) \in \Gamma \quad \text{for all } y, \text{not}((x\ r\ y) \in \Gamma \text{ and } (y : C) \in \Gamma)}{\Gamma \hookrightarrow \Gamma, (x\ r\ z), (z : C)}$$

$$\textsc{subst}\ \frac{(x : (C\tau)) \in \Gamma \quad nnf(push((x : C)\tau)) \notin \Gamma}{\Gamma \hookrightarrow \Gamma, nnf(push((x : C)\tau))}$$

$$\textsc{eq}\ \frac{(x \equiv y) \in \Gamma \quad x \neq y}{\Gamma \hookrightarrow \Gamma[x := y]}$$

$$\textsc{conjF}\ \frac{f_1 \wedge f_2 \in \Gamma \quad \text{not}(f_1 \in \Gamma \text{ and } f_2 \in \Gamma)}{\Gamma \hookrightarrow \Gamma, f_1, f_2}$$

$$\textsc{disjFr}\ \frac{f_1 \vee f_2 \in \Gamma \quad f_1 \notin \Gamma \quad f_2 \notin \Gamma}{\Gamma \hookrightarrow \Gamma, f_1} \qquad \textsc{disjFl}\ \frac{f_1 \vee f_2 \in \Gamma \quad f_1 \notin \Gamma \quad f_2 \notin \Gamma}{\Gamma \hookrightarrow \Gamma, f_2}$$

Fig. 6: Tableau rules

The rule SUBST is applicable for concepts with substitutions. As motivated in Section 5.1, substitutions cannot be eliminated at once, but they can be removed progressively, whenever the tableau prover hits on a fact of the form $(x : C\tau)$. Note that the variable x was possibly not present in the original tableau with which we have started the proof, but may have been introduced by a SOME-rule. If we encounter such a situation, we push the substitution as far as possible. We postpone the details to Section 5.5.

A branch Γ contains a *clash* ($clash(\Gamma)$) if either of the following holds:

- for x a variable, $(x : \bot) \in \Gamma$
- for x a variable and a an atomic concept, $(x : a) \in \Gamma$ and $(x : \neg a) \in \Gamma$
- for x, y variables, $(x\ r\ y) \in \Gamma$ and $(x\ (\neg r)\ y) \in \Gamma$

– for x a variable, $(x \not\equiv x) \in \Gamma$
– $\bot \in \Gamma$

5.4 Tableau Procedure

We can now formulate a depth-first-search function dfs exploring a tableau. The function takes a tableau (here implemented as a list of branches) and returns a list of models. Initially, the tableau is just the formula $[\{f\}]$ to be proved. If the resulting list is empty, f is not satisfiable. Otherwise, the list contains an element which is a model of f.

$$\begin{array}{lll} dfs[\,] & = [\,] & \\ dfs(\Gamma :: \Gamma_s) & = \texttt{if} & clash(\Gamma) \\ & \quad \texttt{then} & dfs(\Gamma_s) \\ & \quad \texttt{else} & \texttt{if}\ reducible(\Gamma) \\ & & \quad \texttt{then}\ dfs(\{\Gamma' | \Gamma \hookrightarrow \Gamma'\} @ \Gamma_s) \\ & & \quad \texttt{else}\ [\Gamma] \end{array}$$

The procedure progressively eliminates all inconsistent branches (with $clash(\Gamma)$). If a branch Γ is not inconsistent, but reducible (*i.e.* , there exists a Γ' with $\Gamma \hookrightarrow \Gamma'$), then we expand the tableau and explore the new branches.

5.5 Eliminating Substitutions

The *push* function used in the *subst* rule of Figure 6 pushes substitutions into formulas, "as far as possible". The remaining tableau rules then decompose formulas until substitutions hidden in subformulas become apparent and the *subst* rule can be applied again. Intuitively speaking, this process decreases the "height" of the substitutions in a formula, until they eventually disappear.

For a formula f, we define $push(f)$ as the formula f' which is the result of the rewrite system spelled out in the following. Thus: $push(f) = f'$ iff $f \rightsquigarrow^* f'$, where the rewrite relation $\rightsquigarrow$ is defined in the following. There are numerous cases to consider, and we do not present all of them.

Substitution in formulas are pushed into subformulas:

– $\bot\tau \rightsquigarrow \bot$
– $(\neg f)\tau \rightsquigarrow (\neg f\tau)$
– $(f_1 \wedge f_2)\tau \rightsquigarrow (f_1\tau \wedge f_2\tau)$
– $(f_1 \vee f_2)\tau \rightsquigarrow (f_1\tau \vee f_2\tau)$

Substitution in facts: Substitutions of individual variables $f[x := y]$ are carried out as expected. Otherwise, we procede as follows:

– $(x : \neg C)\tau \rightsquigarrow x : (\neg C\tau)$
– $(x : C_1 \sqcap C_2)\tau \rightsquigarrow x : (C_1\tau \sqcap C_2\tau)$
– $(x : C_1 \sqcup C_2)\tau \rightsquigarrow x : (C_1\tau \sqcup C_2\tau)$

– For substitutions τ of the form $a := a - \{v\}$ or $a := a + \{v\}$:
  - $(x : c)[a := a - \{v\}] \rightsquigarrow (x : c)$ for $a \neq c$, and similarly for $a := a + \{v\}$
  - $(x : a)[a := a - \{v\}] \rightsquigarrow (x : a) \wedge x \neq v$
  - $(x : a)[a := a + \{v\}] \rightsquigarrow (x : a) \vee x = v$
  - $(x : ([?]\ r\ C))[a := a - \{v\}] \rightsquigarrow (x : ([?]\ r\ C[a := a - \{v\}]))$, and similarly for the other combinations involving constructor [?] or [!] and substitutions $a := a + / - \{v\}$.
– For substitutions τ of the form $r := r - \{(v_1, v_2)\}$ or $r := r + \{(v_1, v_2)\}$:
  - $(x : c)[r := r - \{(v_1, v_2)\}] \rightsquigarrow x : c$, and similarly for $r + \{(v_1, v_2)\}$
  - $(x : ([!]\ r'\ C))[r := r - \{(v_1, v_2)\}] \rightsquigarrow (x : ([!]\ r'\ C))$ for $r \neq r'$
  - $(x : ([!]\ r\ C))[r := r - \{(v_1, v_2)\}] \rightsquigarrow$

$$\begin{array}{l} ite\ ((x = v_1) \wedge (v_2 : (\neg C[r := r - (v_1, v_2)])) \wedge (v_1\ r\ v_2), \\ \quad (x : (<\ 2\ r\ (\neg C[r := r - (v_1, v_2)]))), \\ \quad (x : ([!]\ r\ C[r := r - (v_1, v_2)]))) \end{array}$$

    Here, *ite* is for if-then-else: $ite(a, b, c) = (a \longrightarrow b) \wedge (\neg a \longrightarrow c)$.
    Please note that the logic $\mathcal{ALC}$ cannot completely express the effect of substitution, and we have to resort to the more expressive logic $\mathcal{ALCQ}$, which turns out to be complete for substitutions. Thus, the "then" branch of the *ite* construct expresses that x is r-connected to less than 2 elements of $(\neg C[r := r - (v_1, v_2)])$. We have however not yet implemented tableau rules for $\mathcal{ALCQ}$, so we stick to the simpler logic in this presentation.
  - $(x : ([!]\ r\ C))[r := r + \{(v_1, v_2)] \rightsquigarrow$
    $\neg((x = v_1) \wedge (v_2 : \neg(C[r := r + (v_1, v_2)])) \wedge (v_1\ (\neg r)\ v_2))$
    $\wedge(x : ([!]\ r\ (C[r := r + (v_1, v_2)])))$
  - Similar rules for existential quantification $(x : ([?]\ r\ C))$.

6 Conclusions

We have presented small-t$\mathcal{ALC}$, a framework for executing graph transformations and proving their correctness with a sound and complete calculus. One of the distinctive features of the approach is its formal semantic basis. We are now moving towards application, such as Sparql Query and Update in the knowledge representation world, and model transformations as used in model-driven engineering. The greatest challenge is the development of logics that are more expressive than $\mathcal{ALC}$ but remain decidable. Even though a low proof-theoretic complexity is not a major concern for program correctness proofs (these are not executed on a large knowledge base), the concern changes when wanting to execute programs efficiently on a large data set.

Acknowledgements We are grateful to María Espinoza who has helped us explore the applicability of graph transformations to the RDF world [20].

References

1. Budinsky, F., Brodsky, S.A., Merks, E.: Eclipse Modeling Framework. Pearson Education (2003)
2. Cyganiak, R., Lanthaler, M., Wood, D.: RDF 1.1 Concepts and Abstract Syntax. http://www.w3.org/TR/rdf11-concepts (2014)
3. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Proceedings of MoDELS'10. Volume 6394 of LNCS. Springer (2010)
4. Immerman, N., Rabinovich, A., Reps, T., Sagiv, M., Yorsh, G.: The boundary between decidability and undecidability for transitive-closure logics. In Marcinkowski, J., Tarlecki, A., eds.: Computer Science Logic. Volume 3210 of LNCS. Springer Berlin / Heidelberg (2004) 160–174
5. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. Fundamenta Informaticae **118**(1-2) (2012) 135–175
6. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: PLDI. (2001) 221–231
7. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. STTT **14**(1) (2012) 15–40
8. 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. Volume 4178 of LNCS. Springer Verlag, Berlin (September 2006) 445–460
9. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. MSCS **19**(02) (2009) 245–296
10. Pennemann, K.H.: Resolution-like theorem proving for high-level conditions. In Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G., eds.: Graph Transformations. Volume 5214 of LNCS. Springer Berlin / Heidelberg (2008) 289–304
11. Radke, H.: HR* graph conditions between counting monadic second-order and second-order graph formulas. ECEASST **61** (2013)
12. Asztalos, M., Lengyel, L., Levendovszky, T.: Formal specification and analysis of functional properties of graph rewriting-based model transformation. Software Testing, Verification and Reliability **23**(5) (2013) 405–435
13. Jackson, D.: Software Abstractions: Logic, language, and analysis. MIT Press (2012)
14. Baresi, L., Spoletini, P.: On the use of Alloy to analyze graph transformation systems. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: Graph Transformations. Volume 4178 of LNCS. Springer (2006) 306–320
15. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL. A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer Berlin / Heidelberg (2002)
16. Baader, F., Sattler, U.: Expressive number restrictions in description logics. Journal of Logic and Computation **9**(3) (1999) 319–350
17. Abadi, M., Cardelli, L., Curien, P.L., Lévy, J.J.: Explicit substitutions. Journal of Functional Programming **1**(4) (October 1991) 375–416
18. Nipkow, T., Klein, G.: Concrete Semantics. http://www21.in.tum.de/~nipkow/Concrete-Semantics/ (2014)
19. Baader, F., Sattler, U.: Tableau algorithms for description logics. In Dyckhoff, R., ed.: Automated Reasoning with Analytic Tableaux and Related Methods. Volume 1847 of LNCS. Springer (2000) 1–18
20. Espinoza, M.V.: Transformation de graphes en RDF. Master's thesis, Université de Toulouse (2014)