Verifying Graph Transformations with Guarded Logics

Jon Haël Brenas
UTHSC - ORNL,
Memphis, Tennessee, USA,
jhael@uthsc.edu

Rachid Echahed
CNRS and Université Grenoble Alps
http://membres-lig.imag.fr/echahed/

Martin Strecker

***Abstract*—We consider the problem of verifying graph transformations described by an imperative programming language. This question is particularly relevant for transformation of knowledge bases. We will argue in this paper that previous proof approaches based on dedicated Description Logics were technically complex and often inappropriate for reasoning about preservation of structure of knowledge bases. For these reasons, we explore here an assertion formalism based on the Guarded Fragment of predicate logic, which provides a homogeneous framework. Based on a formal semantics of our transformation language, we show how to extract proof obligations from annotated programs and how to obtain decidable correctness problems.**

I. Introduction

Graphs are structures that are ubiquitous in natural sciences and in engineering, and in particular in computer science. Applications include the topology of a communication network; diagrams in model-driven engineering; pointer structures in traditional imperative programming languages; and graph data or knowledge bases.

We are here concerned with ascertaining that structural properties are preserved under change, and we want to ensure this statically, before effectively carrying out the transformation. Differently said, we aim at full verification of graph transformations and not an *a posteriori* test that the transformation did not go wrong for individual instances. This is a relevant requirement if changes are difficult to be undone; or the graph structure must never be in an inconsistent state; or the graph is too large - indeed, in the following, we make no assumptions about finiteness of graphs.

An ideal we aim at is fully automated verification, and for this reason, we have to make concessions, both on the side of the transformation language and the assertion formalism. General-purpose programming languages are too expressive, containing for example unlimited arithmetic. We have therefore defined an imperative programming language specifically tailored to graph transformations that only addresses structural aspects of a graph. Such a language can be useful as a high-level language to be compiled to and used in conjunction with a general-purpose language. It can also be used in contexts where only a limited expressivity is required, such as manipulation of (graph) data bases and knowledge bases [1], [2], [3]. On the basis of such an imperative language, we use Hoare triples (precondition, statement, postcondition) for specifying the behaviour of a transformation.

Martin Strecker has carried out this research on a private basis without public or third-party funding.

We also have to reduce the expressiveness of the assertion formalism, so as to have a decidable base logic. We have previously investigated Description Logics [4] as an assertion formalism, a family of logics that are particularly suited for reasoning about changes in data and knowledge bases, see [5], [6]. The disadvantage of these logics is their restricted capacity to express nested quantifications, which is a limitation for expressing interesting program properties. A more technical problem is that it is difficult to deal with substitutions that arise during computation of weakest preconditions: Description Logics are typically not closed under substitutions. We also face this problem in the approach described in this paper, but the solution turns out to be cleaner, as we can clearly separate a pre-processing phrase, eliminating substitutions, from theorem proving, whereas in Description Logics, these phases have to be intertwined [7].

To overcome these limitations, in this paper, we propose to use the Guarded Fragment (GF) of first-order predicate logic (FO) as assertion formalism. GF finds its roots in modal logic, just as Description Logics can be understood as extensions of modal logic. GF has been proposed in [8] as a generalisation of the kind of formulas one obtains when translating modal logics to FO. GF is a direct fragment of FO, and so are extensions of GF that we have to adopt temporarily to handle verification conditions arising from programs, and this gives a more homogeneous framework with FO as base formalism. GF has been largely investigated [9], [10], [11]. GF is decidable and several decision procedures have been described [12], [13] and implemented [14].

The paper is structured as follows: In Section II, we provide an example transformation that we will use as running example throughout the paper. In Section III, we recapitulate the Guarded Fragment (GF) and introduce some syntactic extensions that turn out to be reducible to GF. The syntax and semantics of the transformation language will be introduced more formally in Section IV, and we will show how to derive verification conditions, with the aid of a weakest precondition calculus. We will spell out in more detail in Section V how to convert these verification conditions into formulas of the Guarded Fragment. We conclude in Section VI.

II. Example of a Transformation

We start with the presentation of an example program that is given in Figure 1, and which is meant to transform graphs as the one on the left side of Figure 2, to produce the graph on the right side.

Graphs consist of nodes and arcs. The names of nodes ($a_1, a_2, b_1, \dots$) appearing in Figure 2 are for convenience only and are not part of the logical formalism. What is essential is the labelling of nodes with types (or properties or classes), such as A, B, C. These correspond to the unary predicates in Figure 1. Arcs are labelled with arc types. These correspond to the binary predicates in Figure 1. As we only use one arc type R in this example, we have not explicitly annotated the arcs in Figure 2.

The program consists of

- a declarative part (in blue), namely the pre- and postconditions preceded by the keywords `Pre` resp. `Post`, and an invariant following keyword `inv`
- an executable part (in black), consisting of traditional programming constructs (such as a `while` loop) and specifically tailored graph manipulation instructions (`select`, `add`, `delete`) and that will be introduced more formally in Section IV.

The program is meant to identify nodes of type A that have a successor (via relation R) both to nodes of type B and nodes of type C, as specified in the `select` clause. Each such node is added to type D (with the `add` statement) and removed from type A (with the `delete` statement), and the arc to the C type node is deleted. If there are several C-type successors, `select` chooses one of them nondeterministically, such as for example the nodes c_1 or c_2 among the successors of a_3 (in Figure 2, the first choice has been made).

The program continues until no more such nodes can be found in the graph. For better readability, we give the condition and the invariant of the `while`-loop here: The formula *Continue* is $\exists a.A(a) \wedge (\exists b.R(a,b) \wedge B(b)) \wedge (\exists c.R(a,c) \wedge C(c))$. In this example, the invariant *Invar* is the same as the postcondition.

We use some abbreviations which can be translated to formulas in the Guarded Fragment, such as $D = \{\}$ which expands to $\neg\exists d.D(d)$, and $B \cap C = \{\}$ which expands to $\neg\exists n.B(n) \wedge C(n)$.

III. Guarded Fragment

In this section, we define the Guarded Fragment (GF) as a subset of first-order (FO) logic with equality, having arbitrary n-ary relational symbols, but no function symbols. We will also introduce some extensions that arise during verification condition generation, and see how to reduce them to GF. The common basis of these languages is first-order predicate logic, so we start with recalling some essential notions.

A. Predicate Logic

We write $x, y, \dots$ for variables and boldface $\mathbf{x}, \mathbf{y}$ for tuples of variables. If $\mathbf{x} = (x_1, \dots x_n)$ is a tuple of variables,

```
Pre:  (∀ a. A(a) ⟶
             (∃ n. R(a, n) ∧ (B(n) ∨ C(n))))
      ∧ D = {} ∧ B ∩ C = {}

while ( Continue ) inv  Invar  do {
      select (a, b, c) with  A(a) ∧ B(b)
                  ∧ C(c) ∧ R(a,b) ∧ R(a,c);
      add(D(a));
      delete(A(a));
      delete(R(a,c));
   }
Post: A ∩ D = {} ∧
      (∀ d. D(d) ⟶ (∃ b. R(d,b) ∧ B(b)))
```

Fig. 1: An example program

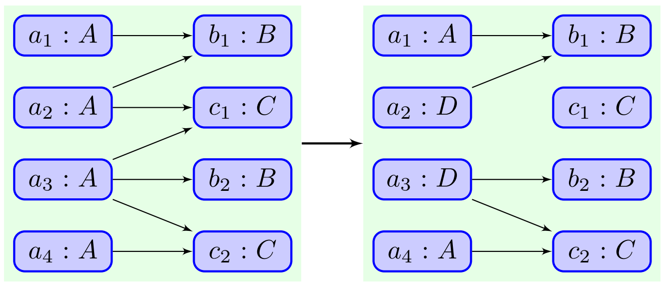

Fig. 2: Resulting transformation

then $set(\mathbf{x}) = \{x_1, \dots x_n\}$ is the set of its variables. We write $\phi(\mathbf{x})$ for a formula whose free variables are among $\mathbf{x}$ without necessarily containing precisely these variables. By $FV(\psi)$, we designate the set of free variables of formula ψ. Quantification with tuples such as $\exists \mathbf{x}.P(\mathbf{x})$ is shorthand for iterated quantification with individual variables such as $\exists x_1 \dots . \exists x_n.P(\mathbf{x})$.

An *atomic formula* or *atom* is defined as an equality $x = y$ or the application of a relation symbol to a tuple of variables, $R(\mathbf{x})$. We henceforth assume that every relation symbol R has a fixed arity.

The logical languages to be defined in the following are all fragments of predicate logic. The semantics we define now is applicable to all of them. The semantics is standard, given by interpretations ρ into relational structures. When restricted to unary and binary relations, these relational structures can be understood as graphs whose nodes are labelled with unary predicates and whose arcs are labelled with binary relations, as in Figure 2. But the entire development is not restricted to unary and binary relations.

More in detail, an interpretation into a relational structure is a triple $\rho = (\rho_d, \rho_r, \rho_i)$ where ρ_d is a domain, ρ_r is a function that assigns to each n-ary relation symbol of the language a subset of ρ_d^n, and ρ_i a function that assigns to each individual variable an element of ρ_d.

Definition 1 (Model of first-order logic): As customary in first-order logic, we write $\rho \models b$ to express that ρ is a model of formula b, which is inductively defined by:

- $\rho \models x = y$ if $\rho_i(x) = \rho_i(y)$
- $\rho \models R(\mathbf{v})$ if $\rho_i(\mathbf{v}) \in \rho_r(R)$, where $\rho_i(\mathbf{v})$ is the obvious mapping of ρ_i on a vector of variables.
- $\rho \models \neg\psi$ if $\rho \not\models \psi$
- $\rho \models \psi \wedge \phi$ if $\rho \models \psi$ and $\rho \models \phi$
- $\rho \models \psi \vee \phi$ if $\rho \models \psi$ or $\rho \models \phi$
- $\rho \models \exists v.\psi(v)$ if there exists an element $vi \in \rho_d$ such that $\rho^{v:=vi} \models \psi(v)$. Here, if $\rho = (\rho_d, \rho_r, \rho_i)$, then $\rho^{v:=vi} = (\rho_d, \rho_r, \rho_i(v := vi))$ and $\rho_i(v := vi)$ is the update of function ρ_i at variable v with value vi.
- $\rho \models \forall v.\psi(v)$ if for all $vi \in \rho_d$, we have $\rho^{v:=vi} \models \psi(v)$.

B. Fragments of Predicate Logic

The Guarded Fragment (GF) of predicate logic is now defined as follows:

Definition 2 (Guarded Fragment, GF):

1) All quantifier-free first-order formulas are formulas of GF.
2) If ψ and ϕ are formulas of GF, then so are $\neg\psi$ and $(\psi \wedge \phi)$ and $(\psi \vee \phi)$.
   Implication $\psi \longrightarrow \phi$ and equivalence $\psi \leftrightarrow \phi$ are defined as usual. We define the formula "if-then-else", $ite(b, t, e)$, as $(b \longrightarrow t) \wedge (\neg b \longrightarrow e)$.
3) 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.

Note that we could have restricted ourselves to the existential quantifier, and by duality, we could have derived the above definition for the universal quantifier. Some of the following developments will indeed only discuss the case of existential formulas.

Proposition 1: The validity of GF is decidable.
See [8], [9], [13] for proofs. For our further developments, we need extension of GF that are described in the following. First of all, *parameters* occur temporarily in formulas as the result of introducing program variables in formulas. These variables occur free and are not correctly guarded, so we need to be able to manipulate formulas outside the Guarded Fragment, which however turn out to be convertible to formulas of GF, as shown below.

Definition 3 (Guarded Fragment with Parameters, GFP): Let P be a set of variables called *parameters*. GFP contains all quantifier-free first-order formulas and is closed under propositional connectives (as GF). Furthermore, if $set(\mathbf{v}) \subseteq P$, $(set(\mathbf{x}) \cup set(\mathbf{y})) \cap P = \{\}$ and $\psi(\mathbf{x}, \mathbf{y}, \mathbf{v})$ is a formula of GFP and $\alpha(\mathbf{x}, \mathbf{y}, \mathbf{v})$ is an atom and $FV(\psi(\mathbf{x}, \mathbf{y}, \mathbf{v})) \subseteq FV(\alpha(\mathbf{x}, \mathbf{y}, \mathbf{v})) \cup P$, then $\exists \mathbf{y}.\alpha(\mathbf{x}, \mathbf{y}, \mathbf{v}) \wedge \psi(\mathbf{x}, \mathbf{y}, \mathbf{v})$ and $\forall \mathbf{y}.\alpha(\mathbf{x}, \mathbf{y}, \mathbf{v}) \longrightarrow \psi(\mathbf{x}, \mathbf{y}, \mathbf{v})$ are formulas of GFP.

We remind the reader that the notation $\alpha(\mathbf{x}, \mathbf{y}, \mathbf{v})$ means that the variables of $\mathbf{v}$ may, but do not necessarily, occur in α. The essential difference is that in GF, all the variables in the body of a quantified formula have to be protected by a guard, whereas in GFP, the body may contain free variables that are not protected by the guard as long as they are parameters. For example, $\exists y.P(x, y) \wedge R(x, y, v)$ is not in GF because v is not guarded, but in GFP if $v \in P$.

Using the dual of a transformation described in [9], we show that every formula in GFP can be transformed into a formula in GF, preserving validity. In the above example, the transformed formula is $\forall v.Par(v) \longrightarrow \exists y.P'(v, x, y) \wedge R'(v, x, y, v)$.

Lemma 1 (Transformation of GFP to GF): Every GFP formula ψ can be transformed into a GF formula ψ' such that ψ is valid iff ψ' is valid.

Proof: Let $P = \{v_1, \ldots, v_p\}$ be the set of parameters of ψ. For every n-ary relation symbol R of ψ, introduce a new $(p+n)$-ary relation symbol R' and a new p-ary relation symbol Par. If ψ is a GFP formula, then define ψ' as $\forall \mathbf{v}.Par(\mathbf{v}) \longrightarrow \psi[R(\mathbf{x}) := R'(\mathbf{v}, \mathbf{x})]$, where $\psi[R(\mathbf{x}) := R'(\mathbf{v}, \mathbf{x})]$ is the formula resulting from replacing every atom of the form $R(\mathbf{x})$ by an atom of the form $R'(\mathbf{v}, \mathbf{x})$, possibly after renaming bound variables in ψ to avoid capture of the free variables of $\mathbf{v}$.

The formula ψ' is in GF, because all parameters contained in the body of a quantified formula are now protected by its guard. Furthermore, take an arbitrary interpretation ρ of ψ and turn it into an interpretation ρ' of ψ' by interpreting Par as always true and by interpreting the relations $R'(\mathbf{v}, \mathbf{x})$ of ψ' like $R(\mathbf{x})$ of ψ. If ψ' is valid, ρ' is a model of ψ', so ρ is a model of ψ. Conversely, take an arbitrary interpretation ρ' of ψ' and turn it into an interpretation ρ of ψ by interpreting $R(\mathbf{x})$ like $R'(\mathbf{v}, \mathbf{x})$, showing that if ρ is a model of ψ, so ρ' is a model of ψ'. Altogether, ψ is valid iff ψ' is valid. ■

Let us go one step further and introduce GFP with universal quantification, defined as follows:

Definition 4 (GFP with universal quantification, GFPU):

1) Every GFP is a GFPU.
2) If $\psi(\mathbf{v})$ is GFP and $\phi(\mathbf{v})$ a GFPU, then $\forall \mathbf{v}.\psi(\mathbf{v}) \longrightarrow \phi(\mathbf{v})$ is a GFPU. The variables $\mathbf{v}$ may be free variables or parameters of the formulas ψ and ϕ.

Our interest in these formulas derives from the fact that we obtain essentially this class of formulas when extracting proof obligations from programs, as described in Section IV-C. To be more precise, these verification conditions still have to undergo a process of elimination of substitutions that will be explicated in Section V-B.

Note that ψ in the above definition is not necessarily the guard of ϕ, because ψ may be non-atomic (and possibly contains complex quantifier alternations) and ϕ may contain free variables not guarded by ψ. The relevant aspect of these non-guarded universal quantifications is that they do not occur below an existential quantification and can therefore be moved to the front of the formula, as shown in the following proof.

Lemma 2 (Transformation of GFPU to GFP): Every GFPU formula ψ can be transformed into a GFP formula ψ' such that ψ is valid iff ψ' is valid.

Proof: Given a formula ψ, we first convert it into a formula $\forall \mathbf{v}.\gamma$ with a possibly empty quantifier prefix $\mathbf{v}$ and such that γ is a GFP formula. To do so, we recursively transform any $\forall \mathbf{v_1}.\psi_1 \longrightarrow (\forall \mathbf{v_2}.\psi_2 \longrightarrow \phi)$ into $\forall \mathbf{v_1}\mathbf{v_2}.\psi_1 \longrightarrow$

$\psi_2 \longrightarrow \phi$, eventually after renaming bound variables. These are equivalence transformations.

In the resulting formula $\forall \mathbf{v}.\gamma$, we drop the outermost universal quantifiers and convert the variables of $\mathbf{v}$ to parameters. Obviously, $\forall \mathbf{v}.\gamma$ is valid if γ is. ■

Taking together Proposition 1 and Lemmas 1 and 2, we see that the problem of validity of GFPU is decidable.

IV. Transformation Language

A. Syntax

We now introduce more formally the transformation language of which we have already presented an example in Section II. The syntax of statements *stmt* and programs *prog* is defined by the following grammar:

$$
\begin{array}{lcl}
\mathit{stmt} & ::= & \texttt{Skip} \\
 & | & \texttt{add}(R(\mathbf{v})) \\
 & | & \texttt{delete}(R(\mathbf{v})) \\
 & | & \texttt{select}\ \mathbf{v}\ \texttt{with}\ \mathit{form} \\
 & | & \mathit{stmt};\ \mathit{stmt} \\
 & | & \texttt{if}\ \mathit{form}\ \texttt{then}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt} \\
 & | & \texttt{while}\ \mathit{form}\ \texttt{inv}\ \mathit{form}\ \texttt{do}\ \mathit{stmt} \\
 & & \\
\mathit{prog} & ::= & \texttt{Pre}:\mathit{form}\ \ \mathit{stmt}\ \ \texttt{Post}:\mathit{form}
\end{array}
$$

The statements define how to modify a given graph; a program is a statement annotated with pre- and postconditions.

The statements feature traditional constructors such as `Skip` (no effect), sequential composition, a conditional statement and loops; the latter are annotated with an invariant (the formula following `inv`).

The more specific constructors are

- `add`, for adding a tuple to a relation. In the context of graph transformations, the relation R is typically binary, and the tuple $\mathbf{v}$ identifies the source and target of an edge to be inserted. Thus, $\texttt{add}(R(n_1, n_2))$ inserts an arc between nodes n_1 and n_2 for relation R (arcs are directed, labelled by relations, and there cannot be multiple arcs of the same relation between the same nodes). Similarly, $\texttt{add}(C(n))$ adds node n to set C, or labels node n with class C. However, note that `add` and also `delete` are not limited to unary or binary relations.
- `delete`, for deleting a tuple from a relation, similar in spirit to `add`.
- `select`, for selecting a tuple satisfying the condition specified by the formula. `select` can be understood as a simultaneous assignment to the variables of the tuple $\mathbf{v}$. It may also be understood as the result of a database query with further processing of one of the result tuples. Since several tuples may satisfy the selection condition, the assignment is non-deterministic. For example, the `select` of Figure 1, when applied to the graph on the left side of Figure 2, may choose the assignment $(a, b, c) := (a_3, b_2, c_1)$, leading to the graph on the right side of Figure 2. It might also choose the assignment $(a, b, c) := (a_3, b_2, c_2)$, which would produce a different graph, but which would satisfy the same specification.

The formulas *form* occurring in statements and programs are syntactically arbitrary first-order formulas. We will have to impose further well-formedness constraints on these formulas to obtain proof obligations that are in the GFPU fragment (see Definition 4). We defer their definition to the discussion of verification conditions in Section IV-C.

B. Operational Semantics

The semantics of the programming language is given by an operational semantics as defined in Figure 3. Programs manipulate interpretations into relational structures, as defined in Section III-A. The rules describe how a relational interpretation (corresponding to a program state) ρ is transformed to a successor state ρ' under the effect of executing a statement c, written as $(c, \rho) \Rightarrow \rho'$.

The rules for the traditional program constructors are as usual. Note that the invariant of loops has no operational significance and is therefore omitted in the rules.

The semantics of `add` and `delete` is defined with the aid of two auxiliary functions:

- *add_rel*, which intuitively adds the tuple $\mathbf{v}$ under the current interpretation to the interpretation of R. More formally, if $\rho = (\rho_d, \rho_r, \rho_i)$ and $\rho' = (\rho'_d, \rho'_r, \rho'_i) = \mathit{add\_rel}\ R\ \mathbf{v}\ \rho$, then $\rho'_d = \rho_d$, $\rho'_i = \rho_i$ and ρ'_r is defined so that $\rho'_r(S) = \rho_r(S)$ for $S \neq R$ and $\rho'_r(R) = \rho_r(R) \cup \{\rho_i(\mathbf{v})\}$, where $\rho_i(\mathbf{v})$ is the obvious extension of ρ_i to tuples.
- *delete_rel*, which removes the tuple $\mathbf{v}$ under the current interpretation to the interpretation of R. More formally, the component ρ'_r is defined so that $\rho'_r(S) = \rho_r(S)$ for $S \neq R$ and $\rho'_r(R) = \rho_r(R) - \{\rho_i(\mathbf{v})\}$.

The semantics of `select v with` b is the following: We try to find a valuation ρ' which modifies the current valuation ρ at variables $\mathbf{v}$ with values $\mathbf{vi}$ and such that ρ' satisfies the selection condition b. If such a valuation ρ' exists, the program continues with this valuation. If no such valuation exists, then the program gets stuck at this point. It is the programmer's responsibility to ensure that this situation cannot happen, for example by protecting the `select` with an appropriate `if` or `while`, as in the example of Section II.

C. Verification Conditions

The aim of program verification is to prove that a statement of the programming language always establishes the postcondition, provided that the precondition is satisfied. The approach is standard and proceeds by computing weakest preconditions of a program. However, the programming language is non-standard and the logic of a restricted form, which will require a special treatment discussed in Section V.

Weakest preconditions wp and verification conditions vc are computed recursively over the structure of formulas according to the definition of Figure 4.

The weakest precondition $wp(c, Q)$ is the weakest condition that is required to ensure the postcondition Q after execution of statement c. The main purpose of function vc is to collect

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

$$(\textsc{Add})\quad \frac{\rho' = add\_rel\ R\ \mathbf{v}\ \rho}{(\texttt{add}(R(\mathbf{v})), \rho) \Rightarrow \rho'} \qquad (\textsc{Delete})\quad \frac{\rho' = delete\_rel\ R\ \mathbf{v}\ \rho}{(\texttt{delete}(R(\mathbf{v})), \rho) \Rightarrow \rho'}$$

$$(\textsc{Select})\quad \frac{\exists \mathbf{vi}.(\rho' = \rho^{[\mathbf{v}:=\mathbf{vi}]} \wedge (\rho' \models b))}{(\texttt{select } \mathbf{v} \texttt{ with } b, \rho) \Rightarrow \rho'}$$

$$(\textsc{IfT})\quad \frac{\rho \models b \quad (c_1, \rho) \Rightarrow \rho'}{(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, \rho) \Rightarrow \rho'}$$

$$(\textsc{IfF})\quad \frac{\rho \not\models b \quad (c_2, \rho) \Rightarrow \rho'}{(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, \rho) \Rightarrow \rho'}$$

$$(\textsc{WhileT})\quad \frac{\rho \models b \quad (c, \rho) \Rightarrow \rho'' \quad (\texttt{while } b \texttt{ do } c, \rho'') \Rightarrow \rho'}{(\texttt{while } b \texttt{ do } c, \rho) \Rightarrow \rho'}$$

$$(\textsc{WhileF})\quad \frac{\rho \not\models b}{(\texttt{while } b \texttt{ do } c, \rho) \Rightarrow \rho}$$

Fig. 3: Big-step semantics rules

all the conditions that are required to ensure local conditions, in particular the correctness of invariants in loops.

We are here interested in partial correctness: showing that a program is correct provided it terminates and does not get stuck during execution, for example due to an unsatisfiable `select` statement. The functions wp and vc are related to the operational semantics by the following theorem:

Theorem 1 (Soundess): If $vc(c, Q)$ is valid and $(c, \rho) \Rightarrow \rho'$, then $\rho \models wp(c, Q)$ implies $\rho' \models Q$.

Differently said, if the initial program state ρ satisfies the weakest precondition $wp(c, Q)$, then the final program state ρ' satisfies the postcondition Q provided that the verification condition $vc(c, Q)$ is valid.

The soundness proof follows a standard argument (see for example [15], [16]) and will not be spelled out here.

Definition 5 (Program correctness): The program $prg = (\texttt{Pre}: P\ \ c\ \ \texttt{Post}: Q)$ is said to be correct, $corr(prg)$, if the formula $vc(c, Q) \wedge (P \longrightarrow wp(c, Q))$ is valid.

We attract the reader's attention to some syntactic aspects: in the clauses for `add` and `delete`, we use substitutions of the form $Q[R := R \pm \mathbf{v}]$. These substitutions are traditional in classical Hoare-style verification [17] but are here not understood as meta-operations, but as new constructors that we add temporarily to the logical language, in the style of explicit substitutions [18]. For the fragments GF, GFP, GFPU defined

$$\begin{array}{l}
wp(\texttt{Skip},\ Q) = Q\\
wp(\texttt{add}(R(\mathbf{v})),\ Q) = Q[R := R + \mathbf{v}]\\
wp(\texttt{delete}(R(\mathbf{v})),\ Q) = Q[R := R - \mathbf{v}]\\
wp(\texttt{select } \mathbf{v} \texttt{ with } b,\ Q) = \forall \mathbf{v}.(b \longrightarrow Q)\\
wp(c_1; c_2,\ Q) = wp(c_1, wp(c_2,\ Q))\\
wp(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2,\ Q) =\\
\qquad ite(b, wp(c_1, Q), wp(c_2, Q))\\
wp(\texttt{while } b \texttt{ inv } iv \texttt{ do } c,\ Q) = iv\\
\\
vc(\texttt{Skip},\ Q) = \top\\
vc(\texttt{add}(R(\mathbf{v})),\ Q) = \top\\
vc(\texttt{delete}(R(\mathbf{v})),\ Q) = \top\\
vc(\texttt{select } \mathbf{v} \texttt{ with } b,\ Q) = \top\\
vc(c_1; c_2,\ Q) = vc(c_1, wp(c_2, Q)) \wedge vc(c_2, Q)\\
vc(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2,\ Q) = vc(c_1, Q) \wedge vc(c_2, Q)\\
vc(\texttt{while } b \texttt{ inv } iv \texttt{ do } c,\ Q) =\\
\qquad (iv \wedge \neg b \longrightarrow Q) \wedge (iv \wedge b \longrightarrow wp(c, iv)) \wedge vc(c, iv)
\end{array}$$

Fig. 4: Weakest preconditions and verification conditions

in Section III, we define extensions GF$^\sigma$, GFP$^\sigma$, GFPU$^\sigma$ that may in addition contain substitutions. In Section V, we will show how to eliminate substitutions.

Definition 6 (Well-formed conditions): We say that a statement has well-formed conditions if the conditions b and invariants iv occurring in `select`, `if` and `while` statements are GFP formulas.

We say that a program $\texttt{Pre}: P\ \ c\ \ \texttt{Post}: Q$ has well-formed conditions if P is a GFP formula and Q a GFPU formula.

Lemma 3:

1) If statement c has well-formed conditions and Q is a GFPU$^\sigma$ formula, then $wp(c, Q)$ and $vc(c, Q)$ are GFPU$^\sigma$ formulas.
2) If program prg has well-formed conditions, then its program correctness formula $corr(prg)$ is a GFPU$^\sigma$ formula.

Proof: The proof of the first part is by induction on the structure of statements, keeping in mind that if Q is a GFPU$^\sigma$ formula, then so is $Q\sigma$ for a substitution σ. Furthermore, GFPU$^\sigma$ formulas are closed by propositional connectives. The most interesting case is the `select` statement: if Q is a GFPU$^\sigma$ formula and b a GFP formula (by well-formedness), then $\forall \mathbf{v}.b \longrightarrow Q$ is a GFPU$^\sigma$ formula. ■

We end this section with a remark on consecutive `select` statements. In the program

```
Pre: $P$
select ($a$, $b$) with ¬ $R(a,b)$;
add($R(a,b)$);
select ($a$, $b$) with $R(a,b)$;
delete($R(a,b)$)
Post: $Q$
```

the variables a (resp. b) in the first and second `select` may refer to the same or different nodes. The weakest precondition $\forall a, b. \neg R(a, b) \longrightarrow$

$(\forall a, b. R(a, b) \longrightarrow Q[R := R - (a, b)])[R := R + (a, b)]$ shows the different bindings to the variables a and b. When computing prenex forms as in Lemma 2, bound variables have to be correctly renamed to avoid conflicts.

V. Correctness Proofs

In this section, we will discuss how to transform the proof obligations derived from programs, as shown in Section IV-C, into formulas of the Guarded Fragment. We will characterise more in detail the nature of substitutions (Section V-A) and then describe how to eliminate them (Section V-B), to obtain formulas of the fragment GFPU. In Section V-C, we summarise the development.

A. Substitutions

Syntactically, substitutions σ can take one of the two forms $[R := R + \mathbf{v}]$ and $[R := R - \mathbf{v}]$. We implicitly assume that the number of elements of $\mathbf{v} = (v_1, \ldots, v_n)$ is the same as the arity of R. The set of free variables $FV(\sigma)$ of a substitution is the set $\{v_1, \ldots, v_n\}$.

Let σ be a substitution and ϕ a formula. We write $\phi\sigma$ for the application of a substitution to the formula. As mentioned before, the application of a substitution is here taken to be a genuine formula constructor and not a meta-operation.

In extension to the semantics of first-order logic (Definition 1), we therefore define:

- $\rho \models \phi[R := R + \mathbf{v}]$ if $add\_rel\ R\ \mathbf{v}\ \rho \models \phi$
- $\rho \models \phi[R := R - \mathbf{v}]$ if $delete\_rel\ R\ \mathbf{v}\ \rho \models \phi$

For the definition of add_rel and $delete\_rel$, see Section IV-B.

In the description below, we use *variable renamings* that we clearly distinguish from the substitutions described above.

Definition 7 (Variable renaming): A renaming of variables $\mathbf{u}$ to $\mathbf{v}$ in formula ψ is written as $\psi[\mathbf{u} := \mathbf{v}]$ and is obtained by replacing the variables $\mathbf{u}$ in ψ by the variables $\mathbf{v}$, eventually renaming bound variables to avoid variable capture.

Lemma 4 (Parameters in renaming): Let P be a set of parameters and ψ a formula with parameters $S \subseteq P$. Then $\psi[\mathbf{u} := \mathbf{v}]$ is a formula with parameters $S' \subseteq (S - U) \cup V$, where $U \subseteq P$ resp. $V \subseteq P$ are the parameters among $\mathbf{u}$ resp. $\mathbf{v}$.

Proof: The proof is by induction on ψ. Note that not all variables of $\mathbf{u}$ necessarily occur in ψ, whence the fact that the parameters of $\psi[\mathbf{u} := \mathbf{v}]$ can be a strict subset of $(S - U) \cup V$. ■

B. Elimination of Substitutions

We recall that the verification conditions extracted from well-formed programs are formulas of GFPU$^\sigma$, *i.e.* of the fragment GFPU with substitutions.

We now show that any formula of GFPU$^\sigma$ can be converted to an equivalent formula of GFPU; and any formula of GFP$^\sigma$ can be converted to an equivalent formula of GFP. We proceed in two steps: We first produce a pre-GFPU (resp. pre-GFP) form, which is syntactically not a GFPU (resp. pre-GFP) formula; and we then convert pre-GFP(U)-forms to GFP(U) formulas.

1) Computing a pre-GFP(U) form: Substitutions are essentially pushed recursively into a formula according to the following equations:

- For propositional connectives: $(\neg\psi)\sigma = \neg(\psi\sigma)$ and $(\psi \wedge \phi)\sigma = (\psi\sigma \wedge \phi\sigma)$ and $(\psi \vee \phi)\sigma = (\psi\sigma \vee \phi\sigma)$
- For quantifiers: $(\exists \mathbf{y}.\psi)\sigma = (\exists \mathbf{y}.\psi\sigma)$ and $(\forall \mathbf{y}.\psi)\sigma = (\forall \mathbf{y}.\psi\sigma)$, possibly after renaming bound variables to make them different from $FV(\sigma)$.
- For propositional constants $\perp \sigma = \perp$, and for equality $(x = y)\sigma = (x = y)$
- For relational applications, depending on the substitution:
  - $R(x_1, \ldots, x_n)[R := R + (v_1, \ldots, v_n)] = R(x_1, \ldots, x_n) \vee \bigwedge_{i=1}^n x_i = v_i$
  - $R(x_1, \ldots, x_n)[R := R - (v_1, \ldots, v_n)] = R(x_1, \ldots, x_n) \wedge \bigvee_{i=1}^n x_i \neq v_i$

Lemma 5: Elimination of substitutions is semantics-preserving.

It is easy to see that applying substitutions to propositional connectives, constants, equality and relational applications preserves the guarded fragment. However, this is no so for quantified formulas where the guard is a relational application, as for example in $(\exists y. R(x, y) \wedge R(y, x))[R := R + (v_1, v_2)] = (\exists y. (R(x, y) \vee x = v_1 \wedge y = v_2) \wedge (R(y, x) \vee y = v_1 \wedge x = v_2))$. There are two problems with this formula: the first conjunct below the quantifier is not an atomic guard; and new variables have been introduced into the second conjunct, and these variables are possibly not guarded any more. The following transformation deals with the first problem, whereas the fact that we allow parameters deals with the second problem.

2) Producing GFP(U) formulas: The transformations of Section V-B1 produce quantified pre-GFP(U) forms that can be of three kinds:

1) $\exists \mathbf{y}.\rho \wedge \gamma$ if the formula resulted from a substitution into an existentially quantified guarded formula, so ρ is $R(\mathbf{x}, \mathbf{y}) \vee (\bigwedge \mathbf{x} = \mathbf{v_x} \wedge \bigwedge \mathbf{y} = \mathbf{v_y})$ or $R(\mathbf{x}, \mathbf{y}) \wedge (\bigvee \mathbf{x} \neq \mathbf{v_x} \vee \bigvee \mathbf{y} \neq \mathbf{v_y})$. Here, we have split up the vector of substitution variables $\mathbf{v}$ into the variables $\mathbf{v_x}$ assigned to the free variables $\mathbf{x}$ and $\mathbf{v_y}$ assigned to the bound variables $\mathbf{y}$. By an inductive argument, we assume that γ is already a GFP formula. We will analyse this situation in detail below.
2) $\forall \mathbf{y}.\rho \longrightarrow \gamma$ if the formula resulted from a substitution into a universally quantified guarded formula, and where ρ and γ are as above. This case is handled in analogy to the case of existential quantification.
3) $\forall \mathbf{y}.\gamma_1 \longrightarrow \gamma_2$ if the formula resulted from a substitution into a universal quantification of the form $\forall \mathbf{v}.\psi \longrightarrow \phi$ where ψ is a GFP but not an atom and ϕ is a GFPU (see Definition 4). By induction, we can assume that γ_1 is a GFP formula and γ_2 a GFPU formula, so $\forall \mathbf{y}.\gamma_1 \longrightarrow \gamma_2$ is a GFPU formula as required.

We discuss in detail the transformation for formulas of the form $\exists \mathbf{y}.\rho \wedge \gamma$, as in the first case above.

- If ρ has the first form, we start with the formula $\exists \mathbf{y}.(R(\mathbf{x}, \mathbf{y}) \vee (\bigwedge \mathbf{x} = \mathbf{v_x} \wedge \bigwedge \mathbf{y} = \mathbf{v_y})) \wedge \gamma$. By distributing conjunctions over disjunctions, we obtain $\exists \mathbf{y}.(R(\mathbf{x},\mathbf{y}) \wedge \gamma) \vee ((\bigwedge \mathbf{x} = \mathbf{v_x} \wedge \bigwedge \mathbf{y} = \mathbf{v_y}) \wedge \gamma)$ and by distributing the existential quantifier over disjunctions: $(\exists \mathbf{y}.(R(\mathbf{x},\mathbf{y}) \wedge \gamma)) \vee (\bigwedge \mathbf{x} = \mathbf{v_x} \wedge (\exists \mathbf{y}. \bigwedge \mathbf{y} = \mathbf{v_y} \wedge \gamma))$. Using equivalences like $(x = v \wedge P(x)) = (x = v \wedge P(v))$ and $(\exists y.y = v \wedge P(y)) = P(v)$, this can further be simplified to $(\exists \mathbf{y}.(R(\mathbf{x},\mathbf{y}) \wedge \gamma)) \vee (\bigwedge \mathbf{x} = \mathbf{v_x} \wedge (\gamma[\mathbf{x} := \mathbf{v_x}, \mathbf{y} := \mathbf{v_y}]))$ where $\gamma[\mathbf{x} := \mathbf{v_x}, \mathbf{y} := \mathbf{v_y}]$ is a renaming of variables in the sense of Definition 7. According to Lemma 4, this formula is a formula of GFP with parameters among those of γ and $\mathbf{v_x}$ and $\mathbf{v_y}$.

- If ρ has the second form, we start with $\exists \mathbf{y}.(R(\mathbf{x},\mathbf{y}) \wedge (\bigvee \mathbf{x} \neq \mathbf{v_x} \vee \bigvee \mathbf{y} \neq \mathbf{v_y})) \wedge \gamma$, which can be rewritten to $\bigvee_i (\exists \mathbf{y}.R(\mathbf{x},\mathbf{y}) \wedge x_i \neq v_{x_i} \wedge \gamma) \vee \bigvee_j (\exists \mathbf{y}.R(\mathbf{x},\mathbf{y}) \wedge y_j \neq v_{y_j} \wedge \gamma)$.
  The first disjunct is equivalent to $(\bigvee_i (x_i \neq v_{x_i})) \wedge (\exists \mathbf{y}.R(\mathbf{x},\mathbf{y}) \wedge \gamma)$ which is a GFP formula.
  Each of the disjuncts of $\bigvee_j (\exists \mathbf{y}.R(\mathbf{x},\mathbf{y}) \wedge y_j \neq v_{y_j} \wedge \gamma)$ is a GFP formula, with parameters among the parameters of γ or v_{y_j}.

From the fact that in the preceding discussion, we have only performed equivalence transformations, we get:

Lemma 6: The transformation of pre-GFP(U) forms to GFP(U) is semantics preserving.

C. Proof Procedure

We summarise the procedure and illustrate it with a part of the proof obligations arising when verifying the example program of Figure 1. Let *lbody* the body of the *while* loop.

The essential steps for verifying the program $(\texttt{Pre} : P \;\; c \;\; \texttt{Post} : Q)$ are:

1) Check that the conditions in the program are well-formed in the sense of Definition 6. Otherwise, the proof obligations are not assured to be convertible to GF formulas.
   In the example, the pre- and postconditions (in this case the same as the formula $Invar$) and the loop condition $Continue$ are already GF formulas. The condition of the `select` clause is a GFP formula with parameters a, b, c.
2) In general, to extract the verification conditions as $corr(prg)$, if $vc(c, Q) \wedge (P \longrightarrow wp(c, Q))$.
   In the example program of Figure 1, one subformula will have the form $Invar \wedge Continue \longrightarrow wp(lbody, Invar)$. In the following discussion, we will concentrate on this subformula.
3) Expand the definitions of vc and wp.
   For our example formula, we obtain $Invar \wedge Continue \longrightarrow \forall a, b, c.\, Selcond \longrightarrow (Invar[R := R - (a,c)][A := A - (a)][D := D + (a)])$, where $Selcond$ is the condition of the `select` statement. We abbreviate the sequence of substitutions by $\vec{\sigma}$.
4) Computing a pre-GFP(U) according to Section V-B1.
   In the example, with formulas $F_1 = (A \cap D = \{\})\vec{\sigma} = (\neg \exists n.A(n) \wedge D(n))\vec{\sigma} = (\neg \exists n.(A(n) \wedge n \neq a) \wedge (D(n) \vee n = a))$ and $F_2 = (\forall d.D(d) \longrightarrow (\exists b.R(d,b) \wedge B(b)))\vec{\sigma} = (\forall d.(D(d) \vee d = a) \longrightarrow (\exists b.(R(d,b) \wedge (d \neq a \vee b \neq c)) \wedge B(b)))$, we obtain $Invar \wedge Continue \longrightarrow \forall a, b, c.\, Selcond \longrightarrow F_1 \wedge F_2$.
5) Computing a GFP(U) formula according to Section V-B2.
   Transforming F_1, we get $(\neg(\exists n.(A(n) \wedge n \neq a \wedge D(n)) \vee (A(n) \wedge n = a \wedge n \neq a)))$, which we simplify on the fly to $F_1 = \neg(\exists n.(A(n) \wedge n \neq a \wedge D(n)))$
   For transforming F_2, we change $(\exists b.(R(d,b) \wedge (d \neq a \vee b \neq c)) \wedge B(b))$ to $F_3 = ((\exists b.R(d,b) \wedge d \neq a \wedge B(b)) \vee (\exists b.R(d,b) \wedge b \neq c \wedge B(b)))$. With this, we separate $(\forall d.(D(d) \vee d = a) \longrightarrow F_3$ into $(\forall d.D(d) \longrightarrow F_3) \wedge (\forall d.d = a \longrightarrow F_3)$, which after some further transformations becomes $F_2 = (\forall d.D(d) \longrightarrow F_3) \wedge (\exists b.R(a,b) \wedge b \neq c \wedge B(b))$.
   Let us pause at this moment to reconsider the notion of parameter. Seen locally, in F_1, the variable a is a parameter because it is a free variable that does not occur in the guard $A(n)$. Similarly, in the subformula $\exists b.R(d,b) \wedge b \neq c \wedge B(b)$, the free variable c is a parameter, but d is not because it is contained in the guard $R(d,b)$. Altogether, when considering a, b, c as parameters, formula $Selcond \longrightarrow F_1 \wedge F_2$ is a correct GFP formula, even though it is not a GF formula. When closing this formula universally, we obtain a GFPU formula with an empty parameter set.
6) Transform the GFPU formula into a GFP formula according to Lemma 2.
   In the example, we move the universal quantifiers of a, b, c outside, to obtain: $\forall a, b, c.\, Invar \wedge Continue \longrightarrow Selcond \longrightarrow F_1 \wedge F_2$, and then drop these three quantifiers. Now, a, b, c are parameters again.
7) Transform the GFP formula into a GF formula according to Lemma 1.
   In the example, with parameters a, b, c, we construct a guarded formula $\forall a, b, c.\, Par(a, b, c) \longrightarrow (Invar' \wedge Continue' \longrightarrow Selcond' \longrightarrow F_1' \wedge F_2')$.
   Let us look at selected subformulas. Following the algorithm of Lemma 1, $A(a)$ in $Selcond$ is transformed to $A'(a, b, c, a)$ in $Selcond'$, which is guarded by $Par(a, b, c)$. The subformula $\exists b.R(a,b) \wedge b \neq c \wedge B(b)$ of F_2, with c not guarded, now becomes $\exists b'.R'(a, b, c, a, b') \wedge b' \neq c \wedge B'(a, b, c, b')$ in F_2', which is a GF formula.

VI. Conclusions

We have presented a graph transformation language and an assertion formalism based on the Guarded Fragment of predicate logic. The focus in this paper has been on the manipulation of formulas that are required to reduce verification conditions to formulas of GF, which establishes the decidability of the reasoning problem.

Several questions remain open, both practical and foundational. We will work on the transformation language itself, introducing, among others, iterators or primitives for allocating or deallocating new nodes in a graph. Whereas the language of Section IV only contains the level of statements, we would also like to introduce a procedural abstraction.

We also intend to work on enhancements of the assertion formalism, to make it more expressive. One of the attractive features of some Description Logics is their ability to count, for example the number of successors of a node via so-called "number restrictions" [19]. Unfortunately, adding counting quantifiers to GF makes it undecidable [9]. Other fragments of first-order logic, such as the two-variable fragment, permit counting quantifiers [20], and their usefulness will have to be evaluated. And a full formal comparison with a formalism [21] based on two-variable logic and PDL [22] still has to be done.

REFERENCES

[1] R. Cyganiak, M. Lanthaler, and D. Wood, "RDF 1.1 Concepts and Abstract Syntax," http://www.w3.org/TR/rdf11-concepts, 2014.

[2] OWL Standard, OWL Standard. http://www.w3.org/2001/sw/wiki/OWL.

[3] SPARQL Standard, SPARQL Standard. http://www.w3.org/2001/sw/wiki/SPARQL.

[4] F. Baader and C. Lutz, "Description logic," in *The Handbook of Modal Logic*, P. Blackburn, J. van Benthem, and F. Wolter, Eds. Elsevier, 2006, pp. 757–820. [Online]. Available: http://www.informatik.uni-bremen.de/tdki/research/papers/2006/BaLu-ML-Handbook-06.ps.gz

[5] J. H. Brenas, R. Echahed, and M. Strecker, "On the closure of description logics under substitutions," in *Proceedings of the 29th International Workshop on Description Logics, Cape Town, South Africa, April 22-25, 2016.*, M. Lenzerini and R. Peñaloza, Eds., 2016. [Online]. Available: http://ceur-ws.org/Vol-1577/paper_47.pdf

[6] ——, "Ensuring correctness of model transformations while remaining decidable," in *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings*, ser. Lecture Notes in Computer Science, A. Sampaio and F. Wang, Eds., vol. 9965, 2016, pp. 315–332. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46750-4_18

[7] M. Chaabani, R. Echahed, and M. Strecker, "Logical foundations for reasoning about transformations of knowledge bases," in *DL – Description Logics*, ser. CEUR Workshop Proceedings, T. Eiter, B. Glimm, Y. Kazakov, and M. Krötzsch, Eds., vol. 1014. CEUR-WS.org, 2013, pp. 616–627. [Online]. Available: http://www.irit.fr/~Martin.Strecker/Publications/dl_transfo2013.html

[8] H. Andréka, I. Németi, and J. van Benthem, "Modal languages and bounded fragments of predicate logic," *Journal of Philosophical Logic*, vol. 27, no. 3, pp. 217–274, 1998. [Online]. Available: http://www.fenrong.net/teaching/Andreka.pdf

[9] E. Grädel, "On the restraining power of guards," *Journal of Symbolic Logic*, vol. 64, pp. 1719–1742, 1999. [Online]. Available: http://www.logic.rwth-aachen.de/pub/graedel/Gr-jsl99.ps

[10] E. Grädel, "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. [Online]. Available: http://www.logic.rwth-aachen.de/pub/graedel/Gr-kalmar03.ps

[11] E. Grädel and I. Walukiewicz, "Guarded Fixed Point Logic," in *Proceedings of 14th IEEE Symposium on Logic in Computer Science LICS '99, Trento*, 1999, pp. 45–54. [Online]. Available: http://www.logic.rwth-aachen.de/pub/graedel/GrWa-lics99.ps

[12] E. Grädel, "Decision procedures for guarded logics," in *Automated Deduction – CADE16. Proceedings of 16th International Conference on Automated Deduction, Trento, 1999*, ser. LNCS, vol. 1632. Springer, 1999. [Online]. Available: http://www.logic.rwth-aachen.de/pub/graedel/Gr-cade99.ps

[13] C. Hirsch, "Guarded Logics: Algorithms and Bisimulation," Ph.D. dissertation, RWTH Aachen, 2002. [Online]. Available: http://www.logic.rwth-aachen.de/pub/hirsch/hirsch.pdf

[14] J. Hladik, "Implementation and optimisation of a tableau algorithm for the guarded fragment," in *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 2002, pp. 145–159.

[15] G. Winskel, *The formal semantics of programming languages*. MIT Press, 1993.

[16] T. Nipkow and G. Klein, *Concrete Semantics*. http://concrete-semantics.org/, 2014. [Online]. Available: http://concrete-semantics.org/

[17] R. Virga, "Efficient substitution in Hoare logic expressions," *Electr. Notes Theor. Comput. Sci.*, vol. 41, no. 3, pp. 35–49, 2000.

[18] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy, "Explicit substitutions," *Journal of Functional Programming*, vol. 1, no. 4, pp. 375–416, October 1991.

[19] B. Hollunder and F. Baader, "Qualifying number restrictions in concept languages," in *KR*, 1991, pp. 335–346. [Online]. Available: http://www.dfki.uni-kl.de/dfkidok/publications/RR/91/03/abstract.html

[20] E. Grädel, M. Otto, and E. Rosen, "Two-Variable Logic with Counting is Decidable," in *Proceedings of 12th IEEE Symposium on Logic in Computer Science LICS '97, Warschau*, 1997. [Online]. Available: http://www.logic.rwth-aachen.de/pub/graedel/gorc2.ps

[21] J. H. Brenas, R. Echahed, and M. Strecker, "Proving correctness of logically decorated graph rewriting systems," in *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal*, ser. LIPIcs, D. Kesner and B. Pientka, Eds., vol. 52. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, pp. 14:1–14:15. [Online]. Available: http://dx.doi.org/10.4230/LIPIcs.FSCD.2016.14

[22] M. J. Fischer and R. E. Ladner, "Propositional dynamic logic of regular programs," *J. Comput. Syst. Sci.*, vol. 18, no. 2, pp. 194–211, 1979. [Online]. Available: http://dx.doi.org/10.1016/0022-0000(79)90046-1