A Precondition Calculus for Correct-by-Construction Graph Transformations

Amani Makhlouf, Christian Percebois, Hanh Nhi Tran

A Precondition Calculus for Correct-by-Construction Graph Transformations

Amani Makhlouf, Christian Percebois, Hanh Nhi Tran

IRIT, University of Toulouse
Toulouse, France
Email: {Amani.Makhlouf | Christian.Percebois | Hanh-Nhi.Tran}@irit.fr

***Abstract*—We aim at assisting developers to write, in a Hoare style, provably correct graph transformations expressed in the $\mathcal{ALCQ}$ Description Logic. Given a postcondition and a transformation rule, we compute the weakest precondition for developers. However, the size and quality of this formula may be complex and hard to grasp. We seek to reduce the weakest precondition's complexness by a static analysis based on an alias calculus. The refined p recondition i s p resented t o t he d eveloper i n t erms of alternative formulae, each one specifying a potential matching of the source graph. The developer chooses then the formulae that correspond to his intention to obtain finally a correct-by-construction Hoare triple.**

Keywords–Graph transformation; Description Logics; weakest precondition calculus; static analysis; alias calculus.

I. Introduction

All approaches applying production rules to a graph require to implement a binary relation between a source graph and a target graph. In the theory of algebraic graph transformations, Habel and Pennemann [1] defined nested graph conditions as a graphical and logical formalism to specify graph constraints by explicitly making use of graphs and graph morphisms. Nested conditions have the same expressive power as Courcelle's first-order g raph l ogic [1][2][3]. H owever, t hey n eed t o be derived into specific i nference r ules i n o rder t o b e proved in a specific t heorem-prover t hat s uits t hem [4][5]. Moreover, this transformation requires the proof of a sound and complete proof system for reasoning in the proposed logic.

Another way to express and reason about graph properties is to directly encode graphs in terms of some existing logic [6]. This solution leads to consider connections between graph constraints and first-order g raph f ormulae. A dopting t his approach, we define graphs a xiomatically b y $\mathcal{ALCQ}$ Description Logic (DL) predicates [7] and manipulate them with specific statements. In this way, we designed a non-standard imperative programming language named Small-t$\mathcal{ALC}$ dedicated to transform labeled directed graphs. Note that $\mathcal{ALC}$ is prototypical for DLs.

Despite the above differences from algebraic graph transformations, we point out the common idea to use satisfiability solvers to prove rules'correctness. This technique requires to assign a predicate transformer to a rule in order to compute the rule's weakest precondition. The setup is rather traditional: given a Hoare triple $\{P\}S\{Q\}$, we compute the weakest (liberal) precondition $wp(S, Q)$ of the rule transformation statements S with respect to the postcondition Q, and then verify the implication $P \Rightarrow wp(S, Q)$. The correctness of the rule is proved by a dedicated tableau reasoning, which is sound, complete and which results in a counter-example when a failure occurs.

Since writing complete and correct specifications may not be easy for novice developers, we aim to assist them in achieving provably correct transformations [8]. In this context, we propose a static analysis of the weakest precondition based on an alias calculus in order to suggest precondition formulae that are easier to understand but still ensuring the correctness of the Hoare-triple. The result is presented to developers in a disjunctive normal form. Each conjunction of positive and negative literals specifies a potential matching of the source graph. By letting developers choose a conjunction as a premise that reflects the rule's intention, our approach can filter and reduce some combinatorial issues.

In this paper, Section II first defines logic-based formulae to annotate pre- and postconditions of a transformation rule. This choice yields manageable proof obligations in a Hoare's style for rules'correctness. Then, we introduce in Section III Small-t$\mathcal{ALC}$ atomic statements that manipulate graph structures. Each statement is characterized by a weakest precondition with respect to a given postcondition. On the basis of an alias calculus that is presented in Section IV, we show in Section V how to reduce some combinatorial issues while ensuring the program correctness by finely analyzing the weakest precondition. An illustrative example is presented in Section VI. We finally give some discussions on related work in Section VII and wrap up the paper with a conclusion and possible improvements in Section VIII.

II. Logic-based conditions

Slightly diverged from the standard approach, we choose a set-theoretic approach for our transformation system [9]. The basic idea is to specify sets of nodes and edges of a subgraph using a fragment of first-order logic. It turns out that replacing graph patterns by graph formulae yields manageable proof obligations for rules'correctness in a Hoare style $\{P\}S\{Q\}$ [6]. A precondition formula P designates a subgraph matching a substructure that should exist in the source graph. The postcondition Q requires the existence of the subgraph represented by Q in the target graph. For instance, consider a rule requiring that: (1) x must be a node (individual) not connected by the relation (role) R to a node y; (2) y is of class (concept) C; (3) x is linked to at most three successors (qualified number of restrictions) of class C via

R. This precondition can be expressed by the logic formula $x\ \neg R\ y\ \wedge\ y : C \wedge\ x : (\leq\ 3\ R\ C)$.

At this point, readers familiar with Description Logics (DLs) may recognize a DL formula. Labeled directed graphs can be directly modeled by entities of DLs, a family of logics for modeling and reasoning about relationships in a domain of interest [10]. Most DLs are decidable fragments of first-order logic. They are organized around three kinds of entities: individuals, roles and concepts. Individuals are constants in the domain, roles are binary relations between individuals and concepts are sets of individuals. Applied to our graphs, individuals are nodes labeled with concepts and roles are edges. Accordingly, pre- and post-assertions are interpreted as graphs by using unary predicates for nodes and binary predicates for edges. The correctness of a graph transformation rule is checked by assigning to each of its statements a predicate transformer in order to compute the corresponding weakest precondition.

To design our own experimental graph transformation language, we chose the $\mathcal{ALCQ}$ logic, an extension of the standard DL Attributive Language with Complements ($\mathcal{ALC}$) [11], which allows qualifying number restrictions on concepts ($\mathcal{Q}$). $\mathcal{ALCQ}$ is based on a three-tier framework: concepts, facts and formulae. The concept level enables to determine classes of individuals ($\emptyset, C, \neg C, C1 \cup C2$ and $C1 \cap C2$). The fact level makes assertions about individuals owned by a concept ($i : C, i : \neg C, i : (\leq\ n\ R\ C)$ and $i : (\geq\ n\ R\ C)$), or involved in a role ($i\ R\ j$ and $i\ \neg R\ j$). The third level is about formulae defined by a Boolean combination of $\mathcal{ALCQ}$ facts ($f, \neg f, f1 \wedge f2$ and $f1 \vee f2$).

Figure 1 depicts a model (graph) satisfying the previous precondition $x\ \neg R\ y\ \wedge\ y : C \wedge\ x : (\leq\ 3\ R\ C)$. In this graph, the white circles designate the nodes variables x and y manipulated by the formula. Nodes variables refer (by a dotted edge) to real nodes represented by black circles. The «◉» node outlines a concept labeled with C. Note that the subgraph having two anonymous nodes each one outfitted with an incoming edge from x and an outgoing edge to the concept C is a model which checks the fact $x : (\leq\ 3\ R\ C)$.

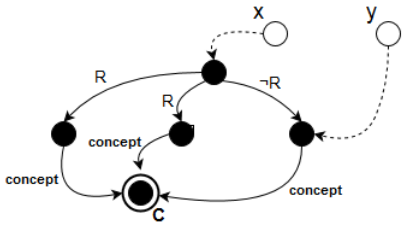


Figure 1. Model satisfying the precondition
$x\ \neg R\ y\ \wedge\ y : C \wedge\ x : (\leq\ 3\ R\ C)$

Our formulae contain free variables that assign references to nodes in a graph. Equality and inequality assertions can be used to define constraints on the value of these variables. If x and y are node variables, $x = y$ means that x and y refer to the same node and $x \neq y$ means that x and y are distinct. The inequality relationship enforces injective graph morphisms.

III. The Small-t$\mathcal{ALC}$ language

The $\mathcal{ALCQ}$ formulae presented in the previous section have been plugged into our Small-t$\mathcal{ALC}$ imperative language and used in atomic transformation actions on nodes (individuals) and edges (roles), as well as in traditional control-flow constructs as loops ($while$) and conditions ($if...then...else...$). In the transformation code, statements manipulate node variables which are bound to the host graph's nodes during the transformation's execution.

We have defined five atomic Small-t$\mathcal{ALC}$ statements according to the following grammar where i and j are node variables, C is a concept name, R is a role name, F is an $\mathcal{ALCQ}$ formula and v is a list of node variables:

atomic_statement ::=

$add(i : C)$	(node labeling)
$\mid delete(i : C)$	(node unlabeling)
$\mid add(i\ R\ j)$	(edge labeling)
$\mid delete(i\ R\ j)$	(edge unlabeling)
$\mid select\ v\ with\ F$	(assignment)

The first four statements modify the graph structure by changing the labeling of nodes and edges. Note that since we consider a set-theoretic approach, the statements $add(i : C)$ and $add(i\ R\ j)$ have no effects if i belongs to the set C and (i, j) to R respectively. Hence, no parallel edges with the same label are allowed. An original construct is the $select$ statement that non-deterministically binds node variables to nodes in the subgraph that satisfies a logic formula. This assignment is used to handle the selection of specific nodes where the transformations are requested to occur. For instance, $select\ i\ with\ i : C$ selects a node labeled with C. If the selection is satisfied the execution continues normally with the value of the node variable i. Otherwise, the execution meets an error situation.

A Small-t$\mathcal{ALC}$ program consists of a sequence of transformation rules. A rule is structured into three parts: a precondition, the transformation code (a sequence of statements) and a postcondition. We illustrate in Figure 2 an example of a transformation rule written in Small-t$\mathcal{ALC}$. The rule first selects a node n of concept A that is R-linked to a. Then, it deletes this link and removes a from the concept A.

pre: $(a : A) \wedge a : (\geq\ 3\ R\ A)$;
$select\ n\ with\ (a\ R\ n) \wedge (n : A)$
$delete(a\ R\ n)$;
$delete(a : A)$;
post: $(a : \neg A) \wedge a : (\geq\ 2\ R\ A)$;

Figure 2. Example of a Small-t$\mathcal{ALC}$ rule

We aim at using a Hoare-like calculus to prove that Small-t$\mathcal{ALC}$ graph programs are correct. This verification process is based on a weakest (liberal) precondition (wp) calculus [12]. Each Small-t$\mathcal{ALC}$ statement S is assigned to a predicate transformer yielding an $\mathcal{ALCQ}$ formula $wp(S, Q)$ assuming the postcondition Q. The correctness of a program prg with respect to Q is established by proving that the given precondition P implies the weakest precondition: every model that satisfies P also satisfies $wp(prg, Q)$. Weakest preconditions of Small-t$\mathcal{ALC}$ statements are given in Figure 3.

The weakest precondition calculus computes predicates which are not closed under substitutions with respect to

$$wp(add\ (i:C), Q) = Q[C + i/C]$$
$$wp(delete\ (i:C), Q) = Q[C - i/C]$$
$$wp(add\ (i\ R\ j), Q) = Q[R + (i,j)/R]$$
$$wp(delete\ (i\ R\ j), Q) = Q[R - (i,j)/R]$$
$$wp(select\ v\ with\ F, Q) = \forall v(F \Rightarrow Q)$$

Figure 3. Small-t$\mathcal{ALC}$ weakest preconditions

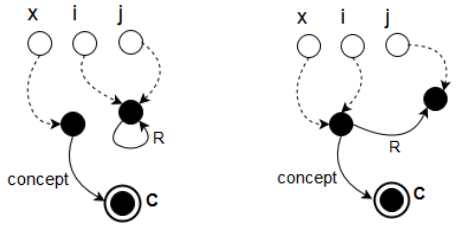
$\mathcal{ALCQ}$. To resolve this situation, substitutions are considered as constructors and should be eliminated. For instance, $wp(add(i:C), x:C) = x:C\ [C+i/C] = x:(C+i) = x:C \vee x = i$.

The conventional precondition calculus presented above does not take into account particular situations of a transformation program and thus may result in a complex precondition. In the following sections, we look at how the precondition's formula can be improved to be more specific and simple on the basis of an alias calculus.

IV. Alias Calculus

The principle of alias calculus was proposed by Bertrand Meyer in order to decide whether two reference expressions appearing in a program might, during some execution, have the same value, meaning that the associated references are attached to the same object [13].

Since our rewriting system allows non-injective morphisms, two or more node variables may reference to the same node in a graph. On the other hand, a node variable can be assigned to a random node of the graph. This is one reason why a Small-t$\mathcal{ALC}$ formula can be represented by several graph patterns. For example, Figure 4 shows two potential models satisfying the formula $x:C\ \wedge\ i\ R\ j$. In Figure 4a, i and j refer to the same node. In 4b, i and j are different but i and x are combined.



(a) $i = j, i \neq x$ (b) $i \neq j, i = x$

Figure 4. Example of models satisfying the formula $x:C\ \wedge\ i\ R\ j$

In this regard, for a transformation program, we apply an alias calculus to determine the node variables that can never refer to the same node. Discerning such specific circumstances helps to discard later unsatisfied subformulae of the weakest precondition. Thus, our method consists in assigning to each node variable *x*, a set of other node variables that may reference to the same node in the graph as *x*. We identify four atomic conditions in which two individuals x and y can never refer to the same node in the graph:

- $x \neq y$
- $\exists C\ /x:C \wedge y:\neg C$
- $\exists R.\exists z\ /x\ R\ z \wedge y\ \neg R\ z$
- $\exists R.\exists z\ /z\ R\ x \wedge z\ \neg R\ y$

The first case ($x \neq y$) states that x and y are naturally distinct so they can never be assigned to the same node. The second one asserts that x and y belong to two complement subsets C and $\neg C$. The same applies to the last two cases where the nodes connected by R and $\neg R$ refer to two disjoint subsets R and $\neg R$.

For each of the above four conditions, x and y are said to be *non-possibly equivalent* nodes. We note this relation by $x \not\simeq y$. As a result we assert that $x \not\simeq y \Rightarrow x \neq y$. However, no conclusion can be drawn from the *possibly equivalent* relation $x \simeq y$.

Consider, as a simple example, the following formula that is presented in the disjunctive normal form: $(x = y \wedge x\ R\ y) \vee (x:C \wedge x\ \neg R\ y)$, and suppose that a static analysis deduces from the code that x and y are non-possibly equivalent, which means that $x \neq y$. As a result, the initial formula can be reduced to $x:C \wedge x\ \neg R\ y$ because the first conjunction $x = y \wedge x\ R\ y$ can never be true in this case. In the section that follows, we show how this calculus helps in reducing the complexness of the weakest precondition.

V. Precondition Extraction

To formally verify the correctness of a Small-t$\mathcal{ALC}$ graph transformation, besides the code, the program's pre- and post-conditions must be properly specified. This task may not be easy for novice developers, so a suggestion of a valid precondition that corresponds to a given code and a postcondition would be useful to them.

Since the computed weakest precondition is often very complex and hard to comprehend, we propose a finer static analysis on the basis of the alias calculus of the program to achieve a simpler precondition. The resulting precondition P is presented in a disjunctive normal form (DNF) where each conjunction of P can be considered as a valid precondition on its own. The analysis consists first in converting the postcondition Q to DNF i.e., $Q = \vee Q_i$ where $Q_i = \wedge q_j$ is a conjunction of facts, then calculating for each statement and for each conjunction Q_i the weakest precondition. This process maintains correctness because $wp(S, Q_1) \vee wp(S, Q_2) \Rightarrow wp(S, Q_1 \vee Q_2)$. In each and every step, the formula of the $wp(S, Q_i)$ may be filtered by discarding subformulae according to the identified non-possibly equivalent node variables. A precondition P is obtained such that $P \Rightarrow wp(prg, Q)$, which makes the transformation program prg correct. This process is applied to *add* and *delete* statements as detailed in Section V-1. Regarding the *select* statement, *wp* is reduced differently as presented later in Section V-2.

1) add *and* delete *statements:*
Let us consider first the *add(i : C)* statement. Its weakest precondition with respect to the postcondition $x : C$ is $x : C \vee x = i$, which means that either the node x was already of concept C before adding i to C, or x and i are equal. Knowing that x and i are non-possibly equivalent, it can be stated that $x \neq i$, and so the weakest precondition can be reduced to the first subformula $x : C$ of the disjunction.

A more glaring example is reducing the weakest precondition of the *add(i R j)* statement with respect to the postcondition $Q = x : (\leq\ n\ R\ C)$ which indicates that there are at most n edges labeled R outgoing from the node x to nodes of concept C. Adding an R-edge between i and

TABLE I. WEAKEST PRECONDITION'S FILTERING FOR THE $add(i : C)$ STATEMENT

Statement	Identified fact	wp	Condition	Precondition
$add(i : C)$	$x : C$	$x : C \vee x = i$	$x \not\simeq i$	$x : C$
	$x : \neg C$	$x : \neg C \wedge x \neq i$	$x \not\simeq i$	$x : \neg C$
	$x : (\leq\ n\ R\ C)$	$(x\ R\ i\ \wedge\ i : \neg C \wedge\ x : (\leq\ (n-1)\ R\ C))$ $\vee\ (x\ \neg R\ i \wedge x : (\leq\ n\ R\ C))$ $\vee\ (i : C \wedge x : (\leq\ n\ R\ C))$ $\vee\ (x : (\leq\ (n-1)\ R\ C))$	$x\ \neg R\ i$	$x : (\leq\ n\ R\ C)$

TABLE II. WEAKEST PRECONDITION'S FILTERING FOR THE $add(i\ R\ j)$ STATEMENT

Statement	Identified fact	wp	Condition	Precondition
$add(i\ R\ j)$	$x\ R\ y$	$(x = i \wedge y = j) \vee x\ R\ y$	$x \not\simeq i \vee y \not\simeq j$	$x\ R\ y$
	$x\ \neg R\ y$	$(x \neq i \vee y \neq j) \wedge (x\ \neg R\ y)$	$x \not\simeq i \vee y \not\simeq j$	$x\ \neg R\ y$
	$x : (\leq\ n\ R\ C)$	$(x = i\ \wedge j : C\ \wedge i\ \neg R\ j\ \wedge\ x : (\leq\ (n-1)\ R\ C))$ $\vee\ (x \neq i\ \wedge\ x : (\leq\ n\ R\ C))$ $\vee\ (j : \neg C \wedge x : (\leq\ n\ R\ C))$ $\vee\ (i\ R\ j \wedge x : (\leq\ n\ R\ C))$ $\vee\ (x : (\leq\ (n-1)\ R\ C))$	$x \not\simeq i \vee j : \neg C$	$x : (\leq\ n\ R\ C)$

j may have a direct impact on Q regarding the concept of j, the existence of a relation between i and j and the equality between i and x. Hence, $wp(add(i\ R\ j), x : (\leq\ n\ R\ C)) =$

$(x = i\ \wedge\ j : C\ \wedge\ i\ \neg R\ j\ \wedge\ x : (\leq\ (n-1)\ R\ C))$
$\vee\ (x \neq i\ \wedge\ x : (\leq\ n\ R\ C))$
$\vee\ (j : \neg C\ \wedge\ x : (\leq\ n\ R\ C))$
$\vee\ (i\ R\ j\ \wedge\ x : (\leq\ n\ R\ C))$
$\vee\ (x : (\leq\ (n-1)\ R\ C))$

Knowing that $x \not\simeq i$ or $j : \neg C$, the first conjunction $x = i \wedge j : C \wedge i\ \neg R\ j \wedge x : (\leq\ (n-1)\ R\ C)$ can be discarded as it will never be satisfied in this case. Furthermore, the whole formula of the wp can be reduced to $x : (\leq\ n\ R\ C)$ according to the second and third conjunction which indicates that the number of restrictions remains unchanged in case one of these two conditions is satisfied.

We illustrated how to reduce the *wp* with respect to a postcondition composed of a single fact. In case of a postcondition consisting of a conjunction of facts, only the facts that manipulate the same concepts and roles given in the statement parameters are identified as a first step. For example, adding an instance to a concept ($add(i : \underline{C})$) results in considering in the given postcondition only the facts that manipulate this concept $(x : \underline{C}, x : \neg\underline{C}, x : (\leq\ n\ R\ \underline{C}))$.

Tables I and II represent the preconditions calculated by our static analyzer for the statement $add(i : C)$ and $add(i\ R\ j)$ respectively. For each statement s, we show in the second column the facts that should be identified in the postcondition to derive a precondition. The third column shows the standard weakest precondition $wp(s, f)$ of the statement s with respect to an identified fact f. To simplify this formula, we present in the fourth column the conditions that allow to discard some conjunctive clauses of the wp. The resulting formula is presented in the last column.

Consider the first row of the Table II. If a fact $x\ R\ y$ is identified within the postcondition during calculation, we look for simplifying $wp(add(i\ R\ j), x\ R\ y) = (x = i \wedge y = j) \vee x\ R\ y$. If the alias calculus asserts that at least one of the conditions $x \not\simeq i$ or $y \not\simeq j$ is true, wp is reduced to $x\ R\ y$.

As observed in Tables I and II, many complex disjunctions in the *wp* can be reduced to only one conjunction on the basis of a condition calculated by the alias calculus or a condition given explicitly in the postcondition. Note that the results of the *delete(i : C)* and *delete(i R j)* statements are similar to the *add* statements.

2) The select *statement:*
So far, the static analysis transforms the predicate Q into a new predicate P regarding statements already presented. However, it operates differently when it comes to the *select* statement where $wp(select\ v\ with\ F, Q) = \forall v\ (F \Rightarrow Q)$. The weakest precondition here involves two formulae that may be complex: F given by the *select*, and the postcondition Q. Consequently, the implication $F \Rightarrow Q$ makes the wp more obscure for the developer. In this case, the static analyzer simplifies the wp by eliminating this implication as further detailed below.

For each conjunction Qi of the postcondition Q, the static analysis isolates first the facts that manipulate the node variables v of the *select* statement. Let Qi_v be the conjunctive formula of these identified facts, and $Qi_{v'}$ the conjunctive formula of the others facts, so that $Qi = Qi_v \wedge Qi_{v'}$. For example, given a formula $Q_1 = x\ R\ y \wedge y : C$ and the statement $select\ x\ with\ x : C$, we have $Q_{1v} = x\ R\ y$ and $Q_{1v'} = y : C$.

Then, the static analysis checks, via our logic formula evaluator, if the implication $\forall v\ (F \Rightarrow Qi_v)$ holds. If so, the precondition $wp(select\ v\ with\ F, Qi) = \forall v\ (F \Rightarrow Qi)$ is reduced to Qi without affecting the validity of the Hoare triple as $Qi \Rightarrow wp(select\ v\ with\ F, Qi)$. Conversely, the non-validity of the implication $\forall v\ (F \Rightarrow Qi_v)$ results in transforming Qi to the predicate $false$ ($\bot$) so that nothing can be concluded

about the transformation correctness. This situation is meant to warn the developer that there are inconsistencies in his transformation between the *select* statement and the predicate formula Q. The two presented cases are given in Table III.

TABLE III. REDUCING THE WP OF THE *select* STATEMENT

Statement	Postcondition	wp	Condition	Precondition
$select\ v\ with\ F$	Qi	$\forall v\ (F \Rightarrow Qi_v)$	$\forall v\ (F \Rightarrow Qi_v)$	Qi
			$\forall v\ (F \not\Rightarrow Qi_v)$	$\bot$

We presented how the static analyzer filters the weakest precondition of a statement with respect to each conjunction $Q_i = \wedge q_j$ of Q where $Q = \vee Q_i$. Hence, the final result of the precondition will be presented as a DNF formula too that expresses different possible alternatives. Each alternative represents a conjunction of facts, constituting a graph that matches a subgraph of the source graph on which the transformation rule is applied.

We filter the weakest precondition by discarding conjunctive clauses that are invalid. This reduction leads to a precondition P stronger than the weakest precondition $wp(S, Q)$. In particular, when two node variables are non-possibly equivalent, a deductive reasoning is carried out by applying equivalence and implication connectives between P and $wp(S, Q)$. We adopt a similar deduction for a node variable belonging to a concept complement and for a role complement. Using these deductions and the well-behaved wp properties, such as distributivity of conjunction and disjunction, we construct the formula P, which satisfies the implication $P \Rightarrow wp(S, Q)$ so that the triple $\{P\}S\{Q\}$ is always correct-by-construction.

VI. EXAMPLE

Using the static analyzer to suggest a precondition formula in the disjunctive normal form, the developer can select the conjunctions that reflect his intention. He can then update his transformation code or refine his specification by injecting into them the facts of the chosen conjunctions.

Consider as an example the transformation code and the postcondition given in Figure 5. The first statement adds a node y to the concept C. The second one adds an R-edge between nodes x and y. The postcondition asserts that x has at most three R-successors to nodes of concept C, and that y belongs to C.

$add(y : C)$;
$add(x\ R\ y)$;
post: $x : (\leq\ 3\ R\ C) \wedge (y : C)$;

Figure 5. Example of an initial code and postcondition

To achieve the given postcondition, a precondition calculus is done in two stages: the first to extract a precondition P with respect to the statement $add(x\ R\ y)$ and the given postcondition, the second to extract a precondition with respect to the statement $add(y : C)$ and P, as $wp(s1; s2, Q) = wp(s1, wp(s2, Q))$. Consequently, the static analyzer extracts seven possible conjunctions as a precondition:

$$x : (\leq 1\ R\ C) \tag{1}$$

$$y : C \wedge x : (\leq 2\ R\ C) \tag{2}$$

$$x\ R\ y \wedge x : (\leq 2\ R\ C) \tag{3}$$

$$x\ \neg R\ y \wedge x : (\leq 2\ R\ C) \tag{4}$$

$$x\ \neg R\ y \wedge y : C \wedge x : (\leq 2\ R\ C) \tag{5}$$

$$x\ R\ y \wedge y : \neg C \wedge x : (\leq 2\ R\ C) \tag{6}$$

$$x\ R\ y \wedge y : C \wedge x : (\leq 3\ R\ C) \tag{7}$$

Each of these conjunctions is a potential precondition that yields a correct Hoare triple. The first formula is the weakest one. It does not take into account neither the concept of y nor the existence of an R-edge between x and y. On the contrary, the other conjunctions are stronger formulae specifying the mentioned properties of x and y. For example, the formula (7) indicates that there exists an R-edge between x and y and that y is of concept C. In this case, both of the two statements of the code have no effects, and so the number of restrictions remains 3 in the fact $x : (\leq 3\ R\ C)$. The various levels of formulae's strength gives the choice to the developer to specify the constraints of rule's applicability in the precondition as much as he wishes to.

Suppose that the developer focuses on the non-existence of an R-edge between x and y before the transformation as it is indicated in the formulae (4) and (5). Thus, he decides to inject the fact $x\ \neg R\ y$ into the transformation by adding the statement $select\ y\ with\ x\ \neg R\ y$ at the beginning of his code. By relaunching the static analyzer, the number of conjunctions extracted decreases from seven to one conjunction which is $x : (\leq\ 2\ R\ C)$. At this point, the developer can choose to put the resulting formula as a precondition as shown in Figure 6.

pre: $x : (\leq\ 2\ R\ C)$;
$select\ y\ with\ x\ \neg R\ y$;
$add(y : C)$;
$add(x\ R\ y)$;
post: $x : (\leq\ 3\ R\ C) \wedge (y : C)$;

Figure 6. The final correct-by-construction triple

In this sense, we help developers to update and annotate their code with specifications based on their intention to achieve finally a correct-by-construction triple.

As described below, our framework guides developers to achieve correct transformation programs. Moreover, it verifies the resulting triple formally using the Small-t$\mathcal{ALC}$ prover. The latter is a formal verification tool that verifies a transformation program with respect to its pre- and postconditions by translating it into Isabelle/HOL logic and generating verification conditions. In case of failure, the prover displays a counter-example which is a model of the precondition that does not satisfy the postcondition when applying the transformation.

VII. RELATED WORK

Most of the logic-based approaches for graph transformations focus on the verification question. Thus, they attempt to encode graph conditions in an appropriate logic that is both expressive and decidable. Like us, Selim et al. [14] proposed a direct verification framework for their transformation language DSLTrans so that no intermediate representation for a specific

proving framework is required. They used symbolic execution to build a finite set of path conditions representing all transformation executions through a formal abstraction relation and thus allow formal properties to be exhaustively proved. Their property language based on graph patterns and propositional logic proposes a limited expressiveness and the property-proving algorithm was presented as a proof-of-concept.

The works in [15] and [16] share with ours some ideas with respect to the assistance in producing a Hoare triple. Given a modeling language with well-formedness constraints and a refactoring specification, Becker et al. [15] uses an invariant checker to detect and report constraint violations via counter-examples and lets developers modify their refactoring iteratively. Similarly to us, Clariso et al. [16] used backward reasoning to automatically synthesize application conditions for model transformation rules. Application conditions are derived from the OCL expression representing the rules's postconditions and the atomic rewriting actions performed by the rule. However, OCL expressions are not really suitable for exploring the graph properties of the underlying model structures. It is thus rather cumbersome when used for verifying complex model transformations.

VIII. Conclusion and Future Work

The distinctive feature of Small-t$\mathcal{ALC}$ is that it uses the same logic $\mathcal{ALCQ}$ to represent graphs, to code a transformation and to reason about graph transformations in a Hoare style. In order to assist users in developing correct transformations, we propose a fine analysis of the weakest precondition to take into account special situations of a program on the basis of an alias calculus. Our approach allows developers to select a precondition to annotate their code according to their intention.

It would be interesting in our framework to automatically infer and test invariant candidates for loop constructs gathered from their corresponding postcondition as proposed in [17]. This attempt is based on the fact that a Small-t$\mathcal{ALC}$ loop often iterates on individuals selected from a logic formula in order to achieve the same property for the transformed elements.

As a complement to a Hoare triple verification, we expect to focus on effects of rules execution in terms of DL reasoning services at the specification rule level. A Small-t$\mathcal{ALC}$ rule execution updates a knowledge base founded upon a finite set of $\mathcal{ALCQ}$ concept inclusions (TBox) and a finite set of $\mathcal{ALCQ}$ concept and role assertions (ABox). This leads to a reasoning problem about a knowledge base consistency embodied by a graph in Small-t$\mathcal{ALC}$ [18].

Acknowledgment

Part of this research has been supported by the *Climt* (Categorical and Logical Methods in Model Transformation) project (ANR-11-BS02-016).

References

[1] A. Habel and K.-H. Pennemann, "Correctness of high-level transformation systems relative to nested conditions," Mathematical. Structures in Comp. Sci., vol. 19, no. 2, Apr. 2009, pp. 245–296.

[2] A. Rensink, "Representing first-order logic using graphs," in Graph Transformations: Second International Conference ICGT,. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 319–335.

[3] B. Courcelle, "Handbook of theoretical computer science (vol. b)." Cambridge, MA, USA: MIT Press, 1990, ch. Graph Rewriting: An Algebraic and Logic Approach, pp. 193–242.

[4] K.-H. Pennemann, "Resolution-like theorem proving for high-level conditions," in Graph Transformations: 4th International Conference, ICGT. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 289–304.

[5] F. Orejas, H. Ehrig, and U. Prange, "A logic of graph constraints," in Fundamental Approaches to Software Engineering: 11th International Conference, FASE. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 179–198.

[6] M. Strecker, "Modeling and verifying graph transformations in proof assistants," Electron. Notes Theor. Comput. Sci., vol. 203, no. 1, Mar. 2008, pp. 135–148.

[7] N. Baklanova, J. H. Brenas, R. Echahed, A. Makhlouf, C. Percebois, M. Strecker, and H. N. Tran, "Coding, executing and verifying graph transformations with small-t$\mathcal{ALCQ}$e," in 7th Int. Workshop on Graph Computation Models (GCM), 2016, URL: http://gcm2016.inf.uni-due.de/ [accessed: 2017-09-25].

[8] A. Makhlouf, H. N. Tran, C. Percebois, and M. Strecker, "Combining dynamic and static analysis to help develop correct graph transformations," in Tests and Proofs: 10th International Conference, TAP. Switzerland: Springer International Publishing, 2016, pp. 183–190.

[9] M. Nagl, "Set theoretic approaches to graph grammars," in Proceedings of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science. London, UK, UK: Springer-Verlag, 1987, pp. 41–54.

[10] M. Krötzsch, F. Simancik, and I. Horrocks, "A description logic primer," arXiv preprint arXiv:1201.4089, 2012, URL: http://arxiv.org/abs/1201.4089 [accessed: 2017-09-25].

[11] M. Schmidt-Schauß and G. Smolka, "Attributive concept descriptions with complements," Artif. Intell., vol. 48, no. 1, Feb. 1991, pp. 1–26.

[12] E. W. Dijkstra and C. S. Scholten, Predicate Calculus and Program Semantics. New York, NY, USA: Springer-Verlag New York, Inc., 1990.

[13] B. Meyer, "Steps towards a theory and calculus of aliasing," Int. J. Software and Informatics, vol. 5, no. 1-2, 2011, pp. 77–115.

[14] G. M. Selim, L. Lúcio, J. R. Cordy, J. Dingel, and B. J. Oakes, "Specification and verification of graph-based model transformation properties," in International Conference on Graph Transformation. Springer, 2014, pp. 113–129.

[15] B. Becker, L. Lambers, J. Dyck, S. Birth, and H. Giese, "Iterative development of consistency-preserving rule-based refactorings," in Theory and Practice of Model Transformations: 4th International Conference, ICMT. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 123–137.

[16] R. Clarisó, J. Cabot, E. Guerra, and J. de Lara, "Backwards reasoning for model transformations," J. Syst. Softw., vol. 116, no. C, Jun. 2016, pp. 113–132.

[17] J. Zhai, H. Wang, and J. Zhao, "Post-condition-directed invariant inference for loops over data structures," in Proceedings of the 2014 IEEE Eighth International Conference on Software Security and Reliability-Companion, ser. SERE-C '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 204–212.

[18] U. Sattler, "Reasoning in description logics: Basics, extensions, and relatives," in Reasoning Web: Third International Summer School. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 154–182.