An Auto-active Approach to Develop Correct Logic-based Graph Transformations

Amani Makhlouf, Christian Percebois, Hanh Nhi Tran

University of Toulouse
IRIT Laboratory, 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 precondition is presented to the developer in terms of alternative formulae, each one specifying a potential matching of the source graph. By choosing some alternatives that correspond to his intention, the developer can interact with an auto-active program verifier, which continuously ensures the correctness of the resulting Hoare triple.**

Keywords–Graph transformation; weakest precondition calculus; static analysis; alias calculus; auto-active program verifier.

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 [2] 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 graph logic [2][3][4]. However, they need to be derived into specific inference rules in order to be proved in a specific theorem-prover that suits them [5][6]. Moreover, this transformation requires the proof of a sound and complete proof system for reasoning in the proposed logic.

The proof of the completeness part of Pennemann's transformation for nested conditions was done by Lambers and Orejas [7] thanks to a tableau reasoning method. The authors introduce nested tableaux, an extension of usual tableaux, to take into account nested conditions. Their proof system requires an $\mathcal{M}$-adhesive category allowing some compatibility of pushouts and pullbacks along $\mathcal{M}$-morphisms. Recently, $\mathcal{M}$-adhesive transformation systems have been generalized to $\mathcal{M},\mathcal{N}$-adhesive ones [8] to cover graph programs dealing with node relabeling as done in GP [9].

Another way to express and reason about graph properties is to directly encode graphs in terms of some existing logic [10]. This solution leads to consider connections between graph constraints and first-order graph formulae. Adopting this approach, we define graphs axiomatically by $\mathcal{ALCQ}$ Description Logic (DL) predicates [11] 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. The suffix is limited to $\mathcal{ALC}$ because this logic is prototypical of 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. This verification can be realized in an auto-active mode [12] where developers annotate their code with specifications to facilitate the automatic verification of the program's correctness. Some verifiers like AutoProof [13] or Dafny [14] use this approach for object-oriented languages.

Aiming to assist developers in writing provably correct transformations [15], we adopt this auto-active approach to provide more interactions with developers to produce a Hoare triple, and thus benefit from their guidance and give them more feedback. In this context, we propose to statically calculate 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 interactively choose a conjunction as a premise that reflects the rule's intention, our approach can filter and reduce some combinatorial issues.

This paper presents an extension of our work originally reported in Proceedings of the Twelfth International Conference on Software Engineering Advances [1]. 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-A, we show in Section IV-B how to reduce some combinatorial issues while ensuring the program correctness

by finely analyzing the weakest precondition. This leads to an auto-active verification of a Hoare triple, which is sketched out in Section V. In Section VI we present our integrated development environment consisting of various tools to assist developers in writing, executing, testing and reasoning about graph transformations. 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 [16]. 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 $\{P\}S\{Q\}$ [10]. 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 [17]. 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 ($\mathcal{ALC}$) [18], 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 $x : (\leq\ 3\ R\ C)$.

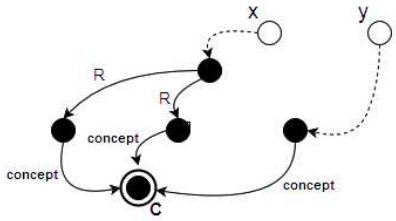


Figure 1. Model satisfying the precondition $x\ \neg R\ y\ \wedge\ y : C \wedge\ x : (\leq\ 3\ R\ C)$

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 is organized into rules. A rule is structured into three parts: a precondition, the transformation code and a postcondition. Small-t$\mathcal{ALC}$ uses the classical control structures to enable sequential composition, branching

and iteration of the atomic statements. Inside a rule, the developer can write an ordered list of statements $s1; s2$; use the statements if or $if-else$ to express a choice between alternatives or the statement $while$ for repeating actions. We illustrate in Figure 3 an example of a transformation rule written in Small-t$\mathcal{ALC}$. The rule r 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.

$$
\begin{array}{l}
rule\ r\ \{ \\
\quad \text{pre: } (a : A) \wedge a : (\geq\ 3\ R\ A); \\
\quad select\ n\ with\ (a\ R\ n) \wedge (n : A); \\
\quad delete(a\ R\ n); \\
\quad delete(a : A); \\
\quad \text{post: } (a : \neg A) \wedge a : (\geq\ 2\ R\ A); \\
\}
\end{array}
$$

Figure 3. Example of a Small-t$\mathcal{ALC}$ rule

The inference rules in Figure 2 define the axiomatic semantics of Small-t$\mathcal{ALC}$. Each rule consists of a premise and a conclusion separated by a horizontal bar. Some ones display prominently the substitutions in the form $P[E\backslash V]$ to indicate that V is replaced by E in P when updating nodes and edges of a graph. For instance, the inference rule ADDC can be interpreted that if the assertion P is valid when substituting the concept $C+i$ for the concept C then P is valid after executing the statement $add(i : C)$. We can say that ADDC is similar to the rule defining the semantics of the assignment $V := E$ in traditional imperative programs. The same observation can be done for DELC defining the semantics of the statement $delete(i : C)$, which modifies the interpretation for C in order to delete the node denoted by i. Rules ADDR and DELR define the semantics of $add(i\ R\ j)$ and $delete(i\ R\ j)$ to add and delete respectively a pair of nodes (i, j) to/from the set of pairs connected by R.

The rules SEQ, IF, IF-ELSE and WHILE give the semantics for the Small-t$\mathcal{ALC}$ control structures. Conditionals in Small-t$\mathcal{ALC}$ are specific DL predicates, which can be considered as Booleans queries on a knowledge base of concept assertions and role assertions. Evaluating a conditional Boolean expression is without side-effects. The statement $select\ v\ with\ F$ is more specific. As we seek and assign individuals when checking the formula F, the inference rule SELECT specifies that any choice for a list v of nodes that satisfies the condition F must provide the postcondition Q assuming the precondition P. If no instance satisfies F, the semantics is blocked.

To shape a sequencing of transformation steps, Small-t$\mathcal{ALC}$ rules will be called inside the $main$ function, which is the entry-point of a Small-t$\mathcal{ALC}$ program. Two separate axiomatic definitions CALL and CALL! are defined for rule invocations inside the $main$ function. Given a rule r and a statement S, we denote $body(r) = S$ in order to ascertain the logical relation of assertions around the execution of the body of r. Note that in Small-t$\mathcal{ALC}$ we only consider rules without parameters. The inference rule CALL refers to a simple rule invocation and says that if we can show that the relation $\{P\}S\{Q\}$ is true where $S = body(r)$, then the relation $\{P\}r\{Q\}$ is true. Thanks to the inference rule CALL!, a body of a rule can be executed many times. Such a call corresponds to an iteration as long as a subgraph matches the rule's precondition formula. To express the semantics of CALL!, we conclude that the current rule call is correct (i.e., $\{P\}r\{Q\}$) if we assume that the previous calls are correct as well (i.e., $\{P\}r\{Q\} \vdash \{P\}S\{Q\}$, which can be interpreted as followed: the sentence $\{P\}S\{Q\}$ is a syntactic consequence ($\vdash$) of the assumption $\{P\}r\{Q\}$).

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 [19]. Each Small-t$\mathcal{ALC}$ statement S is assigned to a predicate transformer yielding an $\mathcal{ALCQ}$ formula $wp(S, Q)$ assuming

$$\frac{}{\{P[C + i \setminus C]\}\ \texttt{add}\ (i : C)\ \{P\}}\ (\text{ADDC}) \qquad \frac{}{\{P[C - i \setminus C]\}\ \texttt{delete}\ (i : C)\ \{P\}}\ (\text{DELC})$$

$$\frac{}{\{P[R + (i, j) \setminus R]\}\ \texttt{add}\ (i\ R\ j)\ \{P\}}\ (\text{ADDR}) \qquad \frac{}{\{P[R - (i, j) \setminus R]\}\ \texttt{delete}\ (i\ R\ j)\ \{P\}}\ (\text{DELR})$$

$$\frac{P\ \wedge \forall v(F \Rightarrow Q)}{\{P\}\ \texttt{select}\ v\ \texttt{with}\ F\ \ \{Q\}}\ (\text{SELECT}) \qquad \frac{\{P\}\ s1\ \{Q\} \quad \{Q\}\ s2\ \{R\}}{\{P\}\ s1; s2\ \{R\}}\ (\text{SEQ})$$

$$\frac{\{P \wedge c\}\ s\ \{Q\} \quad \{P \wedge \neg c\} \Rightarrow \{Q\}}{\{P\}\ \texttt{if}\ c\ \texttt{then}\ s\ \{Q\}}\ (\text{IF}) \qquad \frac{\{P \wedge c\}\ s1\ \{Q\} \quad \{P \wedge \neg c\}\ s2\ \{Q\}}{\{P\}\ \texttt{if}\ c\ \texttt{then}\ s1\ \texttt{else}\ s2\ \{Q\}}\ (\text{IF-ELSE})$$

$$\frac{\{P \wedge c\}\ s\ \{P\}}{\{P\}\ \texttt{while}\ c\ \texttt{do}\ s\ \{P \wedge \neg c\}}\ (\text{WHILE})$$

$$\frac{\{P\}\ S\ \{Q\} \quad body(r) = S}{\{P\}\ r\ \{Q\}}\ (\text{CALL}) \qquad \frac{\{P\}\ r\ \{Q\} \vdash \{P\}\ S\ \{Q\} \quad body(r) = S}{\{P\}\ r\ \{Q\}}\ (\text{CALL!})$$

Figure 2. Axiomatic semantics of Small-t$\mathcal{ALC}$

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 4.

$$
\begin{array}{l}
wp(add\ (i:C), Q) = Q[C + i \backslash C] \\
wp(delete\ (i:C), Q) = Q[C - i \backslash C] \\
wp(add\ (i\ R\ j), Q) = Q[R + (i,j) \backslash R] \\
wp(delete\ (i\ R\ j), Q) = Q[R - (i,j) \backslash R] \\
wp(select\ v\ with\ F, Q) = \forall v (F \Rightarrow Q) \\
wp(s1; s2,\ Q) = wp(s1, wp(s2,\ Q)) \\
wp(if\ c\ then\ s1, Q) = (c \Rightarrow wp(s1, Q)) \wedge (\neg c \Rightarrow Q) \\
wp(if\ c\ then\ s1\ else\ s2, Q) = (c \Rightarrow wp(s1, Q)) \\
\qquad\qquad\qquad\qquad\qquad \wedge (\neg c \Rightarrow wp(s2, Q)) \\
wp(\{inv\}\ while\ c\ do\ s, Q) = inv
\end{array}
$$

Figure 4. Small-t$\mathcal{ALC}$ weakest preconditions

Small-t$\mathcal{ALC}$ axiomatic semantics for add and $delete$ statements introduces substitutions, which build formulae that no longer belong to the $\mathcal{ALCQ}$ logic: $C + i$ in ADDC, $C - i$ in DELC, $R + (i, j)$ in ADDR and $R - (i, j)$ in DELR do not represent concepts and roles anymore. This means that the weakest precondition calculus computes predicates, which are not closed under substitutions with respect to $\mathcal{ALCQ}$ [20]. To resolve this situation, substitutions are considered as constructors for concepts and roles and should be eliminated by predicate transformers. For instance:

$$
\begin{aligned}
wp(add(i:C), x:C) &= x:C[C + i \backslash C] \\
&= x:(C + i) \\
&= x:C \vee x = i.
\end{aligned}
$$

IV. Specification extraction

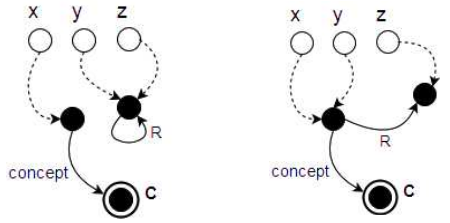
The conventional precondition calculus presented in the previous section does not take into account particular situations of a transformation program and thus may result in a complex precondition. In this section, we look at how the precondition's formula can be improved to be more specific and simple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

A. 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 [21].

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 5 shows two potential models satisfying the formula $x : C\ \wedge\ y\ R\ z$. In Figure 5a, y and z refer to the same node. In 5b, y and z are different but x and y are combined.

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



(a) $y = z, x \neq y$ (b) $y \neq z, x = y$

Figure 5. Example of models satisfying the formula $x : C\ \wedge\ y\ R\ z$

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.

B. 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 IV-B1. Regarding the *select* statement, *wp* is reduced differently as presented later in Section IV-B2.

1) The 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 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, $wp(add(i\ R\ j), x : (\leq\ n\ R\ C)) =$

$$
\begin{aligned}
&(x = i\ \wedge\ j : C\ \wedge\ i\ \neg R\ j\ \wedge\ x : (\leq\ (n-1)\ R\ C))\\
&\vee\ (x \neq i\ \wedge\ x : (\leq\ n\ R\ C))\\
&\vee\ (j : \neg C\ \wedge\ x : (\leq\ n\ R\ C))\\
&\vee\ (i\ R\ j\ \wedge\ x : (\leq\ n\ R\ C))\\
&\vee\ (x : (\leq\ (n-1)\ R\ C))
\end{aligned}
$$

Knowing that $x \not\equiv i$ 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 conjunctions, which indicate 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 III 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 III. 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 $x \not\equiv i$ or $y \not\equiv j$ is true, wp is reduced to $x\ R\ y$.

As observed in Tables I and III, 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 and are respectively described in Tables II and IV.

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 $false$ ($\perp$) 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. These cases are given in Table V for a conjunctive formula Qi.

To clarify the idea, consider an example of a code consisting of the statement $select\ i\ with\ i : C$. First, suppose that the given postcondition is $Q = (i : C) \wedge (j : C)$. So we have $F = i : C$ and $Q_v = i : C$ as $i : C$ is the only fact that manipulates the selected node variable i in the postcondition. The implication $i : C \Rightarrow i : C$ is obviously true, so the precondition is reduced to Q.
Now consider another postcondition for the same code: $Q = (i\ R\ j) \wedge (j\ :\ C)$. In this case, the implication between $F = i : C$ and $Q_v = i\ R\ j$ does not hold. Hence, the static analyzer returns $false$ as a precondition, and so this transformation can not be executed because no state can satisfy the precondition as mentioned above.

3) The other statements:
We presented how the static analyzer filters the weakest precondition of an atomic statement with respect to each conjunction $Q_i = \wedge q_j$ of Q where $Q = \vee Q_i$. The precondition P of a sequence of statements $s1; s2$ is computed conventionally as it is presented above ($wp(s1, wp(s2, Q))$). Similarly, the extracted precondition of the $if\ c\ then\ s1\ else\ s2$ statement is its weakest precondition transformed into a DNF formula: $(c \wedge wp(s1, Q)) \vee\ (\neg c \wedge wp(s2, Q))$.

Apart from loops, weakest preconditions can be computed automatically as it is presented in this section. How-

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 $delete(i : C)$ STATEMENT

Statement	Identified fact	wp	Condition	Precondition
$delete(i : C)$	$x : \neg C$	$x : \neg C \vee x = i$	$x \not\simeq i$	$x : \neg C$
	$x : C$	$x : C \wedge x \neq i$	$x \not\simeq i$	$x : \neg C$
	$x : (\geq\ n\ r\ C)$	$(x\ r\ i\ \wedge i : C \wedge x : (\geq\ n+1\ r\ C))$ $\vee(x\ \neg r\ i \wedge x : (\geq\ n\ r\ C))$ $\vee(i : \neg C \wedge x : (\geq\ n\ r\ C))$ $\vee(x : (\geq\ n+1\ r\ C))$	$x\ \neg r\ y$	$x : (\geq n\ r\ C)$

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

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

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

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

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

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

ever, it is more complicated when it comes to the $while$ statement. In fact, $while\ c\ do\ s$ is semantically equivalent to $if\ c\ then\ \{s; while\ c\ do\ s\}\ else\ skip$, then its weakest precondition is a recursive equation of the form: $wp(while\ c\ do\ s, Q) = (c \Rightarrow wp(s, wp(while\ c\ do\ s, Q))) \vee (\neg c\ \Rightarrow\ Q)$. For this reason, the weakest precondition is approximated by a verification condition, which considers the invariant of the $while$ statement as it is shown in the

TABLE VI. VERIFICATION CONDITION OF THE *while* STATEMENT

Statement	Postcondition	Verification condition	Precondition
$\{inv\}\ while\ c\ do\ s$	Q	$(inv \wedge \neg c \Rightarrow Q)$ $\wedge\ (inv \wedge c \Rightarrow wp(s, inv))$ $\wedge\ inv$	$DNF(inv)$
		else	$\bot$

third column of the Table VI. Basing on the given invariant inv, our logic formula evaluator checks whether all of the implications presented in the third column hold. In this case, the precondition of the loop is simplified by inv. Otherwise, Q is transformed into the predicate $false$ ($\bot$) as the given invariant does not satisfy the loop verification condition.

The final result of the precondition will be presented as a DNF formula 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.

V. AUTO-ACTIVE VERIFIER

Different techniques pertaining to formal program verification exist. An automatic verification requires no interaction with developers; a solver performs autonomously the formal verification of a program. However, it provides weak feedback in case of failure. Conversely, an interactive verification requires an expert to guide the proof assistant through its manipulations to perform the verification.

Since our purpose is to help a novice developer to achieve a correct transformation, we adopt an auto-active verification approach, which lies between automatic and interactive verification [12]. The auto-active approach expects a developer to annotate his code with specifications, then the verification will be done automatically. The process can be repeated in many iterations until the program is proved.

The auto-active approach has two main advantages: on the one hand it promotes an incremental development with rich feedback at each step, on the other hand it bridges the gap between a non-expert developer and a formal verification tool. This technique is used by AutoProof, part of EVE (Eiffel Verification Environment) to verify Eiffel programs [13], and by Dafny to verify functional correctness of Dafny programs [14].

In the next, an example of an incremental development for constructing a correct transformation program with the assistance of our auto-active verifier will be described.

A. Incremental Developement

Algorithm 1 shows an example of an incremental development using the auto-active approach to help developers construct specifications and refine the transformation triple. The idea is alternating the intervention of the static analyzer with the developer's in a progressive process until achieving finally a correct-by-construction triple.

Since the static analyzer calculates many conjunctions as preconditions with respect to a code and a postcondition as explained in Section IV, the number of extracted preconditions may be sizable. In this sense, using an interactive process to enable developers express more precisely their intention can help reduce the number of the extracted conjunctions given by the static analyzer.

```
Input: Code, Post
Output: Pre
repeat
    /* by Static Analyzer                          */
    E_Pre = extractedPre(Code, Post);
    /* by Developer                                */
    S_Pre = selectConjunctions(E_Pre);
    if isCorespondToIntention(S_Pre) then
        Pre = selectPrecondition(S_Pre);
    else
        refineCodeAndPost(Code, Post, S_Pre);

until isValid(Pre, Code, Post);
```

Algorithm 1: Example of an incremental development with Auto-active Verifier

First, the static analyzer suggests a precondition formula in the disjunctive normal form. Then the developer selects some of the suggested conjunctions that reflect his intention. If he is satisfied with the selection, the developer can take directly the chosen conjunctions as the final precondition and terminate the iteration. If not, the developer can refine his code or/and his postcondition to clarify his intention then starts a new iteration. In general, the developer can update his transformation code or refine his specification by injecting into them the facts of the chosen conjunctions. In this way, the transformation program is incrementally enhanced based on the developer's intention.

B. Example

In order to illustrate the application of our auto-active verifier in the proposed incremental process, we present in this section an example based on the development of an application simulating the activities of a hospital [22].

In this application, we consider $Patients$ and $Doctors$ in a hospital composed of several $Departments$. Each

doctor works in (denoted by $worksIn$) a department and $treats$ patients. Each department is directed by (denoted by $directedBy$) a head who is one among its doctors. Each department is in charge of (denoted by $inChargeOf$) some diseases and registers the patients (denoted by $registers$) who $suffer$ one of those diseases. A hospitalized patient has a reference doctor (denoted by $refDr$) and is allocated (denoted by $allocated$) a $Room$.

Supposed that a graph is used to represent different concepts and individuals in the hospital. The following scenarios illustrate a development process of a novice developer to write a graph transformation for updating the hospital's status when patients arrive. We suppose that the developer does not write immediately a correct Hoare triple, but needs many iterations to refine his program with the static analyzer's help.

Rule *assignDoctor*

At the first step, the developer wants to write the rule $assignDoctor$ to assign a doctor dr to treating a patient p. He writes thus the rule's code to add the relations $treats$ and $refDr$ between the patient p and the doctor dr.

The chosen doctor dr has to work in the department dep where the patient p is registered. Moreover, a doctor who is head of a department can not treat more than 3 patients simultaneously. Considering these constraints, the developer writes the first version of his rule as shown in Figure 6.

$rule\ assignDoctor\ \{$
$add(dr\ treats\ p);$
$add(p\ refDr\ dr);$
post: $p : Patient \wedge dr : Doctor$
$\wedge\ dep : Department$
$\wedge\ dep\ registers\ p \wedge dr\ worksIn\ dep$
$\wedge\ head : Doctor \wedge dep\ directedBy\ head$
$\wedge\ head : (\leq 3\ treats\ Patient)$
$\wedge\ dr\ treats\ p \wedge p : (\leq 1\ refDr\ Doctor);$
$\}$

Figure 6. Rule $assignDoctor$ - First version

To complete the rule $assignDoctor$, the developer uses the static analyzer to extract a precondition. From the program in Figure 6, the static analyzer proposes the twelve following conjunctions as possible preconditions:

1) $p : Patient \wedge dr : Doctor \wedge dep : Department$
$\wedge\ dr\ worksIn\ dep \wedge dep\ registers\ p$
$\wedge\ head : Doctor \wedge dep\ directedBy\ head$
$\wedge\ head : (\leq 2\ treats\ Patient)$
$\wedge\ p : (\leq 0\ refDr\ Doctor)$

2) $p : Patient \wedge dr : Doctor \wedge dep : Department$
$\wedge\ dr\ worksIn\ dep \wedge dep\ registers\ p$
$\wedge\ head : Doctor \wedge dep\ directedBy\ head$
$\wedge\ head : (\leq 2\ treats\ Patient)$
$\wedge\ p\ !refDr\ dr$
$\wedge\ p : (\leq 0\ refDr\ Doctor)$

3) $p : Patient \wedge dr : Doctor \wedge dep : Department$
$\wedge\ dr\ worksIn\ dep \wedge dep\ registers\ p$
$\wedge\ head : Doctor \wedge dep\ directedBy\ head$
$\wedge\ head : (\leq 2\ treats\ Patient)$
$\wedge\ p\ refDr\ dr$
$\wedge\ p : (\leq 1\ refDr\ Doctor)$

4) $p : Patient \wedge dr : Doctor \wedge dep : Department$
$\wedge\ dr\ worksIn\ dep \wedge dep\ registers\ p$
$\wedge\ head : Doctor \wedge dep\ directedBy\ head$
$\wedge\ dr\ \neq head$
$\wedge\ head : (\leq 3\ treats\ Patient)$
$\wedge\ p : (\leq 0\ refDr\ Doctor)$

5) $p : Patient \wedge dr : Doctor \wedge dep : Department$
$\wedge\ dr\ worksIn\ dep \wedge dep\ registers\ p$
$\wedge\ head : Doctor \wedge dep\ directedBy\ head$
$\wedge\ dr\ treats\ p$
$\wedge\ head : (\leq 3\ treats\ Patient)$
$\wedge\ p : (\leq 0\ refDr\ Doctor)$

6) $p : Patient \wedge dr : Doctor \wedge dep : Department$
$\wedge\ dr\ worksIn\ dep \wedge dep\ registers\ p$
$\wedge\ head : Doctor \wedge dep\ directedBy\ head$
$\wedge\ dr\ !treats\ p$
$\wedge\ dr\ =\ head$
$\wedge\ head : (\leq 2\ treats\ Patient)$
$\wedge\ p : (\leq 0\ refDr\ Doctor)$

7) $p : Patient \wedge dr : Doctor \wedge dep : Department$
$\wedge\ dr\ worksIn\ dep \wedge dep\ registers\ p$
$\wedge\ head : Doctor \wedge dep\ directedBy\ head$
$\wedge\ dr\ treats\ p$
$\wedge\ p\ !refDr\ dr$
$\wedge\ head : (\leq 3\ treats\ Patient)$
$\wedge\ p : (\leq 0\ refDr\ Doctor)$

8) $p : Patient \wedge dr : Doctor \wedge dep : Department$
$\wedge\ dr\ worksIn\ dep \wedge dep\ registers\ p$
$\wedge\ head : Doctor \wedge dep\ directedBy\ head$
$\wedge\ dr\ \neq head$
$\wedge\ p\ !refDr\ dr$
$\wedge\ head : (\leq 3\ treats\ Patient)$
$\wedge\ p : (\leq 0\ refDr\ Doctor)$

9) $p : Patient \wedge dr : Doctor \wedge dep : Department$
$\wedge\ dr\ worksIn\ dep \wedge dep\ registers\ p$
$\wedge\ head : Doctor \wedge dep\ directedBy\ head$
$\wedge\ dr\ \neq head$
$\wedge\ p\ refDr\ dr$
$\wedge\ head : (\leq 3\ treats\ Patient)$
$\wedge\ p : (\leq 1\ refDr\ Doctor)$

10) $p : Patient \wedge dr : Doctor \wedge dep : Department$
$\wedge\ dr\ worksIn\ dep \wedge dep\ registers\ p$
$\wedge\ head : Doctor \wedge dep\ directedBy\ head$
$\wedge\ dr\ treats\ p$
$\wedge\ p\ refDr\ dr$
$\wedge\ head : (\leq 3\ treats\ Patient)$
$\wedge\ p : (\leq 1\ refDr\ Doctor)$

11) $p : Patient \wedge dr : Doctor \wedge dep : Department$
$\wedge\ dr\ worksIn\ dep \wedge dep\ registers\ p$
$\wedge\ head : Doctor \wedge dep\ directedBy\ head$
$\wedge\ dr\ !treats\ p \wedge p\ refDr\ dr$
$\wedge\ dr\ =\ head$
$\wedge\ head : (\leq 2\ treats\ Patient)$
$\wedge\ p : (\leq 1\ refDr\ Doctor)$

12) $p : Patient \wedge dr : Doctor \wedge dep : Department$
$\wedge\, dr\ worksIn\ dep \wedge dep\ registers\ p$
$\wedge\, head : Doctor \wedge dep\ directedBy\ head$
$\wedge\, dr\ !treats\ p$
$\wedge\, p\ !refDr\ dr$
$\wedge\, dr\ =\ head$
$\wedge\, head : (\leq 2\ treats\ Patient)$
$\wedge\, p : (\leq 0\ refDr\ Doctor)$

Among these conjunctions, eight formulae contain the fact $p : (\leq 0\ refDr\ Doctor)$ stating that the examined patient p does not have yet a reference doctor. This fact reflects the developer's precondition regarding a patient, thus first he selects the formulae 1, 2, 4, 5, 6, 7, 8, 12 as possible preconditions.

Now, to treat a patient p, the developer wants to choose a doctor who is not in charge of the department. To clarify his intention and help the static analyzer reduce the number of extracted conjunctions, the developer injects a *select* statement into his code in order to choose the instance dr that is distinct from the instance $head$. Figure 7 shows the modified body of the rule $assignDoctor$.

```
select dr with
        dr : Doctor ∧ dr worksIn dep ∧ dr ≠ head;
add(dr treats p);
add(p refDr dr);
```

Figure 7. Refined code of the rule $assignDoctor$

Using again the static analyzer with this new code, only one precondition is extracted. It is taken directly by the developer as the precondition of his rule. The final complete rule $assignDoctor$ is shown in Figure 8.

```
rule assignDoctor {
  pre: p : Patient ∧ dep : Department
       ∧ dep registers p ∧ head : Doctor
       ∧ dep directedBy head
       ∧ head : (≤ 3 treats Patient)
       ∧ p : (≤ 0 refDr Doctor);

  select dr with
       dr : Doctor ∧ dr worksIn dep ∧ dr ≠ head;
  add(dr treats p);
  add(p refDr dr);

  post: p : Patient ∧ dr : Doctor
       ∧ dep : Department
       ∧ dep registers p ∧ dr worksIn dep
       ∧ head : Doctor ∧ dep directedBy head
       ∧ head : (≤ 3 treats Patient)
       ∧ dr treats p ∧ p : (≤ 1 refDr Doctor);
}
```

Figure 8. Rule $assignDoctor$ - Final version

Rule *registerPatient*

The rule $assignDoctor$ assumes that the examined patient is already registered in a department. This means that before calling $assignDoctor$ another rule is needed to register the patient. For this purpose, the developer now writes the rule $registerPatient$ to receive and place a patient in an appropriate department according to his illness.

Because $registerPatient$ precedes $assignDoctor$, the postcondition of the rule $registerPatient$ must include the facts regarding $Patient$ and $Department$ concepts in the precondition of the rule $assignDoctor$, i.e., the facts $p : Patient$, $dep : Department$ and $dep\ registers\ p$.

To make sure that a patient is registered in a department that is in charge of his disease, the developer adds to the $registerPatient$'s postcondition the following two facts: $p\ suffers\ disease$ and $dep\ inChargeOf\ disease$.

Since the person p was not considered as a patient before being registered, in the rule's code, the developer writes the statements to declare p belongs to the concept $Patient$ and to create a relationship between the department dep and the patient p. Figure 9 shows the first version of the rule $registerPatient$.

```
rule registerPatient {
  add(p : Patient);
  add(dep registers p);

  post: p : Patient ∧ dep : Department
       ∧ dep registers p
       ∧ p suffers disease
       ∧ dep inChargeOf disease;
}
```

Figure 9. Rule $registerPatient$ - First version

Taking as input the program in Figure 9, the static analyzer extracts the following precondition:

$dep : Department \wedge dep\ inChargeOf\ disease$
$\wedge\, p\ suffers\ disease$

The developer takes the extracted precondition and add to it the fact $p : Person \cap !Patient$ specifying that before applying the rule p is of concept $Person$ and not of type $Patient$. Figure 10 shows the final version of the rule $registerPatient$.

```
rule registerPatient {
  pre: dep : Department
       ∧ dep inChargeOf disease
       ∧ p : Person ∩ !Patient ∧ p suffers disease;

  add(p : Patient);
  add(dep registers p);

  post: p : Patient ∧ dep : Department
       ∧ dep inChargeOf disease
       ∧ p suffers disease ∧ dep registers p;
}
```

Figure 10. Rule $registerPatient$ - Final version

With the help of the static analyzer, the developer can develop two correct-by-construction rules as we've just presented. Now, he can write the main program of his application

as followed to manage all people who have arrived at the hospital as shown in 11. In this code, the operation ! allows applying each rule iteratively as long as possible.

```
main {
    receivePatient!;
    assignDoctor!;
}
```

Figure 11. Program $ManagePatients$

VI. Integrated Development Environment

Aiming at integrating various tools to assist in developing and reasoning about graph transformations, the static analyzer is part of an experimental environment that provides the assistance in coding, executing and verifying transformations written in Small-t$\mathcal{ALC}$ [11].

Figure 12 shows the big picture of our framework and its components. Each component provides a specific support for Small-t$\mathcal{ALC}$ programs: the compiler translates a Small-t$\mathcal{ALC}$ program to an executable code; the dynamic analyzer examines the behavior of a running program; the static analyzer helps achieve correct transformations and the prover verifies the correctness of programs. The development of each component is based on an implementation of Small-t$\mathcal{ALC}$'s semantics in an appropriate foundation.

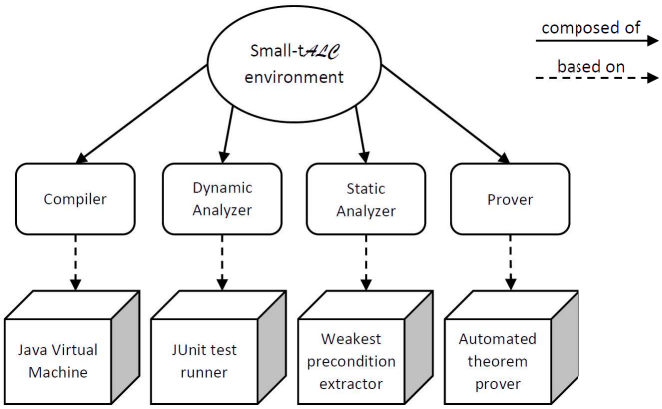


Figure 12. Overview of the Small-t$\mathcal{ALC}$ environment

The compiler, developed using the compiler generator Coco/R, produce the byte code of a Small-t$\mathcal{ALC}$ program for the Java Virtual Machine. This executable code transforms a source graph into a target graph.

The dynamic analyzer can be used to find inconsistencies between a transformation code and its specifications by executing the transformation generated by the compiler then applying automated tests on the target graph. The test cases are generated from the postcondition using our Small-t$\mathcal{ALC}$ testing library, which is based on JUnit assertions. The input graph can be generated automatically from the precondition or can be given by the user.

The prover 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. These verification conditions are passed to an automated theorem prover, which can then formally prove the correctness of the code. 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.

In the following we illustrate the use of different tools in our integrated development environment via a scenario of writing a third rule in the application to manage a hospital.

Rule *allocateRoom*

Suppose that the developer now wants to write a rule to allocate a hospital's room r ($r : Room$) to a patient p ($p : Patient$) who is already assigned to a doctor dr ($dr\ treats\ p$) but is not allocated a room yet ($p : (\leq 0\ allocated\ Room)$).

The allocated room must be available ($r : AvailableRoom$) and in the department dep where p is registered ($p\ allocated\ r \wedge r\ isIn\ dep \wedge dep\ registers\ p$). The developer then writes the first version of the rule $allocateRoom$ as shown in Figure 13.

```
rule allocateRoom {
  pre: p : Patient ∧ r : Room ∩ AvailableRoom
       ∧ r isIn dep ∧ dep registers p
       ∧ dr treats p
       ∧ p : (≤ 0 allocated Room);

  add(p allocated r);
  delete(r : AvailableRoom);

  post: p : Patient ∧ r : Room ∩ !AvailableRoom
       ∧ p allocated r
       ∧ p : (≤ 0 allocated Room);
}
```

Figure 13. Rule $allocateRoom$ - First version

When submitting the program in Figure 13 to the prover, the proof fails and a counter-example is given as shown in Figure 14.

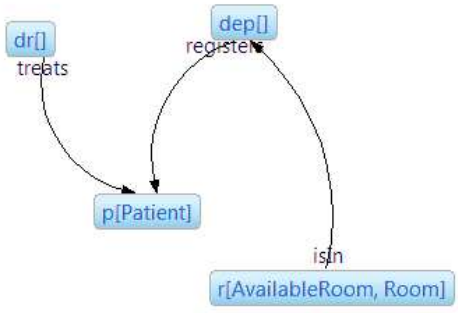


Figure 14. A counter-example of the rule $allocateRoom$

Having difficulties to locate the error from the counter-example given by the prover, the developer uses the dynamic analyzer to examine his program by running tests. From the postcondition, the dynamic analyzer generates the following test cases:

1) $assertExistNode(graph, p, AtomicConcept(Patient))$;

2) $assertExistNode(graph, r, AtomicConcept(Room))$;

3) $assertNotExistNode(graph, r, AtomicConcept(AvailableRoom))$;

4) $assertExistEdge(graph, p, allocated, r)$;

5) $assertAtMostNumberEdges(graph, p, allocated, AtomicConcept(Room), 0)$;

When applying the rule $allocatedRoom$ on a generated graph source from the precondition then executing the tests on the target graph , the first four test cases succeed but the fifth test fails. The failure of this $assertAtMostNumberEdges$ test means that the target graph does not respect the condition "there is at most 0 relation $allocated$ between p and the individuals of the type $Room$". The error is therefore from the fact $p : (\leq 0\ allocated\ Room)$ in the given postcondition.

Thanks to this diagnosis, the developer notices that it is necessary to increase the number of restrictions in the erroneous fact as follows : $p : (\leq 1\ allocated\ Room)$. Now he resubmits the modified program to the prover, which indicates that the triplet is correct.

Our Small-t$\mathcal{ALC}$ environment provides different levels of assistance in writing both rule's statements and their specifications. We choose a testing framework as infrastructure of the dynamic analyzer for providing immediate feedback and detailed diagnostics to help correct rule code with respect to given specifications. On the other hand, the static analyzer helps in an auto-active approach construct a correct-by-construction transformation given a postcondition and partially a code. Consequently, both produce a valid Hoare triple of a rule to be eventually proved formally by the prover.

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. The work in [23] expresses invariants in Computation Tree Logic (CTL). Becker et al. [24] encoded graph patterns as first-order predicates and created symbolic representations for possible violations of the rule's properties. Inaba et al. [25] verified graph transformations against the graph structural constraints in Monadic Second-Order logic (MSO). Calvanese et al. [26] studied the static verification for evolving graph databases where the integrity constraints are expressed in DL $\mathcal{ALCHOIQ}$. Brenas et al. [27] provided a decidable logic based on the DL $\mathcal{SROIQ}$ for reasoning on program processing structures defined as graphs. Like us, Selim et al. [28] 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 [24] and [29] share with ours some ideas with respect to the assistance in producing a Hoare triple. Becker et al. [24] proposed an iterative development of consistency-preserving refactorings, which are specified in a rule-based manner and rely on a graph-transformation formalization. Given a modeling language with well-formedness constraints and a refactoring specification, Becker et al. [24] use 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. [29] 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. To obtain a higher abstraction and benefit the formal verification of graph rewriting systems, recently, there are some graph-based approaches proposing the translations of a set of OCL expressions to graph patterns [30] or nested graph constraints [31].

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 auto-active 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 [32]. 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 are working 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$\mathcal{ALC}$ [33].

Acknowledgment

Part of this research has been supported by the *Climt* (Categorical and Logical Methods in Model Transformation) project (ANR-11-BS02-016). We are grateful for Martin Strecker for the interesting discussions.

References

[1] A. Makhlouf, C. Percebois, and H. N. Tran, "A Precondition Calculus for Correct-by-Construction Graph Transformations," in International Conference on Software Engineering Advances (ICSEA), Athens (Greece), 08/10/2017-12/10/2017. http://www.iaria.org/: IARIA, 2017, pp. 172–177.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] L. Lambers and F. Orejas, "Tableau-based reasoning for graph properties," in Graph Transformation: 7th International Conference, ICGT. Cham: Springer International Publishing, 2014, pp. 17–32.

[8] A. Habel and D. Plump, "M,N-adhesive transformation systems," in Proceedings of the 6th International Conference on Graph Transformations, ser. ICGT'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 218–233.

[9] D. Plump, "The graph programming language gp," in Algebraic Informatics: Third International Conference, CAI. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 99–122.

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

[11] 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: 2018-05-06].

[12] K. R. M. Leino and N. Polikarpova, "Verified calculations," in Verified Software: Theories, Tools, Experiments: 5th International Conference, VSTTE. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 170–190.

[13] C. A. Furia, M. Nordio, N. Polikarpova, and J. Tschannen, "Autoproof: auto-active functional verification of object-oriented programs," International Journal on Software Tools for Technology Transfer, 2016, pp. 1–20.

[14] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, ser. LPAR'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 348–370.

[15] 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.

[16] 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.

[17] 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: 2018-05-06].

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

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

[20] 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, URL: http://ceur-ws.org/Vol-1577/paper_47.pdf [accessed: 2018-05-06].

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

[22] J. H. Brenas, R. Echahed, and M. Strecker, "Ensuring correctness of model transformations while remaining decidable," in Theoretical Aspects of Computing – ICTAC 2016, A. Sampaio and F. Wang, Eds. Cham: Springer International Publishing, 2016, pp. 315–332.

[23] Z. Langari and R. Trefler, "Application of graph transformation in verification of dynamic systems," in Integrated Formal Methods: 7th International Conference, IFM. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 261–276.

[24] 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.

[25] K. Inaba, S. Hidaka, Z. Hu, H. Kato, and K. Nakano, "Graph-transformation verification using monadic second-order logic," in Proceeding of the 13th International ACM SIGPLAN Symposium on Symposium on Principles and Practice of Declarative Programming. ACM Press, Jul. 2011.

[26] D. Calvanese, M. Ortiz, and M. Simkus, "Evolving graph databases under description logic constraints," in Proc. of the 26th Int. Workshop on Description Logics (DL 2013), 2013.

[27] J. H. Brenas, R. Echahed, and M. Strecker, "A hoare-like calculus using the SROIQ σ logic on transformations of graphs," in Theoretical Computer Science - 8th IFIP TC 1/WG 2.2 International Conference, TCS, 2014, pp. 164–178.

[28] 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.

[29] 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.

[30] G. Bergmann, "Translating ocl to graph patterns," in Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS. Cham: Springer International Publishing, 2014, pp. 670–686.

[31] H. Radke, T. Arendt, J. S. Becker, A. Habel, and G. Taentzer, "Translating essential ocl invariants to nested graph constraints focusing on set operations," in Graph Transformation: 8th International Conference, ICGT. Cham: Springer International Publishing, 2015, pp. 155–170.

[32] 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.

[33] 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.