Coding, Executing and Verifying Graph Transformations with small-tALCQe

Nadezhda Baklanova, Jon Haël Brenas, Amani Makhlouf, Christian Percebois, Martin Strecker, Hanh Nhi Tran

Coding, Executing and Verifying Graph Transformations with small-t$\mathcal{ALCQ}$e

Nadezhda Baklanova[1], Jon Haël Brenas[2], Amani Makhlouf[3], Christian Percebois[3], Martin Strecker[3], Hanh Nhi Tran[3]

[1] Systerel, Aix-en-Provence, France
[2] CNRS and University of Grenoble, France
[3] IRIT, University of Toulouse, France

Abstract. This paper gives an overview of small-t$\mathcal{ALCQ}$e, an experimental programming environment for a graph transformation language that is based on the $\mathcal{ALCQ}$ description logic. small-t$\mathcal{ALCQ}$e not only allows developers coding and executing graph transformations but also assists them in analyzing and verifying their codes. We describe the components that make up small-t$\mathcal{ALCQ}$e: the transformation language itself, the compiler for generating executable transformations, the code analyzers and the prover for reasoning about transformations. All of them interact under the hood of an Eclipse user interface to provide different levels of assistance for achieving correct graph transformations.

Keywords: Graph Transformations, Software Analysis, Program Verification, Program Testing, Counterexample Generation

1 Introduction

Transformations of graph structures in computer science appear in a rather pure form as model transformations or modifications of graph databases, and in a more disguised form in programs that manipulate pointer structures. In many cases, it is necessary to associate a notion of correctness with the transformation, such as preservation of coherence of a model *wrt.* a meta model or a database *wrt.* a database schema. Many theoretical studies have been done, often separately, on executing and verifying graph transformations. However, there have been few works offering practical assistance throughout the development to implement correct graph transformations. Thus, writing graph transformations and ensuring their correctness is still challenging, especially for real life applications.

Motivated by this lack, we aim at integrating various tools to assist both developing and reasoning about graph transformations. This paper presents an experimental environment that provides the assistance in coding, executing and verifying transformations written in small-t$\mathcal{ALCQ}$, a graph transformation language based on the $\mathcal{ALCQ}$ description logic.

For reasoning about transformations, two principal methods are employed: automated proofs and code analysis. These two methods are complementary:

proofs provide an infallible evidence correctness, but are limited in expressiveness (at least when considering full automation, as we do), whereas analysis can deal with a wider spectrum of language features and more expressive assertions, but do not provide full coverage. Proofs and static analysis are done statically, whereas dynamic analysis need an execution mechanism, which is also provided in our environment.

We focus on pure graph transformations and do not deal with the transformations that are combined with manipulations of other data types, which are more of theoretical interest than practically feasible. An operational prototype of the presented framework has been implemented and is available for download.[4].

The paper begins with a brief outline of the syntax and semantics of the graph transformation language small-t$\mathcal{ALCQ}$ in Section 2. Then we dive into the description of the tools constituting our framework: the compiler for producing executable code (Section 3.1), the dynamic and static analyzers (Sections 3.2 and 3.3) for helping to construct correct code and deriving appropriate program specifications, the prover (Section 3.4) for verifying the correctness of programs. The possible interactions of these components during the development are sketched out in Section 4. We give some discussion on related works in (Section 5) and wrap up the paper with possible improvements and extensions in (Section 6).

2 Graph Transformation Language

small-t$\mathcal{ALCQ}$ language is an imperative-style programming language based on the $\mathcal{ALCQ}$ description logic [1], which is the logical counterpart of knowledge representation formalisms such as OWL [2] and modeling frameworks such as UML [3]. The distinctive characteristics of this graph transformation language are a precisely and formally defined semantics and the tight integration of logical aspects with the intended execution mechanism, with the overall aim to obtain a decidable calculus for reasoning about program correctness in a pre- / post-condition style.

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

Complex concepts can be constructed, via concept complement, intersection and union. Qualified number restrictions permit to express cardinality constraints of the form `x: (< n R C)` or `x: (≥ n R C)` saying that `x` has less than (respectively at least) `n` successors of class `C` via role `R` (relation between individuals). The abstract syntax of concepts `C` can be defined by the grammar:

$C ::= \bot$	(empty concept)	$\mid a$	(atomic concept)
$\mid\ !C$	(complement)		
$\mid\ C \sqcap C$	(intersection)	$\mid C \sqcup C$	(union)
$\mid\ (> n\ R\ C)$	(at most)	$\mid (< n\ R\ C)$	(at least)

[4] https://www.irit.fr/~Martin.Strecker/CLIMT/Software/smalltalc.html

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

$$
\begin{array}{rll}
fact ::= & i : C & \text{(instance of concept)} \\
| & i\ r\ i & \text{(instance of role)} \\
| & i\ !r\ i & \text{(instance of role complement)}
\end{array}
$$

A formula is a boolean combination of facts. It is represented by the following grammar:

$$
\begin{array}{rlll}
form ::= & \bot & | fact & | \neg form \\
| & form \wedge form & | form \vee form &
\end{array}
$$

The transformation language features first the elementary instructions as *delete* and *add* for manipulating graph elements. The *select* statement selects non-deterministically a node having the property as specified in the formula following *with*.The remaining language constructors are sequence of statements, looping statement and conditional branching statements as it is defined in the following grammar:

$$
\begin{array}{rll}
stmt ::= & \texttt{skip} & \text{(empty statement)} \\
| & \texttt{select}\ i\ \texttt{with}\ form & \text{(assignment)} \\
| & \texttt{delete}(i : C) & \text{(delete element from concept)} \\
| & \texttt{add}(i : C) & \text{(add element to concept)} \\
| & \texttt{delete}(i\ r\ i) & \text{(delete edge from relation)} \\
| & \texttt{add}(i\ r\ i) & \text{(insert edge in relation)} \\
| & stmt\ \texttt{;}\ stmt & \text{(sequence)} \\
| & \texttt{if}\ form\ \texttt{then}\ stmt\ \texttt{else}\ stmt & \\
| & \texttt{while}\ form\ \texttt{do}\ stmt &
\end{array}
$$

A small-t$\mathcal{ALCQ}$ 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 give in Figure 1. an illustrating example of a transformation rule written in small-t$\mathcal{ALCQ}$. A more detailed description of the language can be found in [4].

```
pre: (a : A) && (a : (>= 3 R A));
select n with (a R n) && (n : A);
delete(a R n);
delete(a : A);
post: (a : !A) && (a : (>= 2 R A));
```

Fig. 1: Example of a rule

In this example, the precondition expresses that `a` is a node of concept (or class) `A` and that `a` is linked to at least three successors of class `A` via role (or arc, attribute) `R`. The rule first selects a node `n` that is `R`-linked to `a` and which is of class `A`. Then, it deletes this link and removes `a` from class `A`. It seems plausible that after these transformations, the postcondition holds: `a` is no more of class `A` (*i.e.* , `a` belongs to `A`'s complement `!A`) and `a` has at least two `R`-successors of class `A`.

Many popular applications have been developed using small-t$\mathcal{ALCQ}$, as the model transformation from class diagrams to relational data base models [5], and Ludo, an English game which is one of the case studies of the AGTIVE 2007 Tool Contest [6].

3 Supporting Tools

Figure 2 shows the "big picture" of our framework and its components. Each component provides a specific support for small-t$\mathcal{ALCQ}$ programs: the compiler translates a small-t$\mathcal{ALCQ}$ program to an executable code; the dynamic analyzer examines the behavior of a running a program; the static analyzer and the prover use different reasoning mechanisms to analyze programs without executing them. The development of each component is based on an implementation of small-t$\mathcal{ALCQ}$'s semantics in an appropriate foundation. These components will be further described in the following.

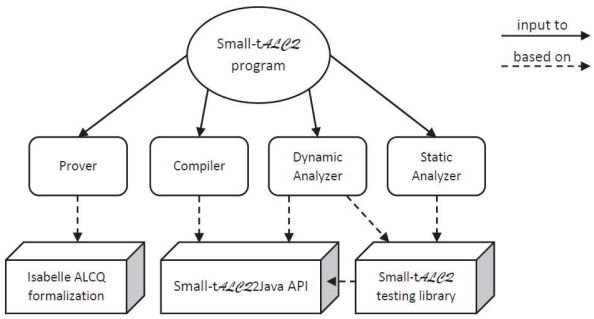


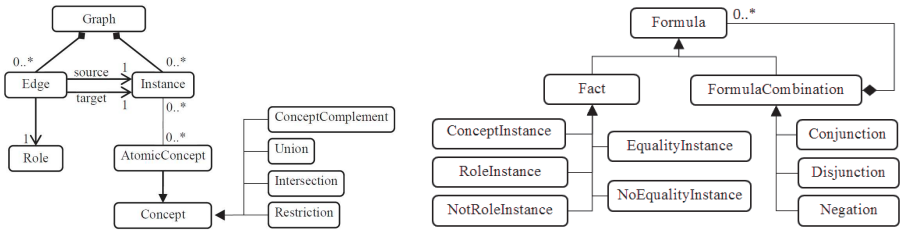
Fig. 2: Overview of the Architecture

3.1 Compiler

The execution engine of small-t$\mathcal{ALCQ}$ is based on Java. In this context, a Java API was developed implementing the semantics of small-t$\mathcal{ALCQ}$'s instructions. This API, called small-t$\mathcal{ALCQ}$2Java, allows defining a graph and translating a small-t$\mathcal{ALCQ}$ program into an executable Java target code. In order to make the execution automatic, a small-t$\mathcal{ALCQ}$ compiler was developed using the compiler generator Coco/R[5].

Within the small-t$\mathcal{ALCQ}$2Java API, a graph is represented by the Java class *Graph* which is composed of sets of *Node* and *Edge* (Figure 3a). An edge typed by a *Role* connects two *Nodes* possibly belonging to one or several concepts. Note that an atomic concept, as well as, a concept complement, intersection, union and restriction are all sub-classes of the class *Concept* as it is defined by the small-t$\mathcal{ALCQ}$ grammar.

[5] http://www.ssw.uni-linz.ac.at/Coco/#Docu



(a) Graph Structure (b) Formula

Fig. 3: Java API classes

As it is mentioned in the grammar also, a formula is a Boolean combination of facts. The most appropriate pattern to describe this composition is the *composite pattern* where *Formula* is the component, *Fact* represents the leaf on the one hand, and *Conjunction*, *Disjunction* and *Negation* represent sub-classes of the composite *FormulasCombination* on the other (Figure 3b). Fact assertions are represented by sub-classes of the class *Fact*. Each statement of our language is implemented by a Java static method defined in a class named *STALCQ*.

The Code 1.1 shows the translation of the program introduced in Figure 1 into Java. Note that the small-t$\mathcal{ALCQ}$ *select* statement allows the assignment of one or more instances satisfying a given formula. This assignment is done by a non-deterministic way. Therefore, this statement requires first translating the given formula into Java by storing all the elements satisfying the formula into the map data structure, then selecting randomly the required element from the map. Thus, each execution of a code in which it occurs a *select* statement, may provide different output graph even if it has the same input graph.

Code 1.1: Java code for the rule in Figure 1

```
// select n with (a R n) && (n : A)
FormulasCombination f1 = new Conjunction();
f1.add(r.createRoleInstances(graph,"a", "R","n")); //a R n
f1.add(i.createConceptInstances(graph,"n",new AtomicConcept("A"))); //n:A
List<String> p = new LinkedList<String>();
p.add("n");
Map<String, Node> m = STALCQ.select(graph, p, f1.instancesOf(graph));
Node n = m.get("n"); // select n
// delete (a R n);
STALCQ.delete(graph, a, "R", n);
// delete (a : A);
STALCQ.delete(graph, a, "A");
```

3.2 Dynamic Analyzer

Our dynamic analysis aims to find inconsistencies between a transformation code and its specifications by executing the transformation code, then applying

automated tests on the output graph. The test cases are generated from the post-condition. The input graph can be generated automatically from the pre-condition or can be given by the user.

Before presenting the diagnostic provided by the dynamic analyzer, let us introduce below how graphs can be generated from a pre-condition formula, and how test cases are also generated from a post-condition using small-t$\mathcal{ALCQ}$ testing library.

3.2.1 Graph Generator

In small-t$\mathcal{ALCQ}$, a formula can be represented graphically by a graph and vice versa. Actually, each fact of a formula represents the existence of a node or an edge in a graph. Thus, the generation of a source graph from the precondition is done by translating each fact in the precondition's formula into a corresponding possible Java statement in order to construct one or more objects of the class Graph. Having a few number of facts in the language makes generation of a graph from a formula not so difficult.

For example, given the precondition of the example in Figure 1, the corresponding typical graph (Figure 4a) consists of a node `a` of concept A, connected by R-edges to three anonymous nodes of concept A.

(a) Typical graph (b) Generated graph

Fig. 4: Input graphs

Having only one graph as data input for testing is not always sufficient because the test coverage is very low. To test more thoroughly graph transformations, more possible input graphs should be generated. For this purpose, fist we vary the number `n` of the restriction in a precondition to generate a set of typical graphs presenting different graph families. Then, for a graph family we can generate more graphs that are isomorphic to the typical graphs of the family[6] on the basis of Molloy-Reed algorithm [7,8]. This algorithm consists of first cutting all the edges between nodes in the graph when preserving the number of each node's outgoing edges, then pairing all the half-edges randomly. In our case, the second phase of the algorithm to match a pair of half-edges has to take into account the types of examined edges and the concepts of examined nodes. For example we cannot pair an outgoing edge `R` with an incoming edge `S`, or connect a node of concept `C` to a concept `C1` in case it was previously connected to a node of concept `C2`. By applying this algorithm to the typical graph in the Figure 4a, we can obtain at least the graph of the Figure 4b if we consider that the number corresponding to the restriction is equal to 3.

[6] Without considering isolated nodes

3.2.2 Test cases Generator

In order to test structural properties of a graph transformed by the execution of a small-t$\mathcal{ALCQ}$ program, a unit testing library was defined and called small-t$\mathcal{ALCQ}$ testing library. Its implementation is based on the small-t$\mathcal{ALCQ}$ Java API and the unit testing framework JUnit. The assertion methods defined in this library enable tests on existence and multiplicity of graph's elements. For the moment, these assertion methods are written in Java, a XUnit framework for the small-t$\mathcal{ALCQ}$ language is under construction. As mentioned before, a formula can be translated into a graph satisfying the formula. In other words, each fact of a small-t$\mathcal{ALCQ}$'s formula represents a property of a graph's element. Thus, by associating each fact to a corresponding assertion in the small-t$\mathcal{ALCQ}$ unit testing library, we can generate a set of test cases from the given formula to test if a given graph satisfies the requires properties. Table 1 shows some of the tests methods that are associated to the facts of small-t$\mathcal{ALCQ}$ language.

Table 1: Assertions associated to small-t$\mathcal{ALCQ}$ facts

`i : C`	assertExistNode(GtlGraph g, Instance i, Concept C)
`i : ! C`	assertNotExistNode(GtlGraph g, Instance i, Concept C)
`i r j`	assertExistEdge(GtlGraph g, Instance i, Role r, Instance j)
`i !r j`	assertNotExistEdge(GtlGraph g, Instance i, Role r, Instance j)
`i : (<= n R C)`	assertAtMostNumberOutgoingEdges(g, i, C, R, Integer n)
`i : (>= n R C)`	assertAtLeastNumberOutgoingEdges(g, i, C, R, Integer n)

3.2.3 Diagnostic

Let us get back to the main issue, our dynamic analyzer. If the pre- and post-conditions of a program are given, small-t$\mathcal{ALCQ}$ unit testing library can be used in the context of a dynamic analyzer to generate test cases that allow detecting possible inconsistencies between a transformation code and its specifications. This can be done automatically by generating an input graph from the given pre-condition, generating test cases from the given postcondition, then executing the examined transformation code on the source graph and finally applying the generated assertions on the program's target graph.

For the post-condition of the given example, the following test cases are automatically generated:

- `assertNotExistNode(graph, a, A)` which corresponds to the fact `a : !A`.
- `assertAtLeastNumberOutgoingEdges(graph, a, A, R, 2)` which corresponds to the fact `a : (>= 2 R A)`

As the select statement in the code is non-deterministic, executing the same transformation code several times may not always give the same output graph. Thus, the generated test cases may pass for one execution and fail for another, even if it always has the same input graph. For example, considering the graph in Figure 4b as the rule input, we have these two output graphs according to the

configuration chosen in the select statement between the two possible configurations: if one of the anonymous nodes is selected as `n`, the output graph will be the graph 5a, contrariwise, if the node `a` is selected as `n`, the output graph will be the graph 5b.

(a) (b)

Fig. 5: Output graphs

The generated tests pass by applying them on the first graph but fail by applying them on the second one. In particular, the second test which verifies that the number of the edges outgoing from the node `a` to nodes of concept `A` is equal 2, is the one that fails. This indicates that there is an inconsistency between fact `a : (>= 2 R A)` in the postcondition and the code. In this case, the developer tends to change this fact into `a : (>= 1 R A)`.

3.3 Static Analyzer

Unlike the *Dynamic Analyzer* which executes the program to find inconsistencies between a code and its specifications, the *Static Analyzer* checks, without executing the code, the inconsistencies between the given specifications and the behavior of the code implementing these specifications. Starting from the given pre-condition, the *Static Analyzer* analyses the code statically to extract the condition that will be satisfied after the execution of the code. Comparing the extracted conditions with the given post-condition using automated test cases, *Static Analyzer* can inform developers if the behavior of the code corresponds to the given condition or not.

To do so, first the *Static Analyzer* analyses the code's control flow to generate all possible execution paths and then executes each path symbolically to construct the post-condition incrementally from the code. Starting by the formula representing the given precondition, the static analysis of the code consists of updating the constructed formula according to each encountered statement on the examined path.

The comparison between the extracted post-condition and the given one will be done by generating test cases from the extracted post-condition and then executing them on the typical graph generated from the given post-condition, also by generating test cases from the given post-condition and then executing them on the typical graph of the extracted post-condition.

Considering always the same example in Figure 1, let us show how our static analyzer uses a forward computation to extract the postcondition with respect to the given code starting from the given precondition. The static analysis starts calculating the postcondition `Q` by assigning to it the precondition's formula as it is shown in Figure 6. After the *select* statement, `Q` stays the same because a

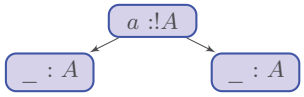
select statement is nothing but an assignment statement that does not affect the condition of the program. After the `delete(a R n)` statement, the static analyzer decreases the number of `R`-successors of class `A`. Finally, `Q` will be updated after the last statement by replacing the fact `(a : A)` in `Q` with the fact `(a : !A)` and decrementing the number of `R`-successors of class `A`, since the node `a` may have been one of the nodes that have an `R`-incoming edge and it is no more of concept `A`. Therefore, the final extracted formula is `Q = (a :!A) && (a : (>= 1 R A))`.

```
    pre: (a : A) && (a : (>= 3 R A));
(1) Q = (a : A) && (a : (>= 3 R A))
    select n with (a R n) && (n : A);
(2) Q = (a : A) && (a : (>= 3 R A))
    delete(a R n);
(3) Q = (a : A) && (a : (>= 2 R A))
    delete(a : A);
(4) Q = (a : !A) && (a : (>= 1 R A))
```

Fig. 6: Calculating the post-condition

In one hand, a typical output graph `g` (Figure 7a) is generated from the given post-condition: `g` is composed of a node `a` that is not of concept `A` connected by `R`-edge to two nodes of concept `A`. Then the static analyzer generates automatically from the extracted post-condition two test cases to be applied on `g`:

- `assertNotExistNode(g, a, A)` which corresponds to the fact `a : !A`.
- `assertAtLeastNumberOutgoingEdges(g, a, A, R, 2)` which corresponds to the fact `a : (>= 2 R A)`.



(a) Graph of the given post-condition

(b) Graph of the extracted post-condition

Fig. 7: Graphs generated from the given and extracted post-conditions

Every test that fails corresponds to an inconsistency between the given and the extracted postconditions. In our case, the tests pass so we conclude that the given post-condition implies the extracted one.

In the other hand, to check whether the implication holds in the opposite direction, tests cases are generated from the given post-condition and applied on the graph generated from the extracted one (Figure 7b). The test that corresponds to the fact `a :!A` passes. However, the test corresponding to the fact `a : (>= 2 R A)` fails. This test result gives the developer a warning and help him realizing that there is less than two `R`-edges outgoing from the node `a` to nodes of concept `A`.

The static analyzer can perform the same process to extract a precondition starting from a postcondition. However, this is done by executing paths statically in a backward mode instead of a forward mode, i.e. by analyzing the code starting from the last statement then going up until the first statement.

3.4 Prover

The purpose of the proof component of our framework is to verify rules with respect to their pre- and postconditions. The setup is rather traditional: given a triple $(pre, statements, post)$, we compute the weakest precondition *wp(statements, post)* of the rule transformations *statements* with respect to the postcondition *post*, and then verify the implication $pre \longrightarrow wp(statements, post)$.

This general, well-understood setup is complicated by several factors:

- the description logic $\mathcal{ALCQ}$ has no Boolean operators for combining facts, such as `(a : !A) && (a : (>= 2 R A))` in our example. These are rather straightforward to add.
- $\mathcal{ALCQ}$ is not closed under substitutions that occur when computing weakest preconditions. In our example, computing *wp* for the statement `delete(a R n)` and postcondition `(a : (>= 2 R A))` would yield a formula `(a : (>= 2 (R - {(a, n)}) A))`, where `(R - {(a, n)})` is the relation `R` from which the pair `(a, n)` has been removed. This is syntactically not a valid $\mathcal{ALCQ}$ formula and demonstrably [9] not equivalent to one.

The solution we propose is a new tableau method that can handle Boolean combinations of facts, that treats substitutions as a separate formula constructor and that progressively eliminates them during the tableau procedure.

A failed run of the prover results in an open tableau from which a counter model can be extracted, which is displayed in the form of a graph with JGraph[7] a modifier]. Thus, when the small-t$\mathcal{ALCQ}$ rule of Figure 8 is submitted to the prover, the proof fails and the counterexample graph of Figure 8 is produced.

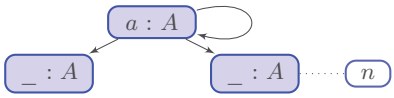


Fig. 8: Counterexample

The counterexample is a model of the precondition which does not satisfy the postcondition when applying the rule of the Figure 8.

We have formalized the operational semantics and the assertion logic in the Isabelle proof assistant [10], and we have formally verified that our logical formalism is sound with respect to the operational semantics [4].

The Isabelle formalization (written in a functional, ML-style language) is extracted to Scala [11], thus providing a highly reliable code base. This Scala

[7] http://www.jgraph.com/

code is integrated into the small-t$\mathcal{ALCQ}$ environment using Java glue code with the parser generated by the compiler generator Coco/R.

4 Intended Interaction

Our ultimate goal is writing correct graph transformations. The prover is therefore the cornerstone of our environment. However, it requires a correct definition of a Hoare's triple $(pre, statements, post)$ coding the transformation, which is sometimes not obvious to obtain at the first attempt. In practice, graph transformations programming is an incremental and iterative cognition process that needs a more flexible and pragmatic supports. Thus, we provide assistance for different levels of program maturity and let developers deciding how to use them together to define a provably correct Hoare triple.

In the ideal case, an erudite developer can successfully suggest the correct Hoare's triple to the prover in just one try. Then he can continue with a reliable transformation process by using the compiler to generate the proved executable program that transforms any source graph complied with the precondition to a target graph respecting the postcondition. As stated earlier, this situation is often too good to be true. A more practical situation is starting to design the initial steps of a transformation program without knowing precisely and fully its technical specification. The developers may need many iterations to integrate all features of the transformation. In that case, we can imagine, as described in Figure 9, some possible interactions between the proposed tools to evolve the rule transformation within a development iteration.

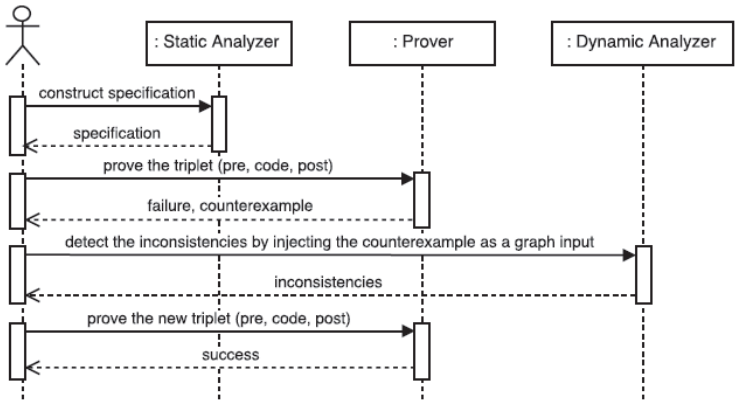


Fig. 9: A scenario of tools interaction

First the developer can use the static analyzer to elaborate the properties concerning pre- and post-conditions of a rule. By examining the rule in backward mode, static analyzer provides a diagnostic on what the precondition must convey with respect to the rule's code and the given post-condition. The static analysis in forward mode helps elicit post-condition from the rule's code and a

given precondition. When having a valid Hoare's triple, the developer can use the prover to verify the correctness of the rule. If the proofs fail, the counterexample generated by the prover can be sent to the dynamic analyzer to be tested using the test cases generated from the postcondition of the rule. By providing an instantaneous feedback about the rule behavior, the dynamic analyzer can allow detecting defects in transformation code that make the proofs failed. After correcting the code, the new Hoare's triple can be submitted again to the prover. If this time the triple is successfully proved, the current iteration is completed and the developer can start a new iteration to add new features or rules to his transformation program.

In summary, our environment aims at providing a user assistance in writing both rule's statements and its specifications. We choose a testing framework as infrastructure for providing immediate feedback and detailed diagnostics. On the one hand, the dynamic analyzer helps correct rule codes with respect to given specifications. On the other hand, the static analyzer helps construct pre- and post-conditions from a given rule code. Both can complement each other to produce a valid Hoare triple of a rule for the prover.

5 Related works

Some graph and model transformation tools such as Groove [12], Moflon [13] and Viatra [14] are very well developed and offer model checking facilities. Our approach aims at a verification method based on deductive program verification. Furthermore, we use a precisely defined semantics that is itself formalized in a proof assistant.

There are several deductive verification tools [15,16] that go in the direction of graph transformations. They are usually based on much more expressive logics, usually first order logic and thus not decidable. A graph transformation computation can also be described by a Monadic Second-Order traduction combining not only schemas, i.e. structural constraints on the source and target graphs, but also transformations [17]. We aim at verification in a decidable, but expressive fragment of first-order logic.

Test cases can be based on verification results. This is the case for CnC (Check 'n' Crash) in order to exhibit Java errors [18]. The approach considers error reports checked by the Extended Static Checker for Java (ESC/Java) as precondition violations. These abstract conditions define constraints on program values which entail a crash when the source code under analysis is executed with such test inputs. In the same way, DyTa, an automated defect detection tool for C# [19], reduces the number of false positives detected by static analysis techniques. To confirm these potential defects reported by the static analyzer, the dynamic phase generates test inputs to cover feasible paths. Our objective is not to automatically detect software defects, but to help coding a valid Hoare triple using independent but complementary tools.

Interactive theorem provers are now more suitable for verifying programs as the interaction between the end-user and the verification system is shortened,

in particular through the logic used by the prover and its decision procedures. This approach is advocated by the Dafny IDE [20], the KeY tool [21] for JAVA CARD applications and AutoProof [22] for Eiffel. However, for imperative languages, one can often observe two specific dialects when reasoning on a program: the "programming logic" itself and the logic needed by the prover. In our case, statements and specifications are based on the same $\mathcal{ALCQ}$ description logic.

Our static analyzer infers specifications from code through symbolic execution. This technique has been used by [23] to suggest loop invariants from loop bodies. A loop body is so analyzed during a single dynamic symbolic execution. considering a rule annotated with its specification as a whole, we are not concerned by loop generation invariants. However, when proving a rule, we characterize all rule executions: assuming the precondition, the code ensures the postcondition. This stamps a summarization of the rule. We envisage to apply the same approach to summarize loop bodies of a rule.

The language GP 2 [24] is close to small-t$\mathcal{ALCQ}$. Building blocks in GP programs are conditional rule schemata. A rule is applied to a left graph and produces a right graph, according to a double-pushout computation with relabeling. Nodes and edges of a rule schema are so labeled by sequences of expressions over parameters of type integer, string and list. Condition of a rule schema can check the existence of a specific labeled edge between two matched nodes, or the in/out degree of a node. small-t$\mathcal{ALCQ}$ does not propose such computations on nodes and edges: individuals (nodes) and roles (edges) within a rule only define local structural properties that the graph must have. We do not define variables and values in order to simplify the small-t$\mathcal{ALCQ}$ computation model. The pre- and postconditions of our calculus are $\mathcal{ALCQ}$ formulae. The pre- and post-conditions of GP are E-conditions [25] i.e. nested graph conditions extended with expressions as labels and assignment constraints for specifying properties of labels [26]. Proof rules for GP programs require two transformations: one (App) to transform a set of conditional rule schemata into an E-condition, and one (Pre) for computing the source graph weakest precondition leading to a target graph. Tools to help the designer when a fail occurs are not addressed in GP.

6 Conclusion

Our tool occupies a particular position in a wider landscape of transformation, verification and testing engines. As relative newcomer, it is much less developed than many specialized tools, but it has a combination of features that, we think, make it interesting.

Several important questions raised from our discussion on how to ensure that the interaction between the small-t$\mathcal{ALCQ}$e components (proofs, tests, execution semantics) is itself sound. For example, is the execution of a program based on the same semantics as the semantics employed by the prover? This paper gives at least an partial answer and sketch further developments.

As verification conditions for loop statements need invariants, we plan to automatically infer and test invariant candidates gathered from their corresponding

postcondition. We also aim at enhancing interface functionalities between the analyzers and the prover. For instance, one can imagine first compute by a static analysis a precondition of a path and then attempt to prove that this condition implies the weakest precondition for this path. Another stimulating topic is to combine formal methods and agile programming in developing small-t$\mathcal{ALCQ}$ to take advantages from agile best practices such as the Test Driven Development (TDD).

7 Acknowledgment

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

References

1. Hollunder, B., Baader, F.: Qualifying number restrictions in concept languages. In: KR. (1991) 335–346
2. Hitzler, P., Krötzsch, M., Rudolph, S.: Semantic Web Technologies. CRC Press (2010)
3. The O. M. G. Group: UML specification (June 2015)
4. Baklanova, N., Brenas, J.H., Echahed, R., Percebois, C., Strecker, M., Tran, H.N.: Provably correct graph transformations with small-talc. In: ICTERI 2015, Lviv, Ukraine, May 14-16, 2015. (2015) 78–93
5. Taentzer, G., Ehrig, K., Guerra, E., Lara, J.D., Levendovszky, T., Prange, U., Varro, D., et al.: Model transformations by graph transformations: A comparative study. In: Model Transformations in Practice Workshop at MODELS 2005, MONTEGO. (2005) 5
6. Rensink, A., Taentzer, G.: AGTIVE 2007 Graph Transformation Tool Contest. In: Applications of Graph Transformations with Industrial Relevance: AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers. Springer Berlin Heidelberg, Berlin, Heidelberg (2008) 487–492
7. Molloy, M., Reed, B.: A critical point for random graphs with a given degree sequence. Random Structures & Algorithms **6**(2-3) (1995) 161–180
8. Molloy, M., Reed, B.: The size of the giant component of a random graph with a given degree sequence. COMBIN. PROBAB. COMPUT **7** (2000) 295–305
9. Brenas, J.H., Echahed, R., Strecker, M.: On the closure of description logics under substitution. In: Book DL – Description Logics. (2016) (to appear)
10. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL. A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer Berlin / Heidelberg (2002)
11. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the Scala programming language. Technical report (2004)
12. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. STTT **14**(1) (2012) 15–40
13. Leblebici, E., Anjorin, A., Schürr, A.: Developing emoflon with emoflon. In: ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings. (2014) 138–145

14. Bergmann, G., Dávid, I., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z., Varró, D.: Viatra 3 : A reactive model transformation platform. In: 8th International Conference on Model Transformations, L'Aquila, Italy, Springer, Springer (07/2015 2015)
15. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In: Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015. Springer Berlin Heidelberg, Berlin, Heidelberg (2015) 566–580
16. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In Clarke, E.M., Voronkov, A., eds.: LPAR (Dakar). Lecture Notes in Computer Science, Springer (2010) 348–370
17. Inaba, K., Hidaka, S., Hu, Z., Kato, H., Nakano, K.: Graph-transformation verification using monadic second-order logic. In: PPDP 2011. (July 2011) 17–28
18. Csallner, C., Smaragdakis, Y.: Check 'n' crash: Combining static checking and testing. In: Proceedings of the 27th International Conference on Software Engineering. ICSE '05, New York, NY, USA, ACM (2005) 422–431
19. Ge, X., Taneja, K., Xie, T., Tillmann, N.: Dyta: Dynamic symbolic execution guided with static verification results. In: Proceedings of the 33rd International Conference on Software Engineering. ICSE '11, New York, NY, USA, ACM (2011) 992–994
20. Christakis, M., Leino, K.R.M., Müller, P., Wüstholz, V.: Integrated environment for diagnosing verification errors. In Chechik, M., Raskin, J., eds.: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Volume 9636 of Lecture Notes in Computer Science., Springer (2016) 424–441
21. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, H.P.: The key tool. Software & Systems Modeling **4**(1) (2004) 32–54
22. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In: Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg (2015) 566–580
23. Godefroid, P., Luchaup, D.: Automatic partial loop summarization in dynamic test generation. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. ISSTA '11, New York, NY, USA, ACM (2011) 23–33
24. Plump, D., Runciman, C., Bak, C., Faulkner, G. In: A Reference Interpreter for the Graph Programming Language GP 2. Volume 181 of Electronic Proceedings in Theoretical Computer Science. (4 2015) 48–64
25. Habel, A., Pennemann, K.h.: Correctness of high-level transformation systems relative to nested conditions†. Mathematical. Structures in Comp. Sci. **19**(2) (April 2009) 245–296
26. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. Fundam. Inf. **118**(1-2) (January 2012) 135–175