Combining Dynamic and Static Analysis to Help Develop Correct Graph Transformations

Amani Makhlouf(✉), Hanh Nhi Tran, Christian Percebois, and Martin Strecker

Institut de Recherche en Informatique de Toulouse (IRIT), University of Toulouse, Toulouse, France
{amani.makhlouf, tran, percebois, strecker}@irit.fr

Abstract. Developing provably correct graph transformations is not a trivial task. Besides writing the code, a developer must as well specify the pre and post conditions. The objective of our work is to assist developers in producing such a Hoare triple in order to submit it to a formal verification tool. By combining static and dynamic analysis, we aim at providing more useful feedback to developers. Dynamic analysis helps identify inconsistencies between the code and its specifications. Static analysis facilitates extracting the pre and post conditions from the code. Based on this proposal, we implemented a prototype that allows running, testing and proving graph transformations written in small $t_{\mathcal{ALC}}$, our own transformation language.

Keywords: Symbolic execution · Test case generation · Graph transformation development

1 Introduction

For most of untrained developers, writing Hoare-style provably correct graph transformations is particularly demanding because besides the transformation code, they have to specify formally the pre- and post-conditions in a suitable logic.

Our ultimate goal is an integrated development environment that allows developing and reasoning about graph transformations written in small-$t_{\mathcal{ALC}}$, a logic-based graph transformation language. In the previous work [1], we focused on using a prover to verify a given Hoare triple presenting the transformation. However, in practice, a proof based on Hoare logic is difficult to perform and often many programing efforts are needed before submitting a transformation to the prover. Thus, in this work, we turn our attention to assisting developers in writing provably correct transformations.

Section 2 presents briefly our graph transformation language small-$t_{\mathcal{ALC}}$. Section 3 presents our approach to help developers analyzing better their transformations. On the one hand, we use dynamic analysis to detect inconsistencies between the code and its specifications (Sect. 3.1). On the other hand, we use static analysis to construct the pre- and post-conditions from a code (Sect. 3.2). This paper reports on how these techniques can complement each other in a testing environment to offer useful feedback to developers.

DOI: 10.1007/978 3 319 41135 4 11

2 Small-t$_{\mathcal{ALC}}$ Environment for Graph Transformations

Our graph transformation language is based on $\mathcal{ALC}$ *(Attributive Language with Complements)* [2], a member of the *Description Logic* family. This logic uses a three-tier framework: *concepts*, *facts* and *formulae*. A concept represents a set of individuals and a role represents a binary relation between the individuals.

At the concept level, a concept C can be empty, atomic or built from other concepts. $\mathcal{ALC}$ provides the following concept constructors: intersection $(C1 \cap C2)$, union $(C1 \cup C2)$, complement $(\neg C)$ and existential or universal restrictions on roles $(\exists r\ C$ and $\forall r\ C)$. The fact level allows making assertions about an individual owned by a concept, or involved in a role. The grammar of facts is summarized in the following: $(i{:}C)$ asserts that an individual i is an instance of a concept C; $(i\ r\ j)$ and $(i\ (\neg r)\ j)$ assert respectively that an instance of a role r exists or not between two individuals i and j. The final level is about formulae defined by a Boolean combination of $\mathcal{ALC}$ facts. This formula level includes negation $(\neg f)$, conjunction $(f1 \wedge f2)$ and disjunction $(f1 \vee f2)$ of formulae.

Concepts, facts and formulae are the core of small-t$_{\mathcal{ALC}}$, a rule-based imperative programming language that we've developed for specifying and reasoning about graph transformations [1]. Note that individuals of a concept can be represented as the nodes of a graph; in the same way, a role between two individuals corresponds to an edge. Thus, a graph can be described by a formula in which each node is represented by a fact $(i{:}C)$ and each edge between two nodes i and j is represented by a fact $(i\ r\ j)$. Manipulating a graph results in modifying the formula representing it.

small-t$_{\mathcal{ALC}}$ provides statements to manipulate the structure of a graph: *add* $(i{:}C)$ and *delete* $(i{:}C)$ for adding and respectively deleting a node (an individual) from a concept[1]; *add* $(i\ r\ j)$ and *delete* $(i\ r\ j)$ for adding and respectively deleting an edge (a role) between two nodes. small-t$_{\mathcal{ALC}}$ also proposes *(select i with f)*, a non-deterministic assignment statement allowing to select a set of individuals satisfying a formula. In this work, we focus only on transformation of graph structure and do not deal with the values of attributes that maybe associated to graph's elements.

small-t$_{\mathcal{ALC}}$ enables sequential composition, branching, iteration and modularity. A small-t$_{\mathcal{ALC}}$ program comprises transformation rules and a *main* function, as the program's entry, that orders the rules to be executed. To allow reasoning about graph transformation programs, a rule is annotated with assertions specifying its pre- and post-conditions. The distinctive feature of small-t$_{\mathcal{ALC}}$ is that formulae occur not only in assertions (such as pre- and post-conditions or loop invariants), but also in statements (branching and iteration conditions, *select* conditions). In this way, assertions are akin to graph manipulation statements and based on the same logic dialect. Assertions lead to a Hoare-like calculus for small-t$_{\mathcal{ALC}}$ with potential tests and proofs.

Figure 1 gives an example of a small-t$_{\mathcal{ALC}}$ rule which redirects the edges between nodes of concept A and nodes of concept B to the edges between nodes of concept A and new nodes of concept C. The rule is structured into three parts: a pre-condition, the code (a set of statements) and a post-condition. small-t$_{\mathcal{ALC}}$ is designed as a domain

[1] The individual is not deleted from the graph because it can be still owned by other concepts.

```
rule EdgeRedirection {
     pre : a : A and b : B and a r b
     post : a : A and b : B and c : C and a r c
     while (a : A and b : B and a r b) do {
        select a, b with a : A and b : B and a r b;
        add(c : C);
        delete(a r b);
        add(a r c);
     }
}
```

Fig. 1. Rule redirecting *(a r b)* to *(a r c)*

specific language, not a general purpose one. Thus, to simplify its syntax, all rules work on the same input and output graphs and the pre- and post-conditions are specified on these global graphs. In this example, the pre-condition expresses that a is a node of concept A, b a node of concept B and that a is linked to b via role (or edge) r. While there are nodes a and b satisfying the while condition, the rule selects these nodes, deletes the link between them, add a new node c of concept C, then connects the selected node a with the new node c via the role r. The post-condition expresses that there are three nodes a, b and c of concepts A, B and C respectively and that a is connected to c via role r[2].

For executing and reasoning on small-t$_{\mathcal{ALC}}$ programs, we developed an environment composed of a Java code generator to enable executing small-t$_{\mathcal{ALC}}$ rules, a JUnit test case generator for rule testing and an Isabelle/HOL verification condition generator coupled to a tableau prover for Hoare triples.

3 Assistance for Writing Small-t$_{\mathcal{ALC}}$ Programs

Our objective is to provide assistance on writing both small-t$_{\mathcal{ALC}}$ code and specifications by combining static and dynamic analysis [3]. In Sect. 3.1 we report how testing can help developers correct their code with respect to given specifications. In Sect. 3.2 we investigate the symbolic execution technique to help a developer construct pre- and post-conditions from a given code.

3.1 Dynamic Analysis for Detecting Defects in Transformation Code

We consider a situation where the correct specifications of a code are given, especially the pre-condition. As presented in Sect. 2, the pre- and post-conditions are formulae specifying graphs before and after a transformation. Each fact of the pre-condition represents the existence of a node or an edge in the source graph. Each fact of the

[2] We can strengthen the post condition by adding the fact *(a ¬r b)* to insist that there is no edge between a and b. However, we intentionally keep it weak to illustrate that developers can write any post condition, not exactly the strongest post condition *wrt.* the given pre condition.

Fig. 2. Source graph and test cases generated from the running example

post-condition represents the existence of a node or an edge in the target graph. Thus, from the given pre-condition we can generate a source graph and, from the given post-condition, generate a set of test cases for the required properties. In our framework, dynamic analysis consists in testing the target graph obtained by the transformation with the generated test cases which are expressed in JUnit. In this context, we defined and implemented a unit testing library for small-t$_{\mathcal{ALC}}$ having about twenty assertion methods allowing testing the existence and multiplicity of nodes and edges.

Figure 2 shows the result of the generation of a source graph and the test cases corresponding to the pre- and post-conditions of the example in Fig. 1. The generated source graph represents the minimal graph configuration satisfying the pre-condition.

Suppose that the developer did not write the statement *add(a r c)* in the code. Because of this missing statement, the corresponding test *assertExistEdge(a r c)* fails. This test result reveals then an inconsistency between the code and its specifications. Moreover, it informs the developer about the non-existence of the edge *r* between *a* and *c*[3].

When the proof fails on verifying a program, the prover can give a counter example without further suggestions about the code's inconsistencies. This counterexample can be used as the program's graph input instead of a graph generated from the precondition to provide more feedback about the behavior of the code in such situation.

3.2 Static Analysis for Constructing Specifications from Code

Assume now that a code is correct, but developers need help to define the formal specification. We aim at computing, from the given code, conditions that must be satisfied before and after applying the transformation. For small-t$_{\mathcal{ALC}}$ programs whose symbolic values are explicitly defined in the code's formulae, such computation can be easily done by using a technique based on symbolic execution. We analyze the code's control flows to generate all possible execution paths and then execute each path symbolically to construct incrementally the pre- and post-conditions by considering the required conditions of each path.

We recall that the axiomatic semantics of each small-t$_{\mathcal{ALC}}$ statement is defined by the formulae representing its pre- and post-conditions, which specify a graph before and after executing the statement. Thus, on tracing the path's statements, we can compute progressively the formulae representing the pre- and post-conditions of the path by updating them according to the pre- or post-conditions of each encountered

[3] If the post condition was strengthened by the fact *(a ¬r b)*, the corresponding test *assertNotExistEdge(a r b)* will have been also generated.

Forward computation

```
FC (add(f), Q)    = delFM (addFM(Q, post(add(f))), pre(add(f)))
FC (delete(f), Q) = delFM (Q, pre(delete(f)))
FC (select(f), Q) = addFM (Q, post(select(f)))
```

Backward computation

```
BC (add(f), P)    = addFM (delFM (P, post(add(f))), pre(add(f)))
BC (delete(f), P) = addFM (P, pre(delete(f)))
BC (select(f), P) = addFM (P, pre(select(f)))
```

Fig. 3. Forward and Backward computations for analyzing small $t_{\mathcal{ALC}}$ statements

statement. An execution path is analyzed in two directions. A forward computation extracts a formula representing the post-condition and a backward computation extracts a formula representing the pre-condition. Path statements are processed differently in each computation mode. Figure 3 presents, in a simplified functional style, the algorithms to update the extracted specification according to the semantics of the encountered statement.

In this figure, *FC* represents the Forward Condition formula and *BC* the Backward Condition formula. *st(f)* denotes a small-$t_{\mathcal{ALC}}$ statement, where *st* can be *add*, *delete* or *select* and *f* is the formula specifying the manipulated graph element. If *st* is *add* or *delete*, *f* can be *(i:C)* to represent a node, or *(i r j)* to represent an edge. The auxiliary functions *pre(st(f))* and *post(st(f))* extract respectively the pre- and post-conditions of *st (f)*. For example, *pre(add(i r j)) = (i (¬r) j)* and *post(add(i r j)) = (i r j)* as we allow only one edge of a given relation between two nodes. For the *select* statement, *pre (select(f)) = post(select(f)) = f*. The auxiliary functions *addFM(C, f)* and *delFM(C, f)* are used respectively to add the formula *f* into the path's conjunction *C* (if *C* does not already contain *f*) or delete the formula *f* in the conjunction (if *C* contains *f*). *C* denotes a post-condition *Q* in a forward computation, or a pre-condition *P* in a backward computation.

For a given path, *FC* and *BC* of the classic control statements are computed in the same way as strongest post-condition and weakest pre-condition computations respectively [4]. Figure 4 illustrates the FC computation for the post-condition *Q* of the example in Fig. 1. We consider only the execution path in which the *while* condition is true.

The computed formula is then presented to developers in the testing framework (c.f. Sect. 3.3) to allow them to verify if the conditions of the analyzed path are respected in the current rule's pre- and post-conditions issued from analyzing previous execution paths or written by developers themselves.

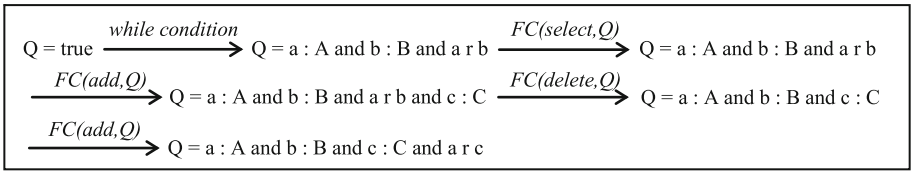

Fig. 4. FC computation for the example

The consequence rule of Hoare logic rule allows to strengthen the precondition and/or to weaken the post-condition of a Hoare triple: given $P1 \rightarrow P2$ and $Q2 \rightarrow Q1$, if $\{P2\}\ S\ \{Q2\}$ then $\{P1\}\ S\ \{Q1\}$. Ideally $P1$ should be the weakest precondition $wp(S, Q1)$ of S with respect to $Q1$ and vice versa (i.e. $Q1$ should be the strongest post-condition $sp(S, P)$ of S with respect to $P1$). However, developers can write a rule with the independent specifications $P2$ and $Q2$ where some facts of the precondition $P2$ are not necessarily considered for the post-condition $Q2$. Considering the rule in Fig. 1, $(a\ r\ b)$ in the pre-condition has been translated into $(a\ r\ c)$ without considering $(a\ \neg r\ b)$ in the post-condition.

3.3 Combining Dynamic and Static Analysis

The two scenarios represented in Sects. 3.1 and 3.2 are the borderline cases of small-$t_{\mathcal{ALC}}$ transformations development. In practice, both of specifications and code are partially and imprecisely defined. Complementary to the diagnostics provided by a prover, we propose an approach that allows treating an incomplete Hoare triple by verifying its consistency in an incremental manner. In general developers prefer testing to proving, so our assistance provides them feedback via a testing framework combining white-box testing and black-box testing [5].

A developer may write a code and weak specifications, apply the white-box testing to detect inconsistencies between them and use the static analysis technique to complete them. On the basis of the static analysis technique, extracted specifications from the code are compared to pre- and post-conditions given by the developer to help him correct or complete his specifications. This comparison yields black-box test cases generated from the extracted pre- and post-conditions then executed on a graph generated from the given pre- and post-conditions respectively. Each test which fails corresponds to a missing or an incorrect fact in the formula representing the given specification. Therefore, during the development of a transformation program, in each iteration, a developer can alternate between the two approaches depending on his needs.

4 Discussion

Our small-$t_{\mathcal{ALC}}$ environment combines two techniques for verifying a Hoare triple. The prover we developed [1] can prove the correctness of a transformation for all arbitrary graphs satisfying the pre-condition without executing the transformation. This formal verification technique, although has been well developed [6, 7], is not really applicable during the transformation development where the Hoare triple is often still incomplete. The testing environment presented in this paper proposes a more pragmatic solution, from the developer's point of view, to detect inconsistencies in an under-developed transformation. By using both of the above techniques, we try to take advantage of multiple complementary approaches [3, 8, 9] for assisting transformation developers.

To assist developers, testing has been used for generic imperative languages. Our approach shares with [5] the idea to use a deductive program verification mechanism for extracting specification by symbolically executing small-$t_{\mathcal{ALC}}$ rules. Our forward

and backward test cases generations are based on code-driven paths exploration as in [5, 10]. The transitions from code to specification and vice versa are straightforward with small-t$_{\mathcal{ALC}}$ because it uses the same logic to specify programs and properties to be verified. This is often less direct for conventional imperative languages where there is a possible gap between the logic defining the semantics of the language and the logic used for formalizing the correctness of programs. In such cases, sometimes it is difficult to identify symbolic values [11] and symbolic execution is often achieved for only a limited subset of the target language features [10].

The design of language GP2 [12] is close to small-$t_{\mathcal{ALC}}$. Building blocks in GP programs are conditional rule schemata whose nodes and edges are labeled by sequences of expressions over parameters of type integer, string and list. Condition of a rule schema can be expressed then on the existence of a specific labeled edge or the in/out degree of a node. small-$t_{\mathcal{ALC}}$ does not propose such computations on nodes and edges: individuals (nodes) and roles (edges) within a rule define only local structural properties of a graph. We do not define variables and values in order to simplify the small-$t_{\mathcal{ALC}}$'s computation model. The conditions of our calculus are $\mathcal{ALCQ}$ formulae while GP uses E-conditions [6], i.e. nested graph conditions extended with expressions as labels and assignment constraints for specifying properties of labels [7]. Tools to help the designer when a fail occurs are not addressed in GP.

Few works have been proposed for testing graph transformation implementations. Close to our work, [13] generates test cases for the graph pattern matching phase; [14] generates JUnit test cases from a Fujaba graphical story diagram. Both approaches are based on the graph pattern matching phase of the transformation rule to generate test cases, not on logical rule specifications as we propose.

5 Conclusion

Thanks to the formal semantic basis of small-t$_{\mathcal{ALC}}$, we can apply both dynamic and static analysis techniques in an effortless way to reason about small-t$_{\mathcal{ALC}}$ programs and give useful feedback to developers during the transformation development.

Our current test data generation is rather simplistic and just covers a minimal configuration of possible source graphs. We are improving our algorithm for generating more graphs from the typical graph on the basis of Molloy-Reed algorithm [15, 16] and allowing also graph inputs provided by developers as the prover's counterexample.

In this paper we did not deal with loop invariants as conditions of a transformation, we plan to automatically infer and test invariant candidates gathered from their corresponding post-condition as proposed in [17]. This attempt is based on the fact that a small-t$_{\mathcal{ALC}}$ loop iterates on all individuals selected from a logic formula in order to achieve the same logic property for all transformed elements. We also aim at enhancing interface functionalities between test and proof processes. For instance, suppose that our testing environment validates a rule's post-condition on a given path. One can imagine, with the help of a prover, computing the strongest post-condition of this path by symbolic execution. The correctness of the path can be proven if the strongest post-condition implies the given post-condition. If this implication holds for all paths in the code, then the original Hoare triple is valid [4].

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

References

1. Baklanova, N., Brenas, J.H., Echahed, R., Percebois, C., Strecker, M., Tran, H.N.: Provably correct graph transformations with small tALC. In: ICTERI 2015, pp. 78 93 (2015)
2. Schmidt Schauß, M., Smolka, G.: Attributive concept descriptions with complements. Artif. Intell. **48**(1), 1 26 (1991)
3. Smaragdakis, Y., Csallner, C.: Combining static and dynamic reasoning for bug detection. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 1 16. Springer, Heidelberg (2007)
4. Gordon, M., Collavizza, H.: Forward with Hoare. In: Roscoe, A.W., Jones, C.B., Wood, K.R. (eds.) Reflections on the Work of C.A.R. Hoare. History of Computing Series, pp. 101 121. Springer, London (2010)
5. Beckert, B., Gladisch, C.: White box testing by combining deduction based specification extraction and black box testing. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 207 216. Springer, Heidelberg (2007)
6. Habel, A., Pennemann, K.H.: Correctness of high level transformation systems relative to nested conditions. Math. Struct. Comput. Sci. **19**(2), 245 296 (2009)
7. Poskitt, C.M., Plump, D.: Hoare style verification of graph programs. Fundam. Inf. **118**(1 2), 135 175 (2012)
8. Liu, S., Nakajima, S.: Combining specification based testing, correctness proof, and inspection for program verification in practice. In: Liu, S., Duan, Z. (eds.) SOFL + MSVL 2013. LNCS, vol. 8332, pp. 1 18. Springer, Heidelberg (2014)
9. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.K.: PVS: combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 411 414. Springer, Heidelberg (1996)
10. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 169 188. Springer, Heidelberg (2007)
11. Xie, T., Marinov, D., Schulte, W., Notkin, D.: Symstra: a framework for generating object oriented unit tests using symbolic execution. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 365 381. Springer, Heidelberg (2005)
12. Bak, C., Faulkner, G., Plump, D., Runciman, C.: A reference interpreter for the graph programming language GP 2. In: Rensink, A., Zambon E. (eds.) Graphs as Models 2015 (GaM 2015). EPTCS 2015, vol. 181, pp. 48 64 (2015)
13. Darabos, A., Pataricza, A., Varró, D.: Towards testing the implementation of graph transformations. Electron. Notes Theor. Comput. Sci. **211**, 75 85 (2008)
14. Geiger, L., Zündorf, A.: Transforming graph based scenarios into graph transformation based JUnit tests. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 61 74. Springer, Heidelberg (2004)
15. Molloy, M., Reed, B.: A critical point for random graphs with a given degree sequence. Random Struct. Algorithms **6**(2 3), 161 180 (1995). Wiley
16. Molloy, M., Reed, B.: The size of the giant component of a random graph with a given degree sequence. Comb. Prob. Comput. **7**(3), 295 305 (1998). Cambridge University Press
17. Zhai, J., Wang, H., Zhao, J.: Post condition directed invariant inference for loops over data structures. In: SERE C 2014, pp. 204 212. IEEE (2014)