Analysis and Abstraction of Graph Transformation Systems via Type Graphs

Dennis Nolte

Abteilung für Informatik und Angewandte Kognitionswissenschaft,
Universität Duisburg-Essen, Germany
dennis.nolte@uni-due.de

Abstract. We generalize verification techniques from the theory of formal languages to the framework of graph languages. For this purpose, we investigate formalisms for specifying graph languages based on type graphs and compare them to existing formalisms. To adapt the verification approaches one needs a specification formalism with suitable closure properties and positive results for decidability problems.

Introduction

The theory of formal languages plays an important role in computer science and there exists a large number of applications for this theory, for instance in compiler construction, parsing and verification. In verification some typical methods are (non-)termination analysis [16,18], reachability analysis [17] and counterexample-guided abstraction refinement [6]. Reachability analysis for example is based on the decidability of the language inclusion problem, in combination with the closure under union property and the possibility to compute postconditions. Starting with an initial set of states, one builds the set of all reachable states iteratively by computing the strongest postcondition and adding all new states to the current set with the union operator. An inclusion check after each iteration is used to check if no new reachable state was added. Using this analysis, one can prove the absence of erroneous states.

Many concurrent and distributed systems, especially those with a dynamically evolving topology, can be naturally modelled by graphs and graph transformation rules. Work on the verification of dynamic, graph-like structures has shown that they introduce an additional level of complexity, compared to rule-based systems where states have either a word or tree structure. While the theory of formal languages is worked out very well in string and tree/term rewriting, it is often non-trivial to solve the same problems when it comes to graph rewriting. Therefore, it is natural to ask for generalizations of these verification techniques for the framework of graph rewriting and additionally for a theory of graph languages, where these techniques can be applied. The analysis of pointer structures, in the research field of heap analysis, is just one example, where the adequate specification of sets of graphs in combination with verification techniques is needed.

For this purpose, one needs a specification formalisms for graph languages with suitable closure properties, positive results for decidability problems (like membership, language inclusion and emptiness) and computable pre- and post-conditions. Instead of just tinkering with fitting existing specification formalisms for any given verification problem, we try to achieve a different main goal here: The contribution of this research is help to understand the essence of different graph specification languages, which grant them the possibility to adapt the verification techniques. Therefore, we have started research on a very simple specification formalism based on type graphs [9]. This formalism is analysed in detail with respect to desirable properties and then refined stepwise to enrich its expressiveness. Each refinement is again analysed and in addition compared to existing formalisms. With this approach, we want to contribute to the comparison of existing formalisms and structure them according to their capabilities to use in certain verification techniques.

Related Work

There already exist several approaches to specifying graph languages, for instance via logics [10,11], grammars [14,19], automata [1,2] or even annotated abstract graphs [22,24]. Most of these formalisms differ in terms of closure properties and there exists a trade off between expressiveness and decidability properties. This might lead to incompatibility with certain verification techniques.

Courcelle's notion of recognizable graph languages [11] (which is equivalent to regular word languages and closely related with monadic second-order graph logic [10]) is widely known and accepted. However, it becomes quite impractical with respect to actual applications due to the large size of the resulting graph automata [1]. Nested application conditions [20] (as the counterpart to first order logic [23]) can already be used to compute pre- and postconditions. However, implication and satisfiability are already undecidable for this formalism. There also exist hyperedge replacement grammars [19] (being equivalent to the notion of context-free (word-)grammars), three valued logic analyser [24] (representing heaps via graphs annotated with predicates from a three-valued logic) and several other approaches one could name.

Usually these specification languages have at least one of the following problems: Language inclusion checks, needed for invariant checking, are usually undecidable. This is already true once the formalism has an expressive power of at least first-order logic. On the other hand, expressiveness might sometimes not be sufficient enough, for instance the existence of paths can not be specified in first-order logic. The computation of postconditions, used in reachability analysis, is often impossible or just too difficult, such that some kind of over-approximation might be necessary to compute it. Or the computation can become way too costly in general, which makes the formalism impractical from some point.

We believe that there is no one-fits-all solution. Our approach is to study graph specification languages and classify them according to their properties.

Proposed solution

We focus on specification languages based on type graphs [8], where the language of a type graph T consists of all graphs that can be mapped homomorphically into T (with potentially extra constraints to extend the framework). Many specification formalisms that are usually used in abstract graph transformation [25] and verification, are based on type graphs. For instance, shape graphs [22] can be seen as type graphs with additional annotations.

We work with the algebraic double-pushout (DPO) approach to graph rewriting [13] to analyse graph transformation systems. Since we are interested in the verification of graphs which may model specific systems, the advantage in using DPO lies in the fact that deletion in unknown contexts is forbidden per default. Therefore, by using DPO instead of other approaches like single-pushout (SPO), we can ensure that the application of our rules never cause unwanted side-effects, which could lead to inadequate models of the described system.

Type graphs are a standard tool for typing graph transformation systems [7,13], but we are not aware of any case where they have been extensively studied from the perspective of specification languages. Usually, one assumes that the rules and the graphs to be rewritten are typed. This idea serves the purpose of introducing constraints on the applicability of the rules and therefore type graphs can be understood as a form of labelling. However, this is different from our point of view, where graphs and rules remain untyped (even while working with labelled graphs) and the type graphs are simply meant to represent a possibly infinite set of graphs. Type graphs retain a nice intuition from regular languages when it comes to specifying graph languages. The language of a given finite state automaton M can be interpreted as the set of all string graphs that can be mapped homomorphically to M (respecting initial and final states).

Preliminary Work

The following section presents my joint work with several co-authors and gives an overview of my research topics. Up until now we have achieved several results, which can be divided into the following two work packages:

– *Termination Analysis.* Proving the termination property of a rewriting system, e.g. the absence of rewriting or derivation sequences of infinite length, is an undecidable problem in general [21]. Nonetheless, given a rewriting system (for instance in graph rewriting), one can try to run several proposed methods in parallel to possibly find a solution for the specific termination problem. One possible approach of proving termination is to construct a monotone function that measures structural properties of the graphs to be rewritten. Afterwards one shows that the value of such a function (or assigned weight of the graph) decreases with every rule application. This is usually achieved by evaluating the weights directly on the left-hand side and right-hand side of every rule in the rewriting system.

We introduced a technique based on type graphs which are weighted over different kinds of semirings (see [3,4]), to check if a given graph transformation system is uniformly terminating, i.e. independently of the initial graph the rules of the system can only be applied a finite number of times. This technique was inspired by an existing method based on matrix interpretations for proving termination in string, cycle and term rewriting systems [15]. The type graph was used to specify the set of all possible graphs by finite means and at the same time assign weights to the graphs to be rewritten. Depending on the semiring chosen for the computation, we were able to prove termination for graph transformation systems consisting of rules, that can be applied up to an exponential number of times.

We implemented the new termination analysis technique (among others) in a prototype Java-based tool named *Grez* [5]. The tool concurrently runs several algorithms to possibly prove the termination of a given graph transformation system. Our work in [3] extended *Grez* to be able to employ an SMT solver, to solve inequalities resulting from our method. The inequalities encode all possible morphisms from the given rule graphs (both left- and right-hand side) into potential weighted type graph candidates. The variables, used in these encodings, represent weights for each element of the type graph. Therefore, whenever the SMT solver returns a valid solution for the inequalities, it gives rise to the weights assigned to the type graph such that it becomes a witness for the termination proof.

Finally, in [26] we translated term rewrite systems from the *Termination Problems Database* (TPDB) into graph transformation systems and let *Grez* automatically prove termination on them. We investigated two different encodings (namely the function and number encoding) in two possible rewriting interpretations (called basic and extended version) for term rewrite rules into graph transformation rules that preserve the termination property, e.g. whenever the graph transformation system terminates, so does the term rewrite system. The following table is an excerpt of our experimental results given in [26]:

Termination Analysis using *Grez*				
Tested Total	**201**			
No Result Found	84			
Terminating Total	**117**		**24**	
Terminating using	117	115	24	24
	Number Encoding	Function Encoding	Number Encoding	Function Encoding
Version	Basic		Extended	

– *Specifying Graph Languages.* In our recent work [8] we analysed decidability and closure properties for graph languages specified by type graphs. While not being as expressive as recognizable graph languages, we proved positive results with respect to decidability problems for the two simplest cases of specification formalisms, namely *type graph languages* and *restriction graph languages*. A *type graph language* contains all graphs which allow a homomorphism into a given

type graph, whereas a *restriction graph language* includes all graphs that do *not* contain an homomorphic image of a given type graph. We also extended the formalism in two different ways: First, we introduced boolean connectives between type graphs to generate a type graph logic and second, we increased the expressiveness of the type graph itself, by adding annotations to every type graph element.

In case of the *type graph logic*, one already obtains the desired closure properties for free since they are semantically given by the logical conjunction, disjunction and negation operators. Due to the presence of these boolean operators the language inclusion problem can be reduced to the emptiness problem. However, it is still impossible to compute postconditions within this formalism. This is due to the fact that one can not express the existence of a subgraph (here the right hand-side graph from a graph transformation rule) in every graph contained in the specified graph language.

We defined a category of *annotated type graphs*, to generate an abstract framework, from which existing formalisms based on type graphs can be instantiated. Each type graph is enriched with a set of annotations, whereas every annotation can be parametrized. For instance, in one of our settings, the annotations are used to globally count all elements that can be mapped to the elements in the type graph. This is different from UML multiplicities, which are locally specified on the edges. The reason to allow several annotations for one type graph, instead of a single annotation, is mainly to ensure closure under union.

By adding annotations to the type graph, the expressiveness is too powerful, such that the language inclusion problem becomes hard to decide. We only obtained positive results for the language inclusion problem by restricting the analysed graph languages to only contain graphs up to a given pathwidth (equivalent to [1]). The situation remains unclear for the unbounded case and is an open problem. At the same time, it remained unclear if the framework of annotated type graphs is closed under the complement operation. However, by adding annotations to the formalism, we were able to compute postconditions of rule applications, which was impossible in the other refinements of the type graph specification language. In addition, we investigated closure under rule application, e.g. invariant checking for our frameworks. An overview of the results given in [8] is shown in the following table, where the checkmark is shown in brackets, whenever the results hold only for the bounded case so far:

Investigation		Pure	Restrict	Logic	Annotated
Decidability	$G \in \mathcal{L}(T)$	✓	✓	✓	✓
	$\mathcal{L}(T) = \emptyset$	✓	✓	✓	✓
	$\mathcal{L}(T_1) \subseteq \mathcal{L}(T_2)$	✓	✓	✓	(✓)
Closure Prop.	$\mathcal{L}(T_1) \cup \mathcal{L}(T_2)$	✗	✓	✓	✓
	$\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$	✓	✗	✓	✓
	$G^* \setminus \mathcal{L}(T)$	✗	✗	✓	?
Invariant Checking		✓	✓	?	(✓)

Future Work

While classifying specification languages via our categorical approach, we are still interested in characterizing a specification language that allows to generalize verification techniques, from the theory of formal languages. Our current results show that one needs to extend the expressiveness of the type graphs by allowing a set of annotations on the type graph elements, to be able to compute postconditions. This is necessary, if we want to extensively use these formalisms in application scenarios such as reachability analysis or non-termination analysis. To compute the pre-/postconditions within the formalism, we will also have to generalize Hoare logic. We still plan to study the possibility to integrate UML multiplicities into our framework and investigate if the framework is able to handle attributes and inheritance (like in [12]). Exploiting universal properties from category theory, we are currently working on a materialisation construction (similar to [24]) for our generalized abstract setting. The basic observation is that in most specification frameworks an abstract rewriting step is performed by computing the (strongest) postcondition in two steps: by first materializing the left-hand side of the rule to be applied (also called shift in some specification formalisms), followed by adding the right-hand side (existentially quantified). Being able to compute postconditions for the specification of graph languages by using annotated type graphs, we plan to implement verification techniques for this formalism in a prototype Java-tool called *DrAGoM*. We further plan to benchmark the techniques of the tool with respect to runtime results.

References

1. Christoph Blume. *Graph Automata and Their Application to the Verification of Dynamic Systems*. PhD thesis, University of Duisburg-Essen, 2014.
2. H.J. Sander Bruggink and Barbara König. On the recognizability of arrow and graph languages. In *Proc. of ICGT '08*. Springer, 2008. LNCS.
3. H.J. Sander Bruggink, Barbara König, Dennis Nolte, and Hans Zantema. Proving termination of graph transformation systems using weighted type graphs over semirings. In *Proc. of ICGT '15*, volume 9151 of *LNCS*. Springer, 2015.
4. H.J. Sander Bruggink, Barbara König, and Hans Zantema. Termination analysis for graph transformation systems. In *Proceedings of IFIP-TCS 2014*, 2014.
5. H.J.S. Bruggink. Grez user manual. `www.ti.inf.uni-due.de/research/tools/grez`, 2015.
6. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.
7. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation—part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*, chapter 3. World Scientific, 1997.
8. Andrea Corradini, Barbara König, and Dennis Nolte. Specifying graph languages with type graphs. In *Proc. of ICGT '17 (International Conference on Graph Transformation)*, pages 73–89. Springer, 2017. LNCS 10373.

9. Andrea Corradini, Ugo Montanari, and Francesca Rossi. Graph processes. *Fundamenta Informaticae*, 26(3/4):241–265, 1996.
10. Bruno Courcelle. The monadic second-order logic of graphs I. Recognizable sets of finite graphs. *Information and Computation*, 85:12–75, 1990.
11. Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic, A Language-Theoretic Approach*. Cambridge University Press, June 2012.
12. Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Attributed graph transformation with node type inheritance. *Theoretical Computer Science*, 376(3):139 – 163, 2007. Fundamental Aspects of Software Engineering.
13. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006.
14. H. Ehrig, M. Pfender, and H. Schneider. Graph grammars: An algebraic approach. In *Proc. 14th IEEE Symp. on Switching and Automata Theory*, 1973.
15. J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40:195–220, 2008.
16. Jörg Endrullis and Hans Zantema. Proving non-termination by finite automata. In *RTA '15*, volume 36 of *LIPIcs*, pages 160–176. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.
17. Laurent Fribourg and Hans Olsén. Reachability sets of parameterized rings as regular languages. In *Proceedings of Infinity '97*, volume 9 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1997.
18. Alfons Geser, Dieter Hofbauer, and Johannes Waldmann. Match-bounded string rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 15(3–4):149–171, 2004.
19. Annegret Habel. *Hyperedge Replacement: Grammars and Languages*. Springer-Verlag, 1992. LNCS 643.
20. Annegret Habel and Karl-Heinz Pennemann. Nested constraints and application conditions for high-level structures. In *Formal Methods in Software and Systems Modeling. Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday*, pages 294–308. Springer, 2005. LNCS 3393.
21. D. Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2):201–209, 1998.
22. Arend Rensink. Canonical graph shapes. In *Proc. of ESOP '04*, pages 401–415. Springer, 2004. LNCS 2986.
23. Arend Rensink. Representing first-order logic using graphs. In *Proc. of ICGT '04*, pages 319–335. Springer, 2004. LNCS 3256.
24. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.
25. Dominik Steenken, Heike Wehrheim, and Daniel Wonisch. Sound and complete abstract graph transformation. In *Proc. of SBMF '11*, pages 92–107. Springer, 2011. LNCS 7021.
26. Hans Zantema, Dennis Nolte, and Barbara König. Termination of term graph rewriting. In *Proc. of WST '16 (Workshop on Termination)*, 2016.