Using Graph Transformations and Graph Abstractions for Software Verification^{*}

Eduardo Zambon

Formal Methods and Tools Group
University of Twente, The Netherlands
`zambon@cs.utwente.nl`

Introduction. In this abstract we present an overview of our intended approach for the verification of software written in imperative programming languages. This approach is based on model checking of graph transition systems (GTS), where each program state is modeled as a graph and the exploration engine is specified by graph transformation rules. We believe that graph transformation [13] is a very suitable technique to model the execution semantics of languages with dynamic memory allocation. Furthermore, such representation provides a clean setting to investigate the use of graph abstractions, which can mitigate the space state explosion problem that is inherent to model checking techniques.

Overview. Figure 1 provides a picture of the whole verification cycle. The input is a program source code. The code is analysed by a compiler that produces as output an abstract syntax graph (ASG). This ASG is essentially the usual abstract syntax tree produced by a language parser enriched with type and variable bindings. The ASG, together with definitions of the language control flow semantics, is the input of a flow construction mechanism, that builds a flow graph for the given ASG. This flow graph represents how the execution point of the program should flow through the ASG, according to the rules of the programming language in use. Together, an ASG and a flow graph form a program graph, an executable representation of the program code as a graph. A program graph serves as input to an exploratory graph transformation system, composed by graph transformation rules that capture the execution semantics of elements of the programming language. This exploration mechanism generates the state space of the given program graph as a GTS that (eventually) captures all possible paths of execution of the program. Usually, these generated GTS are prohibitively large, or even infinite. At this point abstraction techniques come into play, in order to produce a finite over-approximation of the original GTS. After producing a GTS, we can perform model checking against a given set of correctness properties that the program is expected to have. This check produces either a verdict that the program is indeed correct, or a counter-example of an execution path that produces an error. This counter-example can then be traced back to the ASG, or, better yet, the input code, so that the user can inspect the error. As an exploration/model checking engine we use GROOVE [10], a tool specifically developed to perform model checking of graph production systems.

^{*} The work reported herein is being carried out as part of the GRAIL project, funded by NWO (Grant 612.000.632).

Fig. 1. Overview of the verification cycle proposed.

Graph Abstractions. The key point of our verification method is the graph abstraction mechanism. In the context of graph transformation we have seen theoretical studies on suitable abstractions [12, 1, 6]. However, to the best of our knowledge only the last of these is backed up by an implementation, namely AUGUR [7]. Our graph abstractions are based on the concepts of shape analysis, proposed by Sagiv et al. [14], and of abstract interpretation developed by Cousot and Cousot [3]. A *graph shape* is an abstraction that captures the underlying structure of a set of concrete graphs, acting as their representative in the abstracted domain. Each node (resp. edge) of a graph shape is marked with a *multiplicity*, indicating how many nodes (resp. edges) must (or may) be present in a concrete graph. Previous work on graph abstractions were proposed by our group in [8, 11, 2]. One main issue with these abstractions is that they are unable to preserve important structural properties such as connectivity between nodes. This is a necessary information when analysing programs with heap-based data structures. Our current research is focused on elaborating good graph abstractions that keep the state space explosion under control while still allowing the verification of interesting properties on realistic programs.

Development. We chose Java as an initial programming language to handle, due to its wide-spread use. At the time of this writing we have a graph compiler that produces an abstract syntax graph from any legal Java program¹. The details of the construction of this compiler are presented in [9]. The flow construction transformation is performed in GROOVE using a graph grammar that defines the Java control flow semantics. The next step is the elaboration of the

¹ Available at <http://groove.cs.utwente.nl/downloads/java2groove/>

execution semantics of Java also in terms of graph transformation rules. This is future work. It should be noted that all the ingredients of our proposed approach were previously investigated and their feasibility analysed. How graph transformations can be used to capture the execution semantics of a toy programming language was shown in [5]. The construction of a control flow semantics specification for a part of Java was given in [15]. Nevertheless, whether the combination of these techniques will indeed provide good practical results when applied to reasonable sized programs is still to be seen.

References

1. Baldan, P., König, B., König, B.: A logic for analyzing abstractions of graph transformation systems. In: Static Analysis Symposium (SAS). LNCS, vol. 2694, pp. 255–272. Springer (2003)
2. Bauer, J., Boneva, I.B., Kurban, M.E., Rensink, A.: A modal-logic based graph abstraction. In: Ehrig et al. [4], pp. 321–335
3. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages (POPL). pp. 238–252 (1977)
4. Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.): International Conference on Graph Transformations (ICGT), LNCS, vol. 5214. Springer (2008)
5. Kastenberg, H., Kleppe, A., Rensink, A.: Defining object-oriented execution semantics using graph transformations. In: FMOODS. LNCS, vol. 4037, pp. 186–201. Springer (2006)
6. König, B., Kozioura, V.: Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In: TACAS. LNCS, vol. 3920, pp. 197–211. Springer (2006)
7. König, B., Kozioura, V.: Augur – a new version of a tool for the analysis of graph transformation systems. ENTCS 211, 201–210 (2008)
8. Rensink, A.: Canonical graph shapes. In: Programming Languages and Systems (ESOP). LNCS, vol. 2986, pp. 401–415. Springer Verlag, Berlin (2004)
9. Rensink, A., Zambon, E.: A type graph model for Java programs. In: FMOODS/FORTE 2009. LNCS, vol. 5522, pp. 237–242. Springer Verlag (2009)
10. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Applications of Graph Transformations with Industrial Relevance (AGTIVE). LNCS, vol. 3062, pp. 479–485. Springer (2003)
11. Rensink, A., Distefano, D.: Abstract graph transformation. Electr. Notes Theor. Comput. Sci. 157(1), 39–59 (2006)
12. Rieger, S., Noll, T.: Abstracting complex data structures by hyperedge replacement. In: Ehrig et al. [4], pp. 69–83
13. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. World Scientific (1997)
14. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
15. Smelik, R., Rensink, A., Kastenberg, H.: Specification and construction of control flow semantics. In: Visual Languages and Human-Centric Computing (VL/HCC), Brighton, U.K. pp. 65–72. IEEE Computer Society Press (September 2006)