Electronic Communications of the EASST
Volume 38 (2010)

Proceedings of the Fifth International Conference on Graph Transformation - Doctoral Symposium (ICGT-DS 2010)

Using Graph Transformations and Graph Abstractions for Software Verification

Eduardo Zambon and Arend Rensink

13 pages

Guest Editor: Andrea Corradini
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Using Graph Transformations and Graph Abstractions for Software Verification

Eduardo Zambon[*] **and Arend Rensink**

zambon@cs.utwente.nl, rensink@cs.utwente.nl
Formal Methods and Tools Group
Department of Computer Science
University of Twente, The Netherlands

Abstract: In this paper we describe our intended approach for the verification of software written in imperative programming languages. We base our approach on model checking of graph transition systems, where each state is a graph and the transitions are specified by graph transformation rules. We believe that graph transformation is a very suitable technique to model the execution semantics of languages with dynamic memory allocation. Furthermore, such representation allows us to investigate the use of graph abstractions, which can mitigate the combinatorial explosion inherent to model checking. In addition to presenting our planned approach, we reason about its feasibility, and, by providing a brief comparison to other existing methods, we highlight the benefits and drawbacks that are expected.

Keywords: Graph Abstraction, Graph Transformation, Software Model Checking, GROOVE

1 Introduction

The verification of software systems has already been a venerable concern for some computer scientists. However, given the ever-growing use of software in our society, and the consequent problems and damages due to programming errors, over the last years this concern became more wide-spread and hence, attracted more attention from researchers. As a natural consequence of this fact, several different methods and techniques have been proposed and are being studied. One recent key change in such verification techniques is the development of approaches that can analyse the correctness of software written in commonly used imperative programming languages. Previously, those methods were limited to the so-called “modeling languages”, which have a clean and simple definition and are thus more amenable to formal treatment. We can roughly divide current software verification techniques in two types: *deductive* or *exploratory*.

Deductive methods are mostly based on the principles defined by Floyd [Flo67] and Hoare [Hoa69], and later refined by Dijkstra [Dij76]. These methods rely on an axiomatic definition of the semantics of the programming language elements and on inference rules that allow one to reason about the desired correctness properties of a program in a compositional way. Among the tools developed under this approach we can cite the KeY System [BHS07], Why/Krakatoa

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

[FM07], jStar [DP08] and ESC/Java [FLL^{+}02] for the verification of Java programs, and the Spec# tool [BLS04] for analysing code written in a super-set of the C# language.

Exploratory methods try to (partially or exhaustively) enumerate the possible states of a program. A program state corresponds to a snapshot of the program dynamic structures in memory, e.g., heap, stack, threads locks and program counters, etc. The transitions between states are given by the execution semantics of the language in which the program is written. In fact, an exhaustive exploration mechanism can be seen as a non-deterministic machine that generates all possible execution paths of the input program and produces a transition system representing the program state space. A well known exploratory technique is model checking [CGP99, BK08], where the desired correctness properties of the program are checked to hold over a transition system. Among currently available software model checkers we can cite Java PathFinder (JPF) [VHBP00] and Bogor [DHHR05] for Java, and MoonWalker [BNR09] for C# programs.

The purpose of this paper is to present an overview of a new exploratory approach that we plan to develop for the verification of software. This approach is based on model checking of graph transition systems (GTS), where each program state is modeled as a graph and the exploration (execution) engine is specified by graph transformation rules. We believe that graph transformation [Roz97] 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. In fact, given the usual size of state spaces, abstraction is a key factor in the eventual success of the proposed verification approach.

The rest of this paper is organized as follows. In Section 2 we explain the underlying concepts of our approach, viz., graph transformations and graph abstractions. The planned verification approach is given in Section 3, and a discussion about its feasibility is presented in Section 4. Related work is given in Section 5, and Section 6 concludes the paper.

2 Concepts

In this section we present the two techniques on which we base our verification approach: graph transformation and graph abstraction. These techniques are very generic and can be used in various settings, thus we focus our presentation of the concepts to the relevant aspects of our problem at hand.

2.1 Graph Transformations

Graph transformation (or graph rewriting) is a rule-based transformation technique with a solid theoretical foundation [Roz97]. We use the term *graph production system* (GPS) to refer to a set of graph transformation rules. Each rule specifies both the conditions under which it can be applied and the changes to be performed. A transformation rule is composed of two graphs, a left hand side (LHS) and a right hand side (RHS), and is applied to a host graph, i.e., the graph to be transformed. In general terms, a rule is applied by searching for an image of the LHS on the host graph and by replacing the found image with a copy of the RHS. This search for a LHS image corresponds to the subgraph matching problem (see Section 4).

A GPS can be used to simulate the execution semantics of a programming language. We illustrate this with a simple example. Consider the snippet of Java code shown in Figure 1, which implements a circular buffer with three `Cells`. Each `Cell` has two fields: `next`, a reference to its adjacent position, and `val`, which can hold a reference to an `Object`. The state of a `Buffer` object in memory, immediately after the execution of its constructor, can be easily captured by a graph, depicted in Figure 2(a). In this representation, nodes stand for instances of classes, or primitive values, and edges represent the references (object fields). The `Buffer` class has two methods, `put` and `drop`.

Method `put` inserts the given argument `Object` after the current `last` element, provided that the `Buffer` is not full. Figure 2(c) shows the graph transformation rule that models the execution of the `put` method. The LHS defines the nodes and edges of the host graph involved in the transformation and provides the conditions for the rule application, i.e., `last.next.val == null`. The argument of the `put` method is assumed to have an arbitrary non-null value x. The RHS of the `put` rule establishes the effect of the rule application, i.e., the `next` pointer is moved and the argument value is stored.

Method `drop` discard the `first` element of a non-empty `Buffer`. The corresponding transformation rule is given in Figure 2(d). The node label $\{?x[!\texttt{null}]\}$ in the LHS can be seen as a regular expression that matches with any node in the host graph except the `null` node.

The exploration of all possible applications of the `put` and `drop` rules over the initial host graph shown in Figure 2(a) yields a graph transition system (GTS) that captures all possible states of a `Buffer` object. If we consider that the arbitrary value x of the argument of `put` is drawn from a finite set D, with cardinality $\#D = k$, the number n of states of the GTS in this example can be determined *a priory*, and is given by the formula $n = \frac{k^4-1}{k-1}$, for $k > 1$. Assuming that D represents the set of Java integers, we have that $k = 2^{32}$, and thus the value of n, even for such a small example, is already flabbergasting. In order to control such blow-up we need to abstract away irrelevant information.

2.2 Abstract Interpretation

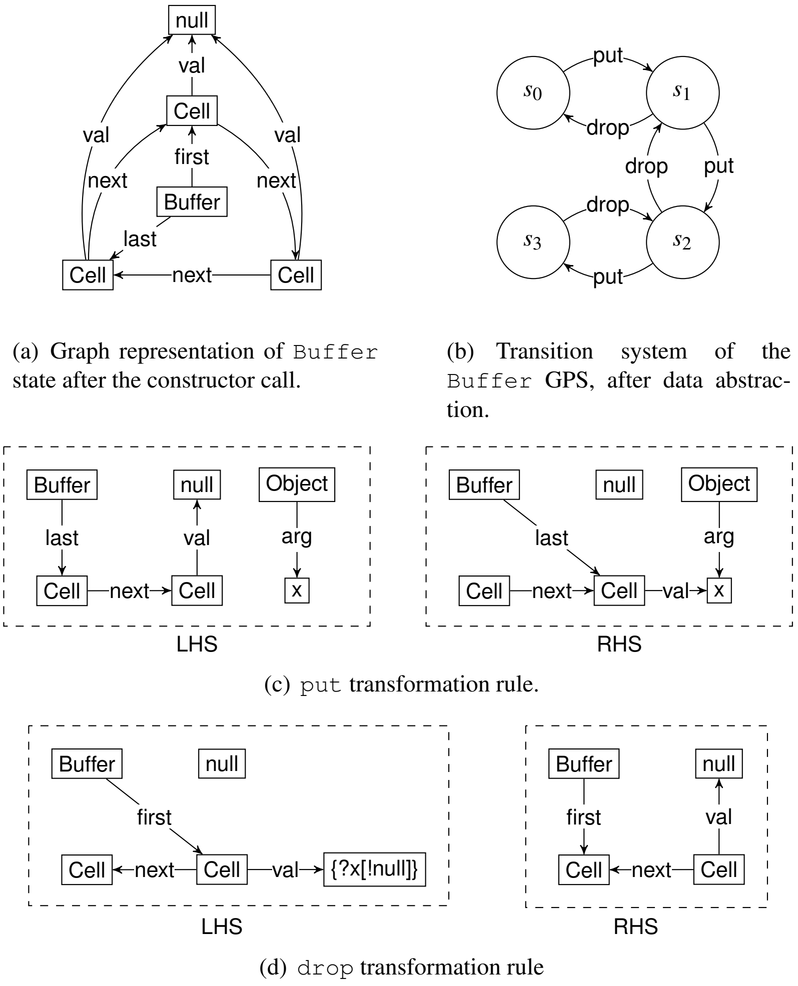
Continuing with the discussion of the previous section, the kind of information that can be considered irrelevant is dependent on the properties that one wishes to verify. For the circular buffer example, we might want to check if indeed no element is inserted if the buffer is full. To verify such property, it is not necessary to keep track of the concrete values stored in the buffer; it suffices to know that the values are `non-null`. This simplifies the GTS to only 4 states, shown in Figure 2(b), and makes the verification of p a trivial task.

The abstraction just described can be placed within the theory of abstract interpretation, developed by Cousot and Cousot [CC77]. An abstraction from a subset of concrete values C to an element of an abstract set A is given by an *abstraction function* $\alpha : 2^C \to A$, and conversely by a *concretization function* $\gamma : A \to 2^C$. The elements of C and A are required to be ordered in a lattice and α and γ must be monotonic with respect to this ordering. In our example, $C = D \cup \{\texttt{null}\}$, with an ordering $(\texttt{null} \sqsubseteq D)$; and $A = \{\bot, \texttt{null}, \texttt{non-null}, \top\}$, with an ordering $(\bot \sqsubseteq \{\texttt{null}, \texttt{non-null}\} \sqsubseteq \top)$. Let $B \subseteq 2^C$ and $a \in A$, the abstraction and concretization functions are defined as

```
public class Cell {
  public Object val;
  public Cell next;
}

public class Buffer {
  private Cell first, last;

  public Buffer() {
    first = new Cell();
    first.next = new Cell();
    first.next.next = new Cell();
    last = first.next.next;
    last.next = first;
  }
```

```
  public void put(Object arg) {
    if (last.next.val == null) {
      last = last.next;
      last.val = arg;
    }
  }

  public void drop() {
    if (first.val != null) {
      first.val = null;
      first = first.next;
    }
  }
}
```

Figure 1: A Java example of a circular buffer with three cells.



(a) Graph representation of `Buffer` state after the constructor call.

(b) Transition system of the `Buffer` GPS, after data abstraction.

(c) `put` transformation rule.

(d) `drop` transformation rule

Figure 2: (a), (c), and (d): Graph production system that captures the execution semantics of the Java program given in Figure 1. (b): A graph transition system, see Section 2.2.

$$\alpha(B) = \begin{cases} \bot & \text{if } B = \varnothing \\ \texttt{null} & \text{if } B = \{\texttt{null}\} \\ \texttt{non-null} & \text{if } B \subseteq D \\ \top & \text{otherwise} \end{cases} \qquad \gamma(a) = \begin{cases} \varnothing & \text{if } a = \bot \\ \{\texttt{null}\} & \text{if } a = \texttt{null} \\ D & \text{if } a = \texttt{non-null} \\ C & \text{otherwise} \end{cases}$$

The $\top$ value is the most coarse abstraction and represents every subset of C. On the other hand, $\bot$ is the most fine-grained approximation and thus maps to the empty set.

The key point of an abstract interpretation is that, with respect to the correctness properties that one wants to verify, the abstraction is an over-approximation of the concrete system. Thus, if a property holds on the abstract domain, it is guaranteed to hold on the concrete domain. This is the reason why we can check the correctness of the program of Figure 1 using the abstract GTS of Figure 2(b). However, it might be the case, due to loss of precision in the abstraction, that a property does not hold on the abstract domain but actually holds in the concrete domain, a so-called *false positive* error report.

2.3 Graph Abstractions

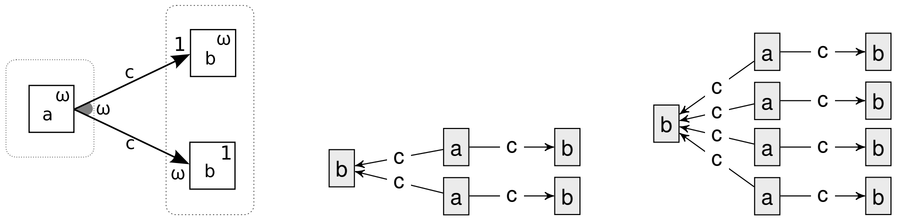
The example of the previous section is an interesting case of abstraction from the data domain of the program, which can be used to shrink the program state space to a reasonable size. However, such abstraction fails to cope with structures of unbounded size, e.g., a linked list. Regardless of data abstraction, the transition system of a such structure is infinite and thus unsuitable to exploratory verification methods without some previous manipulation. What we need is a method to deal with the graph structure, i.e., we need to abstract a possible infinite state graph to a finite graph.

A graph abstraction is based on the concepts of shape analysis, proposed by Sagiv et al. [SRW98, SRW02], and of abstract interpretation. 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. The basis of this technique falls within the same idea of abstract interpretation presented in the previous section, except that now the abstraction function maps a set of concrete graphs to a corresponding graph shape.

2.3.1 Neighbourhood abstraction

One possible type of graph abstraction is *neighbourhood abstraction*, which is based on neighbourhood similarity: two nodes are considered indistinguishable if they have the same incoming and outgoing edges, and the opposite ends of those edges are also comparable. Graphs are abstracted by folding all indistinguishable nodes into one, while keeping count of their original number up to some bound of precision. The incident edges are also combined. Counting up to some bound is done using *multiplicities*. We use $M_k = \{0,\ldots,k,\omega\}$ with $k \in \mathbb{N}$ consisting of exact numbers up to k (which is typically a low value such as 1 or 2) and the value ω standing for "many".

Formally, a graph is a tuple $G = \langle V, E \rangle$ of nodes and edges, where the edges are triples (v,a,w) of source node, label, and target node. The universe of possible labels is denoted by Lab. The abstract graphs (shapes) are 5-tuples $S = \langle G, \sim, \mathsf{mult}^{\mathsf{n}}, \mathsf{mult}^{\mathsf{o}}, \mathsf{mult}^{\mathsf{i}} \rangle$ in which



(a)

(b)

Figure 3: (a) A shape that represents a set of bipartite graphs. Two such concrete graphs are shown in (b).

- G is the underlying graph structure of the shape;
- $\sim \subseteq V \times V$ is a *neighbourhood similarity* relation;
- $\mathsf{mult}^{\mathsf{n}} : V \to M_\nu$ is a *node multiplicity* function, which records how many concrete nodes were folded into a given abstract node, up to bound ν;
- $\mathsf{mult}^{\mathsf{o}}, \mathsf{mult}^{\mathsf{i}} : (V \times \mathsf{Lab} \times V/\!\sim) \to M_\mu$ are outgoing and incoming *edge multiplicity* functions, which record how many edges with a certain label the concrete nodes had that were folded into an abstract node, up to a bound μ and a group of $\sim$-similar opposite nodes.

Figure 3(a) shows an example of a shape. This abstraction represents an infinite set of bipartite graphs with two or more a-labelled nodes and three or more b-labelled nodes connected by c-labelled edges. Two possible concrete graphs are shown in Figure 3(b). Details on the theory of neighbourhood abstraction can be found in [BBKR08] and practical aspects of the theory implementation are given in [RZ10].

3 Approach

Now that we have presented the most important underlying concepts, let us discuss how they can be combined in our intended approach for software verification. Figure 4 provides a picture of the whole verification cycle. The input is the program source code, written in some programming language, e.g., Java. 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, which 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.

We can now feed the program graph to an exploratory graph transformation system, composed by graph transformation rules that capture the execution semantics of the elements of the programming language, to exhaustively explore the state space of the program graph. This exploration produces a graph transition system (GTS) that captures all possible paths of execution

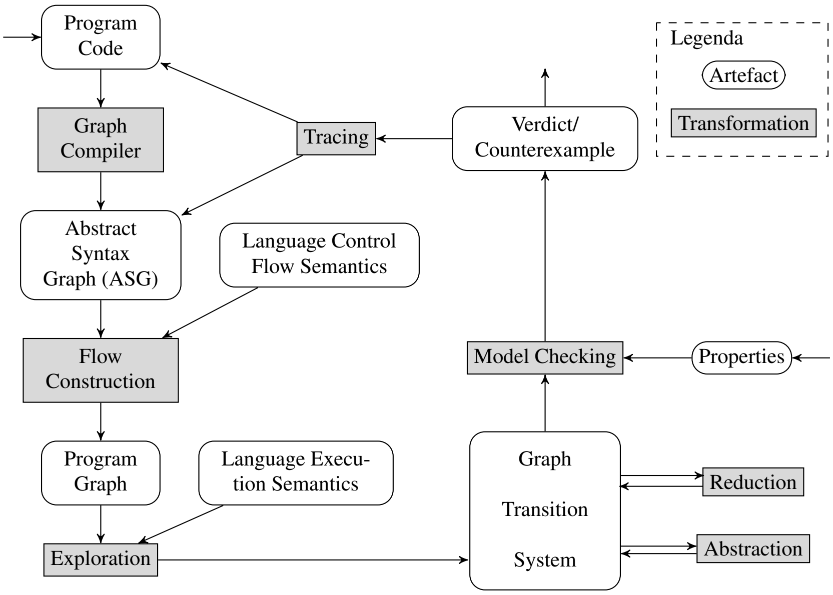


Figure 4: Overview of the verification cycle proposed.

of the program. However, as discussed in Section 2, a program GTS may be finite but still prohibitively large, or even infinite. At this point the previously presented abstraction techniques come into play, in order to produce a finite GTS. Additionally, we may profit from our representation of states as graphs and use a graph isomorphism algorithm to collapse isomorphic states into a single representative, thus reducing the size of the GTS. This is further discussed in Section 4.

After producing the 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 [Ren03], a tool specifically developed to perform model checking of graph production systems.

4 Discussion

As noted by Dwyer et al. [DHR+07], the combination of model checking with abstraction techniques is an emerging trend. The new idea in our proposal is to use graphs and graph transformations as a base of state representation and state space exploration. Although, at first, this may seem to introduce a lot of unnecessary complexity into an already complex problem, after a careful analysis we can see that this is not the case.

On the complexity of the subgraph matching problem. The general case of the subgraph matching problem is known to be NP-complete [GJ79]. However, there are several cases for which solving this problem is much easier. For example, in the case where the rules and the host graph contain a root label, Dodds and Plump [DP06] showed that the matching can be done in constant time. Even on the general case, heuristics can be used to improve the execution time of the matching algorithm. The work of Geiß et al. [GBG$^+$06] shows how the search plan heuristic can be used for this purpose. Another interesting improvement is described in the work of Bergmann et al. [BHRV08], which presents a solution with incremental pattern matching, where sets of matchings of graph transformation rules are stored and incrementally updated as the host graph changes. This idea was recently implemented in GROOVE [GJR10].

Isomorphism for state space reduction. As noted in Section 3, our choice of graphs to represent states allows us to perform symmetry reduction of the state space by using graph isomorphism algorithms. The graph isomorphism problem belongs to the NP complexity class but it is still not known whether the general case of the problem is solvable in polynomial time, or is NP-complete, or is neither. However, in practice the problem can often be solved efficiently [McK81]. Furthermore, this is an optional optimization in our verification cycle that can be switched off if the cost of isomorphism checking is higher than the actual gain on the state space reduction (which is the case if the expected symmetry is low).

The collapsing of states under isomorphism particularly pays off when the states representation of a problem have a high degree of symmetry. An interesting study on one of such cases was conducted by Crouzen, Van de Pol, and Rensink [CPR08]. In this study, the protocol of an ad-hoc self-configuring communication network was analysed with two model checking tools: GROOVE, based on graph transformations; and μCRL, based on process algebra. In that particular problem, the study showed that isomorphism symmetry checks can reduce the state space in several orders of magnitude. In [Ren07], Rensink presents the current implementation of isomorphism checking in GROOVE, based on graph certificates.

Drawbacks. Every verification method has strengths and weaknesses, and ours is no different. A problem with usual model checking techniques is the need of a “whole-world” model of the system. In our case, no provision is currently available for compositional verification, for example, of language libraries. Therefore, the implementation of the libraries elements that are used in the input program must also be given as input.

On capturing the program execution semantics as graph transformation rules. In our running example of the circular buffer (Figure 2) we presented graph transformation rules that simulate the execution of the whole bodies of methods `put` and `drop`. This was done to provide a clearer picture of the concepts that we use. However, it should be noted that it is not very practical to provide an automatic way to translate a whole method body to a single rule. Therefore, we work on a more fine-grained level: we provide transformation rules based on the execution semantics of the elements of the language, e.g., assignments, object creation, method invocation, etc. This once more stresses the analogy of an exploration mechanism and a virtual machine.

5 Related Work

The amount of research dedicated to software verification is enormous. Here we limit ourselves to the investigations that we consider most similar to our proposed approach.

The use of graph isomorphism for state symmetry reduction was investigated by Turner et al. [TLSB07], and Spermann and Leuschel [SL08], in the context of the ProB model checker. In their work, the internal model checker representation of a state was translated to a graph, that was then given to an external isomorphism checking tool: Nauty [McK81]. They report empirical results to show the effectiveness on the state space reduction. A large part of their work was devoted to the translation to a graph representation, a problem that we do not face. Other researchers address the problem of symmetry reduction directly over the internal state representation of their model checker of choice. This is the case of Lerda and Visser [LV01] for Java PathFinder, and Robby et al. [RDHI03] for Bogor.

The automatic extraction of a finite-state model from Java code for the purpose of model checking was addressed by Corbett et al. [CDH$^+$00], with the Bandera tool set. They also propose the use of data abstraction as one of the techniques for building tractable models for verification. Our approach could also benefit from their Slicer component, which removes variables and structures from the code that are not relevant for checking a certain property.

Anand, Păsăreanu and Visser [APV06] proposed the use of abstraction and shape analysis in the context of symbolic model checking. Again, a program state (captured by a symbolic heap configuration) was represented using a graph-based formalism. Their method for symbolic states subsumption and matching is quite similar to our proposal to use graph abstraction and graph shapes. A drawback of their approach is the need for code instrumentation, which in our case we believe will not be required. An interesting aspect of the work by these authors is that the use of symbolic execution allows for modular verification of compilation units, e.g., libraries. They implemented symbolic execution as an extension of JPF [APV07].

In the context of graph transformation we have seen several theoretical studies on suitable abstractions [Ren04, RD06, BBKR08, RN08, BKK03, KK06]. However, to the best of our knowledge, only two of these are backed up by an available implementation [RZ10, KK08]. The Augur tool [KK08] is based on the unfolding of Petri Graphs, and, as usual for unfolding techniques, can normaly only verify *safety* properties of a system. This points to an interesting aspect of our approach, which allows us to also check *liveness* properties.

Our proposed idea of translating Java source code to abstract syntax graphs was developed independently by Striewe, Balz and Goedicke [SBG10]. Their work differs mainly on the choice of the target tool: while we focused in creating input graphs for GROOVE, they chose to use AGG [Tae04], another graph transformation tool.

An interesting line of related research is being done by Poskitt and Plump [PP10], based on the work by Habel, Pennemann and Rensink [HPR06], which concerns the development of a Hoare-like calculus for the verification of graph transformation systems. Such work falls in category of deductive techniques that was introduced in the beginning of this paper, and allows one to reason over the correctness of the rules of a graph grammar.

6 Conclusion

In this paper we present a new approach for software verification that combines the techniques of graph transformation, model checking, and abstract interpretation. Other approaches that combine model checking and abstraction eventually are forced to provide a translation from an internal state representation to a graph-based one. We believe that this translation is often cumbersome and unattractive. The novelty in our method is to use an explicit representation of program states as graphs, and to rely on graph transformations as a computational engine. In doing so, we lift ourselves from the intricacies of a specific model checker implementation and we arrive at a very clean setting to study abstractions over program states.

The development of our approach is ongoing. We chose Java as an initial programming language to handle, due to its wide-spread use. At the moment, we have a graph compiler that produces an abstract syntax graph from any legal Java program[1]. The details of the construction of this compiler are presented in [RZ09]. In addition, the control flow semantics of Java has been translated into a graph grammar. The elaboration of the execution semantics of Java in terms of graph transformation rules is future work. Our current efforts are focused on the abstraction techniques. Details on the implementation of the neighbourhood abstraction of [BBKR08] can found in [RZ10]. In addition, we are working on a different graph abstraction that allows one to reason about structural properties of graphs.

It should be noted 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 programming language was shown in [KKR06]. The construction of a control flow semantics specification for a large part of Java was given in [SRK06]. Initial studies on graph abstraction techniques were proposed in [Ren04], [RD06] and [BBKR08]. Nevertheless, whether the combination of these techniques will indeed provide good practical results when applied to reasonable sized programs is still to be seen.

Bibliography

[APV06] S. Anand, C. S. Păsăreanu, W. Visser. Symbolic Execution with Abstract Subsumption Checking. In Valmari (ed.), *SPIN*. LNCS 3925, pp. 163–181. Springer, 2006.

[APV07] S. Anand, C. S. Păsăreanu, W. Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder. In Grumberg and Huth (eds.), *TACAS*. LNCS 4424, pp. 134–138. Springer, 2007.

[BBKR08] J. Bauer, I. B. Boneva, M. E. Kurban, A. Rensink. A Modal-Logic Based Graph Abstraction. Pp. 321–335 in [EHRT08].

[BHRV08] G. Bergmann, Á. Horváth, I. Ráth, D. Varró. A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. Pp. 396–410 in [EHRT08].

[BHS07] B. Beckert, R. Hähnle, P. H. Schmitt (eds.). *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[BK08] C. Baier, J. P. Katoen. *Principles of Model Checking*. MIT Press, New York, May 2008.

[BKK03] P. Baldan, B. König, B. König. A Logic for Analyzing Abstractions of Graph Transformation Systems. In Cousot (ed.), *Static Analysis Symposium (SAS)*. LNCS 2694, pp. 255–272. Springer, 2003.

[1] Available at http://groove.cs.utwente.nl/downloads/java2groove/.

[BLS04] M. Barnett, K. R. M. Leino, W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004, LNCS vol. 3362*. Pp. 49–69. Springer, 2004.

[BNR09] N. A. de Brugh, V. Y. Nguyen, T. Ruys. MoonWalker: Verification of .NET Programs. In Kowalewski and Philippou (eds.), *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009), York, UK*. LNCS 5505, pp. 170–173. Springer Verlag, Berlin, 2009.

[CC77] P. Cousot, R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. Pp. 238–252. 1977.

[CDH$^+$00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, H. Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE*. Pp. 439–448. 2000.

[CEM$^+$06] A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg (eds.). *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*. LNCS 4178. Springer, 2006.

[CGP99] E. M. Clarke, O. Grumberg, D. A. Peled. *Model Checking*. The MIT Press, 1999.

[CPR08] P. Crouzen, J. C. van de Pol, A. Rensink. Applying Formal Methods to Gossiping Networks with mCRL and GROOVE. *ACM SIGMETRICS performance evaluation review* 36(3):7–16, December 2008.

[DHHR05] M. B. Dwyer, J. Hatcliff, M. Hoosier, Robby. Building Your Own Software Model Checker Using the Bogor Extensible Model Checking Framework. In Etessami and Rajamani (eds.), *CAV*. LNCS 3576, pp. 148–152. Springer, 2005.

[DHR$^+$07] M. B. Dwyer, J. Hatcliff, Robby, C. S. Păsăreanu, W. Visser. Formal Software Analysis Emerging Trends in Software Model Checking. In *FOSE '07: 2007 Future of Software Engineering*. Pp. 120–136. IEEE Computer Society, Washington, DC, USA, 2007.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DP06] M. Dodds, D. Plump. Graph Transformation in Constant Time. Pp. 367–382 in [CEM$^+$06].

[DP08] D. Distefano, M. J. Parkinson. jStar: towards practical verification for Java. In Harris (ed.), *OOPSLA*. Pp. 213–226. ACM, 2008.

[EHRT08] H. Ehrig, R. Heckel, G. Rozenberg, G. Taentzer (eds.). *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*. LNCS 5214. Springer, 2008.

[FLL$^+$02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. PLDI '02, pp. 234–245. ACM, New York, NY, USA, 2002.

[Flo67] R. W. Floyd. Assigning Meanings to Programs. *Proceedings of the Symposium on Applied Mathematics* 19(10):19–32, 1967.

[FM07] J.-C. Filliâtre, C. Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In Damm and Hermanns (eds.), *CAV*. LNCS 4590, pp. 173–177. Springer, 2007.

[GBG$^+$06] R. Geiß, G. V. Batz, D. Grund, S. Hack, A. Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. Pp. 383–397 in [CEM$^+$06].

[GJ79] M. R. Garey, D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, January 1979.

[GJR10] A. H. Ghamarian, A. Jalali, A. Rensink. Incremental Pattern Matching in Graph-Based State Space Exploration. In [LV10].

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12(10):576–580, 1969.

[HPR06] A. Habel, K.-H. Pennemann, A. Rensink. Weakest Preconditions for High-Level Programs. Pp. 445–460 in [CEM$^+$06].

[KK06] B. König, V. Kozioura. Counterexample-Guided Abstraction Refinement for the Analysis of Graph Transformation Systems. In *TACAS*. LNCS 3920, pp. 197–211. Springer, 2006.

[KK08] B. König, V. Kozioura. AUGUR — A New Version of a Tool for the Analysis of Graph Transformation Systems. *ENTCS* 211:201–210, 2008.

[KKR06] H. Kastenberg, A. Kleppe, A. Rensink. Defining Object-Oriented Execution Semantics Using Graph Transformations. In Gorrieri and Wehrheim (eds.), *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*. LNCS 4037, pp. 186–201. Springer, 2006.

[LV01] F. Lerda, W. Visser. Addressing dynamic issues of program model checking. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*. Pp. 80–102. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[LV10] J. de Lara, D. Varro (eds.). *Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)*. Electronic Communications of the EASST 32. European Association for the Study of Science and Technology, 2010.

[McK81] B. D. McKay. Practical Graph Isomorphism. *Congressus Numerantium* 30:45–87, 1981.

[PP10] C. M. Poskitt, D. Plump. A Hoare Calculus for Graph Programs. In Ehrig et al. (eds.), *ICGT*. LNCS 6372, pp. 139–154. Springer, 2010.

[RD06] A. Rensink, D. Distefano. Abstract Graph Transformation. *Electr. Notes Theor. Comput. Sci.* 157(1):39–59, 2006.

[RDHI03] Robby, M. B. Dwyer, J. Hatcliff, R. Iosif. Space-Reduction Strategies for Model Checking Dynamic Software. *Electr. Notes Theor. Comput. Sci.* 89(3), 2003.

[Ren03] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Pfaltz et al. (eds.), *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. LNCS 3062, pp. 479–485. Springer, 2003.

[Ren04] A. Rensink. Canonical Graph Shapes. In Schmidt (ed.), *Programming Languages and Systems (ESOP)*. LNCS 2986, pp. 401–415. Springer Verlag, Berlin, 2004.

[Ren07] A. Rensink. Isomorphism Checking in GROOVE. In Zündorf and Varró (eds.), *Graph-Based Tools (GraBaTs), Natal, Brazil*. Electronic Communications of the EASST 1. European Association of Software Science and Technology, September 2007.

[RN08] S. Rieger, T. Noll. Abstracting Complex Data Structures by Hyperedge Replacement. Pp. 69–83 in [EHRT08].

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[RZ09] A. Rensink, E. Zambon. A Type Graph Model for Java Programs. In *FMOODS/FORTE 2009*. LNCS 5522, pp. 237–242. Springer Verlag, 2009.

[RZ10] A. Rensink, E. Zambon. Neighbourhood Abstraction in GROOVE. In [LV10].

[SBG10] M. Striewe, M. Balz, M. Goedicke. Enabling Graph Transformations on Program Code. In Lara and Varro (eds.), *Pre-Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)*. CTIT Workshop Proceedings WP 10-06. Centre for Telematics and Information Technology, University of Twente, Enschede, 2010.

[SL08] C. Spermann, M. Leuschel. ProB gets Nauty: Effective Symmetry Reduction for B and Z Models. In *TASE*. Pp. 15–22. IEEE Computer Society, 2008.

[SRK06] R. Smelik, A. Rensink, H. Kastenberg. Specification and Construction of Control Flow Semantics. In Grundy and Howse (eds.), *Visual Languages and Human-Centric Computing (VL/HCC), Brighton, U.K.* Pp. 65–72. IEEE Computer Society Press, Los Alamitos, September 2006.

[SRW98] S. Sagiv, T. W. Reps, R. Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. *ACM Trans. Program. Lang. Syst.* 20(1):1–50, 1998.

[SRW02] S. Sagiv, T. W. Reps, R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24(3):217–298, 2002.

[Tae04] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Applications of Graph Transformations with Industrial Relevance, (AGTIVE)*. LNCS 3062, pp. 446–453. Springer, 2004.

[TLSB07] E. Turner, M. Leuschel, C. Spermann, M. J. Butler. Symmetry Reduced Model Checking for B. In *TASE*. Pp. 25–34. IEEE Computer Society, 2007.

[VHBP00] W. Visser, K. Havelund, G. P. Brat, S. Park. Model Checking Programs. In *ASE*. Pp. 3–12. 2000.