Abstract Interpretation of Graph Grammars ⋆

Jörg Bauer and Reinhard Wilhelm

Informatik; Univ. des Saarlandes; Saarbrücken, Germany.
{*joba,wilhelm*}@*cs.uni-sb.de*

Abstract. Many communication systems, distributed algorithms, or heap manipulating programs are hard to verify due to their inherent unboundedness. Their semantics can be described by evolving graphs. Graph grammars are a natural, intuitive, and formally defined method to specify systems of evolving graphs, whereas verification techniques for them are scarce.
We present an abstract interpretation based approach for graph grammar verification. Single graphs are abstracted in two steps. First similar nodes within a connected component, then similar abstracted connected components are summarized. Transformation rules are applied directly to abstract graphs yielding a bounded set of abstract graphs of bounded size that over-approximates the concrete graph grammar semantics and can be used for further verification. Since our abstraction is homomorphic, existential positive properties are preserved under abstraction. Furthermore, we identify automatically checkable completeness criteria for the abstraction. The technique is implemented and successfully tested on the platoon case study.

1 Extended Abstract

We propose a new verification technique for systems with an unbounded number of dynamically created, stateful, linked objects. Prominent examples of such systems are communication systems with an unbounded communication topology, heap manipulating programs, distributed algorithms, and ad-hoc networks. All of them prove hard to verify. They are intuitively modeled using graph grammars, because a state of such a system can be modeled as a node-labeled graph, hence the evolution of graphs as a graph grammar. Graph grammars come in many flavors, and we refer to [1] for an overview.

A graph grammar consists of a set $\mathcal{R}$ of *transformation rules* and an initial graph. For now, the latter is fixed to be the empty graph E. Generally, the semantics $[\![\mathcal{R}]\!]$ of $\mathcal{R}$, *i.e.* all graphs generated by application of $\mathcal{R}$ starting from E, is an infinite set of graphs of unbounded size. Therefore, properties of $[\![\mathcal{R}]\!]$ are hard to verify.

We aim at automatically computing a bounded over-approximation $[\![\mathcal{R}]\!]^\alpha$ of $[\![\mathcal{R}]\!]$. Our technique is based on the abstract interpretation framework of [2]. In abstract interpretation, two things need to be defined. First, an *abstraction* $\alpha(G)$ for a single graph

⋆ This work was supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `www.avacs.org` for more information.

Dagstuhl Seminar Proceedings 06161
Simulation and Verification of Dynamic Systems
http://drops.dagstuhl.de/opus/volltexte/2006/703

G; second, an *abstract transformer* $\rightsquigarrow^\alpha$ mimicking the concrete transformer $\rightsquigarrow$. The latter corresponds to the application of a transformation rule.

The abstraction $\alpha(G)$ of graph G is computed in two steps. The first step is applied *connected component-wise*. In each connected component of G, two nodes are mapped to a *summary node*, if they have the same label, *and* if the set of labels of their respective adjacent nodes are equal. After that, isomorphic abstract connected components are summarized. The first step of this abstraction was inspired by the canonical abstraction of [3].

An abstract transformation is performed by applying a concrete rule of $\mathcal{R}$ directly to an abstract graph. To do so, the abstraction is locally undone to enable matching as in concrete graphs. This process is called *materialization*. Materialized graphs are transformed exactly like concrete graphs. However, the graph resulting from this concrete update needs to be abstracted again to guarantee the boundedness of its size. This implies that our technique computes the over-approximation $[\![\mathcal{R}]\!]^\alpha$ fully automatically given only the set $\mathcal{R}$ of graph transformation rules. Interesting properties of the technique include:

- *Soundness*: $\alpha(G) \in [\![\mathcal{R}]\!]^\alpha$ for each $G \in [\![\mathcal{R}]\!]$
- The abstraction preserves *exactly* the applicability of rules for a restricted set of rules and a precisely specified class of abstract graphs. In this case, a rule is applicable to some $\alpha(G)$ *if and only if* it is applicable to all graphs H with $\alpha(H) = \alpha(G)$.
- The abstraction is homomorphic thus preserving existential positive properties: A graph that does not occur as a subgraph in $[\![\mathcal{R}]\!]^\alpha$ will never occur as a subgraph of any $G \in [\![\mathcal{R}]\!]$.
- *Completeness*: three statically checkable notions of *completeness* are identified for our technique; *i.e.* given $[\![\mathcal{R}]\!]^\alpha$, we can decide whether it is complete under these three notions of completeness. Completeness results are rare in abstract interpretation. Surprisingly, our analysis is able to decide the word problem for a certain class of graph grammars.

Our work is in a line of research with, *e.g.*, [4–7], where the first one is closest to our work. It is the only other approach applying transformation rules directly to abstract graphs. It was developed independently of our approach. The underlying abstraction [8] relies on counting incoming and outgoing labeled edges, *i.e.* it is rather edge-centric compared to our node-centric approach. The idea of materialization-update-abstract occurs in [4], too, and the approach does not need unconstrained node creation. However, our approach provides clear advantages. We are not restricted to deterministic graphs and provide a more refined notion of summary nodes. Most importantly, our layered abstraction is more precise, when it comes to systems of many connected components. We show stronger results like criteria for exact preservation of matches and even completeness, and our technique is implemented. With respect to the considered graph grammars, we are able to include a restricted form of negative application conditions, which is not possible in the approach of [4].

Case Study: Car Platooning Our case study is taken from the California PATH project [9], the relevant part of which is concerned with cars driving on a highway. In order to make better use of the given space, cars heading for the same direction are supposed to

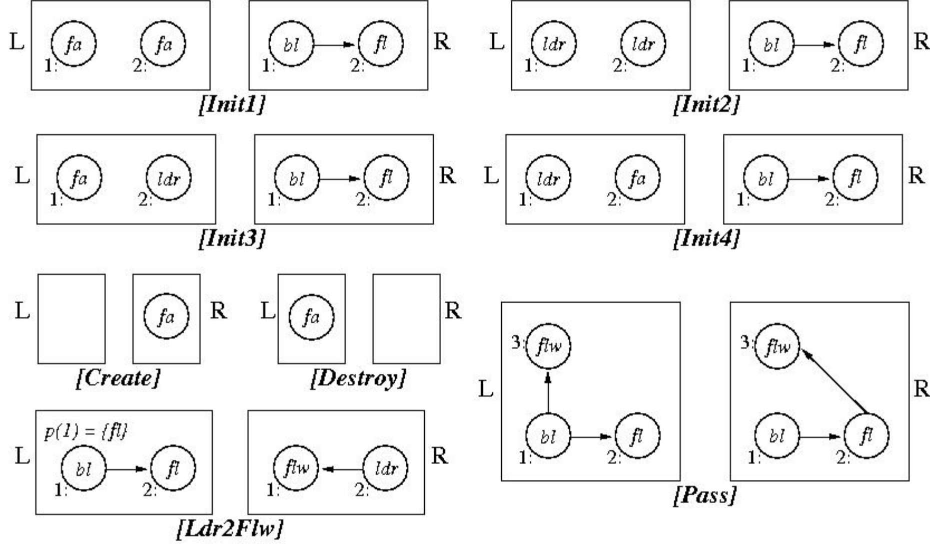

Fig. 1. The GTS $\mathcal{R}_{merge}$ for Platoon Merge. [*Create*] and [*Destroy*] model free agents driving onto and off the highway; [*Init1-4*] initiate a merge; [*Pass*] hands over followers, and [*Ldr2Flw*] makes the back leader a follower. This rule shows an instance of a *partner constraint*.

drive very near to each other building *platoons*. Platoons can perform actions like merging, splitting, or letting cars in and out of a platoon. There are many features that make verification difficult: destruction and dynamic creation (*i.e.*, driving onto and off the highway) of cars, an evolving communication topology, concurrency. All the verification methods developed in [9] are inappropriate, because they consider static scenarios with a fixed number of cars and only limited concurrency.

A *platoon* consists of a *leader*, the foremost car, along with a number of *followers*. A leader without any followers is called *free agent* and is considered a special platoon. Within a platoon there are communication channels between the leader and each of its followers. Inter-platoon communication is only between leaders. As a running example the platoon *merge maneuver* is studied. It allows two approaching platoons to merge. The merge maneuver is initiated by opening a channel between two distinct platoon leaders, *i.e.* leaders or free agents. Then, the rear leader passes its followers one by one to the front leader. Finally, when there are no followers left to the rear leader, it becomes itself a follower to the front leader.

Platoons as GTS The merge maneuver is straightforward to model as GTS with cars modeled as nodes, channels as edges, and internal states of cars as node labels. The GTS $\mathcal{R}_{merge}$ implementing the merge maneuver is given in Figure 1. We refer to this figure for an intuitive understanding. We employ five node labels in $\mathcal{R}_{merge}$. Three of the labels – *ldr*, *flw*, and *fa*– represent the states of a car being a leader, follower, or free agent as described above. Two labels – *bl* and *fl* – are used to model situations during

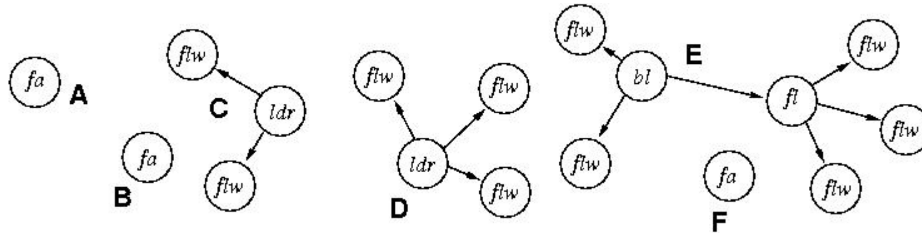

Fig. 2. A graph generated by $\mathcal{R}_{merge}$

a merge maneuver. They distinguish the leader of the back and the front platoon in a merge. Note that the physical position of platoons is not modeled in the GTS.

The rules [*Create*] and [*Destroy*] model free agents driving onto and off the highway. The [*Init1-4*] rules model the initiation of a merge. Followers are handed over from the back to the front leader in rule [*Pass*]. Eventually, after passing all followers, the rule [*Ldr2Flw*] can be applied yielding another platoon. It makes use of a partner constraint requiring the back leader not to have any followers left. Figure 2 shows a sample graph generated by $\mathcal{R}_{merge}$. Subgraphs **A**, **B**, and **F** are free agent platoons. **C** and **D** are well-formed platoons of three and four cars, respectively, whereas **E** depicts a snapshot during a merge maneuver.

For the GTS $\mathcal{R}_{merge}$ our analysis is complete in the strongest sense, *i.e.* we find a bounded abstraction describing exactly all possible states on a highway. We are able to show that all but two examples from our platoon benchmark are complete in the strongest sense. One of the non-complete examples is corrupted, the other generates platoons of even size.

References

1. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. World Scientific (1997)
2. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In: Symp. on Princ. of Prog. Lang., New York, NY, ACM Press (1977) 238–252
3. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems **24**(3) (2002) 217–298
4. Rensink, A., Distefano, D.: Abstract graph transformation. In: International Workshop on Software Verification and Validation. (2005)
5. König, B., Kozioura, V.: Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In: Proc. of TACAS '06, Springer (2006) LNCS. to appear.
6. Baldan, P., Corradini, A., König, B.: Verifying finite-state graph grammars: An unfolding-based approach. In Gardner, P., Yoshida, N., eds.: CONCUR. Volume 3170 of Lecture Notes in Computer Science., Springer (2004) 83–98
7. Heckel, R.: Compositional verification of reactive systems specified by graph transformation. In: FASE. (1998) 138–153
8. Rensink, A.: Canonical graph shapes. In Schmidt, D.A., ed.: ESOP. Volume 2986 of Lecture Notes in Computer Science., Springer (2004) 401–415
9. Hsu, A., Eskafi, F., Sachs, S., Varaiya, P.: The design of platoon maneuver protocols for IVHS. Technical Report UCB-ITS-PRR-91-6, University of California, Berkley (1991)