Hyperedge Replacement Grammars for Lock-sensitive Analysis of Parallel Programs

Sebastian Kenter

Institut für Informatik, Westfälische Wilhelms-Universität, Germany
`s.k@wwu.de`

Abstract. Reachability problems for parallel programs in the presence of locks are often close to the border of decidability. We consider Dynamic Pushdown Networks modelling parallel programs with unboundedly many locks and use Hyperedge Replacement Grammars (HRGs) to analyze the graph structures that represent their executions, exploiting the decidability of the emptiness of a set of graphs generated by an HRG and satisfying certain logical formulas.

Keywords: parallel programs, dynamic pushdown networks, locking, graph grammars

1 Introduction

More and more parallel programs are being used in many areas, and it can be very important to find out if they work properly. Thus we analyze programs with multiple threads and locking, and we are interested in the *lock-sensitive reachability problem* which asks if there is a feasible execution in this setting reaching a state from a given set. Such problems are often close to the border of decidability, and many methods together with restrictions of the general setting have already been developed for solving them. We now analyze programs with unboundedly many locks, which has not been done by prior work in this field. Our approach is to prove decidability by looking at certain graph structures arising from these kinds of programs, and we use Hyperedge Replacement Grammars (HRGs) as a tool to construct those graphs.

2 Program model

The program model we consider in our analysis is based on *Dynamic Pushdown Networks (DPNs)*, which already have been subject to analyses developed in the past (e.g. [1]). A program execution consists of several threads and is run on a state machine with a pushdown for each thread, simulating a call stack for procedures with unbounded recursion depth. New threads can be created dynamically, which leads to unboundedly many threads in this model. Synchronisation between the threads is realized via locking, which is required to abide certain restrictions here:

- Each lock can be *acquired* (once) and *released* later by the same thread.
- As usual, no thread can acquire a lock while it is acquired by another thread.
- At the beginning of an execution, no lock is acquired.
- Whenever a thread acquires another lock l_2 after a lock l_1, it afterwards has to release l_2 before it releases l_1 (*nested locking*).

As a novel extension to this model, called *n-Fold Locking Dynamic Pushdown Network (nLDPN)*, we also allow the dynamic creation of locks, so that the number of locks is unbounded as well. In order to capture the typical situation in common programming languages while keeping the model as simple as possible, we store the locks in a fixed number of procedure-local variables and implement mechanisms to pass them over between procedures.

Formally, an nLDPN is a tuple $M = (V, \mathsf{Act}, P, \Gamma, \Delta, p_0, \gamma_0)$ where

- $V = \{v_1, \dots, v_n\}$ is a finite set of *variables*,
- $\mathsf{Act} = \mathsf{LockOp} \cup \mathsf{Ass}$ where $\mathsf{LockOp} = \{\mathrm{acq}\, v, \mathrm{rel}\, v \mid v \in V\} \cup \{\varepsilon\}$, $\mathsf{Ass} = \{V \to (V \cup \{\mathsf{new}\})\}$,
- P is a finite set of *control states* with *initial state* $p_0 \in P$,
- Γ is a finite set of *stack symbols* with *initial symbol* $\gamma_0 \in \Gamma$,
- and Δ is a finite set of rules of one of the following forms:

$$\begin{array}{ll}
(\mathsf{base}) & p\gamma \overset{lo}{\hookrightarrow} p'\gamma' \\
(\mathsf{push}) & p\gamma \overset{a,\omega}{\hookrightarrow} p'\gamma_1\gamma_2 \\
(\mathsf{pop}) & p\gamma \overset{\varepsilon}{\hookrightarrow} p' \\
(\mathsf{spawn}) & p\gamma \overset{a}{\hookrightarrow} p'\gamma' \triangleright p_s\gamma_s
\end{array}$$

Here $lo \in \mathsf{LockOp}, a \in \mathsf{Ass}$, and ω is a *return value mapping* with signature $V \to (V \cup \{\mathsf{void}\})$.

This model allows for assigning either the caller's variable values or newly created lock objects (new) to the procedure's local variables on procedure call (push) and either overwriting the caller's variables with return values or or not (void) on procedure return (pop). The locks are acquired and released via the names of the variables that store them using the operations $\mathrm{acq}\, v$ and $\mathrm{rel}\, v$, respectively. The semantics, which is not given here in detail, is based on an infinite pool of locks and keeps track of the state, stack content, and variable assignment for each thread, as well as the state of each lock. Because the variables are procedure-local, the assignments are stored on the pushdowns along with the stack symbols so that the caller's values can be accessible again after a procedure returns.

3 Execution Trees

The graph structures that reflect the behaviour of nLDPNs, while abstracting away from the locking mechanism, are called *Execution Trees*. An Execution

Tree is defined such that each vertex represents a program state of a single thread and the paths show all threads' executions that are possible in a given nLDPN. The edges of the tree are labelled with information on the transitions performed by the executions of the threads, taken from the set Δ. Branching in the tree is caused by spawn rules, so that the spawned thread is represented by a different branch, and also by push rules, in which case one branch shows the steps executed up to the corresponding pop operation (i.e., the execution of the "called procedure") and the other one shows everything that happens in the same thread after that (given the called procedure terminates).

Figure 1 shows an example execution of an nLDPN with $P = \{p_0, \ldots, p_9\}, \Gamma = \{\gamma, \gamma', \gamma''\}$, and Δ consisting of the following $\mathsf{base}, \mathsf{push}$ and pop rules:

$$\begin{array}{lll} p_0\gamma \hookrightarrow p_1\gamma & p_4\gamma'' \hookrightarrow p_5 & p_7\gamma \overset{\mathrm{rel}\,v_1}{\hookrightarrow} p_8\gamma \\ p_1\gamma \overset{a_1}{\hookrightarrow} p_2\gamma'\gamma & p_5\gamma' \overset{\mathrm{acq}\,v_1}{\hookrightarrow} p_6\gamma' & p_8\gamma \overset{\mathrm{acq}\,v_1}{\hookrightarrow} p_9\gamma \\ p_3\gamma' \overset{a_2}{\hookrightarrow} p_4\gamma''\gamma' & p_6\gamma' \hookrightarrow p_7 & p_{10}\gamma' \hookrightarrow p_{11}\gamma' \end{array}$$

and the following spawn rule:

$$p_2\gamma' \overset{a_{sp}}{\hookrightarrow} p_3\gamma' \rhd p_{10}\gamma'$$

We furthermore assume the initial assignment $v_1 \mapsto l_1$, let $a_1(v_1) = a_2(v_1) = a_{sp}(v_1) = v_1$, and do not consider return value mappings here.

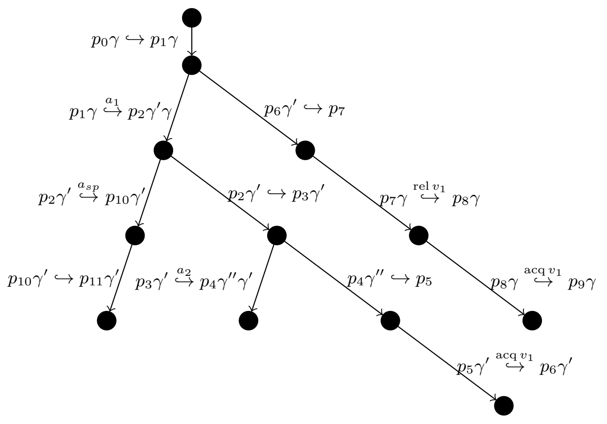
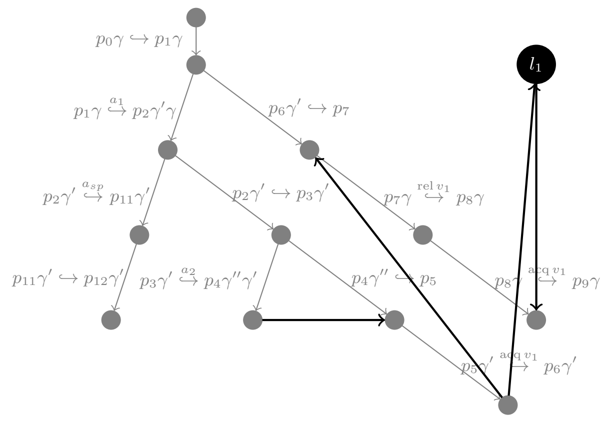

$p_0\gamma \hookrightarrow p_1\gamma$
$p_1\gamma \overset{a_1}{\hookrightarrow} p_2\gamma'\gamma$
$p_6\gamma' \hookrightarrow p_7$
$p_2\gamma' \overset{a_{sp}}{\hookrightarrow} p_{10}\gamma'$
$p_2\gamma' \hookrightarrow p_3\gamma'$
$p_7\gamma \overset{\mathrm{rel}\,v_1}{\hookrightarrow} p_8\gamma$
$p_{10}\gamma' \hookrightarrow p_{11}\gamma'$
$p_3\gamma' \overset{a_2}{\hookrightarrow} p_4\gamma''\gamma'$
$p_4\gamma'' \hookrightarrow p_5$
$p_8\gamma \overset{\mathrm{acq}\,v_1}{\hookrightarrow} p_9\gamma$
$p_5\gamma' \overset{\mathrm{acq}\,v_1}{\hookrightarrow} p_6\gamma'$

Fig. 1. Example Execution Tree.

The existence of a well-formed Execution Tree showing in its leaves a certain nLDPN configuration (i.e., control states and stack content for each thread) proves that this configuration would be reachable from the considered initial configuration in the considered nLDPN if there were no locking instructions. However, in presence of locking, one additionally has to find a *lock-sensitive schedule*, i.e. a global ordering of all the steps given in a tree such as to avoid deadlock situations, and those trees where this is impossible (which are not *schedulable*) have to be ruled out. In order to take the this into account, we enrich the trees by the following elements:

- return edges on terminating push branches
- an additional vertex l for each lock $l \in L$
- an additional edge $v \to l$ if l is non-finally acquired at v
- an additional edge $l \to v$ if l is finally acquired at v

By the term *final acquisition* we mean an acquisition of a lock that is never followed by a release of this lock in the considered tree. These enriched structures are called *Augmented Execution Trees (AETs)* (although they are no longer trees). The augmentation arising in our running example is shown in figure 2.



$p_0\gamma \hookrightarrow p_1\gamma$
$p_1\gamma \overset{a_1}{\hookrightarrow} p_2\gamma'\gamma$
$p_6\gamma' \hookrightarrow p_7$
$p_2\gamma' \overset{a_{sp}}{\hookrightarrow} p_{11}\gamma'$
$p_2\gamma' \hookrightarrow p_3\gamma'$
$p_7\gamma \overset{\text{rel } v_1}{\hookrightarrow} p_8\gamma$
$p_{11}\gamma' \hookrightarrow p_{12}\gamma'$
$p_3\gamma' \overset{a_2}{\hookrightarrow} p_4\gamma''\gamma'$
$p_4\gamma'' \hookrightarrow p_5$
$p_8\gamma \overset{\text{acq } v_1}{\hookrightarrow} p_9\gamma$
$p_5\gamma' \overset{\text{acq } v_1}{\hookrightarrow} p_6\gamma'$
l_1

Fig. 2. Augmentation for the example Execution Tree.

In such a graph all edges between program states represent happens-before relations, and with the help of the other vertices and edges one can tell rather plainly if these temporal constraints allow a lock-sensitive schedule of the tree: We can show that an AET has a lock-sensitive schedule if and only if there is not more than one final acquisition of the same lock and the graph is acyclic.

4 A criterion for decidability

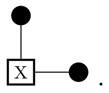
Our main statement now turned into the decidability of the existence of an AET satisfying the above-mentioned conditions. We achieve this by identifying certain structural properties of AETs, which are edge-labelled directed graphs, and expressing the additional conditions in *Monadic Second-Order Logic (MSO)* formulas. MSO formulas on graphs view the graphs as logical structures with vertex set V and edge relations $E_a \subseteq V \times V$ for each label a. Quantification is allowed over vertices $v \in V$ as well as subsets $V' \subseteq V$. We can express the relevant constraints, among those the configuration which has to be reached and acyclicity, in this language. Decidability of MSO satisfiability on graphs has been proven for certain cases. Courcelle and Engelfriet state such a result for so-called *HR-equational sets* of graphs ([2], p. 408, theorem 5.80 (2)), which consist of graphs that can be defined in a certain way by specific algebraic graph operations. One possibility to create such a set of graphs is to specify a *Hyperedge Replacement Grammar (HRG)* of graphs.

5 Hyperedge Replacement Grammars

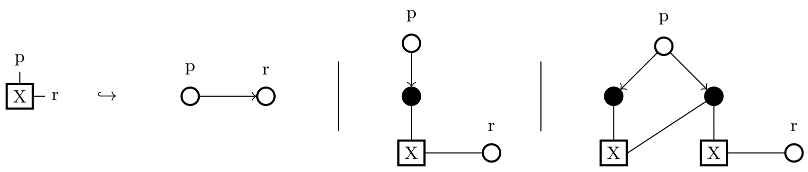
HRGs can be seen as generalizations of context-free grammars of words; they generate HR-equational sets of graphs. We use them also for making sure that the operations given in an AET are consistent to the states and transitions of the nLDPN under consideration.

The non-terminals of an HRG are *hyperedges*, i.e. edges connecting an arbitrary number of vertices via destinguished connectors. The production rules contain on their right-hand sides graphs with possibly inserted non-terminals and destinguished *external vertices* matching the connectors on the left-hand side of the rule. The rules specify how a terminal graph can be derived (in finitely many steps, similarly to context-free grammars) from some initial graph that contains non-terminals. In each derivation step, a non-terminal is replaced by the right-hand side of a corresponding rule, where each external vertex in turn is replaced by the vertex that is referenced by the corresponding connector of the non-terminal.

We show an example HRG with one non-terminal called X having two connectors called v and r that generates tree-like graphs with "return edges" connecting the two branches as if the branching vertices were push operations in an AET, out of the initial structure



Its three production rules are (in a compact representation):



The rules express the termination of a branch, linear continuation, and branching, respectively. The external vertices are depicted without filling and annotated with the name of the connector whose referenced vertex should replace it in a derivation step. This way, each newly introduced vertex has a "previous vertex" (p) and a "return vertex" (r), which always has to be specified in this example, even if there is no branching.

The transition rules of a given nLDPN can be translated into production rules like these, complemented by edge labels, vertices for the locks and other things, in order to create corresponding AETs.

6 Overview and conclusion

To summarize, we show that the lock-sensitive reachability problem for nLDPNs is decidable by reducing it to MSO satisfiability on HR-relational sets of graphs: We transform the given nLDPN into an HRG generating the corresponding AETs and we specify further conditions by MSO formulas. This establishes a new decidability result for programs with unboundedly many locks in the shape of a flexible and practical program model. We also hope to find decidability theorems for otherwise extended program models using the same method in the future.

References

1. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor Sets of Dynamic Pushdown Networks with Tree-Regular Constraints. In: Proceedings of Computer Aided Verification (CAV 2009). Springer, Heidelberg (2009). LNCS 5643.
2. Courcelle, B., Engelfriet, J.: Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach. Cambridge University Press (2012).