Static analysis of unbounded structures in object-oriented programs

Grabe, I.

Part II

Multi-threading

Chapter 3

Deadlock Detection for Reentrant Call-graphs[1]

In this chapter we investigate the synchronization of multithreaded call graphs with reentrance similar to call graphs in Java programs. We model the individual threads as Visibly Pushdown Automata (VPA) and analyse the reachability of a state in the product automaton by means of a Context Free Language (CFL) which captures the synchronized interleaving of threads. We apply this CFL-reachability analysis to detect deadlock.

3.1 Introduction

Due to behavioural complexity formal methods are needed when it comes to reasoning about programs. This is particularly true for concurrent (or multithreaded) programs. Such programs in general involve the synchronization of different processes (or threads) which may lead to undesirable deadlock situations.

A group of activities competing for a number of resources can block each other if each of them holds resources another one needs. A typical example of such a resource is exclusive access to a part of the system, i.e. a class or object, guarded by a lock. Modern programming languages like Java opt for implicit lock handling, i.e. instead of explicitly grabbing a lock via the execution of a lock operation a region is declared to be subject to a lock and the lock handling is done by the execution platform rather than the programmer.

[1] The work presented in this chapter was published as [43].

These languages also allow for reentrance, i.e. a thread is allowed to enter each region guarded by a lock several times if it holds the lock. In such a setting the number of times a thread has entered a lock-guarded region has to be counted to decide when to release the lock again. There are techniques, e.g. preemption or global lock orders, to address the problem of deadlock in such a setting. But these techniques do not provide a general solution to the deadlock problem, i.e. most concurrent programming languages have to deal with deadlocks and how to avoid them on the programming level.

The combination of multithreading, implicit lock handling, and reentrance makes the detection of deadlocks hard. This explains the need for methods and tools to do automatic deadlock analysis.

Contribution In order to develop an automated method for deadlock detection applicable to Java-like languages we abstract from data and focus on the control flow of method calls and returns. The unsynchronized interleaving of a finite number of reentrant (abstract) threads is naturally modelled as a multistack Visibly Pushdown Automaton [14]. In order to analyse the synchronization between threads we apply Context-Free-Language(CFL)-reachability as introduced in [90] to the underlying finite state automaton. Information about the ownership of the locks is included in the CFL to model synchronized sequences of calls and returns and to identify deadlock states.

In general however CFLs are not closed under arbitrary interleavings. We resolve this lack of expressive power of CFL languages in this particular setting by showing that for every (synchronized) interleaving there exists a *rescheduling* which does not affect the synchronization and is included in the CFL language. In fact, the CFL language only restricts the scheduling of the returns and we can anticipate returns of synchronised method calls without affecting the synchronization.

To the best of our knowledge this is the first automata based approach tailored to deadlock detection of the abstract control flow of method calls and returns of multithreaded reentrant programs. A sketch of an implementation is described in the concluding section. In this implementation the programmer only needs to indicate a finite number of threads, i.e., for each thread class the number of threads involved in the deadlock analysis.

Related Work In [91] Rinard gives an overview of recent techniques to analyse multithreaded programs. Deadlock detection is only covered for languages communicating via pairwise rendezvous. Ramalingam (see [89]) has

shown that the analysis of synchronization problems is not decidable even for only two threads if a CCS-style pairwise rendezvous is used to synchronize among the threads.

In [68] Kahlon et al. give a compositional method to reason about programs synchronizing via locks based on Visibly Pushdown Automata. Visibly Pushdown Automata are a kind of pushdown automata tailored to the generation of nested words reflecting the structure of traces generated by languages with nested call and return structures. The languages generated by these automata are closed under intersection. The result from [89] is generalized by showing that the reachability problem is not decidable for two threads communicating via non-nested locks. The language presented is non-reentrant and uses explicit acquire and release primitives. The automata are extended by so called acquisition histories to record relevant locking information to synchronize the threads. These acquisition histories can be used together with the explicit acquire primitives to identify deadlock situations. As soon as reentrance is allowed the setting gets more complicated. Due to reentrance the number of calls to synchronised methods has to be counted to decide whether or not to release a lock. Note that Java provides a nested call and return structure (with respect to one thread) which implies a nested acquire and release of locks.

Kidd et al. [69] introduce a technique called language strength reduction to reduce reentrant locks to non-reentrant locks in the context of Visibly Pushdown Languages. They check for atomic-set serializability violations, i.e. an illegal data access pattern. Due to this goal they take data into consideration. They create a CFL for each thread and for each lock. These languages are approximated by regular languages. Additionally a language describing a violation of a data access pattern is defined and the intersection of all languages is checked for emptiness. Up to our understanding there is no natural way to express a deadlock in this setting.

Lammich and Müller–Olm [72] present a model that can deal with thread creation and reentrant monitors. Their analysis is also focused on atomic-set serializability violations. Their approach is based on a fixpoint construction. They also use acquisition histories but only for synchronization purposes again. To reduce the number of executions to analyse they reduce the executions to a restricted subset involving the notion of a macrostep, i.e. a number of steps by one thread such that the stack only grows that is there are at most new locks taken after a macrostep but none are freed. In general, their analysis answers whether a given program location and a stack of method calls can be reached by a computation. However solutions to this reachability problem do not solve the more abstract problem of checking the reachability of a deadlock

configuration.

In [34] Carotenuto et al. introduce Visibly Pushdown Automata with a finite number of stacks. The languages generated by these automata are also closed under intersection. However the emptiness problem is not decidable for these automata.

A variety of other synchronization and communication mechanisms in concurrent programs with recursion have been studied (cf. for example [24], [35]). In [25] Bouajjani et al. present a formalism to compute abstractions of multithreaded call-graphs.

Outline This chapter is organized as follows. We start with a section on the syntax and semantics of synchronised multithreaded programs. In Section 3.3 we introduce Thread Automata to model the individual threads. Based on Thread Automata we introduce a technique based on CFL-reachability for the analysis of the product automaton in Section 3.4. In Section 3.5 we prove soundness and completeness of our method. We conclude in Section 3.6.

3.2 Synchronized Multithreaded Programs

We abstract from data, which includes object identities. In our setting locks are bound to classes. A system consists of a finite number of given classes and a finite number of given threads synchronizing via locks.

3.2.1 Syntax

We assume a given set of method names M with typical element m. Methods are specified by the following regular expressions. Here τ denotes an internal step and m denotes a call.

$$r ::= \tau \mid m \mid r; r \mid r + r \mid r^*$$

Figure 3.1: Grammar rule for the simple multithreaded language

We denote by M_s the synchronised methods and by M_u the unsynchronised ones. Every method is either synchronised or not:

$$M = M_s \cup M_u \text{ and } M_s \cap M_u = \emptyset.$$

We assume a given set D of method definitions. A method definition consists of a method name m and a method body given by a regular expression $m ::= r$. Furthermore we assume a finite partitioning C of the method names into classes with typical element c.

For every class c, we denote by

- M^c its methods,
- M_s^c its synchronised methods,
- M_u^c its unsynchronised methods.

We assume that every method belongs to exactly one class.

The set of method definitions D and the set of classes C define a program P. The behaviour of a program is defined in terms of a given set of threads T with typical element t. Each thread t has an initial (run) method denoted by $\mathrm{run}(t)$.

3.2.2 Operational Semantics

The operational semantics of a multithreaded program is described by a labelled transition relation between configurations Θ which consist of pairs (t, θ), where θ is a stack of labelled expressions $m@r$. We require Θ to contain for each thread $t \in T$ at most one such pair. The label $m@r$ indicates that r is the continuation of the execution of the body of method m. We record the name of the method to formalize synchronization as described below. The label at the top of the stack represents the method currently executed by the thread. By $\theta \cdot m@r$ we denote the result of pushing the label $m@r$ unto the stack θ.

To describe operationally the return of a method we extend the syntax of expressions by return expression ret. We identify the method body r in a declaration $m ::= r$ with $r; ret$.

Method Calls We have the following transition for unsynchronised methods:

A call of the method m' thus pushes the corresponding label on the stack. Note that upon return of m' the execution of m continues with r.

For synchronised methods we additionally require that no other thread is executing a synchronised method subject to the same lock:

$$\textsf{unsync} \quad \Theta \cup \{(t, \theta \cdot m@m'; r)\} \to \Theta \cup \{(t, \theta \cdot m@r \cdot m'@r')\}$$
with $m' ::= r' \in D$ and $m' \in M_u^c$ for some class c.

$$\textsf{sync} \quad \Theta \cup \{(t, \theta \cdot m@m'; r)\} \to \Theta \cup \{(t, \theta \cdot m@r \cdot m'@r')\}$$
with $m' ::= r' \in D$, $m' \in M_s^c$ for some class c, and there does not exist $(t', \theta') \in \Theta$ such that $t \neq t'$ and θ' contains a continuation $m''@r''$ of method $m'' \in M_s^c$.

$$\textsf{ret} \quad \Theta \cup \{(t, \theta \cdot m@ret)\} \to \Theta \cup \{(t, \theta)\}$$

Figure 3.2: Operational rules

Return Returning from a method is described by The top of the stack thus is simply popped upon return.

The rules for the choice and iteration operators are obtained by a straightforward lifting of the corresponding transitions for regular expressions as described in the next section.

The above transition relation maintains the following synchronization invariant:

Corollary 3.2.1 *For every class c there is at most one $(t, \theta) \in \Theta$ such that θ contains a continuation $m@r$ of a synchronised method $m \in M_s^c$.*

This characterization of threads and locks can be modelled in a straightforward manner as a multistack pushdown automaton with counters for each class. Reachability is not decidable in this general setting. Therefore we model the system differently. Each thread is modelled as a Visibly Pushdown Automata (VPA, for short). We show that the product of these automata are amenable to analysis via a technique based on CFL-reachability. We give a grammar to steer this analysis.

3.3 Thread Automata

In this section we model and analyse the operational semantics of multithreaded programs described above in terms of thread automata. For each thread t a Thread Automaton TA(t) is defined as a VPA, in terms of a *call* alphabet $\Sigma^t_{\text{call}} = \{t_m \mid m \in M\}$ and a *return* alphabet $\Sigma^t_{ret} = \{t_{ret}\}$. By Σ^t we

denote the visible alphabet $\Sigma^t_{\text{call}} \cup \Sigma^t_{ret}$ of thread t. A call of method m by thread t is indicated by t_m. The return of thread t from a method call is indicated by t_{ret}. The idea is that the call alphabet generates push operations, whereas the return alphabet generates pop operations. For each thread t its local alphabet is defined by $\{\tau\}$, used to describe internal steps.

States

The set of states of TA(t) is the set R_t of regular expressions reachable from the run method run(t). Here reachability is defined in terms of the following standard transition relation describing the behaviour of (regular) expressions:

- $m; r \to r$
- $r_1 + r_2; r \to r_i; r \quad \text{for } i \in \{1, 2\}$
- $r^*; r' \to r; r^*; r'$
- $r^*; r' \to r'$

Transitions

The external transitions of TA(t) are of the form (r, a, r', s), where r and r' are states as introduced above, a is an action of the visible alphabet of t, and s a stack symbol. The stack alphabet Γ^t of a Thread Automaton TA(t) is given by the set $\{t_r \mid t \in T,\ r \in R_t\}$, where the regular expression r denotes the return "address" of t. Method calls push a stack symbol upon the stack. This symbol encodes the location to return to later. Method returns pop a symbol from the stack. The location to return to can be derived from this symbol.

Internal transitions are of the form (r, τ, r') with r and r' states in terms of regular expressions, and τ to denote the internal step.

Method call For every state $m; r'$ we have the transition $(m; r', t_m, r, t_{r'})$, where $m ::= r \in D$. This transition models a move of control from state $m; r'$ to state r and a push of token $t_{r'}$ on the stack when reading t_m . The states encode the actual code to execute whereas the stack symbol encodes the location to return to when the method call terminates, i.e. a return is received.

Return For every state r, returning from a method is described by the transition (ret, t_{ret}, r, t_r) which models a move of control from state *ret* to state r and a pop of token t_r from the stack when reading t_{ret}. For each caller of the method a *return* transition exists. The location of return to is determinate by the token popped of the stack. Because the return location being determined by the stack symbol an unspecific return action t_{ret} is sufficient.

Internal transitions The choice construct is described by the transitions $(r_1 + r_2; r, \tau, r_i; r)$ for $i \in \{1, 2\}$ and iteration is described by a transition modelling looping $(r^*; r', \tau, r; r^*; r')$ and a transition modelling termination $(r^*; r', \tau, r')$. Note that internal transitions do not involve an operation on the stack.

Unsynchronised Product

We model the system by the product of the above automata for the individual threads. This automaton does not take synchronization between the individual threads into account. We add this synchronization by means of a grammar in Section 3.4.

Let $T = \{t^1, \ldots, t^n\}$. By $\mathrm{TA}(T)$ we denote the *unsynchronised* product of the automata $\mathrm{TA}(t^i)$. This product is described by a multistack VPA with call alphabet $\Sigma_{\text{call}} = \{t_m \mid t \in T,\ m \in M\}$, return alphabet $\Sigma_{\text{return}} = \{t_{ret} \mid t \in T\}$ and for each thread t a stack over the alphabet Γ^t. We denote by q_0 the initial state $q_0 = \langle \mathrm{run}(t_1), \ldots, \mathrm{run}(t_n) \rangle$.

States

The states of the product automaton are of the form $\langle r_1, \ldots, r_n \rangle$ where r_i denotes the state of t^i.

Transitions

We lift the transitions of the individual threads to transitions of the product in the obvious manner. Note that this lifting still does not provide any synchronization between the threads.

Reachability

Similar to the operational semantics for the definition of reachability in $\mathrm{TA}(T)$ we give a declarative characterization of the synchronization between threads

in terms of arbitrary sequences of calls and returns.

This characterization involves the following language theoretic properties:

- Calls and returns in a sequence are *matched* according to formal language theory, i.e. a bracketed grammar.
- A call without a matching return is called *pending*.
- A return without a matching call is called *pending*.
- A sequence is *well-formed* if it does not contain any pending returns.

Note that the words generated by the unsynchronised product are already well-formed.

Now we define synchronised sequences of calls and returns:
A sequence is called synchronised if for each call t_m to a synchronised method m ($m \in M_s^c$) by thread t there exists no pending call $t'_{m'}$ to a synchronised method m' of c by a thread $t' \neq t$ in the prefix of the sequence up to t_m.

We conclude this section with the definition of reachability and a definition of deadlock freedom in TA(T).

A state $q = \langle r_1, \ldots, r_n \rangle$ of TA(T) is *reachable* in TA(T) if there exists a computation in TA(T)

$$(q_0, \{\bot\}^n) \xrightarrow{W} (q, \bar{\theta})$$

for a synchronised sequence of calls and returns W and a tuple of stacks $\bar{\theta} = \langle \theta_1, \ldots, \theta_n \rangle$. Where $\bot$ denotes the empty stack.

This notion of reachability of a state does not provide enough information for deadlock detection. Therefore we extend the definition in the obvious manner to configurations: A configuration $(q, \bar{\theta})$ of TA(T) is *reachable* in TA(T) if there exists a computation in TA(T)

$$(q_0, \{\bot\}^n) \xrightarrow{W} (q, \bar{\theta})$$

for a synchronised sequence. Furthermore a configuration $(q, \bar{\theta})$ is a deadlock configuration iff $(q, \bar{\theta}) \not\rightarrow$, which indicates there is no transition possible, and at least one thread is not yet terminated, i.e. there exists an i such that $r_i \neq \mathit{ret}$ or $\theta_i \neq \bot$.

Finally we define the automaton TA(T) to be deadlock free iff there does not exist a reachable deadlock configuration.

3.4 CFL-Reachability

For the proof of the decidability of the reachability problem and deadlock freedom we apply CFL-reachability to the finite state automaton FA(T) embodied in TA(T). We first focus on the unsynchronised product and introduce synchronization later.

CFL-Modelling of Unsynchronised Interleavings

The the finite state automaton FA(T) contains all internal transitions (q, τ, q') of TA(T). To model the push and pop transitions of TA(T) we introduce the set of actions

$$\Sigma = \{t_r^m, t_r \mid t \in T,\ m \in M,\ r \in R\}$$

where t_r^m denotes a call of m by t with return expression r and t_r indicates that t returns to the regular expression r. We then model the transitions (q, t_m, q', t_r) and (q, t_{ret}, q', t_r) in TA(T) by (q, t_r^m, q') and (q, t_r, q'), respectively.

In order to compensate for the loss of information we introduce next for each thread t the following context free grammar which describes the structure of recursive call/return sequences.

$$\begin{array}{lcl} S^t & ::= & \epsilon \mid B^t \mid t_r^m S^t \mid S^t S^t \\ B^t & ::= & \epsilon \mid t_r^m r^t \mid B^t B^t \\ r^t & ::= & B^t t_r \end{array}$$

Sequences generated by the non-terminal S^t can contain pending calls, whereas sequences generated by B^t do not contain pending calls. Sequences generated by the non-terminal r^t ($r \in R_t$) describe a return from a method call to the expression r. In these sequences the call itself does not appear, e.g., these sequences contain a return t_r without a matching call. Note that the non-terminal r^t should be distinguished from the corresponding terminal t_r.

Starting with S^t or B^t the grammar produces well-formed sequences.

We lift this grammar to the definition of another CFL grammar describing the *unsynchronised* interleavings of the individual threads. The non-terminals of this grammar are sets G, where G contains for each thread t one of its non-terminals S^t, B^t, and r^t. The above rules are lifted to this grammar as

follows.

$$\begin{array}{lcll} G & ::= & \epsilon & (G \subseteq \{S^t, B^t \mid t \in T\}) \\ G \cup \{S^t\} & ::= & G \cup \{B^t\} \mid t_r^m\ G \cup \{S^t\} & \\ G \cup \{B^t\} & ::= & t_r^m\ G \cup \{r^t\} & \\ G \cup \{r^t\} & ::= & G \cup \{B^t\}\ t_r & \\ G_1 \circ G_2 & ::= & G_1\ G_2 & \end{array}$$

where the composition $G_1 \circ G_2$ contains for every thread a non-terminal U^t for which there exist a rule $U^t ::= V_1^t V_2^t$, with V_1^t in G_1 and V_2^t in G_2. Note that only sets G which contain for each thread t either S^t or B^t can be split (in other words, the non-terminal r^t cannot be split). Note also that the non-terminal r^t cannot be generated by a split.

We denote by $G_0 = \{S^t \mid t \in T\}$ the initial configuration of a derivation.

Note that not all possible interleavings can be derived by this grammar (see the following example). But for any possible interleaving an equivalent one (with respect to synchronization) exists which can be derived by the grammar. Since the non-terminal r^t can not be split the location of a method return is restricted. This does not affect the reachability or deadlock analysis. The method returns can be shuffled within certain limits (a return can be brought forward ignoring steps of other threads and can be delayed by steps of other threads on other locks). This holds also for the synchronised case as we show later. In the synchronized case this property is ensured by the requirements with respect to the lock sets.

Example 3.4.1 *We give an example of a sequence that can not be derived directly. The sequence* $t_r^m, t_{r'}^{\prime m'}, t_r, t_{r''}^{\prime m}, t_{r''}^{\prime}, t_{r'}^{\prime}$ *with* $m \in M_s^c$ *and* $m' \in M_s^{c' \neq c}$. *It is not possible to find a direct derivation for* $G_0 \Rightarrow^* t_r^m, t_{r'}^{\prime m'}, t_r, t_{r''}^{\prime m}, t_{r''}^{\prime}, t_{r'}^{\prime}$. *Since the projection on* t *resp.* t' *contains a matching return for every call it can only be derived by a rule starting from* B^t *resp.* $B^{t'}$. *We have to start with* B^t *to get the* t_r^m *in the front position of the sequence. The next step has to be a* $B^{t'}$ *step to get* $t_{r'}^{\prime m'}$ *to the second position. Now* $G = \{r^t\} \cup \{r'^{t'}\}$. *The next step has to be a* $r'^{t'}$ *to get* $t'_{r'}$ *to the end of the sequence. Now we get* $G = \{r^t\} \cup \{B^{t'}\}$. *Here we are stuck. To get the* t_r *in front of the* $t_{r''}^{\prime m}, t_{r''}^{\prime}$ *we could only use the composition rule but this one is forbidden for* G*s containing a* r^t. *Instead we can derive* $t_r^m, t_r, t_{r'}^{\prime m'}, t_{r''}^{\prime m}, t_{r''}^{\prime}, t_{r'}^{\prime}$. *By reordering the returns we can get the original sequence..*

According to the technique of CFL-reachability we define inductively transitions of the form (q, G, q'), where q and q' are states of FA(T) and G is a set

of non-terminals. Such a transition indicates that q' is reachable from q by a sequence generated by G.

- For every rule $G ::= \epsilon$ and state q we add a transition (q, G, q).
- For transitions (q, τ, q') and (q', G, q'') we add a transition (q, G, q''). Similarly, for transitions (q, G, q') and (q', τ, q'') we add a transition (q, G, q'').
- Given a transition (q, G, q'), an application of a rule $G' ::= G$ generates a transition (q, G', q).
- Given transitions (q_0, t_r^m, q) and (q, G, q_1), an application of a rule $G' ::= t_r^m\ G$ generates a transition (q_0, G', q_1).
- Given transitions (q_0, G, q) and (q, t_r, q_1), an application of a rule $G' ::= G\ t_r$ generates a transition (q_0, G', q_1).
- Given transitions (q_0, G_1, q) and (q, G_2, q_1), an application of rule $G_1 \circ G_2 ::= G_1 G_2$ generates a transition $(q_0, G_1 \circ G_2, q_1)$.

Reachability of a state q in $\mathrm{FA}(T)$ from the initial state q_0 by a word $G_0 \Rightarrow^* W$ then can be decided by checking the existence of a transition (q_0, G_0, q).

CFL-Modelling of Synchronised Interleavings

We now extend the above grammar for unsynchronised interleavings of threads with input/output information about the locks. This information is represented by pairs (I, L), where $I, L \subseteq T \times C$. The set of locks I are taken by some threads at the beginning of a derivation (step), whereas L is the set of locks that are taken by some threads at the end of a derivation (step). We denote an element of $T \times C$ by t_c which indicates that t holds the lock of class c. We implicitly restrict to subsets of $T \times C$ where for each class c at most one thread holds its lock. The non-terminals of this new grammar are annotated sets $(I, L) : G$, where $I \subseteq L$ and G contains for each thread t one of its non-terminals S^t, B^t, and r^t.

We have the following rules (here $I_c = \{t \in T \mid\ t_c \in I\}$ and

$L_c = \{t \in T \mid t_c \in L\}$).

$$
\begin{array}{lcll}
(I,I){:}G & ::= & \epsilon & (G \subseteq \{S^t, B^t \mid t \in T\}) \\
(I,L){:}G \cup \{S^t\} & ::= & (I,L){:}G \cup \{B^t\} & \\
 & \mid & t_r^m\ (I,L){:}G \cup \{S^t\} & (m \notin M_s^c \text{ or } t_c \in I \cap L) \\
 & \mid & t_r^m\ (I \cup \{t_c\},L){:}G \cup \{S^t\} & (m \in M_s^c,\ I_c = \emptyset,\ t_c \in L) \\
(I,L){:}G \cup \{B^t\} & ::= & t_r^m\ (I,L){:}G \cup \{r^t\} & (m \notin M_s^c \text{ or } t_c \in I \cap L) \\
 & \mid & t_r^m\ (I \cup \{t_c\},L \cup \{t_c\}){:}G \cup \{r^t\} & (m \in M_s^c,\ I_c = L_c = \emptyset) \\
(I,L){:}G \cup \{r^t\} & ::= & (I,L){:}G \cup \{B^t\}\ t_r & \\
(I,L){:}G_1 \circ G_2 & ::= & (I,L'){:}G_1\ (L',L){:}G_2 &
\end{array}
$$

The above grammar generates synchronised sequences. The conditions of the rules reflect in a natural manner the locking mechanism. To characterize the language generated by the above grammar we denote for a sequence of calls and returns W the set of locks still taken at the end of W by $\mathrm{Lock}(W)$: $t_c \in \mathrm{Lock}(W)$ iff there exists a pending call to a method m by thread t with $m \in M_s^c$.

Theorem 3.4.2 *For every sequence W generated by $(I, L) : G$ we have the following properties:*

- *W is synchronised*
- *$t_c \in L$ iff $t_c \in I \cup \mathrm{Lock}(W)$.*

Proof: The theorem is proven by induction on the length of the derivation of W. Details of the proof can be found in appendix A.1. □

To check reachability we add inductively transitions $(q, \alpha : G, q')$ to $\mathrm{FA}(T)$ analogous to the unsynchronised case above.

3.5 Soundness and Completeness of CFL-Reachability

Soundness and completeness of our method follows from the general technique of CFL-reachability and the following properties of our specific grammars together with the properties for sequences generated by the grammar established in Theorem 3.4.2.

We define an equivalence relation $W' \approx W$ as follows: For every thread t

- the projection of W' on t equals that of W on t

- $\text{Lock}(W') = \text{Lock}(W)$.

Lemma 3.5.1 *For every well-formed synchronised sequence W there exists a well-formed synchronised sequence W' such that $G_0 \Rightarrow^* W'$ with $G_0 = \{S^t \mid t \in T\}$ and $W' \approx W$.*

Proof: The lemma is proven by induction on the length of the word W. Details of the proof can be found in appendix A.2. □

We extend the notion of a synchronised sequence to a sequence synchronised with respect to a lock set I. A sequence W is synchronised with respect to I if for each $t_c \in I$ W does not contain any calls or returns of a thread $t' \neq t$ to a synchronised method of class c.

Lemma 3.5.2 *If $G_0 \Rightarrow^* W$ with W synchronised then $(\emptyset, \text{Lock}(W)) : G_0 \Rightarrow^* W$.*

Proof: Instead of proving the lemma directly we prove a more general statement: If $G_0 \Rightarrow^* W$ with W synchronised with respect to I, then $(I, I \cup \text{Lock}(W)) : G_0 \Rightarrow^* W$.

The statement is proven by induction on the length of the derivation $G_0 \Rightarrow^* W$. Details of the proof can be found in appendix A.3. □

Theorem 3.5.3 *The reachability problem of* $\text{TA}(T)$ *is decidable.*

Our method for checking reachability of a state q in $\text{TA}(T)$ consists of checking the existence of a transition $(q_0, (\emptyset, L) : G_0, q)$ in $\text{FA}(T)$.

Decidability follows from soundness and completeness proven above.

Theorem 3.5.4 *The problem of deadlock freedom of* $\text{TA}(T)$ *is decidable.*

In this case our method consists of checking reachability of $(q_0, (\emptyset, L) : G_0, q)$ for some state q for which there exists a subset of threads $T' \subseteq T$ such that in q each thread $t \in T'$ is about to execute a synchronised method $m \in M_s^c$ of a class c the lock of which is already held by a different thread $t' \in T'$, i.e., $t'_c \in L$.

Note that this notion of deadlock is a refinement of the notion presented in Section 3.3. With this notion we do not only cover a deadlock of the whole system but also of parts of the system.

3.6 Conclusion

We generalized the technique of CFL-reachability to the analysis of the synchronized interleavings of multithreaded Java programs. By means of this technique we can decide whether a state in the finite state automaton underlying the product of the individual thread automata is reachable by a synchronized interleaving. We also can decide deadlock freedom.

Future Work We are working on an implementation of our approach using the Meta Environment tools (see [101]). This work first involves the development of a suitable ASF specification to rewrite the parse tree of a Java program to the call graphs which form the basis of our analysis. The next step will be to provide a Meta Environment tool to perform the actual CFL-reachability analysis. Once this implementation for Java is finished it will be interesting to extend the method with further static analysis of the control flow graphs and dataflow in Java programs.