Context-Bounded Analysis of Multithreaded Programs with Dynamic Linked Structures

Ahmed Bouajjani[1], Séverine Fratani[1], and Shaz Qadeer[2]

[1] LIAFA, CNRS & University of Paris 7
{abou,fratani}@liafa.jussieu.fr
[2] Microsoft Research, Redmond
qadeer@microsoft.com

Abstract. Bounded context switch reachability analysis is a useful and efficient approach for detecting bugs in multithreaded programs. In this paper, we address the application of this approach to the analysis of multithreaded programs with procedure calls and dynamic linked structures. We define a program semantics based on concurrent pushdown systems with *visible heaps* as stack symbols. A visible heap is the part of the heap reachable from global and local variables. We use pushdown analysis techniques to define an algorithm that explores the entire configuration space reachable under given bounds on the number of context switches and the size of visible heaps.

1 Introduction

Automated analysis of software systems is a challenging problem. The behavior of these systems is usually complex and hard to predict due to aspects such as concurrency and memory management. Reasoning about these behaviors requires considering potentially infinite sets of configurations which makes verification problems undecidable in general. Therefore, approaches based on approximate analysis are needed. While over-approximations are useful for proving properties, under-approximations are useful for finding bugs. In this paper, we propose algorithmic (automata-based) techniques for under-approximate analysis of multithreaded programs that manipulate dynamic linked structures.

A simple way to get an under-approximate analysis is to bound the depth of the explored state space and use finite-state model checking techniques. This approach is interesting only if bugs appear after a small number of computation steps (e.g., [6]), which is unlikely to be the case for multithreaded programs. In such a program, processes interact through shared memory and are executed in alternation according to a schedule. Concurrency bugs appear only after a number of *context switches*, i.e, points in the execution where the active process is stopped and another one is resumed. Between context switches, a process may execute an unbounded number of computation steps. Therefore, a natural approach for under-approximate analysis of multithreaded program is to perform a (precise) analysis for a bounded number of context switches [9]. It has been demonstrated that this is indeed efficient for detecting bugs in multithread programs since they appear in many cases after a small number of context switches [7].

W. Damm and H. Hermanns (Eds.): CAV 2007, LNCS 4590, pp. 207–220, 2007.

However, bounding the number of context-switches does not allow the use of finite-state model checkers since each process may be infinite-state, and unbounded computations are possible between context switches. For example, in concurrent programs with procedure calls, each process may have an unbounded stack. In [9,2], automata-based techniques for symbolic analysis of pushdown systems have been used in order to define bounded context switch analysis of programs with finite data domains. In this paper, we consider the more general case of concurrent programs (with procedure calls) which can create shared objects and manipulate references to them.

In the spirit of under-approximate analysis, we could bound the heap size and reduce this verification problem to the case with finite data domains. This simple approach is unlikely to be effective at getting good state-space coverage for real programs, in which many heap-allocated objects are used either as local variables or for passing parameters to a called procedure. For such programs, it is better to bound only the *visible heap*, i.e., the part of the heap that is reachable from the global variables and the local variables of the running procedure in the active process. Indeed, a program's global heap might be unbounded in spite of its visible heap being bounded due to unbounded nesting of procedure calls.

It is nontrivial to obtain an algorithm for reachability analysis of multithreaded programs based on bounding the number of context switches and the size of the visible heap. In [10,11,8], the idea of visible heaps was used for interprocedural analysis of sequential programs; these techniques are based on procedure summarization and cannot be used for multithreaded programs since the (infinite) sets of stack configurations reached by processes must be stored at each context switch. Moreover, [10,11] consider abstract semantics, whereas we must consider an exact - sound and complete - semantics. Nevertheless, the idea of visible heaps can be used to define a program semantics where the heap is manipulated implicitly and not as a shared global structure: each procedure executed by some process manipulates a local heap structure which is a *copy* of its visible heap. Such an approach leads to a program model based on stack machines where locally visible heaps constitute the (potentially infinite) stack alphabet. To simulate correctly the "real" heap, our algorithm synchronizes the local views of procedures and processes at each procedure call or return and at each context switch. We prove that our new semantics is correct and use it to define an automata-based symbolic reachability analysis algorithm for bounded context switches and bounded visible heap. For lack of space, the proofs of our theorems are omitted here. They can be found in [3].

The contribution of our paper, although theoretical, has important practical applications. Since our algorithm constructs the set of reachable stack configurations, it allows the verification of reachability queries that require stack inspection. This expressiveness is important for specifying various resource-usage scenarios, e.g., user-space buffers are accessed by an operating system kernel only within the scope of an appropriate exception handler, or certain operations on security-critical objects are invoked only if a certain privileged procedure is present on the call stack. Our algorithm allows the verification of such expressive queries for multithreaded programs upto the context-switch and visible heap bounds and for single-threaded programs upto the visible heap bound. Of course, our algorithm can be used iteratively by systematically increasing the bounds in each iteration.

2 Multithreaded Programs with Dynamic Memory

We consider programs with multiple threads and procedure calls, which can dynamically create objects and manipulate pointers on these objects. A sequential program is given as a collection of control flow graphs, one graph for each of its procedures, defined by a set of control nodes N, and a set of transitions between these nodes labeled with actions and tests on the memory heap. We assume that the control flow graph of each procedure p has a unique initial control location n^p_{init} and a unique termination location n^p_{end} reachable after executing a return action. A multithread program consists of a parallel composition of a fixed number of sequential programs sharing the heap.

Let $\mathbb{A}$ be a countable (infinite) domain of memory addresses (pointers), and assume that $\mathbb{A}$ contains a special element $\perp$ representing the null address and a special element $\top$ representing an undefined address. Then, consider a class of objects where each object contains n successor fields $s_1, \ldots, s_n$ ranging over the pointer domain $\mathbb{A}$. Let $S = \{s_1, \ldots, s_n\}$. (We omit aspects related to data manipulation. Data values over finite data domains can of course be encoded in the successor fields.)

We assume that a program has a set of *global* pointer variables G ranging over $\mathbb{A}$. These variables are shared by all parallel threads, and all procedures. Moreover, we consider that each procedure has a set of *local* pointer variables (also ranging over $\mathbb{A}$). We assume w.l.o.g. that all the procedures have the same set of local variables L. Given a (global/local) pointer variable v and a successor field $s \in S$, we denote by $v.s$ the pointer stored in the field s of the object pointed by v. This notation can be extended to sequences of successor fields $\sigma \in S^*$ in the obvious manner. When σ is the empty sequence, we consider that $v.\sigma$ is identical to v.

Programs can perform the following operations on heaps: $v.\sigma := v'.\sigma'$ (*pointer assignment*), $v.\sigma := null$ (*pointer annihilation*), and $v.\sigma := new$ (*object creation*), where v, v' are pointer variables, and $\sigma \in S \cup \{\varepsilon\}$, $\sigma' \in S^*$. They can also perform the following tests: $v.\sigma \# v'.\sigma'$ with $\# \in \{=, \neq\}$ (*equality/disequality test*). In addition, they can perform the following actions: $call(p, v_1.\sigma_1, \ldots, v_m.\sigma_m)$ (*procedure call with parameters*), and *return* (*termination of a procedure call*), where p is a procedure name, $v_1, \ldots, v_m$ are pointer variables, and $\sigma_1, \ldots, \sigma_m \in S^*$. The effect of executing a statement $call(p, v_1.\sigma_1, \ldots, v_m.\sigma_m)$ is to call the procedure p after initialization of its local variables $\ell_1, \ldots, \ell_m$ with the pointer values $v_1.\sigma_1, \ldots, v_m.\sigma_m$ respectively where the variables $v_1, \ldots, v_m$ are either global variables or local variables of the calling procedure.

3 Program Semantics

Heaps as labeled graphs: A *global heap* is a finite directed graph where vertices correspond to memory addresses, edges are labeled by elements of S, and each element of G appears as a label of some vertex. Formally, a global heap is a tuple $GH = (A, \Delta, \Gamma)$ where (1) A is a finite subset of $\mathbb{A}$ containing $\perp$ and $\top$, (2) $\Delta : S \to (A \to A)$ associates with each $s \in S$ a successor mapping, and (3) $\Gamma : G \to A$ associates with a global variable g an object address. We assume that $\forall s \in S.\ \Delta(s)(\perp) = \top$ and $\Delta(s)(\top) = \top$, and $\forall a \in A$, $\Delta(s)(a)$ is defined.

Given $\sigma = s_{i_1} \cdots s_{i_m} \in S^*$, we denote by $\Delta(\sigma)$ the mapping $\Delta(s_{i_m}) \circ \cdots \circ \Delta(s_{i_1})$. When σ is the empty sequence, we consider that $\Delta(\sigma)$ is the identity mapping over A. Then, we define the reachability mapping to be $Reach_\Delta = \bigcup_{\sigma \in S^*} \Delta(\sigma)$.

Then, we consider that a *heap* is a global heap augmented by a mapping associating with each local variable an address. Formally, a heap is a pair $H = (GH, \Lambda)$, where $GH = (A, \Delta, \Gamma)$ is a global heap and $\Lambda : L \to A$.

Programs as concurrent heap pushdown systems: We associate with multithreaded programs concurrent stack machines which act on a (global) shared heap. The behavior of such concurrent stack machines is as follows: at each moment, there is one process which is running and has access to the heap through the global variables (shared by all processes), and its own local variables. Notice that due to nested procedure calls, the heap may be accessible from the local variables of the currently running procedure, and also from the local environments (of the pending procedure executions) stored in the stack of the process. While this process is running, all the other parallel processes are in an idle mode. When a context switch occurs, the active process is interrupted and some other one is resumed, becoming the new active process.

Let us sketch here the construction of the models associated with programs (details can be found in [3]). Let Σ be the (infinite) set of all pairs $\langle n, \Lambda \rangle$, where $n \in N$ and $\Lambda \in [L \to \mathbb{A}]$. We associate with each sequential program (given by a control flow graph) a *heap pushdown system* (H-PDS) which is a stack machine whose control states are global heaps, and whose stack alphabet is Σ: The semantics of basic operations and tests is defined by a transition relation $\xrightarrow{op}$ between heaps, and procedure calls and returns are modeled as usual by push and pop operations on the stack. Then, we associate with a multithreaded program with m parallel threads an m-dim *concurrent heap pushdown system* (CH-PDS). Control states of such a model are pairs (i, GH), where $i \in [1, m]$ is the index of the active process, and GH is a global heap. A configuration of a CH-PDS is a tuple $((i, GH), [u_1, \ldots, u_m])$ where $u_1, \ldots, u_m \in \Sigma^*$ are the local configurations of each of the threads, i.e., the contents of their stacks. A transition relation $\Rightarrow$ is defined between configurations: computations steps are either local to a process (the process i in the global state (i, GH)), or correspond to a context switch (i.e., substitution of i by some $j \neq i$ in the global state (i, GH)).

Bounded heap depth programs: Let $k \geq 1$. Let $c = ((i, GH), [u_1, \ldots, u_m])$ be a configuration, where $GH = (A, \Delta, \Gamma)$. Then, c is k-bounded iff for every $\langle n, \Lambda \rangle$ appearing in any u_i, for any $i \in \{1, \ldots, m\}$, we have $|Reach_\Delta(\Gamma(G) \cup \Lambda(L))| \leq k$. A program is k-bounded for a set of initial configurations C iff every $\Rightarrow$-reachable configuration from C is k-bounded. A program has *bounded heap depth* if it is k-bounded for some $k \geq 1$.

Notice that a heap with a bounded depth may have an unbounded size. Indeed, due to unbounded nesting of procedure calls, it is possible to have an unbounded stack of environments each of them pointing to a different but bounded part of the heap.

4 Program Semantics Based on Locally Visible Heaps

We define in this section a program semantics in terms of concurrent pushdown systems. The difference with the CH-PDS based semantics of the previous section is that the

heap is not manipulated explicitly as a global shared structure, but rather implicitly: each procedure executed by some process manipulates a local heap structure which is a *copy* of the reachable part of the heap from its local and global variables. Therefore, we can associate with programs stack machines where locally visible heaps constitute the stack alphabet. However, to simulate correctly operations on the "real" heap, procedures and processes must exchange (pass/update) their local informations about the heap at each procedure call/return and context switch. These manipulations of the local heaps are quite delicate. Let us give an informal description of the main ideas behind them.

Consider first the case of a sequential program. At each procedure call, the caller passes to the callee a copy of a part of its local heap. At the return of the call, the callee gives back its new local heap, the last one before termination, to the caller which updates accordingly its own local heap (which corresponds to its view of the heap just before the call). To perform correctly these caller/callee communications, a relation between vertices in different copies of a same piece of the "real" heap must be maintained, relating vertices which are copies of each other, representing the same address in the heap. Moreover, since pointer manipulation may disconnect some vertices from the visible part of a procedure during its execution, the deletion of these vertices from the local structure can be problematic since these vertices may still be relevant for further executions of procedures in the call stack. To handle this problem, we introduce a notion of *cut points*: when a heap is passed from the caller to the callee, cut points represent the first vertices in the visible part by the callee which are reachable from the caller or from other procedures in the call stack. Then, during the execution of the procedure, a vertex can be removed from the locally visible heap only if it becomes unreachable from the local variables, the global variables, and the cut points.

In the case of parallel programs, an additional but similar mechanism has to be introduced for passing/updating locally visible heaps at context switches. Intuitively, parallel processes synchronize their views of the heap at each context switch. The active process passes its local heap to all of the other processes which can update accordingly their local heaps (corresponding to the configuration at the previous context switch), taking into account the modifications on the heap performed by the active process while they were idle. For that, the active process maintains in its control state, which is a global state for all its procedures, a relation between its current local heap and its initial local heap corresponding to the configuration at the last context switch, allowing to determine for each vertex in the (old) local heaps whether it has a copy in the (new) local heap returned by the active process. Also, to deal with vertex deletion, it is necessary to extend the use of cut points by considering the reachable vertices from the stacks of each process. In fact, each process needs to distinguish his own cut points from the ones of the other processes.

We prove that the new semantics is correct (sound and complete) w.r.t. the original semantics (in section 3) in the sense that they define bisimilar transition systems. An important property allowing to prove this fact is that isomorphism between locally visible heaps preserves bisimilarity (i.e., substitution of isomorphic local heaps does not modify behaviors). This holds because the performed operations and tests on the heaps do not refer to the precise values of the addresses. (Notice that there is an infinite number of isomorphic heaps of a same size since the address domain is infinite). Then, we

consider a normalization operation for visible heaps which associates to each of them an equivalent heap modulo isomorphism, and we prove that the semantics based on normalized visible heaps is also correct w.r.t. the original semantics. Our normalization operation matches all the isomorphic visible heaps to a finite number of representatives. (We do not have a unique representative due to the presence of cut points.) Therefore, for sequential programs, our pushdown model construction terminates for bounded heap depth programs. For concurrent programs, the construction terminates for a finite number of a context switches, given a bound on the heap depth.

4.1 Locally Visible Heaps

A *cut heap* is a tuple $CH = (A, \Delta, \Gamma, \Lambda, \overrightarrow{C})$ where $(A, \Delta, \Gamma, \Lambda)$ is a heap and $\overrightarrow{C} \in (2^A)^m$ is a vector of sets of cut points. For each $i \in \{1, \ldots, m\}$, $\overrightarrow{C}(i)$ is the set of cut points reachable from the local environments (of the pending procedure executions) stored in the stack of the process i. Then, a *visible heap* is a cut heap such that $A = Reach_\Delta(\Gamma(G) \cup \Lambda(L) \cup \bigcup_{i=1}^m \overrightarrow{C}(i) \cup \{\bot\})$, i.e., it is a cut heap without garbage. We define an operation of *garbage elimination* allowing to obtain a visible heap from a cut heap: given a cut heap $CH = (A, \Delta, \Gamma, \Lambda, \overrightarrow{C})$, the visible heap $Clean(CH)$ is given by $(A', \Delta', \Gamma, \Lambda, \overrightarrow{C})$ where $A' = A \cap Reach_\Delta(\Gamma(G) \cup \Lambda(L) \cup \bigcup_{i=1}^m \overrightarrow{C}(i) \cup \{\bot\})$ and $\forall s \in S.\ \Delta'(s) = \Delta(s) \cap [A' \to A']$, i.e., the restriction of $\Delta(s)$ to A'.

We define an equivalence relation $\simeq$ between visible heaps which is essentially graph isomorphism modulo renaming of vertices. Let $VH_1 = (A_1, \Delta_1, \Gamma_1, \Lambda_1, \overrightarrow{C_1})$ and $VH_2 = (A_2, \Delta_2, \Gamma_2, \Lambda_2, \overrightarrow{C_2})$ be two visible heaps and let $\beta : A_1 \to A_2$ be a bijection s.t. $\beta(\top) = \top$ and $\beta(\bot) = \bot$. Then, $VH_1 \simeq_\beta VH_2$ iff (i) $\forall s \in S.\ \forall a \in A_1.\ \beta(\Delta_1(s)(a)) = \Delta_2(s)(\beta(a))$, (ii) $\forall v \in G.\ \Gamma_2(v) = \beta(\Gamma_1(v))$, and $\forall v \in L.\ \Lambda_2(v) = \beta(\Lambda_1(v))$, and (iii) $\forall i \in [1, m]$, $\overrightarrow{C_2}(i) = \{\beta(c) : c \in \overrightarrow{C_1}(i)\}$. Then, $VH_1 \simeq VH_2$ if there is a β s.t. $VH_1 \simeq_\beta VH_2$.

4.2 Sequential Programs as Pushdown Systems

Let us define an *environment* to be a tuple $e = \langle n, VH, \pi \rangle$ where $n \in N$, VH is a visible heap, and $\pi \subseteq \mathbb{A} \times \mathbb{A}$ is an injective function (π relates vertices in the heap of the caller procedure with vertices in the current visible heap VH). Let $\mathbb{V}$ be the set of all environments. Then, we associate with each sequential process a *visible heap pushdown system* (VH-PDS) which is a stack machine whose stack alphabet is $\mathbb{V}$ and whose control states are injective functions from $\mathbb{A}$ to $\mathbb{A}$. These functions are used to maintain a link between the current visible heap of the running sequential process and its initial visible heap since its last activation (at the initial configuration of the system or at the last context switch). This information is needed at the next context switch for updating the visible heaps of the idle processes. (Informations in control states can be omitted in the case of a purely sequential program.) Now, let i be the index of a sequential process. We define the (infinite) set of transition rules R_i of the stack machine associated with process i by means of the three inference rules given hereafter.

Basic operations and test: We extend the relation $\xrightarrow{op}$ defined on heaps (see section 3) to a relation on visible heaps (which also informs about the correspondence between

vertices in the original and final heaps). Let $VH_1 = (H_1, \overrightarrow{C_1})$, and $VH_2 = (H_2, \overrightarrow{C_2})$ be two visible heaps. For every function $\pi : \mathbb{A} \times \mathbb{A}$, we have $VH_1 \xrightarrow{op}_\pi VH_2$ iff there exists a heap H_2' s.t. $H_1 \xrightarrow{op} H_2'$ and $Clean(H_2', \overrightarrow{C_1}) \simeq_\pi VH_2$. Then:

$$\frac{n_1 \xrightarrow{op} n_2 \quad VH_1 \xrightarrow{op}_\pi VH_2}{(\Pi, \langle n_1, VH_1, \pi_1 \rangle) \hookrightarrow (\Pi \circ \pi, \langle n_2, VH_2, \pi_1 \circ \pi \rangle) \in R_i} \quad \mathsf{HeapOp}$$

Procedure calls: Let $i \in [1,m]$ be the index of the active sequential process, let $VH_1 = (A_1, \Delta_1, \Gamma_1, \Lambda_1, \overrightarrow{C_1})$ be the visible heap of the caller procedure, and let $v_1.\sigma_1, \ldots, v_k.\sigma_k$ be the effective parameters which must be assigned respectively to the local variables $\ell_1, \ldots, \ell_k$ of the called procedure (considered as its formal parameters). The visible heap passed to the callee procedure (of the process i) is obtained as follows: (1) construct first the cut heap $CutPass_i(VH_1)$ which is a copy of VH_1, where local variables of the callee procedure are assigned with their new values, and local variables of the caller procedure are memorized as cut points, (2) then determine the new vector of sets of cut points: cut points of processes $j \neq i$ stay unchanged, cut points of the process i are the first addresses reachable from cut points of $CutPass_i(VH_1)$ in the sub-heap corresponding to reachable addresses from the new local variables, the global variables, and the cut points of all processes, (3) finally remove the garbage of $CutPass_i(VH_1)$ according to the new set of cut points.

Let us define formally the operation $CutPass_i$. Let $VH = (A, \Delta, \Gamma, \Lambda, \overrightarrow{C})$ be a visible heap. Then, $CutPass_i(VH, [\ell_1, v_1.\sigma_1], \ldots, [\ell_k, v_k.\sigma_k])$ is the cut heap $(A, \Delta, \Gamma, \Lambda', \overrightarrow{C'})$ where (1) $\forall j \in [1,k], \Lambda'(\ell_j) = \Delta(\sigma_j)((\Lambda \cup \Gamma)(v_j))$ and $\Lambda'(\ell) = \top$ for all other variables $\ell \in L$, (2) $\forall j \in [1,m]$, $j \neq i$, we have $\overrightarrow{C'}(j) = \overrightarrow{C}(j)$, and (3) $\overrightarrow{C'}(i) = \overrightarrow{C}(i) \cup \Lambda(L)$.

We give now the formal definition of the visible heap passed by the caller to the callee. Consider $CH = (A, \Delta, \Gamma, \Lambda, \overrightarrow{C})$ to be any cut heap. Let $A' = Reach_\Delta(\Gamma(G) \cup \Lambda(L) \cup \bigcup_{j \neq i} \overrightarrow{C}(j))$ (the set A' is the visible part of the heap from local and global variables and from cut points in $\overrightarrow{C}(j)$, for $j \neq i$), and let Δ' be the restriction of Δ to vertices in A'. Then, let $Visible_i(CH) = Clean(A, \Delta, \Gamma, \Lambda, \overrightarrow{C'})$ where, for every $j \in [1,m]$ s.t. $j \neq i$, $\overrightarrow{C'}(j) = \overrightarrow{C}(j)$, and $\overrightarrow{C'}(i) = (A' \cap Reach_\Delta(\overrightarrow{C}(i))) \setminus \Delta'(S)(Reach_\Delta(\overrightarrow{C}(i)))$. The set $\overrightarrow{C'}(i)$ contains the first vertices in A' which are Δ-reachable from $\overrightarrow{C}(i)$. The cleaning operation removes all vertices which are Δ-reachable only from $\overrightarrow{C}(i)$, i.e., the set of vertices in $Visible_i(CH)$ is precisely A'.

Finally, given a visible heap VH, we define $\mathsf{Pass}_i(VH, p, v_1.\sigma_1, \ldots, v_k.\sigma_k)$ to be the set of pairs (VH_π, π) such that $Visible_i(CutPass_i(VH, [\ell_1, v_1.\sigma_1], \ldots, [\ell_k, v_k.\sigma_k])) \simeq_\pi VH_\pi$. The injective relation π allows to relate addresses in the visible heap of the caller with addresses in the new visible heap. It is used to update the heap of the caller after termination of the callee procedure. Then, we consider the inference rule:

$$\frac{n_1 \xrightarrow{call(p, v_1.\sigma_1, \ldots, v_k.\sigma_k)} n_2 \quad (VH_2, \pi_2) \in \mathsf{Pass}_i(VH_1, p, v_1.\sigma_1, \ldots, v_k.\sigma_k)}{(\Pi, \langle n_1, VH_1, \pi_1 \rangle) \hookrightarrow (\Pi \circ \pi_2, \langle n^p_{init}, VH_2, \pi_2 \rangle \langle n_2, VH_1, \pi_1 \rangle) \in R_i} \quad \mathsf{Call}$$

Procedure returns: Given the current visible heap VH_2 of the terminating procedure and the visible heap VH_1 of the caller procedure stored in the stack, we define an operation updating VH_1 according to the effect of the procedure call on the heap.

Let $\pi_2 \subseteq A_1 \times A_2$ be an injective relation giving the correspondence between the vertices of VH_1 and VH_2 (i.e., if $(a_1,a_2) \in \pi$ and $(a_1',a_2) \in \pi$, then $a_1 = a_1'$). We suppose that $A_1 \cap A_2 = \{\top, \bot\}$ (otherwise, instead of VH_2 and π_2, consider a new visible heap VH_2' and $\pi_2 \circ \pi$ for some π s.t. $VH_2 \simeq_\pi VH_2'$).

Then, let $B_1 = \{a \in A_1 : \nexists a' \in A_2.\ (a,a') \in \pi_2\}$, let $B_2 = \{a \in A_1 : \exists a' \in A_2.\ (a,a') \in \pi_2\}$, and let $B_3 = \{a' \in A_2 : \nexists a \in A_1.\ (a,a') \in \pi_2\}$.

Intuitively, B_1 is the set of vertices in A_1 such that either they were not reachable in the initial visible heap passed to the called procedure, and therefore they should be restored in the heap of the caller procedure after the call return, or they became invisible during the call execution due to garbage deletion and therefore they should not appear in the heap after the call return since they were not reachable from cut points. (We explain below how to get rid of the vertices in this second category). Vertices in B_2 are those which were present before the call, and which are still present after termination of the call. Finally, B_3 is the set of the created vertices during the call.

It can be easily seen that these three sets are disjoint. Moreover, we have $B_3 \cup \pi_2(B_2) = A_2$. Let us consider the bijection $\beta : A_2 \to B_2 \cup B_3$ defined in the obvious way (for every $a \in A_2$, if $a \in B_3$ then $\beta(a) = a$, otherwise $\beta(a) = {\pi_2}^{-1}(a)$).

Let $(A_1', \Delta_1', \Gamma_1', \Lambda_1', \overrightarrow{C_1'})$ be the cut heap such that (1) $A_1' = B_1 \cup B_2 \cup B_3$, (2) $\forall g \in G.\ \Gamma_1'(g) = \beta(\Gamma_2(g))$, (3) $\forall \ell \in L,\ \Lambda_1'(\ell) = \Lambda_1(\ell)$, (4) $\overrightarrow{C_1'}(i) = \overrightarrow{C_1}(i)$ and $\forall j \neq i,\ \overrightarrow{C_1'}(j) = \beta(\overrightarrow{C_2}(j))$, and (5) $\forall s \in S$, (i) $\forall a \in B_1.\ \Delta_1'(s)(a) = \Delta_1(s)(a)$, (ii) $\forall a \in B_2.\ \Delta_1'(s)(a) = \beta(\Delta_2(s)(\pi_2(a)))$, and (iii) $\forall a \in B_3.\ \Delta_1'(s)(a) = \beta(\Delta_2(s)(a))$.

Then, we define $\mathsf{Update\text{-}seq}_i(VH_1, VH_2, \pi_2)$ to be the set of all pairs (VH_π, π) such that $Clean(A_1', \Delta_1', \Gamma_1', \Lambda_1', \overrightarrow{C_1'}) \simeq_\pi VH_\pi$.

Notice that (1) the cleaning operation removes the vertices of A_1 which were garbage collected during the procedure call, and (2) for every VH_2', and every β s.t. $VH_2 \simeq_\beta VH_2'$, $\mathsf{Update\text{-}seq}_i(VH_1, VH_2, \pi_2) = \mathsf{Update\text{-}seq}_i(VH_1, VH_2', \pi_2 \circ \beta)$.

Then, we consider the inference rule:

$$\frac{n \xrightarrow{return} n_{end} \quad (VH_1', \pi') \in \mathsf{Update\text{-}seq}_i(VH_1, VH_2, \pi_2)}{(\Pi, \langle n, VH_2, \pi_2\rangle\langle n', VH_1, \pi_1\rangle) \hookrightarrow (\Pi \circ \pi', \langle n', VH_1', \pi_1 \circ \pi'\rangle) \in R_i} \quad \mathsf{Return}$$

4.3 Multithreaded Programs as Concurrent Pushdown Systems

We associate with a multithreaded program with m parallel threads an m-dim *concurrent visible heap pushdown system* (CVH-PDS). The stack alphabet is $\mathbb{V}$, and the (infinite) set of control states is $\mathbb{S} = \{(i, \overrightarrow{\Pi}) : i \in [1,m],\ \overrightarrow{\Pi} = (\overrightarrow{\Pi}(1), \ldots, \overrightarrow{\Pi}(m)),\ \overrightarrow{\Pi}(j) : \mathbb{A} \to \mathbb{A}\}$. We define hereafter the set of transition rules $\mathcal{R}$ of the model.

Local transitions: Transitions of each sequential process are obviously transitions of the whole system. For every $i \in [1,m]$, let R_i be the set of transition rules associated with the process of index i (defined in the previous subsection). Then, we have:

$$\frac{(\Pi_i,w_i)\hookrightarrow(\Pi'_i,w'_i)\in R_i \qquad \overrightarrow{\Pi}(i)=\Pi_i \qquad \overrightarrow{\Pi'}(i)=\Pi'_i \qquad \forall j\neq i.\ w_j\in\mathbb{V} \text{ and } \overrightarrow{\Pi}(j)=\overrightarrow{\Pi'}(j)}{((i,\overrightarrow{\Pi}),[w_1,\ldots,w_i,\ldots,w_m])\hookrightarrow((i,\overrightarrow{\Pi'}),[w_1,\ldots,w'_i,\ldots,w_m])\in\mathcal{R}}\ \textsf{Local}$$

Context switches: Finally, consider the case of a context switch. Assume that process i was the last active process and let $VH_i = (A_i,\Delta_i,\Gamma_i,\Lambda_i,\overrightarrow{C_i})$ be its last visible heap. At the context switch, process i communicates to all the other ones (which were idle) informations about the new heap configuration. For that, the part which is *shared* by all processes is extracted from VH_i and passed to them. Then, each process $j \neq i$ updates accordingly its old visible heap (which corresponds to the heap configuration at the previous context switch) before the next active process starts its computation.

Formally, let us define $Shared_i(VH_i) = Visible_i(CutPass_i(VH_i,\forall \ell \in L.\ [\ell,\top]))$, and assume that $Shared_i(VH_i) = (A'_i,\Delta'_i,\Gamma'_i,\Lambda'_i,\overrightarrow{C'_i})$. Notice that we remove from A_i only vertices that are reachable from local variables in L or from cut points in $\overrightarrow{C_i}(i)$, but which are not reachable from global variables, nor from cut points of other threads. These vertices are not visible from other threads and therefore they do not belong to the shared part of the heap. The cut points in $\overrightarrow{C'_i}(i)$ allow to know which vertices in the shared part are reachable from local variables in the stack of the current thread. The removed vertices will be added to the heap when the current process will resume later. This is done by an updating operation described below.

Let $j \in [1,m]$, let $VH_j = (A_j,\Delta_j,\Gamma_j,\Lambda_j,\overrightarrow{C_j})$ be the visible heap of the process j, and suppose $\Pi \subseteq A_j \times A'_i$ is an injective relation connecting vertices in VH_j with vertices in $Shared_i(VH_i)$. We suppose that $A_i \cap A_j = \{\top,\bot\}$ (otherwise, instead of VH_j and Π, consider VH'_j and $\Pi \circ \pi$ for some π s.t. $VH_j \simeq_\pi VH'_j$).

Then, let $B_1 = \{a \in A_j \ :\ \nexists a' \in A'_i.\ (a,a') \in \Pi\}$, let $B_2 = \{a \in A_j \ :\ \exists a' \in A'_i.\ (a,a') \in \Pi\}$, and let $B_3 = \{a' \in A'_i \ :\ \nexists a \in A_j.\ (a,a') \in \Pi\}$. These sets are disjoint and we have $B_3 \cup \Pi(B_2) = A'_i$. Then, consider the bijection $\beta_j : A'_i \to B_2 \cup B_3$ defined in the obvious way (for every $a \in A'_i$, if $a \in B_3$ then $\beta_j(a) = a$, otherwise $\beta_j(a) = \Pi^{-1}(a)$).

We define the operation $\textsf{Update-par}_j$, for $j \in [1,m]$, which updates the local heap of process j using the shared heap passed by process i. For $j \neq i$, let $(A'_j,\Delta'_j,\Gamma'_j,\Lambda'_j,\overrightarrow{C'_j})$ be the cut heap s.t. (1) $A'_j = B_1 \cup B_2 \cup B_3$, (2) $\forall g \in G.\ \Gamma'_j(g) = \beta_j(\Gamma'_i(g))$, (3) $\forall \ell \in L$, $\Lambda'_j(\ell) = \Lambda_j(\ell)$, (4) $\overrightarrow{C'_j}(j) = \overrightarrow{C_j}(j)$, and $\forall k \neq j$, $\overrightarrow{C'_j}(k) = \beta_j(\overrightarrow{C'_i}(k))$, and (5) $\forall s \in S$, (i) $\forall a \in B_1.\ \Delta'_j(s)(a) = \Delta_j(s)(a)$, (ii) $\forall a \in B_2.\ \Delta'_j(s)(a) = \beta_j(\Delta'_i(s)(\Pi(a)))$, and (iii) $\forall a \in B_3.\ \Delta'_j(s)(a) = \beta_j(\Delta'_i(s)(a))$. Then, $\textsf{Update-par}_j(VH_j,VH_i,\Pi)$ is defined to be the set of all (VH'_j,π,Π_j) such that (1) $Clean(A'_j,\Delta'_j,\Gamma'_j,\Lambda'_j,\overrightarrow{C'_j}) \simeq_\pi VH'_j$, and (2) $\Pi_j = \pi^{-1} \circ \beta_j^{-1}$. Moreover, for every mapping $\Pi_i : \mathbb{A} \to \mathbb{A}$, we define $\textsf{Update-par}_i(VH_i,VH_i,\Pi_i)$ to be the set of all (VH'_i,π,π^{-1}) such that $VH_i \simeq_\pi VH'_i$. Then, we consider the inference rule:

$$\frac{\begin{array}{c} i,k\in\{1,\ldots,m\} \qquad i\neq k \qquad \forall j\in\{1,\ldots,m\}.\ w_j=\langle n_j,VH_j,\pi_j\rangle \\ \forall j\in\{1,\ldots,m\}.\ (VH'_j,\pi'_j,\Pi_j)\in\textsf{Update-par}_j(VH_j,VH_i,\overrightarrow{\Pi}(j)\circ\overrightarrow{\Pi}(i)) \\ \forall j\in\{1,\ldots,m\}.\ w'_j=\langle n_j,VH'_j,\pi_j\circ\pi'_j\rangle \end{array}}{((i,\overrightarrow{\Pi}),[w_1,\ldots,w_m])\hookrightarrow((k,(\Pi_1,\ldots,\Pi_m)\circ{\Pi_k}^{-1}),[w'_1,\ldots,w'_m])\in\mathcal{R}}\ \textsf{Switch}$$

with the notational convention $(\Pi_1,\ldots,\Pi_m)\circ\Pi = (\Pi_1\circ\Pi,\ldots,\Pi_m\circ\Pi)$.

4.4 Associated Transition System and Correctness of the Semantics

A *configuration* of a CVH-PDS is a tuple $((i,\overrightarrow{\Pi}),[v_1,\ldots,v_m]) \in \mathbb{S}\times[\mathbb{V}^*]^m$ where $i\in[1,m]$ is the index of the active process, and $v_1,\ldots,v_m\in\mathbb{V}^*$ are the local configurations of all the processes. Let $\mathbb{C}$ be the set of all configurations.

Given $r=((i_1,\overrightarrow{\Pi_1}),[u_1,\ldots,u_m])\hookrightarrow((i_2,\overrightarrow{\Pi_2}),[u'_1,\ldots,u'_m])\in\mathcal{R}$, let $\mapsto_r\subseteq\mathbb{C}\times\mathbb{C}$ be the relation s.t. $b\mapsto_r b'$ iff $b=((i_1,\overrightarrow{\Pi_1}),[v_1,\ldots,v_m])$, $b'=((i_2,\overrightarrow{\Pi_2}),[v'_1,\ldots,v'_m])$, and $\forall k\in[1,m]$, $\exists w_k\in\mathbb{V}^*$ s.t. $v_k=u_kw_k$ and $v'_k=u'_kw_k$.

Let $\mapsto_{\mathsf{loc}}$ (resp. $\mapsto_{\mathsf{sw}}$) be the union of all relations $\mapsto_r$ where r is a Local rule (resp. Switch rule), and let $\mapsto$ be the union of all the relations $\mapsto_r$, for $r\in\mathcal{R}$. Then, we consider the relation $\rightsquigarrow=\mapsto_{\mathsf{loc}}^*\circ\mapsto_{\mathsf{sw}}$. For each $K\geq 1$, the relation $\rightsquigarrow^K$ (Kth power of $\rightsquigarrow$) corresponds to $\mapsto$-reachability with $K-1$ context switches (or K consecutive contexts).

We extend the equivalence relation $\simeq$ defined on visible heaps to environments in $\mathbb{V}$: we consider that $\langle n_1,VH_1,\pi_1\rangle\simeq_\beta\langle n_2,VH_2,\pi_2\rangle$ iff (1) $n_1=n_2$, and (2) β is a bijection from A_1 to A_2 such that $VH_1\simeq_\beta VH_2$ and $\pi_2=\pi_1\circ\beta$. Given two environments e_1,e_2, we write $e_1\simeq e_2$ iff there exists β such that $e_1\simeq_\beta e_2$. We extend this equivalence relation to sequences of environments in the obvious manner ($e_1\cdots e_j\simeq e'_1\cdots e'_k$ iff $j=k$ and for every $i\in[1,j]$, $e_i\simeq e'_i$).

Finally, we extend $\simeq$ to configurations: let $b=((i,\overrightarrow{\Pi}),[e_1\alpha_1,\ldots e_m\alpha_m])$ and $b'=((j,\overrightarrow{\Pi'}),[e'_1\alpha'_1,\ldots,e'_m\alpha'_m])$ be two configurations. Then, $b\simeq b'$ iff (1) $i=j$, (2) $\forall k\in[1,m]$, $\alpha_k\simeq\alpha'_k$, and (3) $\exists\pi_k:\mathbb{A}\to\mathbb{A}$ s.t. $e'_k\simeq_{\pi_k}e_k$ and $\overrightarrow{\Pi'}(k)\circ\pi_k=\overrightarrow{\Pi}(k)$.

Proposition 1. *For every configurations b_0,b,b', if $b_0\mapsto^* b$ and $b'\simeq b$, then $b_0\mapsto^* b'$.*

We prove that given a multithreaded program, its associated CH-PDS and CVH-PDS are bisimilar. For that, we exhibit a bisimulation relation between configurations of the two systems. Intuitively, this relation maps a configuration of the CVH-PDS to a configuration of the CH-PDS by applying the updating operations through the configuration. (Indeed, visible heaps stored in the stacks of the CVH-PDS model are not up to date since they correspond to views of the heap at the moment of their memorization.)

Theorem 1. *The relations $\Rightarrow$ and $\mapsto$ define bisimilar transition systems.*

4.5 Program Semantics Based on Normalized Visible Heaps

Normalized visible heaps: Visible heaps in *normal form* are obtained by numbering nodes according to a depth-first traversal of the heap, for a given ordering of global and local variables, and the fixed order on the successor fields $\{s_1,\ldots,s_n\}$.

Let us consider a bijection $\eta:L\cup G\to[1,|L\cup G|]$. Then, given an environment $\langle n,VH,\pi\rangle$ with $VH=(A,\Delta,\Gamma,\Lambda)$, we define the class $[\![\langle n,VH,\pi\rangle]\!]_\eta$ to be the set of all environments $\langle n,VH',\pi\circ\beta\rangle$ with $VH'=(A',\Delta',\Gamma',\Lambda')$ s.t. (1) $VH\simeq_\beta VH'$, (2) $A'=[1,|A|]$, and (3) in the graph (V,Δ'_V) where $V=Reach_{\Delta'}(\Lambda'(L)\cup\Gamma'(G))$ and $\Delta'_V=\Delta'\cap[S\to(V\to V)]$, vertices correspond to the depth-first-traversal induced by the order η on root nodes (and the fixed order on successor fields labeling the edges of the graph).

Notice that, if all vertices in VH are reachable from $L\cup G$, then $[\![\langle n,VH,\pi\rangle]\!]_\eta$ contains a *single* element. Otherwise, there must be cut points (and may be other vertices

reachable from them) which are not reachable from $L \cup G$. In that case, $[\![\langle n, VH, \pi\rangle]\!]_\eta$ contains environments (with identical reachable parts from $L \cup G$) corresponding to different permutations of the vertices reachable from cut points but not from $L \cup G$.

Proposition 2. *For every η, the set $[\![\langle n, VH, \pi\rangle]\!]_\eta$ is finite (if VH is finite).*

Concurrent pushdown systems on normalized visible heaps: Let us fix η. We define a program semantics where visible heaps are considered modulo the η-equivalence. Let $\mathbb{V}_\eta$ be the set of all $[\![e]\!]_\eta$, for $e \in \mathbb{V}$. We consider the set of transition rules $\mathcal{R}_\eta$ corresponding to the restriction of the set $\mathcal{R}$ to the alphabet $\mathbb{V}_\eta$:

$$\mathcal{R}_\eta = \{((i, \overrightarrow{\Pi}), [\alpha_1, \ldots, \alpha_m]) \hookrightarrow ((j, \overrightarrow{\Pi'}), [\alpha'_1, \ldots, \alpha'_m]) \in \mathcal{R} \mid \forall j,\ \alpha_j, \alpha'_j \in {\mathbb{V}_\eta}^*\}.$$

Let $\mathbb{C}_\eta$ be the set of all configurations $((i, \overrightarrow{\Pi}), [\alpha_1, \ldots, \alpha_m])$ such that $\forall j,\ \alpha_j \in {\mathbb{V}_\eta}^*$. Then, let $\mapsto_{r,\eta}$ be the restriction of the transition relation $\mapsto_r$ to configurations in $\mathbb{C}_\eta$. Let $\mapsto_\eta$ be the union of the relations $\mapsto_{r,\eta}$ for all transition rules r. The relations $\mapsto_{\mathsf{loc},\eta}$, $\mapsto_{\mathsf{sw},\eta}$, and $\rightsquigarrow_\eta$ are also defined as previously in terms of the restricted relations $\mapsto_{r,\eta}$.

Theorem 2. *The relations $\mapsto$ and $\mapsto_\eta$ define bisimilar transition systems.*

5 Reachability Analysis

Bounded visible heap depth programs: Let $k \in \mathbb{N}$. A visible heap $VH = (A, \Delta, \Gamma, \Lambda, C)$ is k-bounded if (1) $|Reach_\Delta(\Lambda(L) \cup \Gamma(G))| \leq k$, and (2) $\forall i \in [1, m].\ |Reach_\Delta(C(i))| \leq k$. Notice that k-boundedness does not imply that $|A| \leq k$ since there may exist vertices which are reachable from cut points but not from local/global variables. A sequence $\langle n_1, VH_1, \pi_1\rangle \ldots \langle n_j, VH_j, \pi_j\rangle \in \mathbb{V}^*$ is k-bounded iff $\forall i \in [1, j]$, VH_i is k-bounded. A configuration $((i, \overrightarrow{\Pi}), [\alpha_1, \ldots, \alpha_m])$ is k-bounded iff $\forall j \in [1, m]$, α_j is k-bounded.

Theorem 3. *Given $k \geq 1$, for every k-bounded $\Rightarrow$-computation in the CH-PDS model of a program there is a bisimilar k-bounded $\mapsto$-computation in its CVH-PDS model.*

Bounded heap depth reachability analysis of sequential programs: We extend to VH-PDS the automata-based construction of $post^*/pre^*$ images for pushdown systems (see, e.g., [1,5,4]). We assume in the sequel that visible heaps are in normal form according to some fixed ordering η on local and global variables. Sets of configurations are recognized by finite-state automata over the alphabet of normalized environments called *Conf-automata*. More precisely, initial states in these automata correspond to mappings Π, and edges are labeled by elements of $\mathbb{V}_\eta$. Then, a configuration (Π, α) is accepted by the automaton if starting from the initial state Π there is an accepting run for the sequence $\alpha \in \mathbb{V}_\eta^*$.

Given $k \geq 1$, and a regular set of k-bounded (local) configurations recognized by a Conf-automaton $\mathcal{A}$, we apply a saturation based algorithm $\mathsf{Closure}_\eta(\mathcal{A}, k)$ which constructs a sequence of Conf-automata with increasing languages, each of them being obtained from the previous one using one of the transition rules of the pushdown system. The difference here with the existing constructions for pushdown systems is that

the stack alphabet of the built automata may increase at each step. Therefore, we restrict the construction to the k-bounded environments. Then, by Proposition 2, the algorithm terminates and produces a finite Conf-automaton representing all (forward/backward) reachable configuration by k-bounded $\mapsto_\eta$-computations, which is a subset of the set of all reachable configurations from $L(\mathcal{A})$ (by Theorems 1 and 2), and which contains the set of all reachable configurations by k-bounded $\Rightarrow$-computations (by Theorem 3).

Reachability analysis of concurrent programs: Let us first introduce some definitions and notations: A *special* Conf-automaton is a Conf-automaton for which there exists a pair (Π, e) such that, for every local configuration $(\Pi', \alpha) \in L(\mathcal{A})$, $\Pi' = \Pi$ and there exists $\alpha' \in \mathbb{V}_\eta^*$ such that $\alpha = e\alpha'$, i.e., all words accepted by $\mathcal{A}$ start with (Π, e). The pair (Π, e) is then denoted $\widehat{\mathcal{A}}$. An *aggregate* is a tuple $(\mathcal{A}_1, \ldots, \mathcal{A}_m)$ of special Conf-automata. Such an aggregate defines the set of global configurations $L(\mathcal{A}_1, \ldots, \mathcal{A}_m) = \{((i, \overrightarrow{\Pi}), [\alpha_1, \ldots, \alpha_m]) \ : \ i \in [1,m], \forall j \in [1,m].\ (\overrightarrow{\Pi}(j), \alpha_j) \in L(\mathcal{A}_j)\}$. A finite set of aggregates defines the union of the languages defined by all its elements.

Given a Conf-automaton $\mathcal{A}$ and (Π, e) with $e \in \mathbb{V}_\eta$, let $\mathcal{A}^{(\Pi,e)}$ be an automaton recognizing the language $L(\mathcal{A}) \cap (\Pi \times e\mathbb{V}_\eta{}^*)$. Clearly, $\mathcal{A}^{(\Pi,e)}$ is a special automaton. Given (Π, e) and a special automaton $\mathcal{A}$, we denote by $(\Pi, e) \rhd \mathcal{A}$ the special automaton recognizing the language $\{(\Pi, e\alpha) \mid (\Pi', e'\alpha) \in L(\mathcal{A})\}$, i.e., the language of $\mathcal{A}$ where the first symbol of every word is replaced by e and the initial state is replaced by Π.

We consider w.l.o.g. that the reachability analysis starts from a single initial configuration where the heap is empty and all pointer variables are equal to *null*: For each $i \in [1,m]$, let $\mathcal{A}_\emptyset^i$ be the special automaton recognizing the (singleton) language $\{(Id_{\{\top,\bot\}}, \langle n_{init}^i, VH_\emptyset, Id_{\{\top,\bot\}}\rangle)\}$, where n_{init}^i is the entry node of the main procedure of process i, and $VH_\emptyset = (\{\top, \bot\}, \Delta_\top, \Gamma_\bot, \Lambda_\bot, \overrightarrow{\emptyset})$ with $\Delta_\top$ being the function mapping $\top$ to each address for all successor fields, $\Gamma_\bot$ (resp. $\Lambda_\bot$) being the functions mapping $\bot$ to each global (resp. local) variable, and $\overrightarrow{\emptyset}$ being the vector of functions mapping an empty set of cut points to each process.

Finally, given $h_1, \ldots, h_m$ such that for every $\ell \in [1,m]$, $h_\ell = (\overrightarrow{\Pi}(\ell), e_\ell)$, and given $i, j \in [1,m]$, $i \neq j$, we denote by $\mathsf{Update}_{i,j}(h_1, \ldots, h_m)$ the set of tuples $(h'_1, \ldots, h'_m)$ such that: (1) for every $\ell \in [1,m]$, $h'_\ell = (\overrightarrow{\Pi'}(\ell), e'_\ell)$, and (2) $((i, \overrightarrow{\Pi}), [e_1, \ldots, e_m]) \hookrightarrow_{\mathsf{sw}} ((j, \overrightarrow{\Pi'}), [e'_1, \ldots, e'_m])$, where $\hookrightarrow_{\mathsf{sw}}$ refers to the Switch inference rule (see section 4.3).

We are now ready to present our bounded reachability analysis algorithm. The algorithm is given in Figure 1. The input is a CVH-PDS, the bound on context switches K, and the bound on the size of visible heaps k. The algorithm computes (in *Reach*) the set of all reachable configurations by k-bounded $\rightsquigarrow_\eta{}^K$-computations: A set of *tasks* is maintained in a structure *todo*. A task is a triple $(\ell, i, \mathcal{A})$ where ℓ is the number of context switches done so far, i is the index of the process chosen to be active, and $\mathcal{A}$ is an aggregate. Initially, *todo* contains the initial configuration of the program with all possible starting active process (lines 2-3). Then, the treatment of a task is as follows: if ℓ has already reached K then $\mathcal{A}$ is added to *Reach* (lines 6-7). Otherwise, the set of reachable k-bounded configurations from $\mathcal{A}$ by process i is computed (line 9), and then new tasks are produced corresponding to all possible context switches from i to some other

Input: An m-dim CVH-PDS, and two integers $K, k \geq 1$
Output: The set of reachable configurations by k-bounded $\leadsto_{\eta}^{K}$-computations.

```
1  Reach ← ∅;
2  for all i ∈ {1,...,m}
3     todo ← {(0, i, A_∅^1, ..., A_∅^m)};
4  while todo ≠ ∅ do
5     pop (level, i, A_1, ..., A_m) from todo with minimal level;
6     if level = K then
7        Reach ← Reach ∪ {(A_1, ..., A_m)};
8     else
9        let B_i = Closure_η(A_i, k) in
10          for all (Π, e_i') such that L(B_i^(Π,e_i')) ≠ ∅
11             for all j ∈ {1,...,m} such that j ≠ i
12                for all (h_1, ..., h_m) ∈ Update_{i,j}(Â_1, ..., (Π, e_i'), ..., Â_m)
13                   todo ← (level + 1, j, h_1 ▷ A_1, ..., h_i ▷ B_i^(Π,e_i'), ..., h_m ▷ A_m);
14 return Reach
```

Fig. 1. Algorithm for bounded context-switch/heap depth reachability on CVH-PDS

process j (lines 10-13). For that, a case splitting is performed according to all possible visible heaps reached by process i (line 10), corresponding to head environments of its possible stacks. Then, the local heaps of all processes are updated, and tasks (where ℓ is incremented) are defined for all possible next active processes $j \neq i$ (lines 11-12) and added to $todo$ (line 13).

References

1. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, Springer, Heidelberg (1997)
2. Bouajjani, A., Esparza, J., Schwoon, S., Strejcek, J.: Reachability analysis of multithreaded software with asynchronous communication. In: Ramanujam, R., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, Springer, Heidelberg (2005)
3. Bouajjani, A., Fratani, S., Qadeer, S.: Bounded context switch analysis of mutithreaded programs with dynamic linked structures. Technical Report, 2007-02, LIAFA lab (January 2007) Available at `http://www.liafa.jussieu.fr/~abou/publis.html`
4. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, Springer, Heidelberg (2000)
5. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. Electr. Notes Theor. Comput. Sci., 9 (1997)
6. Godefroid, P.: Model checking for programming languages using Verisoft. In: POPL 1997, ACM Press, New York (1997)
7. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI 1997, ACM Press, New York (2007)
8. Qadeer, S., Rajamani, S.: Deciding assertions in programs with references. Technical Report MSR-TR-2005-08, Microsoft Research (September 2005)

9. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, Springer, Heidelberg (2005)
10. Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: POPL 2005, ACM Press, New York (2005)
11. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, Springer, Heidelberg (2005)