Accepted for NSAD 2019 informal proceedings (https://staticanalysis.org/nsad2019/)

Towards an abstraction for data structures that implement cooperation mechanisms

Guillaume Cluzel[1] and Cezara Drăgoi[2]

[1] École Normale Supérieure de Lyon
[2] INRIA, ENS, CNRS, PSL

Abstract. Abstract interpretation techniques for concurrent programs are one of the main current challenges for static analysis. In this paper we propose a trace abstraction for concurrent data structures that implement cooperation mechanisms, for example java.exchanger or stack elimination. Our abstraction is designed for programs with unboundedly many threads and unbounded data structures.

1 Introduction

The design of concurrent data structures is an active research area. The first approach towards concurrent data structures uses locks, that ensure synchronization between threads by allowing at most one thread to write (read) a shared resource. The problem with lock-based implementations is that they are blocking, i.e., if the thread holding the lock is suspended or crashes before releasing the lock, the entire execution blocks, no other thread being able to acquire a non-released lock. Consequently, non-blocking implementations were developed, where the threads that cannot access the shared memory are not blocked, instead of retry, or report the operation as unsuccessful. Typical examples are Treiber's stack [9] or Michael-Scott queue [11] (they are implemented using instructions like compare-and-swap). However, even for these data structures, concurrent accesses are sequentialized at one or two entry points of the data structure, despite its potentially unbounded size. Therefore, the latter data structures suffer also from a performance bottleneck, all accesses to the data structure being done at the top of the stack, respectively the head/tail of the queue, therefore the degree of parallelism is heavily restricted.

The advantage of these data structures is that they have a sequential specification, easily understood by programmers. More precisely, the standard correctness criteria for concurrent data structures is linearizability, which says that a client cannot distinguish the concurrent implementation from a sequential (atomic) one. The client communicates with the data structure using an API, and the client cannot distinguish a concurrent implementation of the methods in the API from a sequential one, that we refer to as the specification. The methods in the library represent the different operation that the library can perform. Given their sequential specifications, verification methods [1, 17, 18] and tools [17] have been developed, that check a simulation relation between concurrent executions

Accepted for NSAD 2019 informal proceedings (https://staticanalysis.org/nsad2019/)

of the (concurrent) implementation and sequential executions in the specification. The simulation is based on the user annotations the concurrent program with the so-called linearization points. When a concurrent execution reaches a linearization point, in the corresponding sequential execution a new operation is added, matching the effect of the concurrent operation whose linearization point was reached.

More recent non-blocking data structures like, stack-elimination [9] or combined funnels [14], increase the parallel access to the data structure by implementing a *collaboration mechanism*. Roughly, two concurrent operations, e.g., a push and a pop, are allowed to exchange values without accessing the shared stack, using instead a pair of registers. The performance of these data structures emerges in heavily concurrent environments where hundred of threads try to access the same data structure in parallel. The implementation of stack-elimination uses an auxiliary (elimination) array, where each array cell is a pair of registers that allows two operations to exchange values. The size of the array determines the number of operations that can be done in parallel. The exchange mechanism is implemented also as a standalone data structure, `java.exchanger`, in the library `java.concurent`, being available as a synchronization primitive between two threads. All collaboration mechanisms are based on a shared memory implementation of a message passing protocol between the threads that exchange values. Therefore these data structures have a concurrent specification [8], that captures the effect of the collaboration between two threads.

The verification of data structures that have a concurrent specification has been less studied in the literature. With the exception of [7] all other formal methods techniques tackle the bug finding problem [3, 4] or study decidability issues when the number of threads and the size of the data structure are bounded. In [7] the verification method proposed does not address directly the verification of cooperation mechanisms, instead it reduces it to the verification of a library that has a sequential specification, and can be addressed with existing verification techniques.

In this paper we study the verification problem for concurrent data structures that have concurrent specifications. We define a trace abstraction of concurrent executions that captures relations between threads, namely the essence of the collaboration between two threads. The main ingredients of the abstraction are 1) it uses a bounded number of threads to abstract the behavior of unboundedly many threads, 2) for each operation it remembers a subset of the events produced, i.e., the call, the return, and writes to the share memory, and 3) it is parametrized by an abstract domain that captures the global state of the program projected on a bounded number of threads. This is the first approach, as far as we know, that proposes an abstract domain for proving correctness of concurrent data structures that have a concurrent specification. Our results are preliminary, we have manually tried our abstraction on a few algorithms like, `java.exchanger`, (a simpler version of) stack-elimination, and validity, a concurrent object with one method that tries to write a value and returns either the

value it wrote if the write was successful, or the value written by a concurrent operation, whose write was unsuccessful.

The paper presents in Section 2 the syntax and semantics of the language we consider, Section 3 presents the abstract domain we propose to capture the interference between threads, in Section 4 we give an example of analysis, and finally related works and conclusions are discussed in Section 5.

2 Description of the language

2.1 Syntax

$$
\begin{aligned}
\mathtt{p} &::= \mathtt{s}_{init}; (\mathtt{s}_1 \parallel \cdots \parallel \mathtt{s}_n \parallel \cdots) \\
\mathtt{s} &::= \textbf{if } \mathtt{c} \textbf{ then } \mathtt{s} \textbf{ else } \mathtt{s} \mid \textbf{while } \mathtt{c} \textbf{ do } \mathtt{s} \mid x := \mathtt{e} \mid *p := \mathtt{e} \mid \mathtt{s}; \mathtt{s} \\
&\quad \mid \mathsf{ATOMIC}() \mid \mathsf{END\_ATOMIC}() \\
\mathtt{e} &::= n \in \mathbb{Z} \mid x \in \textsc{Nvar} \mid *p \in \textsc{Pvar} \mid \mathtt{e} + \mathtt{e} \mid \mathtt{e} - \mathtt{e} \mid \mathtt{e} \times \mathtt{e} \\
\mathtt{c} &::= \mathtt{e} = 0 \mid \mathtt{e} \neq 0 \mid \mathtt{e} < 0 \mid \cdots
\end{aligned}
$$

Fig. 1. Syntax of our concurrent language

We consider a simple language that is described in figure 1. Let Vars be the set of program variables, with Nvar the set of variables of basic types and Pvar the set of variables of composite (record) type. Without loss of generality we consider only integer variables and pointers to user defined struct types. Furthermore we distinguish the shared variables, of basic or composite type, and we denote them by SNvar and SPvar. The shared variables are declared with the **global** keyword.

For simplicity, we will group different variables inside a structure, like in C language to allow us to create more powerful structures. We will define this structure with the keyword **struct** following by the name of the struct, and access its members by *structvar.fieldname*. For simplicity, we consider that our structures are not recursive.

We introduce the operator of parallel composition $\parallel$. The rule $\mathtt{p}$ means that a program fist calls an *init* function which corresponds to the initialization of the shared variables used by the data structure and then it calls an unbounded number of threads $t_1, \ldots, t_n, \ldots$ that execute in parallel $\mathtt{s}_1, \ldots, \mathtt{s}_n, \ldots$ respectively. We denote as $\mathcal{T}$ the set of all running threads.

We also introduce two built-in functions ATOMIC() and END_ATOMIC() that are used in the way ATOMIC(); s; END_ATOMIC() to make the statements s atomic, which means that no process can interfere and the code s is executed without interruption.

We will also group different instructions inside functions that can take arguments, and can return a value. A function call can be used as an expression, and we note as $f(\mathsf{e}_1, \ldots, \mathsf{e}_n)$ a function call where f is the function name and $\mathsf{e}_1, \ldots, \mathsf{e}_n$ are the arguments of the function.

2.2 Concrete state semantic

The state of a program is given by an evaluation of the shared variables and an evaluation of the local variables of each thread. Let $t_1, \ldots, t_n$ be n threads running a program $\mathcal{P}$.

A memory state $\sigma = (\sigma_s, \sigma_1, \ldots, \sigma_n)$ is an evaluation of the program variables, where σ_s is an evaluation of the shared variables and σ_i is an evaluation of the local variables of thread t_i, for each $i \in [1, n]$. We define $\sigma(x) = \sigma_s(x)$ if x is a shared variable, otherwise $\sigma(x) = \sigma_i(x)$ if x is a local variable of t_i, for some $i \in [1, n]$. We denote $\mathbb{M}$ the set of all memory states.

We consider the heap as a labeled graph $G = (V, E)$, where nodes in V are objects of struct type, and an edge in E represents the interpretation of pointer fields of struct objects. For simplicity, we model numeric fields as node labels. Therefore, $\sigma_s(x), \sigma_i(x)$ is an integer if x is a basic type variable, and $\sigma_s(x), \sigma_i(x)$ is a node in the graph, representing memory location storing an object of the x's type, if x is a pointer variable.

For each comparison, expression and statement we can define a concrete semantic $[\![\mathsf{c}]\!] : \mathbb{M} \to \mathsf{bool}$, $[\![\mathsf{e}]\!] : \mathbb{M} \to \mathbb{M}$ and $[\![\mathsf{s}]\!] : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M})$ for all $\sigma \in \mathbb{M}$ and $\mathcal{E} \in \mathcal{P}(\mathbb{M} \to \mathbb{M})$ by:

$$
\begin{aligned}
&[\![n]\!](\sigma) = n \qquad n \in \mathbb{Z} \\
&[\![x]\!](\sigma) = \sigma(x) \qquad x \in \textsc{Vars} \\
&[\![\mathsf{e}_1 \diamond \mathsf{e}_2]\!](\sigma) = [\![\mathsf{e}_1]\!](\sigma) \diamond [\![\mathsf{e}_2]\!](\sigma) \qquad \diamond \in \{+, -, \times\} \\
&[\![\mathsf{e} \bowtie 0]\!](\sigma) = \begin{cases} \mathsf{t} & \text{if } [\![\mathsf{e}]\!](\sigma) \bowtie 0 \\ \mathsf{f} & \text{if } [\![\mathsf{e}]\!](\sigma) \not\bowtie 0 \end{cases} \qquad \bowtie \in \{=, \neq, >, \ldots\} \\
&[\![x := \mathsf{e}]\!](\mathcal{E}) = \{\sigma[x \leftarrow [\![\mathsf{e}]\!](\sigma)] \mid \sigma \in \mathcal{E}\} \\
&[\![\textbf{if } \mathsf{c} \textbf{ then } \mathsf{s}_1 \textbf{ else } \mathsf{s}_2]\!](\mathcal{E}) = [\![\mathsf{s}_1]\!]\{\sigma \in \mathcal{E} \mid [\![\mathsf{c}]\!](\sigma) = \mathsf{t}\} \cup [\![\mathsf{s}_2]\!]\{\sigma \in \mathcal{E} \mid [\![\mathsf{c}]\!](\sigma) = \mathsf{f}\} \\
&[\![\mathsf{s}_1; \mathsf{s}_2]\!](\mathcal{E}) = [\![\mathsf{s}_2]\!]([\![\mathsf{s}_1]\!](\mathcal{E}))
\end{aligned}
$$

To describe the semantic of the loops we use the notion of the least fix-point (lfp):

$$
[\![\textbf{while } \mathsf{c} \textbf{ do } \mathsf{s}]\!](\mathcal{E}) = \left\{ \sigma \in \bigcup_{i \in \mathbb{N}} F_b^i(\mathcal{E}) \;\middle|\; [\![\mathsf{c}]\!](\sigma) = \mathsf{f} \right\} = \operatorname{lfp} F_b
$$

$$
\begin{aligned}
\text{with } F_b : \mathcal{P}(\mathbb{M}) &\longrightarrow \mathcal{P}(\mathbb{M}) \\
\mathcal{E} &\longmapsto [\![\mathsf{s}]\!](\{\sigma \in \mathcal{E} \mid [\![\mathsf{c}]\!](\sigma) = \mathsf{t}\})
\end{aligned}
$$

For the call of the function, we use the call-by-value semantic.

The semantic of our concurrent programs that we consider is the *interleaving semantic*. After each execution of an atomic bloc of instructions, any other thread can execute an atomic bloc of instructions. This behavior of concurrent programs can be modeled by a transition system (Σ, I, τ) where Σ is the set of every reachable memory states, I is the initial memory state after having executed the *init* method. The transition function τ is derived from the sequential execution of each thread, where $\mathtt{s}_t$ always represents an atomic bloc in this case.

$$\tau_t(\mathcal{E}) := [\![\mathtt{s}_t]\!](\mathcal{E})$$

This states that each step of the program execution is the execution of any single thread t which only updates its local memory and the global memory and leaves the memory of the other threads unchanged. Thus, we can define $\tau = \bigcup_{t \in \mathcal{T}} \tau_t$.

The set of reachable states of the whole program can be defined with a fixpoint.

$$[\![\mathtt{s}_1 \parallel \cdots \parallel \mathtt{s}_n \parallel \cdots]\!](\{I\}) = \bigcup_{i \geq 0} \tau^i(\{I\}) = \mathrm{lfp}\, \tau$$

The semantic of the two built-in functions ATOMIC() and END_ATOMIC() is different from the other instructions. They are used by the scheduler to know that the code inside an atomic bloc cannot be interrupted by another thread. The transition function τ can only be applied on atomic bloc as specified previously. For example, for two threads:

$$[\![\mathsf{ATOMIC}(); \mathtt{s}_1; \mathsf{END\_ATOMIC}() \parallel \mathtt{s}_2]\!] = [\![\mathtt{s}_1; \mathtt{s}_2]\!] \cup [\![\mathtt{s}_2; \mathtt{s}_1]\!]$$

2.3 Concrete trace semantic

We can use another semantic with our parallel programs which is a trace semantic. Better than keeping the memory states at each point of the program, we can only keep the instructions executed, and we keep the order in which they were executed. A trace is a sequence of atoms of the form $(t, \mathtt{op})$ where $t \in \mathcal{T}$ is the thread that executes the operation `op`. It can be an assignment, a comparison or an arithmetic operation. We will also add the call of function of the form **inv** `fun`, and the return of function with the value returned **ret** `fun` $\triangleright$ *val*. We denote as $\mathcal{T}r$ the set of all traces T.

We notice that the trace is more expressive than the memory state at a given point of the program because we can obtain the memory state from the trace. Indeed, if we execute the different operations in the trace we can compute the memory state exactly like if we executed the program. But the traces keep more information. They can express relations between threads.

3 Description of the abstract domain

Definition 1. *An* ***abstract trace*** $T \in \mathcal{T}r^\sharp$ *is a partially ordered sequence of atoms* $a_1, \ldots, a_n$*, that we denote as* $T = a_1 \cdot \ldots \cdot a_n$*. These atoms can be an*

invocation of a function $(t, \textbf{inv } \texttt{f})$*, a return of a function* $(t, \textbf{ret } \texttt{f} \rhd v)$*, or a write on the shared memory. Let* $a \prec a'$ *if* $\mathrm{tid}(a) = \mathrm{tid}(a')$ *(where, if* $a = (t, \texttt{op})$*,* $\mathrm{tid}(a) = t$*) and* a *is executed before* a' *or if there exists a data dependency between* a *and* a' (i.e. a' *involves data written by* a*). A trace is* ***complete*** *if every invocation has a matching response. The* ***contretization*** $\gamma_{\mathcal{T}r}$ *of an abstract trace* T *is the set of all concrete traces that contain the atoms of* T*, that respect the order verified by the atom in* T *and that have a shared memory state at the end of the execution included in the concretization of the corresponding abstract memory state.*

The partial order $\prec$ on the atoms induces a partial order on the abstract traces, *i.e.* if $T = T_1 \cdot T_2$, we said that $T_1 \prec T_2$ if there exist $a \in T_1$ and $a' \in T_2$ such that $a \prec a'$.

Definition 2. *Two abstract traces* $T = a_1 \cdot \ldots \cdot a_n$ *and* $T' = a'_1 \cdot \ldots \cdot a'_k$ *are* ***equal*** *if there exists a bijective function* $\varphi : \{a_1, \ldots, a_n\} \to \{a'_1, \ldots, a'_k\}$ *that conserves the order.*

The equality between two traces takes into account that we can reorder the atoms in a trace because some of then are not ordered. This definition implies that the two traces must have the same length, and then n must be equal to k in the definition above. If for example we consider a trace $T = a_1 \cdot a_2 \cdot a_3$ with the only relation between $a_1 \prec a_3$ the atoms, then the trace $T' = a_2 \cdot a_1 \cdot a_3$ is equal to T.

Definition 3. *An abstract trace* T *is* ***reduced*** *if for all traces* T_1 *and* T_2 *such that* $T = T_1 \cdot T_2$ *then* $T_1 \prec T_2$*.*

Example 1. We consider the trace $T = (t, \textbf{inv } \texttt{f}) \cdot (t, \texttt{X} = \texttt{v}) \cdot (t, \textbf{ret } \texttt{f} \rhd 0)$. This trace T because all the atom in the trace are ordered by the execution order.

Now we consider another trace $T' = (t, \textbf{inv } \texttt{f}) \cdot (t', \textbf{inv } \texttt{g}) \cdot (t, \texttt{X} = \texttt{v}) \cdot (t', \textbf{ret } \texttt{g} \rhd 0) \cdot (t, \textbf{ret } \texttt{f} \rhd 0)$ and we state that there is no data dependence between the threads t and t'. This trace is not reduced because, thanks to 2, we can reorder the atoms of T' such that $T' = (t, \textbf{inv } \texttt{f}) \cdot (t, \texttt{X} = \texttt{v}) \cdot (t, \textbf{ret } \texttt{f} \rhd 0) \cdot (t', \textbf{inv } \texttt{g}) \cdot (t', \textbf{ret } \texttt{g} \rhd 0)$. Since there is no interaction between the two thread, we have rewritten T' in the form $T' = T_1 \cdot T_2$ and we don't have $T_1 \prec T_2$. Then we obtain that T' is not reduced.

We do not want to make our traces infinite with irrelevant partial traces. That could happen when we will have finished our analysis and we reanalyze a function that have no dependency with the previous traces already found. Thus, we consider the function $\mathrm{red} : \mathcal{T}r^\sharp \to \mathcal{T}r^\sharp$ that associates to an abstract trace T the smallest reduced trace:

$$\mathrm{red}(T) = \min\{T_0 \mid \exists T_1, T = T_0 \cdot T_1 \wedge \mathrm{Mem}(T_0) \sqsubseteq \mathrm{Mem}(T)\} \tag{1}$$

where $\mathrm{Mem}(T)$ is the abstract memory state after execution of the operation in the trace T.

More than an abstract trace, we need to keep the abstract memory state. Our aim is to connect the different values input and output by the function. For that, we need to use a relational abstract domain to represent the relations between the values of the variables. We can use, for example, the octagon abstract domain [12] to express these relations and we will use this abstract domain to represent the abstract shared memory.

Definition 4 (Abstract value). *An abstract value is a disjunction of pairs* (T_i, S_i) *where* $T_i \in \mathcal{T}r^\sharp$ *is the abstract trace and* S_i *is the abstract shared memory state corresponding to the abstract trace* T*, existencially quantified on the threads* $t_1, \ldots, t_n$*. The* ***concretization*** γ *associate the most general concrete traces and abstract states,* i.e.

$$\gamma\left(\bigvee_i (T_i, S_i)\right) = \bigcup_i \{(T, M_{T_i}) \mid T \in \gamma_{\mathcal{T}r}(T_i)\}$$

where M_{T_i} *is the memory state after the execution of the trace* T*.*

The join of two abstract values $A_1 = (T_1, S_1)$ and $A_2 = (T_2, S_2)$ cannot only be the join of the abstract memory states S_1 and S_2, because the traces T_1 and T_2 that have lead to these memory states are not necessary the same. If we analyze the code **if** $\text{rand}(0,1) = 0$ **then** $x := 1$ **else** $x := 2$ on a thread t, we will get after the affectation in the then part the abstract value $(T_1 = T \cdot (t, x := 1), \{x = 1\})$ and in the else part $(T_2 = T \cdot (t, x := 2), \{x = 2\})$. Since the two traces T_1 and T_2 are not equal, we cannot consider to join the two abstract traces into a unique trace. We would lose too much information.

Definition 5 (Join). *Let* $A = \bigvee_i (T_i, S_i)$ *and* $A' = \bigvee_j (T'_j, S'_j)$ *two abstract values. We define*

$$A \sqcup A' := \left(\bigvee_{\substack{i,j \\ T_i \neq T'_j}} (T_i, S_i)\right) \vee \left(\bigvee_{\substack{i,j \\ T_i = T'_j}} (T_i, S_i \sqcup S_j)\right)$$

Widening We can extend the notion of reduction to an abstract value, and

$$\text{red}((T, M)) = (\text{red}(T), \text{Mem}(\text{red}(T)) \sqcup \text{Mem}(T)).$$

This definition would useful to define the widening operator.

The idea of the widening here is to reduce the trace to only keep the relevant part. In particular, in the concurrent data structures we consider, at the end of every loop the global state is reset to have to its initial value to avoid to lock the library. We haven't found a widening that works in every case, but for our needs we only need the following property of the widening:

$$\begin{cases} T_1 \\ S \cup \{\mathtt{X} = 0\} \end{cases} \nabla \begin{cases} T_1 \cdot (t, \mathtt{X} = 1) \cdot (t, \mathtt{X} = 0) \\ S \cup \{\mathtt{X} = 0\} \end{cases} = \begin{cases} T_1 \\ S \cup \{\mathtt{X} = 0\} \end{cases}$$

This can be obtain by using the red operation.

stabilization The red function is not only used during the widening of the loop. It is also used during the stabilization operation. The stabilization might increase the length of the trace infinitely with irrelevant parts in the trace that are the traces of other threads without interaction with the analyzed thread. For example, if we consider a stack, where a push and a pop can exchange their value without using the stack (see the next part), we only want to capture the operation push+pop, and we are not interested in the other execution of pushs and pops. Then, to prevent our traces to be similar to the following trace

$$T = (t, \textbf{inv pop}(3)) \cdot (t', \textbf{inv pop}(2)) \cdot (t', \textbf{ret pop} \rhd 0) \cdot (t, \textbf{ret pop} \rhd 0)$$

we will reduce it to keep only the trace minimal. Here the minimal trace corresponds to $T = \varepsilon$ because the abstract state before the execution of the two function is the same that the initial abstract state.

4 Example of analysis

```
1  int F = 0, W = 1, C = 2;
2  struct exchanger {
3    int flag, value;
4  };
5  exchanger E
6        := {flag=F,value=0};
7
8  int push(e) {
9    while (1 = 1) {
10     ATOMIC();
11     if (E.flag = W) {
12       E.flag := C;
13       E.value := e;
14       END_ATOMIC();
15       return 1;
16     }
17     END_ATOMIC();
18   }
19 }
20
```

```
21 int pop() {
22   while (1 = 1) {
23     ATOMIC();
24     if (E.flag = F) {
25       E.flag := W;
26       END_ATOMIC();
27       ATOMIC();
28       if (E.flag ≠ C) {
29         E.flag := F;
30       }
31       else {
32         E.flag := F;
33         int l := E.value;
34         END_ATOMIC();
35         return l;
36       }
37     }
38     END_ATOMIC();
39   }
40 }
```

Fig. 2. Exchanger stack

In this section we show an example of static analysis using our abstract domain. The fixpoint computation we use is the one defined in [18] where thread interfaces (rely-garantee actions) capture relations between relation of thread operations where instead of using thread modular abstract of the state space we use thread interfaces in our abstract domain that capture operation of different

threads. After having analyze each operation, if the shared memory state is modified an *action* is generated. Before analyzing an operation, we stabilize with all the actions that can be performed by the other threads.

We consider the exchanger stack, which is simply the Treiber's stack [9] with an exchange mechanism. An example of a such exchange mechanism is given in the figure 2. The functions `push` and `pop` can also write on the stack but we do not consider this part, and we focus on the exchange mechanism.

In this stack, a `push` and a `pop` can exchange their value without using the stack. If the value of the variable `flag` is equal to `F` (Final), then a `pop` executed by a thread t can change it to `W` (Waiting) meaning that it is waiting for exchanging its value with a `push`. Meanwhile, a `push` executed by a second thread t' will detect that the value of the `flag` variable is equal to `W`, and the `push` will change it in `C` (Collided) and puts its value `e` in the variable `value`. The thread t that execute the `pop` can resume its execution and since the value of the flag has changed and is now equal to `C` then it resets the flag and returns the value passed by the `push`.

The aim of our analysis is to highlight the dependencies that exist between a push and a pop. For example we will show that for every `pop` that returns a value v, there exists a corresponding `push` that was called with the argument v.

Analysis of the init part The library is first called in a sequential mode to be initialized. Here the initialization corresponds to the line 1 to 6, which is the code executed when the library is initialized. The initialization step only sets the variable `E` to its initial value, which means that we obtain as an initial state for the memory, the state $\{\texttt{E.flag} = \texttt{F} \wedge \texttt{E.value} = 0\}$.

First step of the analysis The analysis starts with an empty trace and the initial state for the shared memory discovered in the previous step. We start, for example, by analyzing the `pop` function on a thread t. We could have chosen another function to start our analysis, this choice is random.

We add to the trace the atom $(t, \textbf{inv}\ \texttt{pop}())$. At line 24, since the abstract state contains the constraint $\texttt{E.flag} = \texttt{F}$, we analyze the following code and in particular the write of the shared memory at line 25. Since the line 24 and 25 are into an atomic bloc, we have to analyze both lines together without considering interference by other thread between them, which mean that we do not stabilize with the actions produced by other threads (even if, at this point we have not generated any action yet).

At this point we add an atom at the end of the trace and we modify the abstract state that represents the memory state. Since the shared memory has been written, an action is generated, which is the transition between the previous write on the memory (or the initial state if there is no previous write) and the state after the write on the memory. In our case, the action generated is the

following:

$$\begin{cases} T = \varepsilon \\ M = \{\texttt{E.flag} = \texttt{F} \wedge \texttt{E.value} = 0\} \end{cases} \rightsquigarrow \begin{cases} \exists t, \\ T = (t, \textbf{inv pop}()) \cdot (t, \texttt{E.flag} = \texttt{W}) \\ M = \{\texttt{E.flag} = \texttt{W} \wedge \texttt{E.value} = 0\} \end{cases}$$

The abstract state generated by the action contains a "$\exists t$" which means that the action generated by the current thread become an action generated by any thread. Thanks to this tricks we can keep the thread identity.

If we continue the analysis, we obtain that `E.flag` is flipped back to `F` because we know in this case that `E.flag` is different than `C`. At this point we have to generate another action because we have written the shared memory, and then we go back at the beginning of the loop. When we apply the widening operator after one iteration in the loop, we get the same abstract state than before the first iteration because:

$$\begin{cases} T = (t, \textbf{inv pop}()) \\ M = \{\texttt{E.flag} = \texttt{F} \wedge \texttt{E.value} = 0\} \end{cases} \triangledown \begin{cases} T = (t, \textbf{inv pop}()) \cdot (t, \texttt{E.flag} = \texttt{W}) \cdot (t, \texttt{E.flag} = \texttt{F}) \\ M = \{\texttt{E.flag} = \texttt{F} \wedge \texttt{E.value} = 0\} \end{cases}$$
$$= \begin{cases} T = (t, \textbf{inv pop}()) \\ M = \{\texttt{E.flag} = \texttt{F} \wedge \texttt{E.value} = 0\} \end{cases}$$

Then the analysis of the `pop` is finished.

Second step of the analysis Now the analysis uses another thread t' to analyze a function of the library. We will consider that the function `push` is analyzed during the second step. Before the call of the `push` function, we have to stabilize with all the possible actions from the other thread. Applying the reduce operation on top of the stabilization give us the following initial abstract state for the `push` analysis.

$$\begin{cases} T = \varepsilon \\ M = \{\texttt{E.flag} = \texttt{F} \wedge \texttt{E.value} = 0\} \end{cases} \vee \begin{cases} \exists t \\ T = (t, \textbf{inv pop}()) \cdot (t, \texttt{E.flag} = \texttt{W}) \\ M = \{\texttt{E.flag} = \texttt{W} \wedge \texttt{E.value} = 0\} \end{cases}$$

At the line 11, the test imposes to refine the abstract state with only the right part of the disjunction, and then we get a new action after the affectations and the return, at the end of the atomic bloc. After this first iteration we need to widen with the abstract state got at the end of the loop after the first iteration which is:

$$\begin{cases} T = (t', \textbf{inv push}(\texttt{e})) \\ M = \{\texttt{E.flag} = \texttt{F} \wedge \texttt{E.value} = 0\} \end{cases}$$

because the other part of the abstract state was use in the if-then part, and finished with a return. It left only the left part of the initial abstract state that has not changed at the end of the loop.

We get the same abstract value than before the loop after a widening, because we haven't changed anything. This step is finished.

Third step of the analysis We continue our analysis with another thread t. We will analyze for example the `pop` function. We can start by stabilizing with all the possible actions which give us the initial state to start the analysis of the `pop`. After the stabilization and the execution of the call of the function, we might have called 1) no function (then `E.flag` is equal to `F`), 2) only a pop and changed the `flag` value to `W` or 3) a pop and a push and the value of the flag is `C` and the variable `value` contains the value of the argument of the push. To verify the condition at the line 24, it is mandatory that the flag is equal to `F`. In this case, only the 1) in the abstract state verifies this condition.

When the analysis reaches the line 26, the atomic bloc is finished and we can stabilize with all the possible actions. The action

$$\begin{cases} \exists t_1, \\ T_1 = (t_1, \textbf{inv } \texttt{pop}()) \cdot (t_1, \texttt{E.flag} = \texttt{W}) \\ M = \{\texttt{E.flag} = \texttt{F} \wedge \texttt{E.value} = 0\} \end{cases} \rightsquigarrow \begin{cases} \exists t_1, t_2, \\ T = T_1 \cdot (t_2, \textbf{inv } \texttt{push}(e_{t_2})) \cdot (t_2, \texttt{E} = \{\texttt{W}, e_{t_2}\}) \\ M = \{\texttt{E.flag} = \texttt{F} \wedge \texttt{E.value} = e_{t_2} \wedge e_{t_2} = \top\} \end{cases}$$

can be applied now if we suppose that $t_1 = t$. It is the only abstract value that does not verify the condition at line 28. With this abstract value, we generate a new action at line 32, that we can complete with the return that directly follow the write in the global memory.

After one iteration of the loop, we need to widen with our operator ∇. Like in the first iteration, the traces have increased with irrelevant atoms. The widening operation will reduce our traces to remove them and to obtain, like in the first step, a stable abstract value.

Next steps Up to this point we have partially prove what we wanted to obtain, *i.e.* the trace

$$\begin{aligned} T = (t_1, \textbf{inv } \texttt{pop}()) \cdot (t_1, \texttt{E.flag} = \texttt{W}) \cdot (t_2, \textbf{inv } \texttt{push}(e_{t_2})) \cdot (t_2, \texttt{E} = \{\texttt{W}, e_{t_2}\}) \\ \cdot (t_2, \textbf{ret } \texttt{push} \rhd 1) \cdot (t_1, \texttt{E.flag} = \texttt{F}) \cdot (t_1, \textbf{ret } \texttt{pop} \rhd e_{t_2}) \end{aligned}$$

that highlight that the input value of the push is returned by the pop. But the analysis is not finished because the trace might increase infinitely. But as we have designed the stabilization operation, this issue will not happen. In our case, if we consider for example the trace T_0 defined above, we observe that the abstract memory after the execution of T is equal to $M = \{\texttt{E.flag} = \texttt{F} \wedge \texttt{E.value} = \top\}$ and the abstract memory after the execution of the empty trace T_0, is simply the memory in the initial state, which corresponds to $M_0 = \{\texttt{E.flag} = \texttt{F} \wedge \texttt{E.value} = 0\}$. Then we have $M_0 \sqsubseteq M$, and after the reduction, we get the following abstract state:

$$\begin{cases} T = \varepsilon \\ M = \{\texttt{E.flag} = \texttt{F} \wedge \texttt{E.value} = \top\} \end{cases}$$

The analysis will restart with this new abstract value. We will get the same results than previously, with the initial value containing $\texttt{E.value} = 0$, but with a more general beginning value. The analysis will finish with this new abstract value as an initial state.

Finally, we have proved the property we wanted, which was that for every `pop` that returns a value there exists a `push`, and the value returned by the `pop` was an input parameter of the `push`. This property can directly be read on the final traces we obtain when the analysis is finished. This example shows how our abstract domain can be used to prove properties on a concurrent data structures that uses cooperation between threads. Other data structures can be analyzed with this abstract domain, like the Java's exchanger.

5 Related work and conclusions

Static analysis by abstract interpretation is well perfected for numeric programs in the sequential setting and also well studied for the concurrent setting with a bounded number of threads [15, 12, 6]. Several successful approaches exist also for programs manipulating sequential data structures [13, 10, 2]. However, static analysis of concurrent data structures accessed concurrently by an unbounded number of threads is a research area in its beginnings.

The current state of art is the thread modular approach [1, 16, 17], which captures the effect of all threads on the shared memory, but cannot represent the relation between the local states of two threads. Thread modular analysis are incomplete, and cooperation based algorithms do not have a thread modular proof.

The closest related works are [7, 3–5, 8]. In [3–5] the authors give a new definition of correctness for concurrent data structures, that implies linearizability, but does not rely on linearization points. This way of specifying data structures reduces linearizability to a reachability problem, and it was used to develop powerful bug finding techniques based on model checking or SMT-solvers.

In [8] the authors highlight the need for concurrent specifications, for a subclass of concurrent data structures, including some of the structures we are interested in. The paper defines a language to express concurrent specifications and proposes a proof methodology to establish correctness of concurrent implementations with respect to a concurrent specification. No automation technique for these proofs is discussed.

Probably the work closest to our is [7]. It presents a static analysis for concurrent data structures with external linearization points, which in fact correspond to data structures that have a concurrent specification (as shown later in [8]). The paper proposes a static analysis that captures relations between the local state of threads. However, the proof method consists in rewriting the original library into one that has a sequential specification, and static analysis is used to prove the correctness of the rewriting. In the current paper we are interested in proving directly the implementation correct with respect to the concurrent specification, and we design an abstract domain that can capture the concurrent specification and the invariants required to prove it.

In conclusion we study abstract interpretation for data structures that implement cooperation mechanisms. We propose the first abstract that tackles this class of programs. The abstraction captures relations between thread local

states and combines them with a trace abstraction. Our results are preliminary but encouraging. We manually verified (a simpler version of) stack elimination, java.exchanger, and validity.

References

1. Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2008.
2. Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. On inter-procedural analysis of programs with lists and data. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 578–589, 2011.
3. Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. On reducing linearizability to state reachability. In *ICALP (2)*, volume 9135 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 2015.
4. Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. Proving linearizability using forward simulations. In *CAV (2)*, volume 10427 of *Lecture Notes in Computer Science*, pages 542–563. Springer, 2017.
5. Ahmed Bouajjani, Constantin Enea, and Chao Wang. Checking linearizability of concurrent priority queues. In *CONCUR*, volume 85 of *LIPIcs*, pages 16:1–16:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
6. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM Press, 1978.
7. Cezara Dragoi, Ashutosh Gupta, and Thomas A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 174–190. Springer, 2013.
8. Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. Modular verification of concurrency-aware linearizability. In *DISC*, volume 9363 of *Lecture Notes in Computer Science*, pages 371–387. Springer, 2015.
9. Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, pages 206–215. ACM, 2004.
10. Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival. Semantic-directed clumping of disjunctive abstract states. In *POPL*, pages 32–45. ACM, 2017.
11. Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275. ACM, 1996.
12. Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
13. Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
14. Nir Shavit and Asaph Zemach. Combining funnels: A new twist on an old tale.. In *PODC*, pages 61–70. ACM, 1998.
15. Thibault Suzanne and Antoine Miné. Relational thread-modular abstract interpretation under relaxed memory models. In *APLAS*, volume 11275 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2018.
16. Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI*, volume 5403 of *Lecture Notes in Computer Science*, pages 335–348. Springer, 2009.

17. Viktor Vafeiadis. Automatically proving linearizability. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2010.
18. Viktor Vafeiadis. Rgsep action inference. In *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 345–361. Springer, 2010.