On Context Switch Upper Bound for Checking Linearizability

Vadim S. Mutilin

Institute for System Programming, RAS
Moscow, Russia

***Abstract*—Approaches that tackle multithreaded programs suffer from state explosion problem. Promising idea is bounding the number of context switches of running threads. Recent work [10] shows that most bugs can be detected even with two context switches. Despite of the fact that it was successful in practice we still can not be sure that no bug has escaped. In this paper we use context-bounding for checking linearizability property, which proved to be useful both for simplifying specifications and usage of programs and as a common property for finding potential bugs in the same way as race conditions. For linearization we provide an algorithm, which returns an upper bound of context switches. Having the upper bound we can be sure that if the program is not linearizable then context-bounding algorithm will show it.**

I. Introduction

Lately multithreaded programming has become widely spread and gained popularity. Aiming at increasing effectiveness, programs consist of several threads that do the work in parallel. But their development is much harder than development of sequential ones. This is caused by the fact that the scheduling order of instructions from different threads cannot be predicted before actual execution and the developer should foresee correct work of the program for all possible interleavings of instructions.

Here we consider programs providing interface (API), which other programs use for interacting with them. Interface consists of *operations* (procedures) which can be *executed* (invoked) with different values of *parameters*. Operation execution ends with returning a value called a result. Moreover, operations can be executed in different threads simultaneously. Execution of operations in several threads we will call *parallel execution*, and execution in a single thread – *sequential execution*.

Parallel execution is *linearizable*, if it is equivalent to some sequential execution conforming to specification. Formally, the notion of linearizability will be defined in section II. As we can see, the task of checking linearizability is a special case of functional testing where we check whether the program meet functional requirements given as specification. But in contrast to general case, linearizable programs require specification only for sequential executions.

As shown in [7], linearizability is local and nonblocking property. Locality means that if we have shown that operations working with the same object are linearizable then they are linearizable together with any other linearizable operations working with different objects. Hence hereafter we will consider operations working with single shared object. In linearizable program running operation does not require invocation of new operations for its completion. This is nonblocking property. Moreover, proof of properties for programs which use linearizable operations is simplified because behaviour of linearizable program is reduced to sequential executions.

Linearizability has much in common with such properties as serializability [8], [11], atomicity [3], [15], sequential consistency [9]. In contrast to it linearizability requires a specification while these properties impose restrictions on the program only. In some works the term atomicity is used as synonymous to linearizability, in the others it means what we call self-linearizability.

The problem of checking linearizability in general case is not decidable. The key issue is lack of restrictions on the number of threads and operations which occur in parallel executions. There is only manual proof approaches for general case [7], [14]. The automatized approaches work only in the special case, which we consider in our work. In these settings the number of threads is finite and the number of operations in each thread is also finite. For example, in [1] it is shown that with limited number of threads, states of implementation and specification the problem can be solved by model checking.

The principle source of the complexity is a great amount of executions resulting from interleavings of instructions. Partial order reduction, heuristic search, as well as context-bounding methods are used to reduce the amount of them. The last one is used in our paper. The main idea of it is to bound the number of context switches in parallel executions. As shown in [10] in practice tricky errors can be detected even with small number of context switches. But the question of choosing an upper bound of switches which is sufficient for ensuring linearizability is still open. That is the question we give answer here that forms novelty of our work.

Next section gives a formal definition of linearizability. Section III describes a formal model of program. In section IV we prove a theorem on which the algorithm given in section V is based. This algorithm gives an upper bound of context switches sufficient for checking linearizability. Section VI discusses significancy of upper bound produced by the algorithm. Section VII contains comparison with related works.

```
1   volatile int x;
2   volatile boolean b = false;
3
4   boolean insert(int i) {
5     if(b==false) {
6       x=i;
7       b=true;
8       return true;
9     } else return false;
10  }
11  void delete() {
12    b = false;
13    return;
14  }
15
16  int lookUp() {
17    if(b==true)
18      return x;
19    else return -1;
20  }
```

Fig. 1. Cell Example

insert:
$\{true\}$
$op\langle insert\rangle\_begin\langle i\rangle$ $op\langle insert\rangle\_end\langle r\rangle$
$\{r = \neg b \wedge (b \wedge (b' = b) \wedge (x' = x) \vee \neg b \wedge (b' = true) \wedge (x' = i))\}$
delete:
$\{true\}$
$op\langle delete\rangle\_begin\langle\rangle$ $op\langle delete\rangle\_end\langle\rangle$
$\{b' = false\}$
lookUp:
$\{true\}$
$op\langle lookUp\rangle\_begin\langle\rangle$ $op\langle lookUp\rangle\_end\langle i\rangle$
$\{(b' = b) \wedge (b \wedge (i = x' = x) \vee \neg b \wedge (i = -1))\}$

Fig. 2. Cell Specification

II. The Notion of Linearizability

A. History

Suppose we have a set of operations $op\langle name\rangle$, each of which has *begin* (invocation) $op\langle name\rangle\_begin\langle parameters\rangle$ and *end* (response) $op\langle name\rangle\_end\langle result\rangle$.

History is a finite sequence of events α:$op\langle name\rangle\_begin\langle args\rangle$ and α:$op\langle name\rangle\_end\langle res\rangle$, where α is a thread. End *matches* begin, if thread names and operation names agree.

Consider an example of Cell program shown in Fig.1 written in Java language, which will be used hereafter. Examples of histories:

$h_1 =$
$\alpha : op\langle insert\rangle\_begin\langle 0\rangle$,
$\beta : op\langle delete\rangle\_begin\langle\rangle$,
$\alpha : op\langle insert\rangle\_end\langle true\rangle$,
$\gamma : op\langle lookUp\rangle\_begin\langle\rangle$,
$\beta : op\langle delete\rangle\_end\langle\rangle$,
$\gamma : op\langle lookUp\rangle\_end\langle 0\rangle$

$h_2 =$
$\alpha : op\langle insert\rangle\_begin\langle 0\rangle$,
$\alpha : op\langle insert\rangle\_end\langle true\rangle$,
$\gamma : op\langle lookUp\rangle\_begin\langle\rangle$,
$\gamma : op\langle lookUp\rangle\_end\langle 0\rangle$,
$\beta : op\langle delete\rangle\_begin\langle\rangle$,
$\beta : op\langle delete\rangle\_end\langle\rangle$

$h_3 =$
$\alpha : op\langle insert\rangle\_begin\langle 0\rangle$,
$\alpha : op\langle insert\rangle\_begin\langle 1\rangle$,
$\beta : op\langle insert\rangle\_end\langle true\rangle$,
$\beta : op\langle insert\rangle\_end\langle true\rangle$

Definition 1: History is sequential if

1) The first event is a begin of operation.
2) Each event except the last one is immediately followed by matching end.

In the example, history h_2 is sequential and h_1, h_3 are not sequential.

Thread history(projection, subhistory) in a history H ($H \mid \alpha$) is a subsequence of all events in H, which have thread name α. For instance, $h_1 \mid \alpha = \alpha\ op\langle insert\rangle\_begin\langle 0\rangle$, $\alpha\ op\langle insert\rangle\_end\langle true\rangle$. Two histories H, H' are *equivalent*, denoted as $H \sim H'$, if for any thread α subhistory $H \mid \alpha$ equals to $H' \mid \alpha$. In the example $h_1 \sim h_2$. History is a *well-formed* if any subhistory $H \mid \alpha$ is sequential. All histories considered in the paper are well-formed.

Operation is *pending* in a history if some begin is not followed by matching end. $complete(H)$ is maximal subsequence of H consisting only from begins and matching ends (pending operations are removed).

A set S is *prefix-closed* if for any history H in S holds that any prefix of H is also in S. *Sequential specification of a program* is a prefix-closed set if sequential histories. History H *conforms to specification* if $H \in S$. Specification can be presented in different forms. Fig. 2 shows specification of Cell in the form of pre and postconditions.

B. Definition of Linearizability

History H induces irreflexive partial order on operations $<_H$, such that $e_0 <_H e_1$ if $end(e_0)$ precedes $begin(e_1)$ in H

Definition 2: History H is linearizable if it can be extended (appending zero or more responses) to some history H' for which

1) $complete(H')$ is equivalent to some sequential history S which conforms to specification.
2) $<_H \subseteq <_S$.

History h_1 is linearizable, because it is equivalent to h_2 while preserving partial order of operations. History h_3 is not linearizable, because any sequential specification which is equivalent to it contains two sequential successful $insert$ operations that contradicts to specification.

By *reachable* history of a program we shall mean a history which can actually occur in the program. Later on we will define the notion of reachable history on the base of execution trace. Program is *linearizable* if any reachable history is linearizable.

C. Self-linearizability of Program

For checking linearizability it is helpful to define a notion of self-linearizability independent from specification. In work [6] self-linearizability is called atomicity, but there is no formal definition. They introduce atomicity using sufficient conditions. Here we give a formal definition.

Definition 3: Program is self-linearizable if for any reachable history H there exists a reachable sequential history H' such that $H' \sim H$.
If a program is self-linearizable then by checking that all reachable sequential histories conform to specification we show that program is linearizable.

III. Program Model

Program (system, implementation) is a triple

$$Sys = \langle s_0, S, P \rangle$$

where $s_0 \in S$ is an initial state, S is a set of shared states, P is a finite set of opeartion subprograms.

Each subprogram P is a quadruple

$$P = \langle l_0, L, \nu, T \rangle$$

where $l_0 \in L$ is an initial local state, L is a set of local states (control states), $\nu : T \to \Sigma$ is a labeling function, $T \subseteq L \times G \times C \times L'$ is a set of transitions.

$\Sigma = \{\tau, op\_begin\langle parameters\rangle, op\_end\langle result\rangle\}$. All transitions from initial local state are labeled by $op\langle name\rangle\_begin\langle arguments\rangle$, intermediate are transitions labeled by τ. Transitions which are labeled by $op\langle name\rangle\_end\langle result\rangle$ finish the subprogram.

In a transition $c \in C$ is a command $S \to S$ changing the state (instruction, sequence of instructions), $g \in Gd$ is guard condition $S \to \{true, false\}$. We assume that sets of local states of different subprograms do not intersect. Start and end transitions of operations do not change the shared state.

In Cell example, the program model consists of a set S including states of variables x,b and stacks of each subprogram, an initial state $s_0 = \{x = 0, b = false,$ empty stacks $\}$ and a set $P = \{p_{insert}, p_{delete}, p_{lookUp}\}$. Evidently, that subprogram stacks can be separated from S thus showing that this part of the state is accessible to one subprogram only, but this feature is not essential in the paper.

p_{insert}:
$L^i = \{4 \dots 10\}$, $l_0^i = 4$
$T^i = \{\ t_1^i = (4, true, invoke\langle i\rangle, 5)$,
$t_2^i = (5, true, \hat{b} = get\langle b\rangle, 5')$,
$t_3^i = (5', \hat{b} = false, nop, 6)$,
$t_4^i = (5', \hat{b} \neq false, nop, 9)$,
$t_5^i = (6, true, put\langle x, i\rangle, 7)$,
$t_6^i = (7, true, put\langle b, true\rangle, 8)$,
$t_7^i = (8, true, ret\langle true\rangle, 10)$,
$t_8^i = (9, true, ret\langle false\rangle, 10)\ \}$
$\nu(t_1^i) = op\langle insert\rangle\_begin\langle i\rangle$
$\nu(t_7^i) = op\langle insert\rangle\_end\langle true\rangle$
$\nu(t_8^i) = op\langle insert\rangle\_end\langle false\rangle$
For the others $\nu(t^i) = \tau$

p_{delete}:
$L^d = \{11 \dots 14\}$, $l_0^d = 11$
$T^d = \{\ t_1^d = (11, true, invoke, 12)$,
$t_2^d = (12, true, put\langle b, false\rangle, 13)$,
$t_3^d = (13, true, ret, 14)\ \}$
$\nu(t_1^d) = op\langle delete\rangle\_begin\langle\rangle$
$\nu(t_3^d) = op\langle delete\rangle\_end\langle\rangle$
$\nu(t_2^d) = \tau$

p_{lookUp}:
$L^l = \{16 \dots 20\}$, $l_0^l = 16$
$T^l = \{\ t_1^l = (16, true, invoke, 17)$,
$t_2^l = (17, true, \hat{b} = get\langle b\rangle, 17')$,
$t_3^l = (17', \hat{b} = true, nop, 18)$,
$t_4^l = (17', \hat{b} \neq true, nop, 19)$,
$t_5^l = (18, true, \hat{x} = get\langle x\rangle, 18')$,
$t_6^l = (18', true, ret\langle\hat{x}\rangle, 20)$,
$t_7^l = (19, true, ret\langle -1\rangle, 20)\ \}$
$\nu(t_1^l) = op\langle lookUp\rangle\_begin\langle\rangle$
$\nu(t_6^l) = op\langle lookUp\rangle\_end\langle\hat{x}\rangle$
$\nu(t_7^l) = op\langle lookUp\rangle\_end\langle -1\rangle$
For the others $\nu(t^l) = \tau$

In order to *execute a program* it is necessary to provide user threads $\Psi = \psi_u^1, \dots, \psi_u^n$, which will be executed. User thread ψ_i is defined as a sequence of operation subprograms $p_0, \dots, p_{n_i}$ with values of input parameters. Thread starts execution in an initial state of subprogram p_0. After the end of each subprogram it moves from the end state to the initial state of the next subprogram. Thread finishes execution after finishing the last subprogram p_{n_i}.

For the given user threads Ψ we define *execution state* as $g = (s, l^1, \dots, l^n) \in G$, where s is a shared state, l^i is a local state of the thread ψ_i. $g_0 = (s_0, l_0^1, l_0^2, \dots, l_0^n)$ is an initial state. A set of all execution states we denote as G.

We will use the following definitions:

1) $enabled(t, s) \equiv t.guard(s)$.
2) $pre(t)$ is a start state of t, $post(t)$ is an end state.
3) $local(\alpha, g)$ returns a local state of the thread p in g.
4) $shared(g)$ returns a shared state s.
5) $t(\alpha)$ means that t is executed in thread $\alpha \in \Psi$.
6) $active(t(\alpha), g) \equiv pre(t) = local(\alpha, g)$.
7) $enabled(t(\alpha), g) \equiv active(t(\alpha), g) \wedge enabled(t, shared(g))$.

Define a transition relation $\longrightarrow$. There is a transition $g \xrightarrow{t(\alpha)} g'$ from $g = (s, l^1, \dots, l^\alpha, \dots, l^n)$ if $enabled(t(\alpha), g) = true$ and $g' = (\hat{s}, l^1, \dots, \hat{l}^\alpha, \dots, l^n)$, where $\hat{s} = t.command(s)$ and $\hat{l}^\alpha = post(t)$.

Execution trace of a program is a sequence $t_1(\alpha_1), \dots, t_m(\alpha_m)$ such that $g_0 \xrightarrow{t_1(\alpha_1)} g_1 \xrightarrow{t_2(\alpha_2)} \cdots \xrightarrow{t_m(\alpha_m)} g_m$. *Thread trace* is a projection of execution trace on a thread. *Operation trace* in a thread is a projection of thread trace on an operation. Execution history for a trace $\sigma = t_1(\alpha_1), \dots, t_m(\alpha_m)$, denoted as $H(\sigma)$, is a sequence of labels $\nu(t_i(\alpha_i))$ with all τ labels removed. History H is reachable if there exists a trace σ such that $H = H(\sigma)$.

Let a thread α executes $insert\langle 0\rangle$, β executes $delete\langle\rangle$, γ executes $lookUp\langle\rangle$. Consider the examples of traces.

$$\sigma_1 = t_1^i(\alpha), t_2^i(\alpha), t_1^d(\beta), t_3^i(\alpha), t_5^i(\alpha), t_6^i(\alpha), t_7^i(\alpha), \\ t_1^l(\gamma), t_2^l(\gamma), t_3^l(\gamma), t_5^l(\gamma), t_2^d(\beta), t_3^d(\beta), t_6^l(\gamma) \quad (1)$$

$H(\sigma_1) = h_1$.

$$\sigma_2 = t_1^i(\alpha), t_1^d(\beta), t_2^i(\alpha), t_3^i(\alpha), t_5^i(\alpha), t_2^d(\beta), t_6^i(\alpha), t_3^d(\beta), t_7^i(\alpha) \quad (2)$$

$$\sigma_3 = t_1^l(\gamma), t_1^i(\alpha), t_2^l(\gamma), t_2^i(\alpha), t_3^l(\gamma), t_5^i(\alpha), t_5^l(\gamma), t_6^i(\alpha), t_6^l(\gamma), t_7^i(\alpha) \quad (3)$$

IV. An Upper Bound of Context Switches

A. The Notion of Independence

We will use classical definition of independence [4], [5], [12] (Definition 4) and extend it for an arbitrary set of user threads (Definition 5).

Definition 4: $D(\Psi)$ is a symmetric dependence relation for an execution Ψ, iff for all $(t_1(\alpha), t_2(\beta)) \notin D(\Psi)$ (independent) implies that the two following conditions hold for all reachable states g:

1) From $enabled(t_1(\alpha), g)$ and $g \xrightarrow{t_1(\alpha)} g'$ follows that $enabled(t_2(\beta), g') = enabled(t_2(\beta), g)$,
2) If $enabled(t_1(\alpha), g)$ and $enabled(t_2(\beta), g)$ then there exists unique state $\hat{g}$ such that $g \xrightarrow{t_1(\alpha), t_2(\beta)} \hat{g}$ and $g \xrightarrow{t_2(\beta), t_1(\alpha)} \hat{g}$.

Definition 5: D is a symmetric dependence relation for a program iff $(t_1, t_2) \notin D$ (independent) implies that $\forall \Psi$, $\forall \alpha, \beta$ $\exists D(\Psi) : (t_1(\alpha), t_2(\beta)) \notin D(\Psi)$.

Note 1: Independent transitions can be interchanged in a trace, but the history of the trace and the final state will stay unchanged. [5]

Consider an example of dependence relation for Cell program. We can see that some transitions do not access shared data. Such transitions are obviously independent. Consider transitions accessing shared data $t_2^i, t_5^i, t_6^i, t_2^d, t_2^l, t_5^l$. Among them we can distinguish transitions which only read or write variables. $r_b(w_b)$, $r_x(w_b)$ denote transitions performing read (write) of variables b and x correspondingly. Then r_b: t_2^i, t_2^l; w_b: t_6^i, t_2^d; r_x: t_5^l; w_x: t_5^i. Transition pairs performing read/read of arbitrary variables, write/write or read/write of different variables are independent with each other. The others are considered as dependent. Hence dependence relation D includes $\{(t, t') \mid t \in \{t_6^i, t_2^d\}, t' \in \{t_6^i, t_2^d, t_2^i, t_2^l\}\} \bigcup \{(t, t') \mid t \in \{t_5^i\}, t' \in \{t_5^i, t_5^l\}\}$ and symmetric pairs.

B. The Notion of Dependence Cycle

Let $\sigma = t_1(\alpha_1), \ldots, t_m(\alpha_m)$ be a trace of a program. Define successor relation.

Definition 6: Successor relation (without transitive closure)

1) $t_i(\alpha_i) < t_j(\alpha_j)$, if $t_i(\alpha_i)$ precedes $t_j(\alpha_j)$ in the trace and one of the following conditions holds
   a) $t_i, t_j \neq \{op\_begin, op\_end\}$, $(t_i, t_j) \in D$ and $\alpha_i \neq \alpha_j$,
   b) $t_i = op\_end$, $t_j = op\_begin$ and t_i, t_j do not belong to the same operation.
2) $t_i(\alpha_i) = t_j(\alpha_j)$, if t_i, t_j belong to the same operation.

Statement 1: Let a trace σ contains neighbouring pair $t_i(\alpha_i)$, $t_j(\alpha_j)$ and $t_i(\alpha_i) \not\leq t_j(\alpha_j)$. Let σ' be the trace obtained from σ by interchanging $t_i(\alpha_i)$, $t_j(\alpha_j)$ in the reverse order. Then the history of σ' is equivalent to the history of σ ($H(\sigma') = H(\sigma)$).
Follows from independence of $t_i(\alpha_i)$, $t_j(\alpha_j)$, because $t_i(\alpha_i) \not\leq t_j(\alpha_j)$.

Sequence by successor relation is a sequence of transitions related by definition 6: $t_{i_1}(\alpha_{i_1}) \leq t_{i_2}(\alpha_{i_2}) \leq \ldots \leq t_{i_p}(\alpha_{i_p})$, where $t_{i_j}(\alpha_{i_j})$ is a trace element.

Cycle is a sequence by successor relation $t_{i_1}(\alpha_{i_1}) < t_{i_2}(\alpha_{i_2}) \leq \ldots < t_{i_{p-1}}(\alpha_{i_{p-1}}) = t_{i_p}(\alpha_{i_p})$, in which $t_{i_1}(\alpha_{i_1}) = t_{i_p}(\alpha_{i_p})$ and $t_{i_1}(\alpha_{i_1})$ precedes $t_{i_{p-1}}(\alpha_{i_{p-1}})$. Element $t_{i_1}(\alpha_{i_1})$ is called the *start* of a cycle and $t_{i_{p-1}}(\alpha_{i_{p-1}})$ is the *end*.

Consider examples of cycles. Trace $\sigma_2(2)$ has a cycle

$$t_2^i(\alpha) < t_2^d(\beta) < t_6^i(\alpha) = t_2^i(\alpha) \quad (4)$$

Trace $\sigma_3(3)$ has two cycles

$$t_2^l(\gamma) < t_6^i(\alpha) = t_5^i(\alpha) < t_5^l(\gamma) = t_2^l(\gamma) \quad (5)$$

$$t_5^i(\alpha) < t_5^l(\gamma) = t_2^l(\gamma) < t_6^i(\alpha) = t_5^i(\alpha) \quad (6)$$

Schematically these cycles are shown in Fig. 3(a,b). Arrows mean that transitions are related by $<$. Trace $t_1^d(\beta), t_1^l(\gamma), t_2^l(\gamma), t_2^d(\beta), t_3^l(\gamma), t_3^d(\beta), t_5^l(\gamma), t_6^l(\gamma)$ has no cycles (Fig. 3(c)).

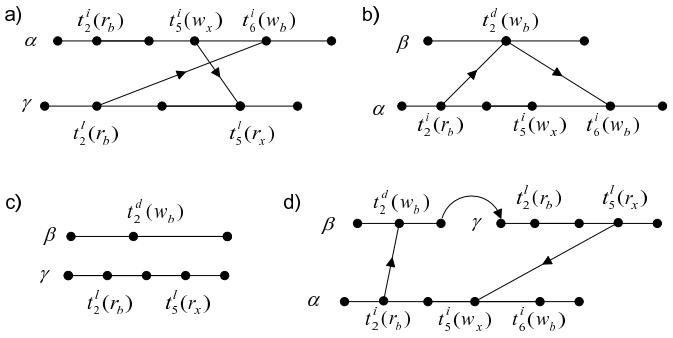

Fig. 3. Examples of Cycles

Two cycles with ends t_{k_1}, t_{k_2}, belonging to the same operation, where t_{k_1} precedes t_{k_2} are *equivalent* if there is no cycle with a start t' preceding t_{k_2} and not preceding t_{k_1} ($t' \in [t_{k_1}, t_{k_2})$).

The equivalence of cycles defines an equivalence relation and corresponding division of cycles into equivalence classes.

Holding cycle is an equivalence class of cycles.

C. The Notion of Context Switch

Let $\sigma = t_1(\alpha_1), \ldots, t_k(\alpha_k), t_{k+1}(\alpha_{k+1}), \ldots, t_n(\alpha_n)$ be a trace.

We say that there is a *context switch* between $t_k(\alpha_k)$, $t_{k+1}(\alpha_{k+1})$ if $\alpha_k \neq \alpha_{k+1}$. Context switch is *nonpreemptive* if $t_k(\alpha_k)$ is the last transition in the operation trace. Otherwise context switch is called *preemptive* which means that a scheduler suspends the executions of the running thread at an arbitrary point. Note that context switches between different operations in the same thread are defined as nonpreemptive.

The number of preemptive context switches in a trace σ we denote as $csw(\sigma)$. For instance, $csw(\sigma_2) = 5$, $csw(\sigma_3) = 8$.

Theorem 1: If the number of holding cycles in a trace σ is k then there exists a trace σ' such that $H(\sigma) = H(\sigma')$ and $csw(\sigma') \leq k$.

Proof of the theorem can be found in the Appendix on page 6.

Corollary 1: If there is no reachable traces with cycles then program is self-linearizable.

If there is no cycles in a trace σ then it is equivalent to a trace σ' without preemptive switches. Hence σ' is a sequential trace and the program is self-linearizable.

V. CycleCount Algorithm

Traditional model checking algorithms suffer from state explosion problem. It is not difficult to show that the number of executions grows exponentially both in the length of thread traces and in the number of threads. More precisely, $\frac{(nm)!}{(m!)^n}$, where n is the number of threads, m is the maximal length of thread trace. Promising approach for solving this problem is bounding the number of context switches. This idea appeared in [13] and in [10] it was evaluated in practice. In the approach the number of executions with k switches is estimated as $(n^2m)^k n!$, i.e. polynomial in the trace length. Empirical results show that 90% of state coverage can be achieved with eight context switches and the majority of errors are detected even with two switches. The novel result of our paper is that we can guarantee linearizability if we apply context-bounding search algorithm [10] with a bound calculated by CycleCount algorithm (Fig. 4). The linearizability of each trace found during the search we check using one of existing methods. For example, if we have a recognizing automaton as in [1] we can check it in $O(m)$.

CycleCount algorithm takes user threads Ψ, i.e. $\psi_1, \ldots, \psi_n$, where $\psi_i = p_0, \ldots, p_{n_i}$ and for each subprogram p it takes a complete set of operation traces $traces(p)$. The completeness means, that any operation trace that can occur in executions of $Sys(\Psi)$ should be in $traces(p)$. Execution traces of threads in the algorithm are overapproximated as $traces(\psi_i) = traces(p_0) \times \cdots \times traces(p_{n_i})$. Execution traces of program with Ψ are overapproximated as all possible interleavings of thread traces $traces(\psi_1), \ldots, traces(\psi_n)$.

Complexity of the algorithm is $O(n(mk)^3 l^2)$, where n is the max number of threads, m is the max number operations in a thread, k is the max number of traces for a single operation, l is the max operation trace length.

For Cell example the complete set of operation traces is as follows:

insert:
$\sigma_1^i = t_1^i, t_2^i, t_3^i, t_5^i, t_6^i, t_7^i$
$\sigma_2^i = t_1^i, t_2^i, t_4^i, t_8^i$
delete:
$\sigma_1^d = t_1^d, t_2^d, t_3^d$
lookUp:
$\sigma_1^l = t_1^l, t_2^l, t_3^l, t_5^l, t_6^l$
$\sigma_2^l = t_1^l, t_2^l, t_4^l, t_7^l$

$K := 0$
For each thread ψ_i
  For each operation $p_j \in \psi_i$
    For each operation trace $\sigma_k \in traces(p_j)$
      For each $\sigma' \in op' : \psi(op') \neq \psi(op_j)$
        Find $D_k(\sigma') = \{t \in \sigma \mid \exists t' \in \sigma' : (t', t) \in D(\Psi)\}$
        If $D_k(\sigma') \neq \emptyset$, then
          Let $E(\sigma') = \{\sigma'\}$
            $\cup \{traces(\hat{op}) \mid \psi(\hat{op}) \neq \psi(op'), \psi(op_j)\}$.
        Else let $E(\sigma') = \emptyset$.
      For each $\sigma' \in op' : \psi(op') \neq \psi(op_j)$
        Mark transitions $t \in \sigma_k$: $L(t) = \{B, E, \tau\}$.
          Let $L_E = \bigcup_{\hat{\sigma} \in E(\sigma')} D_k(\hat{\sigma})$.
          If $L_E \neq \emptyset$
            Let $L_B = \{t' \in D_k(\sigma') \mid t'$ precedes $t_r\}$.
            where t_r is the most right end in L_E.
          Else let $L_B = \emptyset$.
          Transitions in L_B we mark as B (starts),
            L_E – E(ends), the others – τ.
        $K(\sigma_k)$ is the number of continuous intervals
          of $\{E, \tau\}$, i.e. ends not separated by starts.
        $K := K + K(\sigma_k)$

Fig. 4. CycleCount Algorithm

Suppose that a thread α executes operation p_{insert}, β executes p_{delete}, γ executes p_{lookUp}. Then $CycleCount(\beta, \gamma) = 0$, because there is no cycles. $CycleCount(\alpha, \gamma) = 2$, because we have cycles with ends t_6^i, t_5^l. $CycleCount(\alpha, \beta, \gamma) = 3$, because we have cycles with ends t_5^i, t_6^i, t_5^l. Cycle with end t_5^l (5) is shown in Fig. 3(a). Cycles (4,6) with end t_6^i are shown in Fig. 3(a), 3(b). Cycle

$$t_2^i(\alpha) < t_2^d(\beta) = t_3^d(\beta) < t_1^l(\gamma) = t_5^l(\gamma) < t_5^i(\alpha) = t_2^i(\alpha) \tag{7}$$

is shown in Fig. 3(d). Here $t_3^d(\beta) < t_1^l(\gamma)$, because $\nu(t_3^d(\beta)) = op\_end$ and $t_1^l(\gamma) = op\_begin$.

VI. Discussion

While formulating CycleCount algorithm in section V we assumed that we have complete set of operation traces. One way to get the traces in practice is to run a test suite. Moreover, our experience in analysing multithreaded programs suggests that in most cases complete set of traces can be generated even by sequential executions. In the worst case it requires minimal number of switches. For ensuring completeness the existing tools measuring path coverage [2] can be used.

Note, that the number of context switches in execution is bounded by maximal number. There are at most

$$\sum_{\psi_i \in \Psi} \sum_{p_1^i, \ldots, p_{n_i}^i} Max_{\sigma \in traces(p_j^i)}(|\sigma| - 1),$$

context switches, i.e. not more than the sum of context switches in the longest operation traces for each thread. Besides, context switches can occur only between transitions inside operation trace, i.e. $|\sigma| - 1$. Table 5 shows maximal

Threads	CycleCount	POR	Maximum
β, γ	0	1	6
α, γ	2	3	9
α, β, γ	3	3	11

Fig. 5. Context Switch Bounds

bounds for Cell example. We can see, that in comparison with maximal bounds CycleCount algorithm has an advantage.

Results of the algorithm CycleCount can be improved. We notice three reasons why the algorithm gives crude estimates. First, theorem 1 does not take into account the fact that several cycles can be broken by one switch. For instance, two cycles in Fig.3(a) can be broken by one context switch instead of two.

Second, CycleCount algorithm does not detect contradictory cycles, i.e. cycles which can not appear together in one trace, but can appear in different ones. For example, while running $CycleCount(\alpha, \beta, \gamma)$ cycle (7) does not appear together with (4), because the same transition t_5^i is both start of the cycle (4), and end of (7) related with the same transition t_5^l.

Third, algorithm does not consider nonappearance of transitions between the other ones. Such situation occur in acquiring locks. Transitions from mutual exclusion intervals cannot interfere with each other, i.e. at first the instructions from the one interval will be executed then the rest ones.

VII. Related Work & Conclusions

Novelty of the results of the paper is in estimation of upper bound of context switches which provided to context-bounding algorithm will guarantee linearizability of the program. Along with context bounding algorithm there are heuristic search and partial order reduction (POR) which reduce the number of executions. In comparison with heuristic search our method can guarantee linearizability. The same as in our estimations the key notion in POR is independence. In this sense our derivation of upper bound is a reduction of non linearizable traces to non linearizable traces with at most k switches.

Table 5 shows the number of context switches in traces found with POR search in the lucky case of a search order. The benefit of CycleCount can be explained by the fact that it knows operation traces in advance, hence reductions can be calculated before search of actual program executions.

Our future work is to improve CycleCount algorithm and implement it with one of model checking engines.

References

[1] Rajeev Alur, Ken McMillan, and Doron Peled. Model-checking of correctness conditions for concurrent objects. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 219–228, Washington, DC, USA, 1996. IEEE Computer Society.

[2] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[3] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, New York, NY, USA, 2004. ACM.

[4] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, 2005.

[5] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. Foreword By-Pierre Wolper.

[6] John Hatcliff, Robby, and Matthew B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *In Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation*, pages 175–190. Springer, 2004.

[7] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[8] A. Khoroshilov. *Specification and testing components with asynchronous interfaces. PhD thesis*. ISP RAS, Moscow, 2006.

[9] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[10] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6):446–455, 2007.

[11] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[12] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

[13] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *In TACAS*, pages 93–107. Springer, 2005.

[14] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 129–136, New York, NY, USA, 2006. ACM.

[15] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32(2):93–110, 2006.

Appendix

Theorem 2: If the number of holding cycles in a trace σ is k then there exists a trace σ' such that $H(\sigma) = H(\sigma')$ and $csw(\sigma') \leq k$.

Proof.

Let $\beta_1, \ldots, \beta_n$ are operations in a trace σ.

$\beta_1 = t_1^1 \ldots t_{q_1}^1$

$\ldots$

$\beta_n = t_1^n \ldots t_{q_n}^n$

A *division* Ξ is

$\beta_1 = \beta_1^1 \ldots \beta_1^{p_1}$

$\ldots$

$\beta_n = \beta_n^1 \ldots \beta_n^{p_n}$

where β_i^l is a union of neighbouring transitions in the same operation trace.

Given Ξ we can define partial order HB ($\beta_i^l \prec \beta_j^m$) on it.

Definition 7: Partial order HB on Ξ is *consistent* if

1) $\forall i \; \forall \; l < m$ holds $\beta_i^l \prec \beta_i^m$
2) $\forall \; t_1 < t_2$ in σ, t_1 in β_i^l, t_2 in β_j^m holds $\beta_i^l \prec \beta_j^m$.

Lemma 1: If partial order HB is consistent on the division Ξ then $\exists \sigma'$ such that $H(\sigma) = H(\sigma')$ and $csw(\sigma') \leq (p_1 - 1) + \cdots + (p_n - 1)$.

As σ' it is sufficient to take any trace composed from β_i^l, with respect to partial order HB. The number of context switches for β_i is $p_i - 1$. σ' can be derived from σ by interchanging transitions $t_i(\alpha_i) \not\leq t_j(\alpha_j)$, because it preserves equivalence of histories (Statement 1). From consistency of HB it follows

that transitions for which $t_i(\alpha_i) \leq t_j(\alpha_j)$ need not to be interchanged.

Now for proving theorem we need to choose division Ξ such that $\sum_{i=1}^{n}(p_i - 1) \leq k$ and to choose consistent partial order HB on it.

Choosing division. For each β_i. If there is no cycles then $\beta_i = \beta_i^1$. Otherwise, suppose there is s holding cycles. $t_{k_1}, t_{k_2}, \ldots, t_{k_s}$ are the most left ends of cycles in each holding cycle (i.e. t_{k_j} is the most left end of the cycles in a holding cycle j).

Suppose that $t_{k_1}, t_{k_2}, \ldots, t_{k_s}$ appear in β_i in the same order. Chose division β_i:
$\beta_i^1 = [op\_begin, t_{k_1})$
$\beta_i^2 = [t_{k_1}, t_{k_2})$
$\beta_i^3 = [t_{k_2}, t_{k_3})$
...
$\beta_i^{s+1} = [t_{k_s}, op\_end]$

We define relation $<$ on Ξ as follows. $\beta_i^l < \beta_j^m$ if one of the following conditions holds

1) $i = j$, $l < m$
2) $\exists\ t_1$ in β_i^l, t_2 in β_j^m: $t_1 < t_2$.

We define HB ($\prec$) on Ξ as a transitive closure of $<$. Lets show that $\prec$ is consistent partial order. Consistency immediately follows from definition of $<$. Lets show that $\prec$ is a partial order.

Proof by contradiction. Let exists β_i^l, β_j^m such that $\beta_i^l \prec \beta_j^m$ and $\beta_j^m \prec \beta_i^l$.

Hence there exists chains
$\beta_i^l < \beta_{i_1}^{l_1} < \ldots < \beta_j^m$
$\beta_j^m < \beta_{j_1}^{m_1} < \ldots < \beta_i^l$
or
$\beta_i^l < \beta_{i_1}^{l_1} < \ldots < \beta_j^m < \beta_{j_1}^{m_1} < \ldots < \beta_i^l$

Hence there exists a sequence $t_1(\beta_i) \leq t_2(\beta_{i_1}) \leq \ldots \leq t_w(\beta_j^m) \leq t_{w+1}(\beta_{j_1}) \leq \ldots \leq t'(\beta_i)$. Two signes are strictly less, because there should appear $\hat{\beta} \neq \beta_i$.

Either t_1 precedes t' then let $t_a = t_1$, $t_b = t'$ or t_1 does not precede t'. Then in one of the equivalences $\ldots < t_b(\hat{\beta}) = \ldots = t_a(\hat{\beta}) < \ldots$, t_a precedes t_b. Because otherwise, if in all equivalencies t_b precedes t_a, then t_1 precedes t'.

t_a precedes t_b then exists a cycle: $t_a(\hat{\beta}) < \ldots \leq t'(\beta_i) = t_1(\beta_i) \leq \ldots < t_b(\hat{\beta})$. This cycle can not be equals to any cycle with the end which is not later than t_a. Hence one of the most left ends of cycles in holding cycle lies after t_a and not after t_b. That contradicts to the chosen division. •