Correctness Proofs and Complexity Analysis of EventTrack Algorithm

Pallavi Maiya
Indian Institute of Science, India
pallavi.maiya@csa.iisc.ernet.in

Aditya Kanade
Indian Institute of Science, India
kanade@csa.iisc.ernet.in

This is a supplementary material to the paper [2] which presents an algorithm called EventTrack to compute happens-before (HB) relation for event-driven programs such as Android applications. This material presents the correctness proofs and complexity analysis of the EventTrack algorithm.

Assumptions. The correctness proofs as well as complexity analysis presented here make an assumption related to the `NO-PRE` event-ordering rule presented in Table 1 of [2]. The `NO-PRE` rule states that for a pair of events e and e' posted to the same event queue, if an operation in the handler of e is found to happen before an operation in the handler of e' then $\texttt{end}(e) \prec_{hb} \texttt{deq}(e')$. Let function $hbSrc(z)$ return the set of operations in the trace that happen before an operation z due to some HB rule except purely due to transitive closure. Typically, the set $hbSrc(z)$ can be computed for an operation z in a trace τ just by inspecting the prefix of τ upto the index of operation z in the trace. This makes vector clock based HB relation computation complexity linear in the number of operations in the trace. However, the presence of `NO-PRE` event-ordering rule may cause EventTrack to HB order `deq` of an event e' w.r.t. `end` of an event e, only after analyzing an operation y such that $\texttt{target}(y)$ (defined in Table 3 in [2]) is an operation in the handler of e'. This in turn may cause the operations in between $\texttt{deq}(e')$ and y to be reprocessed to reflect the new HB information obtained via $\texttt{end}(e)$ in their respective vector clock timestamps. In general, computation of vector clock timestamps for operations in the program trace of a multi-threaded event-driven program like an Android application, may not complete in fixed number of passes over the trace.

However we assume that the application of `NO-PRE` rule does not result in reprocessing of already processed operations in the input execution trace. We make this assumption when giving the correctness proofs and algorithm complexity, as we did not observe such trace reprocessing due to `NO-PRE` in our experimentation. However, our implementation of EventTrack handles such cases as well. We refer the readers to Section 3 of `README.pdf` at the root of EventTrack tool repository[1] for more information about our implementation strategy to handle trace reprocessing due to `NO-PRE`.

1 CORRECTNESS PROOFS

We give correctness proofs for EventTrack algorithm and prove that it indeed computes the happens-before relation for a given execution trace τ of an event-driven program P. These proofs have a framework similar to the correctness proofs of FastTrack algorithm [1] which performs vector clock based HB computation for multi-threaded programs.

[1]http://bitbucket.org/iiscseal/eventtrack

An operation z belonging to a trace τ may be indicated as $z \in \tau$. Henceforth, for any operation z in a trace τ we refer to the analysis state prior to processing z and after processing z by the application of transfer functions, as $\sigma^z = (T^z, G^z)$ and $\ddot{\sigma}^z = (\ddot{T}^z, \ddot{G}^z)$ respectively. For an operation z we use a notation $\mathbb{T}^z$ and $\mathbb{G}^z$ to mean the following. If z is of type `tinit`, `texit`, `join`, `trigger`, `deq` or `end` then $\mathbb{T}^z = \ddot{T}^z$, $\mathbb{G}^z = \ddot{G}^z$, else $\mathbb{T}^z = T^z$ and $\mathbb{G}^z = G^z$. For an operation z in the trace τ, $(\mathbb{T}^z, \mathbb{G}^z)$ is chosen as given above because of the following intuitive reasons. If z is of type `join`, `trigger` or `deq`, then the vector clock and incoming edges of $task(z)$ is updated through `VC-JOIN` and edge propagation respectively when EventTrack analyzes z, which intuitively means that some HB information is passed to z only after z is evaluated. In case of other types of operations, all the required HB information is received by the executing task at state (T^z, G^z) itself. If z is of type `fork`, `notify`, `post` or `enable` which pass on HB information to other operations (see Table 4 and 5 in [2]) but do not receive any additional HB information through `VC-JOIN` or edge propagation after processing z, we define $(\mathbb{T}^z, \mathbb{G}^z)$ as (T^z, G^z).

We define a notion of well-formedness of a state as follows.

Definition 1.1. An analysis state $\sigma = (T, G)$, with $G = (H, \xi)$, reached on processing a prefix α of a program trace τ is said to be well-formed if,

(1) for an event $e \in Ev$ if $\texttt{deq}(e) \in \alpha$ and either $\texttt{end}(e) \notin \alpha$ or $\texttt{end}(e)$ is the last operation in α, then for every task $b \in H$ such that $b \neq e$, $T(b)[C(e)] < T(e)[C(e)]$.
(2) for a thread $t \in Th$ if $\texttt{tinit}(t) \in \alpha$ and either $\texttt{texit}(t) \notin \alpha$ or $\texttt{texit}(e)$ is the last operation in α, then for every task $b \in H$ such that $b \neq t$, $T(b)[C(t)] < T(t)[C(t)]$.
(3) for an event $e \in Ev$ if $\texttt{post}(e) \notin \alpha$ then for every task $b \in H$, $(e \xrightarrow{l} b) \notin G(b)$ for any value of edge label l.
(4) for an event $e \in Ev$ if $\texttt{deq}(e) \notin \alpha$ then for every task $b \in H$, $(e \xrightarrow{1} b) \notin G(b)$.
(5) for an event $e \in Ev$ if $\texttt{end}(e) \in \alpha$ then $T(e) = \ddot{T}^{\texttt{end}(e)}(e)$.
(6) for a thread $t \in Th$ if $\texttt{texit}(e) \in \alpha$ then $T(t) = \ddot{T}^{\texttt{texit}(t)}(t)$.

Lemma 1.2. *An analysis state $\sigma = (T, G)$ reached on processing a prefix α of a program trace τ is well-formed.*

Proof. Let $\sigma^{\epsilon} = (T^{\epsilon}, G^{\epsilon})$ be the initial state of EventTrack. Then for every task $b \in H$, $T^{\epsilon}(b) = \perp_{\mathbb{V}}$ and $G^{\epsilon}(b) = \emptyset$. Then, trivially σ^{ϵ} is well-formed since α is an empty sequence. With this we can do a case by case analysis of each transfer function of EventTrack and using induction prove that any state σ reached from σ^{ϵ} on processing a prefix α of τ is well-formed. □

Lemma 1.3. *In a trace τ of a program P for an operation $z \in \tau$ and b being $task(z)$,*

(1) *if an operation $x \in \tau$ is such that $\mathbb{T}^x(task(x))[C(task(x))] \leq \mathbb{T}^z(b)[C(task(x))]$ then $x \prec_{hb} z$.*
(2) *for an event $e \in Ev$ if $(e \xrightarrow{0} b) \in \mathbb{G}^z(b)$ then* `post(e)` $\prec_{hb}$ *z.*
(3) *for an event $e \in Ev$ if $(e \xrightarrow{1} b) \in \mathbb{G}^z(b)$ then* `deq(e)` $\prec_{hb}$ *z.*

Proof. We prove this by induction on the number of steps of application of transfer functions required for EventTrack to derive antecedent of property (1), (2) or (3) of the lemma.

We first prove property (1) of the lemma. Let $d = task(x)$ and $b = task(z)$. If $d = b$ or $C(d) = C(b)$ then by program order and total order property of chains, $x \prec_{hb} z$. Assume $d \neq b$ and $C(d) \neq C(b)$. From Lemma 1.2 by which analysis state $(\mathbb{T}^x, \mathbb{G}^x)$ is well-formed, and the antecedent of property (1), we have $\mathbb{T}^x(b)[C(d)] < \mathbb{T}^x(d)[C(d)] \leq \mathbb{T}^z(b)[C(d)]$. Then, there exists an operation y executed between x and z in τ which passes $\mathbb{T}^x(d)[C(d)]$ to vector clock of b resulting in the following.

$$\mathbb{T}^x(b)[C(d)] \leq T^y(b)[C(d)] < \mathbb{T}^x(d)[C(d)] \leq \ddot{T}^y(b)[C(d)] \leq \mathbb{T}^z(b)[C(d)] \quad \texttt{[I]}$$

Base case. In the base case, either $x = y$ (*e.g.*, $x =$ `fork(d,b)`) or $y = z$ (*e.g.*, $z =$ `join(b,d)`) and the happens-before from x to z can be established by single application of a non event-ordering HB rule.

Induction Hypothesis. For a pair of operations o_1 and o_2 in the trace τ, if the relation $\mathbb{T}^{o_1}(task(o_1))[C(task(o_1))] \leq \mathbb{T}^{o_2}(task(o_2))[C(task(o_1))]$ is derived within η steps of application of transfer functions of EventTrack then $o_1 \prec_{hb} o_2$. For an event $e' \in Ev$, an operation o and a task $a = task(o)$, if $(e' \xrightarrow{0} a)$ can be established to be a member of $\mathbb{G}^o(a)$ within η steps of application of transfer functions of EventTrack then `post(e')` $\prec_{hb}$ *o*. For an event $e' \in Ev$, an operation o and a task $a = task(o)$, if $(e' \xrightarrow{1} a)$ can be established to be a member of $\mathbb{G}^o(a)$ within η steps of application of transfer functions of EventTrack then `deq(e')` $\prec_{hb}$ *o*.

Induction step. Let $\mathbb{T}^x(d)[C(d)] \leq \mathbb{T}^z(b)[C(d)]$ be established in $\eta + 1$ applications of transfer functions. From `[I]`, the transfer function of y modifies the vector clock of b. Then from the transfer functions given in Table 4 and 5 in [2], $b = target(y)$. Let $a = source(y)$. Then y is one of the operations among `fork(a,b)`, `join(b,a)`, `notify(o)` with b being the target, `trigger(b)`, `post(b)`, or `deq(b)`. We prove the induction step for the case when $y =$ `deq(b)` which involves reasoning about event-ordering rules. The other cases can be proved similarly using an applicable non event-ordering rule. Further, we prove the case when event b is posted with a delay. The case when b is posted to the front of the queue can be proved using similar arguments. Let A be the set of events identified by the transfer function of `deq` in Table 4 in [2], and thus whose vector clock values and incoming edges will be used to update the vector clock and incoming edges respectively of the task b. Due to `[I]`, set A must contain atleast one event whose vector clock was used to update $T^y(b)[C(d)]$. Let $e \in A$ be the event with whose vector clock `VC-JOIN` was performed to result in $\mathbb{T}^x(d)[C(d)] \leq \ddot{T}^{\texttt{deq}(b)}(b)[C(d)]$. Projecting `VC-JOIN` between $T^y(b)$ and $T^y(e)$ to component $C(d)$ we have,

$$\ddot{T}^y(b)[C(d)] = max(T^y(b)[C(d)], T^y(e)[C(d)]) \quad \texttt{[II]}$$

$$\ddot{T}^y(b)[C(d)] = T^y(e)[C(d)] \quad \text{from } \texttt{I} \text{ \& } \texttt{II} \quad \texttt{[III]}$$

Since event $e \in A$, $(e \xrightarrow{l} b) \in G^y(b)$, for $l \in \{0, 1\}$. Assume $l = 1$. Note that case when $l = 0$ can be proved similarly. Then, the edge $(e \xrightarrow{1} b)$ has been added to event graph in $k \leq \eta$ steps. Therefore by induction hypothesis, `deq(e)` $\prec_{hb}$ `deq(b)`. Then, by `NO-PRE` rule in Table 1 in [2], `end(e)` $\prec_{hb}$ `deq(b)`. This also indicates that EventTrack has processed `end(e)` prior to y. Since analysis state σ^y is well-formed and from Definition 1.1 we have $T^y(e) = \mathbb{T}^{\texttt{end}(e)}$. Then, from `[I]` and `[III]`, $\mathbb{T}^x(d)[C(d)] \leq \ddot{T}^{\texttt{end}(e)}(e)[C(d)]$ and clearly this has been established within η steps. Therefore,

$$x \prec_{hb} \texttt{end}(e) \quad \text{by induction hypothesis} \quad \texttt{[IV]}$$

Since $y =$ `deq(b)`, $task(y) = b$. Then, y and z are HB ordered by program order. Combining `IV`, `NO-PRE` HB rule and program order we have thus proved that $x \prec_{hb} z$.

We can prove property (2) and (3) of the lemma by referring to properties (3) and (4) of the well-formedness definition and using analogous arguments as above. □

The following lemma (Lemma 1.4) states the properties of vector clocks and event graph observed when a pair of operations get HB ordered. The operator $\uplus$ defined for event graph (see Table 2 in [2]) prunes or does not propagate certain edges if the information conveyed by those edges are already subsumed by vector clocks. Hence, the lemma also specifies the condition when an otherwise expected edge is not present in the event graph.

Lemma 1.4. *Consider a trace τ of a program P containing operations $x, z \in \tau$ with $b = task(z)$ such that $x \prec_{hb} z$. Then,*
(1) *$\mathbb{T}^x(task(x)) \sqsubseteq \mathbb{T}^z(task(z))$.*
(2) *Additionally if $x =$ `post(e)` for an event $e \in Ev$ then $e = b$, or $(e \xrightarrow{l} b) \in \mathbb{G}^z(b)$ for $l \in \{0, 1\}$, or $\mathbb{T}^{\texttt{end}(e)}(e) \sqsubseteq \mathbb{T}^z(b)$.*
(3) *Additionally if $x =$ `deq(e)` for an event $e \in Ev$ then $e = b$, or $(e \xrightarrow{1} b) \in \mathbb{G}^z(b)$, or $\mathbb{T}^{\texttt{end}(e)}(e) \sqsubseteq \mathbb{T}^z(b)$.*

Proof. We prove this by induction on the length of derivation of happens before ordering from x to z in τ. If there are multiple ways to derive $x \prec_{hb} z$ then induction is on the derivation which applies minimum number of happens-before rules. We prove this for the case where $x =$ `post(e)` such that e is *not* posted to the front of an event queue (*i.e.*, $!foq(e)$). Proof for the rest of the types of x can be given similarly. Let $b = task(z)$.

Base case ($k = 1$). Let $d = task(x)$. Operation $x =$ `post(e)` is established to happen before z in one step only if $z =$ `deq(e)` or z is the immediate next operation in x's task d. In the former case $task(z) = b = e$. In the latter case where $d = b$, on processing `post(e)` the transfer function of `post` adds an edge $(e \xrightarrow{0} d)$. Also, in both the cases the application of transfer function of `post` results in $\mathbb{T}^x(d) \sqsubseteq \mathbb{T}^z(b)$. Hence proved.

Induction Hypothesis. If the happens-before ordering from an operation o_1 to o_2 can be established within η steps of application of HB rules, where $h_1 = task(o_1)$ and $h_2 = task(o_2)$, then $\mathbb{T}^{o_1}(h_1)$

$\sqsubseteq \mathbb{T}^{o_2}(h_2)$. In addition if $o_1 = \texttt{post}(e)$ then $e = h_2$, or $(e \xrightarrow{0/1} h_2) \in \mathbb{G}^{o_2}(h_2)$, or $\mathbb{T}^{\texttt{end}(e)}(e) \sqsubseteq \mathbb{T}^{o_2}(h_2)$. Instead if $o_1 = \texttt{deq}(e)$ then $e = h_2$, or $(e \xrightarrow{1} h_2) \in \mathbb{G}^{o_2}(h_2)$, or $\mathbb{T}^{\texttt{end}(e)}(e) \sqsubseteq \mathbb{T}^{o_2}(h_2)$.

Induction step ***(**$k = \eta + 1$**).*** There exists an intermediate operation y such that $\texttt{post}(e) \prec_{hb} y$ in k_1 steps and $y \prec_{hb} z$ in k_2 steps such that $k_1 + k_2 = \eta + 1$. We choose y such that $y \prec_{hb} z$ is established by the application of a HB rule other than transitivity, *i.e.*, any event-ordering rule, program order or a base HB rule such as `notify` happens before `wait` and so on. Such an intermediate operation exists because even if $y \prec_{hb} z$ due to transitivity, then by the antecedent of transitive HB rule there must be an intermediate operation y' such that $y \prec_{hb} y'$ and $y' \prec_{hb} z$.

Let $d = task(y)$ and $b = task(z)$. Since $k_1 + k_2 = \eta + 1$, we have $k_1 \le \eta$. Then by induction hypothesis,

$$\mathbb{T}^{x}(task(x)) \sqsubseteq \mathbb{T}^{y}(d), \text{ where } x \text{ is } \texttt{post}(e) \qquad \texttt{[I]}$$

$$d = e, \text{ or } (e \xrightarrow{0/1} d) \in \mathbb{G}^{y}(d), \text{ or } \mathbb{T}^{\texttt{end}(e)}(e) \sqsubseteq \mathbb{T}^{y}(d) \qquad \texttt{[II]}$$

If $b = e$ then we can trivially prove the relevant properties of the lemma for the induction step. Assume $b \neq e$ henceforth. We prove the induction step either by appealing to the action of the transfer function corresponding to the HB rule that establishes $y \prec_{hb} z$, or by establishing the existence of a HB derivation which takes less than k_2 steps to derive $y \prec_{hb} z$ and consequently less than or equal to η steps to derive $x \prec_{hb} z$.

We firstly consider the case where $y \prec_{hb} z$ due to some HB rule other than an event-ordering rule. Then from the corresponding transfer function in Table 4 or Table 5 (see [2]), we can establish the following. The vector clock updates performed by any of the transfer functions establishing $y \prec_{hb} z$ results in $\mathbb{T}^{y}(d) \sqsubseteq \mathbb{T}^{z}(b)$. Combining this with `[I]` we can derive $\mathbb{T}^{x}(task(x)) \sqsubseteq \mathbb{T}^{z}(b)$, thus proving property (1) of the lemma. We will now prove property (2) of the lemma. If $d = e$ is the disjunct satisfied in `[II]`, then the transfer function adds an edge $(e \xrightarrow{1} b)$. Assume $d \neq e$ and $\mathbb{T}^{\texttt{end}(e)}(e) \sqsubseteq \mathbb{T}^{y}(d)$ to be the disjunct satisfied in `[II]`. Then the transfer function establishing $y \prec_{hb} z$ computes $\mathbb{T}^{z}(b)$ by performing `VC-JOIN` of vector clock of b with $\mathbb{T}^{y}(d)$, either by consulting the lookup table or directly with the current vector clock value of d. Hence, $\mathbb{T}^{y}(d) \sqsubseteq \mathbb{T}^{z}(b)$. Combining this with the assumption made we thus have $\mathbb{T}^{\texttt{end}(e)}(e) \sqsubseteq \mathbb{T}^{z}(b)$. If the above two cases do not hold, the remaining case in `[II]` is that $(e \xrightarrow{0/1} d) \in \mathbb{G}^{y}(d)$. If $d = b$ then it is trivial. If not, the transfer function establishing the ordering between y and z propagates $\mathbb{G}^{y}(d)$ to b resulting in an edge $(e \xrightarrow{0/1} b)$. From the definition of $\uplus$ given in Table 2 in [2], this edge is not added only if $\mathbb{T}^{\texttt{end}(e)}(e) \sqsubseteq \mathbb{T}^{z}(b)$. We have thus proved property 2 of the lemma for the induction step when $y \prec_{hb} z$ is established by a *non* event-ordering HB rule and $x \prec_{hb} y$ causes any disjunct in `II` to be satisfied.

We now consider the case where $y \prec_{hb} z$ due to an event-ordering rule. We provide a proof for the `FIFO` rule and the rest can be proved similarly. Then d and b are events on the same thread such that $y = \texttt{end}(d)$ and $z = \texttt{deq}(b)$. Since y and z are ordered by `FIFO(a)` or `FIFO(b)` rule we have $\texttt{post}(d) \prec_{hb} \texttt{post}(b)$. As assumed for the induction step, the ordering from y to z is established in $k_2 \le \eta$ steps. This implies that,

$$\text{The ordering from } \texttt{post}(d) \text{ to } \texttt{post}(b) \text{ is established in at most } k_2 - 1 \text{ steps, where } k_2 \le \eta \qquad \texttt{[III]}$$

Let $a = task(\texttt{post}(b))$. From `[III]`, induction hypothesis is applicable to the derivation of $\texttt{post}(d) \prec_{hb} \texttt{post}(b)$. Then, the following property holds.

$$a = d \text{ or } (d \xrightarrow{0/1} a) \in \mathbb{G}^{\texttt{post}(b)}(a) \text{ or } \mathbb{T}^{\texttt{end}(d)}(d) \sqsubseteq \mathbb{T}^{\texttt{post}(b)}(a) \qquad \texttt{[IV]}$$

If either of the disjuncts $a = d$ or $(d \xrightarrow{0/1} a) \in \mathbb{G}^{\texttt{post}(b)}(a)$ in `[IV]` holds, then the transfer function of `post` adds an edge $(d \xrightarrow{1} b)$ or $(d \xrightarrow{0/1} b)$ respectively. For both these cases on processing $z = \texttt{deq}(b)$ the transfer function of `deq` adds event d to set A (see transfer function of `deq` in Table 4 in [2]) which results in the propagation of all incoming edges of d to b using $\uplus$ and computation of $\mathbb{T}^{z}(b)$ such that $\mathbb{T}^{y}(d) \sqsubseteq \mathbb{T}^{z}(b)$. Then, we can make arguments similar to those made when y and z were assumed to be ordered by a non event-ordering rule and thus prove the induction statement for properties 1 and 2 of the lemma.

Consider the case where the disjunct $\mathbb{T}^{\texttt{end}(d)}(d) \sqsubseteq T^{\texttt{post}(b)}(a)$ of `[IV]` holds. Note that from our definition, for any `post` operation o we have $\mathbb{T}^{o} = T^{o}$. Then from the transfer function of `post` we get $\mathbb{T}^{\texttt{end}(d)}(d) \sqsubseteq \ddot{T}^{\texttt{post}(b)}(b)$. The transfer function of $\texttt{deq}(b)$ preserves this resulting in $\mathbb{T}^{y}(d) \sqsubseteq \mathbb{T}^{z}(b)$, where $y = \texttt{end}(d)$ and $z = \texttt{deq}(b)$. Then from property (1) of Lemma 1.3, $y \prec_{hb} z$. After deriving $\mathbb{T}^{\texttt{end}(d)}(d) \sqsubseteq T^{\texttt{post}(b)}(a)$ with at most $k_2 - 1$ steps of application of HB rules (from `[III]` and `[IV]`), we derived $\mathbb{T}^{y}(d) \sqsubseteq \mathbb{T}^{z}(b)$ and consequently $y \prec_{hb} z$ purely by using transfer functions and Lemma 1.3. Therefore, $y \prec_{hb} z$ is established in at most $k_2 - 1$ steps which in turn means $\texttt{post}(e) \prec_{hb} z$ in at most η steps. Then properties 1 and 2 of the Lemma hold by induction hypothesis. □

THEOREM 1.5. *In a trace τ of a program P evaluated by EventTrack, for a pair of operations $x, z \in \tau$ such that $d = task(x)$ and $b = task(z)$,*

(1) $x \prec_{hb} y$ iff $\mathbb{T}^{x}(d) \sqsubseteq \mathbb{T}^{z}(b)$.
(2) if $x = \texttt{post}(e)$ for an event $e \in Ev$ then, $x \prec_{hb} z$ iff $e = b$, or $(e \xrightarrow{l} b) \in \mathbb{G}^{z}(b)$ for $l \in \{0, 1\}$, or $\mathbb{T}^{\texttt{end}(e)}(e) \sqsubseteq \mathbb{T}^{z}(b)$.
(3) if $x = \texttt{deq}(e)$ for an event $e \in Ev$ then, $x \prec_{hb} z$ iff $e = b$, or $(e \xrightarrow{1} b) \in \mathbb{G}^{z}(b)$, or $\mathbb{T}^{\texttt{end}(e)}(e) \sqsubseteq \mathbb{T}^{z}(b)$.

PROOF. From Lemma 1.3 and Lemma 1.4. □

2 COMPLEXITY ANALYSIS

We present a detailed complexity analysis for EventTrack and compare it with that of EventRacer. Computations performed on vector clocks and event graph determine the time complexity of EventTrack. Whereas for EventRacer, computations performed on vector clocks and HB graph determine this. Both these HB computation techniques for event-driven programs have the same time complexity to perform vector cock updates. Hence, we present the complexity of EventTrack only in terms of computing candidate events using event graph and compare it with that of computing candidate events using HB graph.

Table 1: Complexity of event graph operations with a bound k on the number of incoming edges from tasks on each thread.

Cost of edge retrieval for source events from a given thread	$O(k)$
Cost of inserting an edge into a group of edges with source events from the same thread	$O(k)$
Cost of edge propagation with pruning	$O(k * \|Th\|)$

Analysis of event graph of EventTrack. The incoming edges of all the nodes in an event graph $G = (H, \xi)$ are maintained in a sorted manner as mentioned in Section 4.3 in [2]. The main operations performed on an event graph G are (1) inspecting incoming edges of a task b by performing edge retrieval $G(b)$, (2) inserting an edge into G using $\oplus$ or $\otimes$, and (3) propagating incoming edges of a task d to task b along with pruning incoming edges of b using operator $\uplus$. Clearly, the cost of retrieving edges of b is $O(|G(b)|)$. The incoming edges of every node in G are stored as a linked list which is always maintained sorted as per the Σ IDs of source events. Hence, the cost of inserting an edge into a node b is $O(|G(b)|)$. Propagating edges with pruning involves inspection of all the incoming edges of the source task d as well as source task b, making its complexity $O(|G(b)| + |G(d)|)$.

If we exclude edge retrieval performed internally by edge propagation with pruning (using operator $\uplus$), we observe that EventTrack performs edge retrieval on G only when evaluating an operation posting an event e to the front of a queue, and when identifying candidate events of an event e when evaluating `deq(e)`. In both of these cases it is sufficient to only inspect a subset of $G(e)$ with source events belonging to $dest(e)$. Hence, the incoming edges of every node are grouped into threads corresponding to their source tasks and are sorted using Σ IDs only within these thread groups. The edges in the event graph are transitively propagated thus making the event graph generation cost $O(|H|^3)$ in the worst case, where H is the set of all tasks in the given trace. However, the operator $\uplus$ used to transitively propagate edges also prunes edges, thus preventing further propagation of pruned edges. In the presence of such a pruning let us assume that every task has atmost k incoming edges from tasks belonging to each thread. Then, the maximum in-degree of any node in event graph is $in_{max} = k * |Th|$. With this bound on the number of incoming edges from each thread and thread based grouping of incoming edges, the revised complexities of operations are given in Table 1.

The transfer function of each of the operations except `deq` inserts at most two edges with a cost of $O(k)$, and performs an edge propagation with cost $O(k * |Th|)$. EventTrack orders each event directly only with a corresponding event-covering set. Let the size of event-covering set of each event be at most K. Then, including the cost of inspecting incoming edges to obtain the set of candidate events, the complexity of event graph computations for `deq`'s transfer function is $O(k) + O(k * |Th| * K)$. Hence, the complexity of performing event graph computations on all the operations so as to aid in identifying sets of candidate events is:

$$O(|Op| * (k + k * |Th|)) + O(|Ev| * k * |Th| * K)$$

If we assume the number of events to be linearly proportional to the number of operations and retain only dominant components of the complexity computed, we can derive the complexity of event graph computations as $O(|Op| * k * |Th| * K) = O(|Op| * in_{max} * K)$.

Analysis of HB graph of EventRacer. EventRacer performs DFS traversals on HB graph over operations in the given trace, to identify the set of candidate events for each event. Let the number of such DFS traversals performed by EventRacer be at most d per event in Ev. Then, the complexity of computing candidate events using HB graph and identifying HB ordered events for all the events is $O(d * |Op| * |Ev|)$.

From our complexity analysis, complexity of computing sets of candidate events for EventTrack is linear in the number of operations in the trace, linear in the size of event-covering sets and linear in the maximum in-degree of nodes in event graph. Whereas for EventRacer, the complexity of computing sets of candidate events is linear in the number of operations, linear in the number of events and linear in the number of DFS traversals performed per event. These two complexities are incomparable. This is because, the actual values of in_{max} and K which feature in the complexity computed for EventTrack and the value of d which features in the complexity computed for EventRacer, are an artifact of the HB relation, and hence hard to compare analytically. Thus, we have performed an empirical evaluation and comparison of these two HB computation techniques.

REFERENCES

[1] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 121–133.

[2] Pallavi Maiya and Aditya Kanade. 2017. Efficient Computation of Happens-before Relation for Event-driven Programs. In *Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM. (To appear).