

# Checking Consistency of Event-driven Traces

Parosh Aziz Abdulla<sup>[0000-0001-6832-6611]</sup>, Mohamed Faouzi  
 Atig<sup>[0000-0001-8229-3481]</sup>, R. Govind<sup>\*[0000-0002-1634-5893]</sup>, Samuel  
 Grahn<sup>[0009-0004-1762-8061]</sup>, and Ramanathan S. Thinniyam<sup>[0000-0002-9926-0931]</sup>

Uppsala University, Sweden  
 {parosh.abdulla,mohamed\_faouzi.atig,govind.rajanbabu,  
 samuel.grahn,ramanathan.s.thinniyam}@it.uu.se

**Abstract.** Event-driven programming is a popular paradigm where the flow of execution is controlled by two features: (1) shared memory and (2) sending and receiving of messages between multiple *handler threads* (just called handler). Each handler has a mailbox (modelled as a queue) for receiving messages, with the constraint that the handler processes its messages sequentially. Executions of messages by different handlers may be interleaved. A central problem in this setting is checking whether a candidate execution is *consistent* with the semantics of event-driven programs. In this paper, we propose an axiomatic semantics for event-driven programs based on the standard notion of *traces* (also known as execution graphs). We prove the equivalence of axiomatic and operational semantics. This allows us to rephrase the consistency problem axiomatically, resulting in the *event-driven consistency problem*: checking whether a given trace is consistent. We analyze the computational complexity of this problem and show that it is NP-complete, even when the number of handler threads is bounded. We then identify a tractable fragment: in the absence of nested posting, where handlers do not post new messages while processing a message, consistency checking can be performed in polynomial time. Finally, we implement our approach in a prototype tool and report on experimental results on a wide range of benchmarks.

**Keywords:** Event-driven programs · Consistency-checking · Verification.

## 1 Introduction

Event-Driven (ED) programming has emerged as a powerful paradigm for building scalable and responsive systems capable of handling a large number of user interactions concurrently [14,34,33,45,19,26,20,7]. It is widely used across various domains, including file systems [39], high-performance servers [15], systems programming [16], and smartphone applications [40]. Event-driven programs have become so common that they are considered a core topic under *Programming Fundamentals* according to IEEE and ACM computing curricula [35]. ED programming extends multi-threaded shared-memory programming through the use of messages, thus using both shared-memory as well as message-passing.

---

\* Corresponding author

Verification of ED programs, in addition to the usual challenges associated with shared-memory multi-threaded program verification, has to deal with the non-determinism introduced by the sending and receiving of messages between multiple *handler threads* (just called handler). Each handler has a mailbox (modelled as FIFO queue, following [36,27]) for receiving messages, with the constraint that the handler processes its messages sequentially. Executions of messages by different handlers may be interleaved. A well-established technique for verifying multi-threaded programs is stateless model checking (SMC)[23], which has proven effective for detecting concurrency bugs. SMC has been implemented in several tools - including VeriSoft[24], CHES[42], Concuerror[12], NIDHUGG [1], rInspect [50], CDSCHECKER [43], RCMC [28], and GENMC [31] - and applied to realistic programs [25,30]. To efficiently explore execution traces, SMC tools often employ dynamic partial order reduction (DPOR) [49,44,13,22,6,1,5,2]. DPOR avoids redundant exploration by recognizing and pruning equivalent executions. DPOR does this by exploring the space of all *traces* (also called execution graphs [28]). Intuitively, a trace is a summary of the important concurrency information contained in a program execution, represented as a directed graph whose edges are the union of certain relations (defined further below).

A central component of DPOR techniques is *consistency checking* (e.g., see Section 5 of [4] and Section 4 of [29]), which involves determining whether a candidate trace is realizable, i.e., whether there exists an execution of the program that respects all the relations implied by the trace. The consistency checking problem has been extensively studied on its own for different programming models, notably by Gibbons and Korach (for Sequential Consistency) [21] from 1997 and continued in several works (e.g., [9,48,11]).

We refer to the consistency checking in the event-driven setting as the event-driven sequential consistency problem. In this work, we study the event-driven sequential consistency problem. We first propose an axiomatic semantics for ED programs. We then establish the equivalence between the operational and axiomatic semantics of ED programs. Next, we explore the complexity landscape of the event-driven sequential consistency problem.

Concretely, we consider as input a trace represented by a set of events and relations among them. The goal is to determine whether this trace can arise from a valid event-driven execution. These relations include *Program Order* (the order in which instructions are fetched from the code associated with the message), *Read-From* relation (which relates each read to the write that it reads from), *Coherence Order* (which specifies the order between writes on the same variable). The above relations already exist for general multi-threaded shared-memory programs. In addition, our traces contain the *Execution Order* (which fixes the order in which the messages of the same handlers are executed), *Message order* (ordering the posting of messages to the same handler) and *Posted-by* (relating the instruction posting the message to the instruction starting the execution of the message). We prove that event-driven consistency problem is NP-complete, *even when the number of handler threads is bounded*. On the positive side, we identify a tractable fragment: in the absence of nested posting - where handlers

do not post new messages while processing a message - consistency checking can be performed in polynomial time. Finally, we implement our approach in a prototype tool and report on experimental results on a wide range of event-driven benchmarks, both synthetic and from real-world event-driven programs.

**Related work.** Race detection in event-driven programs has been studied [46,38]. There has also been work on partial order reduction in this setting [36,3] and stateless model checking [27]. The paper [3] considers the consistency problem in the case of mailboxes modeled as multisets, showing its NP-hardness. In the specialised setting of ED programs for real-time systems, the work [18] shows that checking safety properties is undecidable. The robustness problem, which asks if a given an ED program has the same behaviour as if it were to be run on a single thread, has been studied in [8]. Several efforts have been made to provide language support for ED programming such as Tasks [17] as well as the P programming language [16]. In particular, P programming was built to provide safe asynchronous ED programming from the ground up and used to implement and validate the USB driver in Windows 8. Finally, the consistency problem has been extensively studied for different programming languages (e.g., [21,9,48,11,3]). However, as far as we know, this is first time that the consistency problem is studied in the context of ED programs with FIFO queues as mailboxes.

## 2 Event-driven programs: Syntax and Semantics

In the following, we will first give the syntax of Event-driven (ED) programs. Then, we will describe the operational semantics of ED programs. Next, we will define the notion of traces of ED programs and give an equivalent axiomatic definition of traces. Finally, we will define the event-driven consistency problem.

### 2.1 Syntax of Event-Driven Programs

The syntax of event-driven programs we consider is shown in Fig. 1.

An event-driven program  $\mathcal{P}$  has a finite set  $H$  of *handlers*<sup>1</sup>, each  $h \in H$  having a finite set  $R_h$  of *local registers*. We denote

$R = \bigcup_h R_h$ . The handlers interact via a finite set  $X$  of (*shared*) *variables*, as well as via a finite set  $M_h$  of *messages* which are posted to the mailbox  $b_h$  associated

```

< prog > ::= vars < var >* handlers < handler >* msgs < msg >*
          < handler > ::= < handlerId > regs < reg >*
          < msg > ::= < msgname > < inst >* < label > : < last >
          < inst > ::= < label > : < stmt >
          < stmt > ::= < var > = < reg > | < reg > = < var > | < reg > = < exp >
          if < cond > goto < label > | goto < label >
          post(< handlerId >, < msgname >)

```

Fig. 1: Syntax of Event-Driven Programs

<sup>1</sup> ED programs often have designated handler threads with mailboxes as well as *non-handler* threads which do not have an associated mailbox. However, we can think of a non-handler thread as a handler to which a message is never posted and hence simplify notation by assuming that all threads are handlers.

with each handler  $h$ . We assume that the local registers and shared variables take values from a data domain  $D$ . The message sets of different handlers are assumed to be disjoint with  $M = \bigcup_h M_h$  the set of all messages. Each message has a *message name* and comprises of a sequence of instructions, ending in a special instruction *last* which indicates the end of the message.

Each instruction consists of a unique *label* followed by a *statement*. A statement of the form  $\langle \text{var} \rangle = \langle \text{reg} \rangle$  in the grammar indicates the writing of a register value into a shared variable. A statement of the form  $\langle \text{reg} \rangle = \langle \text{var} \rangle$  indicates the read operation of a shared variable which is then stored in the local register of a handler. More complex manipulations of data domain values are assumed to be performed within handlers through the use of *expressions* as in  $\langle \text{reg} \rangle = \langle \text{exp} \rangle$ . These expressions are assumed to only use local registers. The **goto** statement moves the program control to the indicated label, with conditional branching allowed using the *if* construct. The condition *cond* in the *if* statement uses only the local registers of the handler which executes the statement. The *post* statement posts a message to the mailbox of the indicated handler. We assume that the labels in  $\mathcal{P}$  form a finite set  $L$  and there is a successor function  $\text{succ}: L \mapsto L$  which indicates the flow of program control. Each label  $l \in L$  has an associated instruction  $\text{inst}(l)$  which is given by the function  $\text{inst}$ .

## 2.2 Operational Semantics of Event-Driven Programs

We now describe the operational semantics of ED programs, focusing on how handlers interact with mailboxes during the execution of an event-driven program.

**Handler.** A handler  $h$  repeatedly extracts a message from its mailbox, executes the code of the message to completion, then extracts another message and executes its code, and so on. This extraction is modelled as a **get** event. We use a counter at each handler in order to generate unique message IDs. Note that execution of messages by different handlers could be interleaved. Further, while executing the code of a message by a handler, messages could be added to its mailbox. The execution of a message is done one instruction at a time. At any point of time, a handler has at most one *active message* which is being executed. An underlying *nondeterministic scheduler* decides which handler to run at a step.

**Mailbox.** A mailbox is a labelled transition system  $\text{MB} = \langle \mathcal{B}, \beta_{\text{init}}, \{\text{get}, \text{post}\}, \Sigma, \rightarrow \rangle$ , where  $\mathcal{B}$  is the set of configurations of MB,  $\beta_{\text{init}} \in \mathcal{B}$  is the initial configuration, and  $\Sigma$  is the set of messages (including a special symbol  $\perp$ ). We assume that the operations that can be performed on MB are  $\{\text{get}, \text{post}\}$  and the transition relation  $\rightarrow \subseteq \mathcal{B} \times \{\text{get}, \text{post}\} \times \Sigma \times \mathcal{B}$  specifies the semantics of the operations. In this paper, the operations are of two kinds: **get** which downloads a message from the mailbox, and **post** which adds a new message into the mailbox. Since the mailbox is modelled as a FIFO queue and follows the first-in-first-out semantics, we have  $\mathcal{B} = \Sigma^*$  and  $\beta_{\text{init}} = \varepsilon$ . We write  $\beta \xrightarrow{o, \sigma} \beta'$  to denote that the message  $\sigma$  is returned by (resp. posted by) the operation  $o$  if it is a **get** (resp. **post**) and  $\beta = \beta' \cdot \sigma$  (resp.  $\beta' = \sigma \cdot \beta'$ ), while transitioning from

configuration  $\beta$  to configuration  $\beta'$ . A *run*  $\rho = \beta_{\text{init}} \xrightarrow{o_1, \sigma_1} \beta_1 \xrightarrow{o_2, \sigma_2} \dots \beta_n$  of MB is a finite sequence of transitions starting from the initial configuration  $\beta_{\text{init}}$ .

**Configuration of ED-programs.** We use  $h, g, \dots$  for handlers,  $m$  for messages,  $x, y, z$  for shared variables, and  $a, b, c$  for local registers. Members of  $D$  will be denoted by  $v$ . Configurations of programs and mailboxes will be denoted by  $\alpha$  and  $\beta$  respectively. The local state  $s_h = \langle \text{val}, \beta, \text{line}, \text{mid}, \text{mcount} \rangle$  of a handler  $h$  is a tuple containing the valuation  $\text{val}: R_h \mapsto D$  of its local registers, the configuration of its mailbox  $\beta$ ,  $\text{line}$  which is the label of the next instruction that will be executed by the handler, the message id  $\text{mid}$  of its currently active message and a counter  $\text{mcount}$ . The valuation function  $\text{val}$  is extended to expressions in the standard way. When a message  $m$  is to be posted by handler  $h$  to the mailbox of handler  $h'$ , the local counter  $\text{mcount}$  is incremented by 1. A unique mid  $(h, \text{mcount})$  is generated and associated with the message instance. We will write  $s_h.\text{val}, s_h.\beta$  etc to denote the components of  $s_h$ . Given a particular message name  $m$ , let  $m.\text{init}$  denote the label of the first instruction in the code of  $m$ .

A configuration  $\alpha = (\{s_h \mid h \in H\}, \nu)$  of a program consists of the local state of each handler  $h$  along with a valuation  $\nu: X \mapsto D$  of the shared variables. We sometimes write  $\alpha.\nu, \alpha.s_h$  etc to denote the valuation of global variables and the local state of handler  $h$  respectively, in configuration  $\alpha$ . A program  $\mathcal{P}$  starts in some initial configuration<sup>2</sup>  $\alpha_0 = (\{s_h^0 \mid h \in H\}, \nu)$  which satisfies the condition that for each handler  $h$ , we have  $s_h^0.\text{line} = m.\text{init}$  for some  $m \in M_h$  i.e. the execution of some message is initialised in each handler. Note that there is no post event associated with this initialization. Furthermore, the mailboxes are all empty i.e.  $s_h^0.\beta = \beta_{\text{init}}$  for all  $h$ ,  $s_h^0.\text{mid} = (h, 0)$  and  $s_h^0.\text{mcount} = 1$ .

A transition  $\alpha \xrightarrow{a} \alpha'$ , between two configurations  $\alpha$  and  $\alpha'$ , occurs on either the execution of an instruction or a **get** operation. The subset of rules dictating transitions relevant to concurrency is shown in Fig. 2. The other rules i.e. local transitions within a thread, can be found in Fig. 6 in Appendix A. Given a tuple/mapping  $f$ , we use  $f(x \leftarrow d)$  to denote the tuple/mapping  $f'$  which agrees with  $f$  on all parameters except  $x$ , on which it takes the value  $d$ . We write  $f(x \leftarrow d)$  (resp.  $f(x_1 \leftarrow d_1, \dots, x_n \leftarrow d_n)$ ) instead of  $f(g \leftarrow g(x \leftarrow d))$  (resp.  $f(x_1 \leftarrow d_1) \dots (x_n \leftarrow d_n)$ ) when it is clear from the context.

An *execution sequence* or run  $\rho$  of program  $\mathcal{P}$  is a finite sequence of transitions  $\alpha_0 \xrightarrow{a_1} \alpha_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} \alpha_n$  starting with an initial configuration  $\alpha_0$ .

### 2.3 Events, Traces and Axiomatic Consistency

In this subsection, we introduce an axiomatic semantics for ED programs. We first define the relevant types of events and then formalize the notion of a trace.

<sup>2</sup> Note that we intentionally do not specify the initial values of local registers and shared variables, or the initial message each handler should execute, as the event-driven consistency problem treats the program code as a black box. However, our framework can be easily extended to take into account such initial conditions.

$$\begin{array}{c}
\text{WRITE} \\
\frac{\text{inst}(\alpha.s_h.\text{line}) = l_i : x = a \wedge \alpha.s_h.\text{val}(a) = v}{\alpha \xrightarrow{\langle h, \text{write}, x, v \rangle} \alpha(\nu \leftarrow \nu(x \leftarrow v), s_h.\text{line} \leftarrow \text{succ}(s_h.\text{line}))} \\
\\
\text{POST} \\
\frac{\alpha.s_h.\text{line} = l : \text{post}(h', m) \quad \alpha.s_{h'}.\beta \xrightarrow{\text{post}, (m, \text{newmid})} \beta' \quad \text{newmid} = (h, \alpha.s_h.\text{mcount})}{\alpha \xrightarrow{\langle h, \text{post}, h', \text{newmid} \rangle} \alpha(s_{h'}.\beta \leftarrow \beta', s_h.\text{mcount} \leftarrow s_h.\text{mcount} + 1, s_h.\text{line} \leftarrow \text{succ}(s_h.\text{line}))} \\
\\
\text{READ} \\
\frac{\text{inst}(\alpha.s_h.\text{line}) = l : a = x}{\alpha \xrightarrow{\langle h, \text{read}, x \rangle} \alpha(s_h.\text{val} \leftarrow s_h.\text{val}(a \leftarrow \nu(x)), s_h.\text{line} \leftarrow \text{succ}(s_h.\text{line}))} \\
\\
\text{GET} \\
\frac{\alpha.s_h.\text{line} = l : \text{last} \quad \alpha.s_h.\beta_h \xrightarrow{\text{get}, (m, \text{mid})}_{\text{MB}} \beta'}{\alpha \xrightarrow{\langle h, \text{get}, \text{mid} \rangle} \alpha(s_h.\beta \leftarrow \beta', s_h.\text{mid} \leftarrow \text{mid}, s_h.\text{line} \leftarrow m.\text{init})}
\end{array}$$

Fig. 2: A subset of transition rules of ED programs with  $\alpha = (\{s_h \mid h \in H\}, \nu)$ .

*Events* An event is a collection of information about a transition that is meant to be made visible. Transitions which contain such information are called event-transitions (observe that local-transitions do not have corresponding events). As shown in Fig. 2, there are four types of event-transitions: reads, writes, posts and gets. The event is obtained from the event-transition by dropping the mid and newmid information. Note that newmid is basically a newly created mid written this way for clarity. Formally,

- A **write** event is a tuple  $e = \langle h, \text{write}, x, v \rangle$  which denotes the writing of the value  $v$  by handler  $h$  into global variable  $x$ . We say  $e.\text{var} = x$  and  $e.\text{val} = v$ . We denote by  $\mathcal{W}_x$  the set of all write events on  $x$ .
- A **read** event is a tuple  $e = \langle h, \text{read}, x \rangle$  which denotes the reading of the value stored in global variable  $x$  by handler  $h$ . We say  $e.\text{var} = x$ . We denote by  $\mathcal{R}_x$  the set of all read events on  $x$ .
- A **post** event is a tuple  $e = \langle h, \text{post}, h' \rangle$  which denotes the posting of a message by handler  $h$  to the mailbox of the handler  $h'$ . We write  $e.\text{sender}$  to denote  $h$  and  $e.\text{receiver}$  to denote  $h'$ .
- A **get** event is a tuple  $e = \langle h, \text{get} \rangle$  which denotes the downloading of a message by handler  $h$ .

In general, we write  $e.h$  to denote the handler on which a message is being executed. In particular,  $e.h$  is the same as  $e.\text{sender}$  for a post event. Given a transition  $\alpha \xrightarrow{a} \alpha'$  we write  $e(a)$  for the event corresponding to  $a$  if  $a$  is an event-transition. In case the transitions are indexed e.g.  $a_i$  then we just write  $e_i$  instead of  $e(a_i)$ . For an event  $e$ , we denote by  $e.\text{type}$  the *type* of the event i.e. whether it is a read, write, **post** or **get**. Note that unless necessary, we omit handler identifiers from events for readability.

*Traces* Let  $\text{rels} = \{\text{rf}, \text{co}, \text{po}, \text{eo}, \text{pb}, \text{mo}\}$  be a set of relation names. A trace is a directed graph  $\tau = (E, \Delta)$  where  $E$  is a finite set of events,  $\Delta \subseteq E \times \text{rels} \times E$  is a set of edges on  $E$  with labels from  $\text{rels}$ . Let  $E_h = \{e \mid e.h = h\}$  be the

set of events occurring on handler  $h$ . Let  $G_h = \{e \mid e.type = get\} \cap E_h$  and  $P_h = \{e \mid e.type = post \wedge e.receiver = h\}$  be respectively the get events of handler  $h$  and the post events to  $h$ . The following conditions are satisfied by  $\Delta$ :

- rf: (reads-from) maps each read instruction to a write instruction. For each  $x \in X$  and each  $e \in \mathcal{R}_x$ , there exists exactly one  $e' \in \mathcal{W}_x$  such that  $e' \text{ rf } e$ .
- po: (program order) is a union of total orders on the set of events  $E_h$  which occur on a particular handler. This is a total ordering on all events which happen as part of the execution of a particular message instance. Formally, we have
  - For any  $e \in E_h \setminus G_h$ , there exists at most one event  $e' \in G_h$  s.t.  $e' \text{ po } e$ .
  - For every  $e, e'' \in E_h \setminus G_h$  such that  $e' \text{ po } e$  and  $e' \text{ po } e''$  for some  $e' \in G_h$ , it is the case that either  $e \text{ po } e''$  or  $e'' \text{ po } e$ .
  - Let  $E'_h$  be the set of events  $e \in E_h \setminus G_h$  such that there is no event  $e' \in G_h$  with  $e' \text{ po } e$ . Then,  $\text{po}$  is a total order over  $E'_h$ . Furthermore, for every  $e \in E'_h$  and  $e' \in G_h$ , we have  $e \text{ po } e'$ . This means that all events of the initial message are ordered before the events of any other message.
- co: (coherence order) For each pair of writes  $e, e' \in \mathcal{W}_x$ , either  $e \text{ co } e'$  or  $e' \text{ co } e$ . Note that  $\text{co}$  is a total order on the set  $\mathcal{W}_x$  for each  $x \in X$ .
- eo: (execution order) is a total order on the set of get events occurring on a handler. Let  $e, e' \in G_h$  for some  $h$ . Then either  $e \text{ eo } e'$  or  $e' \text{ eo } e$ .
- pb: (posted by) is a relation which relates each get event on a handler to the corresponding post event. In other words, it is a bijection between the sets  $P_h$  and  $G_h$ : for each  $e \in G_h$  there is exactly one  $e' \in P_h$  such that  $e' \text{ pb } e$ .
- mo: (message order) orders the events that posts messages to the mailbox of a particular handler. For every  $e, e' \in P_h$  either  $e \text{ mo } e'$  or  $e' \text{ mo } e$ .

A partial trace is a subgraph of a trace. A partial trace  $\tau' = (E', \Delta')$  is said to *extend* a partial trace  $\tau = (E, \Delta)$  if  $E \subseteq E', \Delta \subseteq \Delta'$ . A linearization  $\pi = (E, \leq_\pi)$  of a partial trace  $\tau = (E, \Delta)$  is a total ordering  $\leq_\pi$  satisfying  $\delta \in \Delta \Rightarrow \delta \in \leq_\pi$ .

*Traces of Programs* Given a program  $\mathcal{P}$  and its execution  $\rho = \alpha_0 \xrightarrow{a_1} \alpha_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} \alpha_n$ , we define the set  $E(\rho) = \{e_i \mid a_i \text{ is an event-transition}\}$  to be the *event set of  $\rho$* . Clearly  $\rho$  induces a total order  $\leq_\rho$  on  $E(\rho)$  defined in the natural way:  $e_i \leq_\rho e_j$  iff  $i \leq j$ . Each **get** event-transition specifies an mid for the message instance which is obtained from the mailbox. The execution of this message instance may contain more event-transitions later in  $\rho$ . Hence we extend the notion of mid to all non-**get** event-transitions in the following way. For each event  $e_i$  which is not a get event, let  $e_j$  be the first preceding get event  $e_j$  in the order  $\rho$  such that  $e_i.h = e_j.h$ , if such an event exists. We assign the message id  $a_j.mid$  to the transition  $a_i$ , since by the event-driven semantics, only one message can be executed by a handler at any point in time. If no such get event exists, then we assign message id  $(h, 0)$  to  $a_i$ . Note that a post event has both an mid from the message it is part of as well as a newmid for the message it is creating.

Recall that for  $x \in X$ , we have  $\mathcal{R}_x = \{e \in E \mid e.type = read, e.var = x\}$ ,  $\mathcal{W}_x = \{e \in E \mid e.type = write, e.var = x\}$ . The event set  $E$  together with the total order  $\leq_\rho$  derived from a run *induces* a trace  $\tau(\rho)$  in the following way:

- rf: If  $e_i \leq_\rho e_j$  where  $e_i \in \mathcal{W}_x, e_j \in \mathcal{R}_x$ , and for all  $e_i \leq_\rho e_k \leq_\rho e_j$  we have  $e_k \notin \mathcal{W}_x$ , then  $e_i \text{ rf } e_j$ .

- co: If  $e_i, e_j \in \mathcal{W}_x$  for some  $x$  and  $e_i \leq_\rho e_j$  then  $e_i \text{ co } e_j$ .
- po: If  $e_i, e_j$  are such that  $a_i.\text{mid} = a_j.\text{mid}$  and  $e_i \leq_\rho e_j$  then  $e_i \text{ po } e_j$ . Further, if  $e_i, e_j$  are such that  $e_i.h = e_j.h$ ,  $a_i.\text{mid} = (h, 0)$  and  $a_j.\text{mid} \neq a_i.\text{mid}$  then  $e_i \text{ po } e_j$ .
- eo: If  $e_i \leq_\rho e_j$  satisfies  $e_i.\text{type} = e_j.\text{type} = \text{get}$  and  $e_i.h = e_j.h$ ,  $a_i.\text{mid} \neq a_j.\text{mid}$  then  $e_i \text{ eo } e_j$ .
- pb: If  $e_i \leq_\rho e_j$  satisfies  $e_i.\text{type} = \text{post}$ ,  $e_j.\text{type} = \text{get}$  and  $a_i.\text{newmid} = a_j.\text{mid}$  then  $e_i \text{ pb } e_j$ .
- mo: If  $e_i, e_j$  are both post events such that  $e_i.\text{receiver} = e_j.\text{receiver}$  and  $e_i \leq_\rho e_j$  then  $e_i \text{ mo } e_j$ .

The definition of rf ensures that every read on a variable reads from the latest write on that variable. The coherence order co is just the sequence of writes on a variable. The mid information tells us which set of instructions belong to the same message from which we can infer po. Similarly the mid information also tells us which get is posted-by which post. The sequence of message executions on a handler (eo) and the order in which messages were posted to a handler (mo) can be inferred from  $\leq_\rho$ . This gives us the following lemma:

**Lemma 1.** *For any program  $\mathcal{P}$  and its execution  $\rho$ ,  $\tau(\rho)$  is a trace.*

*Remark 1.* Recall that a handler processes a message in its entirety before accessing the next message from its mailbox. Hence all events belonging to one message of  $h$  must occur before all events of all subsequent messages of  $h$ . Further, note that since the mailbox is a FIFO queue, then another requirement is that the order in which messages are removed from the mailbox needs to respect the order in which they are added to the mailbox. In this case, mo is a total ordering on all events that post messages to the same handler. The restriction here is that the order in which messages are extracted should be according to the mo between the events that posts the messages to the same mailbox.

*Axiomatic Consistency* We introduce conditions under which a trace  $\tau$  is said to be *axiomatically consistent* and show that this happens iff  $\tau$  can be derived from the run of some event driven program. The conditions are given as is standard by means of acyclicity of the union of relations. To this end, we introduce new relations: The *queue order* qo is defined as  $\text{qo} = \text{pb}^{-1}.\text{mo}.\text{pb}$ . The from-reads relation be defined as  $\text{fr} = \text{rf}^{-1}.\text{co}$ . Let  $\text{eo}^\dagger = (\text{po}^{-1})^*.\text{eo}.\text{po}^*$ .

**Definition 1.** *A trace  $\tau$  is said to be axiomatically consistent if the happens-before relation  $\text{hb} = (\text{po} \cup \text{rf} \cup \text{fr} \cup \text{co} \cup \text{pb} \cup \text{mo} \cup \text{eo}^\dagger \cup \text{qo})$  is acyclic.*

**Theorem 1.** *A trace  $\tau$  is axiomatically consistent iff there exists an event-driven program  $\mathcal{P}$  and a run  $\rho$  such that  $\tau = \tau(\rho)$ .*

The proof of the theorem can be found in Appendix B.



## 2.4 Event-driven consistency problem

Having defined both operational and axiomatic semantics, we now introduce the event-driven consistency problem, which asks whether a partial trace can be extended to an axiomatically consistent trace.

We first recall a related problem in the non-event-driven setting, where only the relations **po** (program order), **rf** (reads from) and **co** (coherence order) are relevant. When all three are given, consistency checking is tractable. However, if only **po** and **rf** are provided (aka **rf-consistency**), is known to be NP-complete [21]. In ED programs, we additionally deal with **pb**, **mo** and **eo**. Given the NP-completeness of the **rf-consistency** problem, it is natural to ask about the complexity of ED-consistency where **mo** and **eo** are not provided<sup>3</sup>. As in the nonED case, our proof of equivalence in Theorem 1 implies that consistency is in polynomial time if all the relations are given. This leads to the well-motivated consistency problem for event-driven programs. Let  $\text{rels}' = \{\text{po} \cup \text{rf} \cup \text{co} \cup \text{pb}\}$

**Definition 2 (ED-Consistency Problem).** *Given a partial trace  $\tau' = (E', \Delta')$  with  $\Delta' \subseteq E \times \text{rels}' \times E$ , decide whether there exists an axiomatically consistent extension trace  $\tau = (E, \Delta)$  of  $\tau'$  such that  $\Delta' \cap (E \times \text{rels}' \times E) = \Delta \cap (E \times \text{rels}' \times E)$ .*

## 3 Complexity of Event-driven Consistency

In this section, we study the complexity of the event-driven consistency problem. We show that the problem is NP-hard. Further, since our reduction uses only 12 handlers, this also implies the hardness for the more restricted version of the problem, with only a bounded number of handler threads.

The proof will follow from a reduction from *3-Bounded Instance 3SAT* (3-BI-3SAT in short), a problem known to be NP-complete [47]. Further details and a proof of correctness are given in Appendix C.

**Definition 3 (3-BI-3SAT).** *A 3-BI-3SAT problem is the Boolean satisfiability problem restricted to conjunctive normal form formulas such that: (1) each clause contain two or three literals, (2) each variable occurs in at most 3 clauses, and (3) each variable appears at most once per clause.*

**Theorem 2.** *The ED-consistency problem is NP-complete for traces with at most 12 handlers.*

The proof is done by reduction from the 3-BI-3SAT. Let  $\phi$  be a 3-BI-3SAT instance with variables  $x_1, x_2, \dots, x_n$  and clauses  $C_1, C_2, \dots, C_m$ . We will construct a partial ED trace  $\tau = (E, \Delta)$ , with  $\Delta \subseteq E \times \text{rels}' \times E$ , such that  $\tau$  can be extended to a axiomatically consistent trace  $\tau' = (E, \Delta')$ , with  $\Delta' \cap (E \times \text{rels}' \times E) = \Delta \cap (E \times \text{rels}' \times E)$ , iff  $\phi$  is satisfiable.

<sup>3</sup> Note that the **eo** order should respect the **mo** order, since the mailbox is a FIFO queue. When we talk about partial traces, we do not explicitly mention the relations present. The understanding is that both **mo** and **eo** are missing.

**High level structure.** The construction of the trace  $\tau$  is divided into two stages, which we call Stage 1 and Stage 2 respectively. There are 8 handlers in Stage 1 and 5 handlers in Stage 2. One handler, namely  $h_W$  is common to both stages, hence totally there are 12 handlers. If a satisfying assignment exists for  $\phi$ , then there is a program which can execute the events in Stage 1 followed by those in Stage 2, i.e.,  $\tau$  is consistent. If  $\phi$  is unsatisfiable then there is no witnessing execution possible which executes both stages and  $\tau$  is inconsistent.

Stage 1 corresponds to the selection of a satisfying assignment  $f$  for  $\phi$ . We can encode the information of whether a variable  $x_i$  is assigned true or false using the order of execution of two messages  $m_{i,1}$  and  $m_{i,0}$  on the same handler, where  $x_i$  is assigned true (resp. false) if  $m_{i,1}$  (resp.  $m_{i,0}$ ) is executed later. Unfortunately, this will not work due to technical difficulties faced in clause verification (see Remark 3 in Subsection 3.2). This necessitates our extremely technical reduction which makes use of the structure of the 3-BI-3SAT instance where each variable occurs in *at most* three clauses and the variables occurring in a clause are all different. We have to create (at most) 3 copies of the messages, one for each clause in which  $x_i$  occurs and find a way to synchronise the assignment between these three copies.

Hence the messages for  $x_i$  are actually of the form  $m_{i,j,b}$  where  $j$  refers to clause  $C_j$  and  $b \in \{0, 1\}$ . Using the technique of *post sequences* which use nested posting (explained using example in Subsection 3.2), we post the set  $M$  of  $m_{i,j,b}$  messages in the queue of  $h_W$  in some order  $\sigma$ . The remaining 7 handlers of Stage 1 are used to shuffle the messages in  $M$  with certain restrictions on the order  $\sigma$  of messages.

The set  $S$  of all the possible orders  $\sigma$  is such that, every  $\sigma$  is constrained to be *consistent* (see Challenge 2 of Subsection 3.1) with some particular assignment  $f$  of variables of  $\phi$ . There are no other constraints on the order  $\sigma$ . At the end of Stage 1, the queues of all other Stage 1 handlers is empty and the queue of  $h_W$  is populated in some order  $\sigma \in S$  consistent with some assignment  $f$ . Note that there are multiple  $\sigma$  which are consistent with a particular  $f$ , this fact will be important later.

Stage 2 Let us fix  $\sigma$  and  $f$  from Stage 1. Then Stage 2 verifies that  $f$  indeed satisfies all the clauses of  $\phi$ . For this, we build a clause gadget  $G_j$  corresponding to each clause  $C_j$ . The set  $E_G$  of events of these clause gadgets occupy the 4 non- $h_W$  handlers of Stage 2. The  $E_G$  events belong to an initial message of each of the 4 handlers, and consist purely of read and write events. Recall that the queue of  $h_W$  is populated at this point with message set  $M$ . The information regarding the assignment is encoded in the order of the messages in  $h_W$ . This information is transferred to the other 4 non- $h_W$  handlers via a technique we call *sandwiching* (explained using example in Subsection 3.1). There are now two possibilities:

(1) If  $f$  is not a satisfying assignment, then some clause  $C_j$  is not satisfied by  $f$ . In this case, any order  $\sigma$  of messages consistent with  $f$  will induce a hb (happens before) cycle in the corresponding gadget  $G_j$  via the sandwiching. Therefore

Stage 2 cannot be executed by any witnessing execution. If there are no satisfying assignments, then  $\phi$  is unsatisfiable and hence  $\tau$  is not consistent.

(2) If  $f$  is a satisfying assignment then there is some order  $\sigma$  of the messages in  $M$  which is consistent with  $f$  such that there is a witnessing execution. The clause gadgets are executed interleaved with the messages in  $M$  due to the sandwiching. The execution happens sequentially i.e.  $G_1$  is executed, then  $G_2$  etc. This implies  $\tau$  is consistent.

*Remark 2.* When we say that message  $m$  is posted to handler  $h$  before message  $m'$  is posted to  $h$ , we refer to the order of post operations as executed in the witnessing execution.

Through the use of examples, we now give intuition on how the two Stages work, beginning with Stage 2.

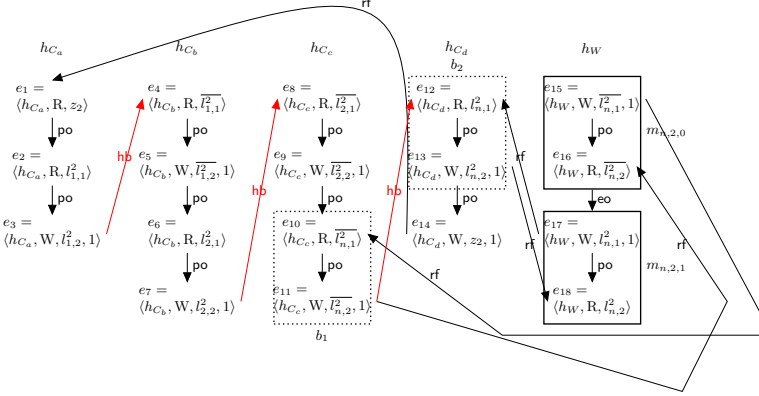
### 3.1 Stage 2: Checking satisfaction of clauses.

We assume that  $h_W$  has been populated with messages in accordance with a variable assignment function  $f$  in Stage 1. Each clause  $C_j$  is associated with a *clause gadget*  $G_j$  implemented using handlers  $h_{C_a}, h_{C_b}, h_{C_c}, h_{C_d}$  (the non- $h_W$  handlers of Stage 2). For example, for  $C_2 = x_1 \vee x_2 \vee \bar{x}_n$ , Figure 3 shows the corresponding gadget  $G_2$  and two messages posted to  $h_W$  in Stage 1. For reasons of space we use  $W$  and  $R$  for *write* and *read* to describe events.

Within each gadget, there are *boxes* that contain a read followed by a write event in **po**. Each box is linked to messages in  $h_W$  through **rf** relations. Consider the box  $b_1$  in Figure 3. Here, the variable  $l_{n,1}^2$  in  $e_{10}$  has the information: superscript 2 for clause  $C_2$ , subscript  $n, 1$  indicating the literal  $\bar{x}_n$  and 1 indicating it is the first event in the box. The events in  $b_1$  are linked to the read and write events in the message  $m_{n,2,0}$  via **rf** arrows. The direction of the arrows implies that the events in box  $b_1$  have to be executed after event  $e_{15}$  and before  $e_{16}$  which are both in message  $m_{n,2,0}$ . This is the technique we call *sandwiching*.

Similarly  $b_2$  has to be executed during the execution of  $m_{n,2,1}$ . Suppose  $m_{n,2,0}$  is executed before  $m_{n,2,1}$  as indicated by the **eo**, this means that  $x_n$  is assigned the value **true**. This sandwiching induces the red **hb** relation shown between  $e_{11}$  and  $e_{12}$ .

Clause satisfaction. Notice that similar boxes can be drawn around events  $e_2, e_3$  and  $e_4, e_5$  corresponding to copying the assignment to variable  $x_1$  and for  $e_6, e_7$  and  $e_8, e_9$  for variable  $x_2$ . Each of these boxes has similar sandwiching **rf** relations to messages in  $h_W$  which are not shown in the figure. The three red **hb** arrows correspond to setting each of the three variables in  $C_2$  to a value that falsifies the corresponding literal in  $C_2$ . The events  $e_1$  and  $e_{14}$  use a variable  $z_2$  (where the subscript refers to the clause  $C_2$ ) and are connected by an **rf**. Under these conditions, a cycle is formed and thus the clause gadget cannot be executed. On the other hand, if even one of the red arrows is flipped (indicating that a literal of  $C_2$  is set to **true**), then the arrows form a partial order allowing execution of the clause gadget  $G_2$ .

Fig. 3: The structure of clause gadget  $G_2$  of  $C_2$  for the example in Figure 4.

Note that the clause gadgets  $G_1, G_1, \dots, G_m$  are placed in that order in the handlers  $h_{C_a}, h_{C_b}, h_{C_c}, h_{C_d}$  and connected by **po** arrows. For example,  $G_j.e_3$  will be **po** before  $G_{j+1}.e_1$  in  $h_{C_a}$ ,  $G_j.e_7$  will be **po** before  $G_{j+1}.e_4$  in  $h_{C_b}$ , etc. In other words, the events of each of these four handlers can be assumed to be in an initial message in the respective handlers. There is no posting of events either from or to these 4 handlers.

### 3.2 Stage 1: Encoding variable assignments.

In this Stage, we use the handlers  $h_V, h_{t_1}, h_{t_2}, h_{t_3}, h_{t_4}, h_{t_5}, h_{t_6}$  in order to post messages to  $h_W$ . We stated that an assignment to variable  $x_i$  can be encoded as the order between  $m_{i,j,0}, m_{i,j,1}$ .

**Challenge 1:** How can we ensure that the messages  $m_{i,j,0}, m_{i,j,1}$  can be posted in any order to  $h_W$ ?

In order to solve this, we use *nested posting*.  $h_V$  posts  $m'_{i,j,0}$  to  $h_1$  and  $m'_{i,j,1}$  to  $h_2$ , which in turn post  $m_{i,j,0}$  and  $m_{i,j,1}$  to  $h_W$ . Since  $m'_{i,j,0}$  and  $m'_{i,j,1}$  are on different handlers, they can be executed in any order, thus ensuring that  $m_{i,j,0}, m_{i,j,1}$  can be posted in any order to  $h_W$ . Next we take up the reason for using multiple messages for each variable.

*Remark 3.* Consider the sandwiching technique that we presented in Stage 2 in order to copy the assignment of a variable to the clause gadget. Suppose we were to use a single pair of messages  $m_{i,0}, m_{i,1}$  in  $h_W$  for a variable  $x_i$  from which this value was copied to the different clauses in which  $x_i$  occurs. This means that any handler in which a clause gadget is being executed would be blocked from running till all of the clauses containing  $x_i$  are able to finish executing the boxes corresponding to  $x_i$ . This leads to a cascading set of blocked handlers, requiring an unbounded number of handlers to execute the clause gadgets. In order to overcome this difficulty, we have to use upto three copies of the two messages  $m_{i,0}, m_{i,1}$  as mentioned before. But this leads to a different challenge.

Challenge 2: How can we ensure that the different copies of the messages corresponding to the same variable exhibit the same value?

To solve this, we rely on the structure of the 3-BI-3SAT formula. Figure 4 depicts a grid with variables as rows and clauses as columns. Marked cells indicate variable occurrences. For each literal  $l$ , we define a *post sequence*  $p^l$  composed of segments  $p_1^l, \dots, p_7^l$  corresponding to marked and unmarked cells.

To address this, we further extend the nesting of posts, relying on the structure of the 3-BI-3SAT formula  $\phi$ . Figure 4 depicts a grid with variables as rows and clauses as columns. Marked cells indicate variable occurrences. Each row contains 2 or 3 marked cells and each column contains 4 or 6 marked cells as per the restriction on 3-BI-3SAT. For each literal  $l$ , we define a *post sequence*  $p^l$  composed of segments  $p_1^l, \dots, p_7^l$  corresponding to marked and unmarked cells. Marked cells trigger the posting of messages to  $h_W$ , while unmarked ones simply post to  $h_V$  and defer execution. The post sequence of each marked cell is designed to enforce consistent ordering of the associated messages. We will now describe the post sequences in detail.

A post sequence is a partial trace of the form

$$\langle h_0, \text{post}, h_1 \rangle \xrightarrow{\text{pb}} \langle h_1, \text{get} \rangle \xrightarrow{\text{po}} \langle h_1, \text{post}, h_2 \rangle \xrightarrow{\text{pb}} \dots \xrightarrow{\text{po}} \langle h_{n-1}, \text{post}, h_n \rangle$$

We will simply write this as  $p = \langle h_1, \text{post}, h_2, \text{post}, \dots, \text{post}, h_n \rangle$ . In case  $h_i = h_{i+1} = \dots = h_j$  we will further shorten this to  $\langle h_1, \text{post}, h_2, \text{post}, \dots, h_i, \text{post}^{j-i-1}, h_j, \text{post}, h_{j+1}, \text{post}, \dots, \text{post}, h_n \rangle$ .

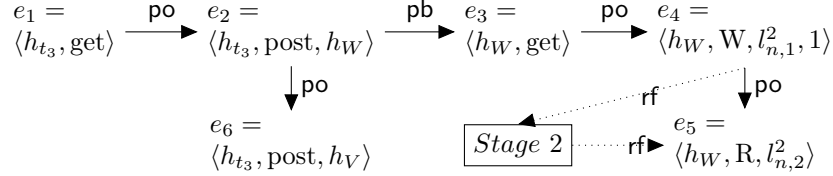
Consider the row labelled by  $\bar{x}_1$  in the Figure 4. The post sequence is the concatenation of 7 post sequences  $p^{\bar{x}_1} = p_1^{\bar{x}_1} p_2^{\bar{x}_1} p_3^{\bar{x}_1} p_4^{\bar{x}_1} p_5^{\bar{x}_1} p_6^{\bar{x}_1} p_7^{\bar{x}_1}$  where  $p_2^{\bar{x}_1}, p_4^{\bar{x}_1}, p_6^{\bar{x}_1}$  correspond to the cells marked  $\bar{1}, 1, 1, 2$  and  $\bar{1}, 3$  respectively, while the others correspond to the part of the row consisting of unmarked cells, with  $p_1^{\bar{x}_1}$  for the part from the beginning till the first marked cell, etc. Each of the post sequences  $p_1^{\bar{x}_1}, p_3^{\bar{x}_1}, p_5^{\bar{x}_1}, p_7^{\bar{x}_1}$  consists of a long sequence of posts to  $h_V$  of length the number of unmarked cells in the segment they correspond to. For example  $p_1^{\bar{x}_1} = p_3^{\bar{x}_1} = \langle h_V, \text{post}, h_V \rangle$  while  $p_5^{\bar{x}_1} = \langle h_V, \text{post}^3, h_V \rangle$  since there are 3 empty cells in between (in the figure they are not explicitly shown, but rather by  $\dots$ , but once can infer that the boxes corresponding to  $C_5, C_6, C_7$  are empty along this row). The idea is that the post sequences are executed column by column. The empty cell post sequences simply ‘send to back of queue’

	$C_1$	$C_2$	$C_3$	$C_4$	$\dots$	$C_8$	$\dots$	$C_m$
$x_1$		(1, 1)		(1, 2)	$\dots$	(1, 3)	$\dots$	
$\bar{x}_1$		(1, 1)		(1, 2)	$\dots$	(1, 3)	$\dots$	
$x_2$	(1, 1)	(2, 2)	(3, 1)		$\dots$		$\dots$	
$\bar{x}_2$	(1, 1)	(2, 2)	(3, 1)		$\dots$		$\dots$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$x_n$		(1, 3)			$\dots$		$\dots$	
$\bar{x}_n$		(1, 3)			$\dots$		$\dots$	

Fig. 4: Relationship between variables and clauses dictating the nesting of posts. Empty cell means variable does not occur in clause (not all nonempty cells are shown). A cell marked  $(u, v)$  or  $(\bar{u}, v)$  indicates that it is the  $u$ -th occurrence of the variable in a clause and is the  $v$ -th variable of the clause. The bars on the tuple indicate the polarity of the variable occurrence.  $C_2 = x_1 \vee x_2 \vee \bar{x}_n$ ,  $x_1$  occurs in  $C_2, C_8$  and  $\bar{x}_1$  occurs in  $C_4$ .

while the marked cells are responsible for populating  $h_W$  with an appropriate sequence of messages as explained below.

We now describe the post sequences made in the marked cells. Consider the 6 marked cells corresponding to column  $C_2$ . Top to bottom, these are  $p_2^{x_1}, p_2^{\bar{x}_1}, p_4^{x_2}, p_4^{\bar{x}_2}, p_2^{x_n}, p_2^{\bar{x}_n}$ . Let us consider the post sequence for a cell labelled  $(u, v)$  (resp.  $(u, \bar{v})$ ), indicating that it is the  $u$ -th occurrence of the variable in a clause and is the  $v$ -th variable of the clause, with the bar indicating whether the variable or its negation occurs in the clause. Suppose  $u \neq 1$ , then the post sequence is  $\langle h_V, \text{post}, h_{t_v}, \text{post}, h_V \rangle$  for both  $(u, v)$  as well as  $(u, \bar{v})$ . If  $u = 1$  then the post sequence is  $\langle h_V, \text{post}^2, h_{t_v}, \text{post}, h_V \rangle$  for  $(u, v)$  but it is  $\langle h_V, \text{post}, h_{t_{v+3}}, \text{post}, h_{t_v}, \text{post}, h_V \rangle$  for  $(u, \bar{v})$ . For example,  $p_2^{x_n}$  which is marked  $(1, 3)$  has the post sequence  $\langle h_V, \text{post}, h_{t_6}, \text{post}, h_{t_3}, \text{post}, h_V \rangle$ . Intuitively, the post sequences of the variable and its negation move to different handlers  $h_{t_k}$  before coming back to the same handler iff a variable is occurring for the first time in a clause i.e., if  $u = 1$ . We modify each post sequence of a marked cell to post a message  $m_{i,j,b}$  (corresponding to the occurrence of  $x_i$  in  $C_j$  in positive or negative form based on the value of the bit  $b$ ) to  $h_W$  just before its return to  $h_V$ . For example, in  $p_2^{x_n}$  we insert the events  $e_2, e_3, e_4, e_5$  between the events  $e_1$  and  $e_6$  which are part of  $p_2^{x_n}$  as follows:



This means that six messages are posted to  $h_W$  corresponding to  $C_2$ . The assignment to  $x_1$  and  $x_n$  are chosen by using different handlers  $h_{t_i}$  for them, but the assignment to  $x_2$  was already chosen when executing the post sequence for  $C_1$  corresponding to  $x_2$ . Hence  $p_4^{x_2}, p_4^{\bar{x}_2}$  will both contain  $h_{t_2}$  and the corresponding messages will be posted to  $h_W$  in the order already chosen. Crucially, we prevent orderings in  $h_W$  which do not correspond to consistent assignment of variables. However we allow all other possible reorderings of messages and this is essential for the verification in Stage 2, where only some of these reorderings may be allowed based on the partial order of events in a satisfiable clause i.e. one where not all red hb arrows are present (see Figure 3).

In this way, the post sequences ensure that:

1. all copies of the same variable agree on the assignment,
2. messages are posted in an order reflecting this assignment,
3. the clause gadgets can detect satisfiability by checking the consistency of the induced trace  $\tau$ .

We provide more details of the reduction and a formal proof of correctness in Appendix C.

## 4 Event-driven programs without nested posting

We have shown that the event-driven consistency problem is NP-hard, even when the number of handlers is bounded. However, a closer examination of our reductions reveals that they rely critically on the ability of a message to post another message to a handler—a feature we refer to as *nested posting*. This observation motivates the study of a restricted setting in which nested posting is disallowed: does the problem become easier in this case? We answer this question affirmatively in Theorem 3 (proof provided in Appendix D), which shows that the event-driven consistency problem becomes tractable under this restriction. Specifically, we present a polynomial-time procedure for checking consistency when the number of handlers is bounded and nested posting is not allowed.

**Theorem 3.** *Given a trace  $\tau = (E, \Delta')$  containing  $k$  handlers and no nesting of posts, the ED-Consistency problem for  $\tau$  can be solved in time polynomial in  $|E| = n$  and exponential in the number  $k$  of handlers.*

**Proof Sketch.** Since there is no nesting of posts, all post events occur in the initial message of each handler. The **po** order within the initial message therefore implies an **mo** order on all posts made by a given handler  $h$ . Due to queue semantics, this also implies the corresponding **eo** order on the corresponding **get** events and also the **eo**<sup>†</sup> relation between all pairs of events occurring in two messages ordered by **eo**. This implies that for the messages posted *to* a handler  $h'$ , we obtain  $k$  different total orders of based on the handler  $h$  making the post. We can now express the consistency problem in terms of program termination as follows. We define the notion of a configuration  $C$  which consists of

- (P1)  $k$  pointers for each handler  $h'$  which indicates the messages which have been executed so far in each of the  $k$  total orders (for a total of  $k^2$  pointers), and
- (P2)  $k$  additional pointers which indicate the event to be executed next in the message currently being executed in each handler.

This information requires  $O(k^2 \log(n))$  space and thus the total number of configurations is exponential in  $k$  and polynomial in  $n$ .

We can now construct the configuration graph  $G$  which consists of nodes representing the configurations and edges which indicate when a transition is possible (e.g. when the a pointer in P1 is moved to the successor event inside a message, all other pointers remaining the same). Thus, the trace is consistent iff there is a path from the initial configuration to the final configuration in  $G$ .

## 5 Consistency checking: Procedure and Optimisations

In this section, we propose two concrete procedures for checking event-driven consistency. Both the procedures take as input the trace as a graph  $G$  whose nodes are events and whose edges are the program relations **rf**, **co**, message relation **po**, and posted-by relation **pb**. The first procedure is based on some *saturation rules*, designed to accelerate consistency checking. The second procedure involves encoding the consistency problem as a constraint satisfaction problem and uses

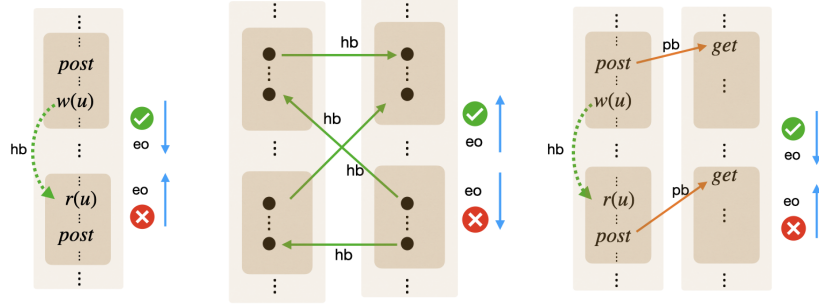


Fig. 5: Patterns corresponding to saturation rules.

the Z3 SMT solver to solve it. Observe that the ED-consistency problem is in NP. Since checking consistency is in polynomial time if all the relations are given, it suffices to guess the missing relations.

### 5.1 Procedure using Saturation Rules

The  $\text{CheckQconsistency}(G)$  procedure iterates over all possible assignments of  $\text{eo}$  and  $\text{mo}$  edges for  $G$ , and for each of these assignments, (1) adds the  $\text{fr}$ ,  $\text{qo}$  and  $\text{eo}^\dagger$  edges that are defined in Section 2.3, and checks whether there is one for which

#### Algorithm 1: Consistency checking

---

**Input:** A partial trace  $G = (E, \Delta)$  where  
 $\Delta \subseteq E \times \text{rels}' \times E$ .

```

1 if  $G$  contains a cycle then
2   return Inconsistent
3 else
4   Apply saturation rules (1), (2), and (3) to  $G$ ;
5   CheckQconsistency( $G$ );

```

---

$G$  is acyclic after the addition of these edges. If there is an assignment for which the graph  $G$  is acyclic, the procedure returns true, otherwise, it returns false.

**Saturation rules.** To reduce the search space and speed up the consistency check, we define the following saturation rules, depicted in Figure 5:

**Rule 1:** If there is a  $\text{hb}$  edge between two events in two different messages of the same handler, the  $\text{eo}$  edge between these messages have to respect this order.

**Rule 2:** Consider the sequence of  $\text{hb}$  edges between events as depicted in the second figure. It can be seen that this pattern forces the  $\text{eo}$  edge to be as shown in the figure - the inverse  $\text{eo}$  edge immediately creates a cycle in the trace.

**Rule 3:** This rule checks that the order in which messages are processed follows the queue semantics. Consider the sequence of  $\text{hb}$  edges between events as depicted in the third figure. It can be seen that this pattern forces the  $\text{eo}$  edge to be as shown in the figure - the inverse  $\text{eo}$  edge immediately creates a cycle in the trace.

### 5.2 Procedure using SMT encoding

Here, we present an algorithm to check consistency of a partial event-driven trace using the Z3 solver [41]. Note that each event  $e$  is part of one message, and each message is part of one handler. The algorithm uses the *Special Relations* theory in Z3.



The algorithm takes the input trace  $G$  and the set of events  $E$ . An enum datatype  $T$  is created, where each value corresponds to an event. Then, a partial order  $O$  is declared over the events in  $T$  using Z3's Special Relations theory - this order will represent the orderings that must be satisfied for the trace to be consistent. Then, a Z3 solver instance  $I$  is created to solve the logical constraints defined over the events and their orderings. Then, in lines 4-5, we enforce

that each known relation is a subrelation of the partial order  $O$ , meaning that  $O$  must respect all the orders that are already known from the event-driven semantics. Then, for each handler, and for every pair of messages  $m_1, m_2$  on this handler, we require  $(m_1.\text{last} <_O m_2.\text{first} \vee m_2.\text{last} <_O m_1.\text{first})$  where  $m.\text{first}$  and  $m.\text{last}$  denote the first (respectively last) instruction of a message  $m$ . This constraint enforces that the event order (eo) is total within each handler, i.e., the execution of messages is serial. Similarly, we let  $m.\text{post}$  denote the posting event of message  $m$ , and require  $(m_1.\text{post} <_O m_2.\text{post} \vee m_2.\text{post} <_O m_1.\text{post})$  which corresponds to the requirement of mo being total for the posts to a given handler. Finally, the Z3 solver is called to check if the given set of constraints are satisfiable. If the solver returns Yes, then there exists a global partial order  $O$  that extends the known relations and satisfies all handler-level ordering constraints and therefore, the trace is consistent. Otherwise, no such order exists, which implies that the trace is inconsistent.

---

**Algorithm 2:** Consistency checking

---

**Input:** A partial trace  $G = (E, \Delta)$  where  $\Delta \subseteq E \times \text{rels}' \times E$ .

```

1  $T \leftarrow \text{Enum}(E)$ ;
2  $O \leftarrow \text{PartialOrder}(T)$ ;
3  $I \leftarrow \text{Z3instance}$ ;
4 for  $(a, b) \in \Delta$  do
5    $I.\text{assert}(O(a, b))$ ;
6 for  $h \in \text{handlers}$  do
7   for  $m_1, m_2 \in h.\text{messages}$  do
8      $I.\text{assert}(O(m_1.\text{last}, m_2.\text{first}) \vee$ 
9        $O(m_2.\text{last}, m_1.\text{first}))$ ;
9      $I.\text{assert}(O(m_1.\text{post}, m_2.\text{post}) \vee$ 
10       $O(m_2.\text{post}, m_1.\text{post}))$ ;
10 return  $I.\text{check}()$ ;
```

---

## 6 Implementation and Experimental Evaluation

We have implemented a prototype tool for consistency checking of event-driven traces, based on the algorithms described in Section 5. The prototype verifies whether a given partial trace admits a consistent extension, and, when successful, produces a witness: a concrete assignment to the missing relations (i.e.,  $mo$  and  $eo$ ). From this witness, a valid execution can be reconstructed. All experiments were conducted on a machine running Debian 12.4 with an Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz and 192 GB of RAM. Selected results are presented in Table 1. Further results can be found in Appendix E.

**Experiment setting.** To evaluate our approach, we used two independent methods to generate event-driven traces:

*Synthetic Traces via NIDHUGG.* Our first method involves using the open-source model-checking tool NIDHUGG to generate traces from [32] and new synthetic programs<sup>4</sup>. While NIDHUGG supports an event-driven execution model, it inter-

<sup>4</sup> A detailed discussion of these benchmarks is given in Appendix E.

Benchmark	Algorithm 1							Algorithm 2		
	Max # E	Max # M	Max # H	# T	# Consistent traces	# T/O traces	Time in sec.	# Consistent traces	# T/O traces	Time in sec.
SampleApp	4776	13	5	1	0	1	-	1	0	2.9816
Tomdroid	4776	13	5	2	0	2	-	2	0	1.8621
Opensudoku	11292	14	5	1	0	1	-	1	0	22.0052
Sgtpuzzles	18406	18	5	2	0	2	-	2	0	34.6982
Remindme	6870	23	5	1	0	1	-	1	0	16.7987
Modelcheckingserver	4057	26	4	1	0	1	-	1	0	3.6556
Messenger	5034	26	7	2	0	2	-	2	0	5.4606
Music	4253	33	5	3	0	3	-	3	0	4.3871
Fbreader	6159	35	8	6	0	6	-	6	0	9.4964
K9Mail	5309	46	9	2	0	2	-	2	0	13.2162
Aarddict	1220	12	5	2	1	1	2.0932	2	0	0.0982
AdobeReader	15717	140	7	1	0	1	-	0	1	-
Facebook	5319	14	12	1	0	1	-	1	0	1.9775
Twitter	9889	30	12	1	0	1	-	1	0	18.7847
Browser	9762	34	15	6	0	6	-	3	3	46.9663
Flipkart	116945	61	15	1	0	1	-	0	1	-
Mytracks	3671	32	16	3	1	2	27.7057	3	0	2.2841

Table 1: Experimental results for benchmark programs collected from droidracer. The field # T denotes the number of traces. The traces can differ in size (events # E), messages # M, handlers # H), and the field contains the maximum of its traces. The field # Consistent traces denotes the number of these traces for which the implementation reports the existence of a satisfying execution. The field # T/O traces denotes the number of traces for which our tool timed out (with a timeout of 120s). For any remaining traces, the tool concludes inconsistency. The time fields represent the average runtime for the traces that did not time out. A value of - indicates that the corresponding algorithm timed out on every trace.

pretends asynchronous semantics using multisets rather than FIFO queues. As a result, some of the generated traces may be inconsistent under queue semantics. For each benchmark program, we randomly sampled traces from NIDHUGG’s output. These traces are guaranteed to satisfy multiset semantics, but may violate the stricter queue-based consistency.

*Android Traces via DROIDRACER.* Our second source of benchmarks is DROIDRACER [38], a tool for systematic exploration of Android application behaviors. DROIDRACER’s Trace Generator executes Android binaries on an emulator and exhaustively generates event sequences up to a bound  $k$  using depth-first search. We developed a custom parser to transform these sequences into partial traces suitable for our tool. These traces extracted from Android apps available at [37], originally used by Maiya et al. [38]. Static edges (e.g., program order, reads-from, and posted-by) are added during parsing, while the other relations are left unspecified. The tool then checks whether a consistent extension exists. Since Android’s semantics closely follow queue-based message handling, we expect all DROIDRACER traces to be consistent.

**Experimental Results** We compared the performance of our two algorithms described in Section 5 for event-driven consistency.

The results, shown in Table 1, clearly demonstrate the advantage of Algorithm 2 in both performance and scalability. While Algorithm 1 performs acceptably on small traces (e.g Aardict), it fails to scale to complex instances due to the

combinatorial explosion in possible execution orderings. In contrast, Algorithm 2 benefits from Z3’s efficient constraint-solving capabilities, enabling fast detection of consistency or inconsistency even in challenging benchmarks.

To summarise, our experiments indicate that SMT-based techniques can be effectively leveraged for consistency checking in event-driven programs. The integration of SMT solvers, which are already widely adopted in verification tools, provides a scalable and precise foundation for reasoning about partial traces.

## 7 Conclusion and Future Work

In this paper, we investigate the problem of ED consistency under the sequential consistency memory model. We propose axiomatic semantics for event-driven programs and show equivalence of axiomatic and operational semantics. Furthermore, we establish that checking event-driven consistency is NP-hard, even when the number of handler threads is bounded. Further, when there is no nested posting in the trace, we show that checking consistency can be done in polynomial time. Finally, we also implement our event-driven consistency checking in a prototype tool, and provide promising experimental results on standard event-driven examples.

In the future, we plan to extend this work to the setting of other memory models such as Release-Acquire, Total Store Ordering etc. We also plan to integrate our implementation in procedures for Dynamic partial order reduction for event-driven programs, race detection and predictive analysis. Finally, we also plan to use evaluate our implementation on a wider range of examples, for instance the traces generated from android applications.

## References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. *Acta Inf.* **54**(8), 789–818 (2017)
2. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: *POPL*. pp. 373–384. ACM (2014)
3. Abdulla, P.A., Atig, M.F., Bonneland, F.M., Das, S., Lang, M., Jonsson, B., Sagonas, K.: Tailoring stateless model checking for event-driven multi-threaded programs. In: *Automated Technology for Verification and Analysis - 21st International Symposium, ATVA 2023, Proceedings. Lecture Notes in Computer Science*, vol. 14216. Springer (2023)
4. Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. *Proc. ACM Program. Lang.* **3**(OOPSLA), 150:1–150:29 (2019)
5. Abdulla, P.A., Atig, M.F., Jonsson, B., Leonardsson, C.: Stateless model checking for POWER. In: *CAV (2). Lecture Notes in Computer Science*, vol. 9780, pp. 134–156. Springer (2016)
6. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. *PACMPL* **2**(OOPSLA), 135:1–135:29 (2018)
7. kernel-mode driver architecture., M.I.W.: [http://msdn.microsoft.com/en-us/library/windows/hardware/ff557560\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/hardware/ff557560(v=vs.85).aspx)
8. Bouajjani, A., Emmi, M., Enea, C., Ozkan, B.K., Tasiran, S.: Verifying robustness of event-driven asynchronous programs against concurrency. In: *ESOP. Lecture Notes in Computer Science*, vol. 10201, pp. 170–200. Springer (2017)
9. Bouajjani, A., Enea, C., Guerraoui, R., Hamza, J.: On verifying causal consistency. In: *POPL*. pp. 626–638. ACM (2017)
10. Castro-Perez, D., Hu, R., Jongmans, S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.* **3**(POPL), 29:1–29:30 (2019)
11. Chakraborty, S., Krishna, S.N., Mathur, U., Pavlogiannis, A.: How hard is weak-memory testing? *Proc. ACM Program. Lang.* **8**(POPL), 1978–2009 (2024)
12. Christakis, M., Gotovos, A., Sagonas, K.: Systematic testing for detecting concurrency errors in erlang programs. In: *ICST*. pp. 154–163. IEEE Computer Society (2013)
13. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.A.: State space reduction using partial order techniques. *Int. J. Softw. Tools Technol. Transf.* **2**(3), 279–287 (1999)
14. Cunningham, R., Kohler, E.: Making events less slippery with eel. In: *HotOS. USENIX Association* (2005)
15. Dabek, F., Zeldovich, N., Kaashoek, M.F., Mazières, D., Morris, R.: Event-driven programming for robust software. In: *ACM SIGOPS European Workshop*. pp. 186–189. ACM (2002)
16. Desai, A., Gupta, V., Jackson, E.K., Qadeer, S., Rajamani, S.K., Zufferey, D.: P: safe asynchronous event-driven programming. In: *PLDI*. pp. 321–332. ACM (2013)
17. Fischer, J., Majumdar, R., Millstein, T.D.: Tasks: language support for event-driven programming. In: *PEPM*. pp. 134–143. ACM (2007)
18. Ganty, P., Majumdar, R.: Analyzing real-time event-driven programs. In: *FORMATS. Lecture Notes in Computer Science*, vol. 5813, pp. 164–178. Springer (2009)
19. Gay, D., Levis, P., von Behren, J.R., Welsh, M., Brewer, E.A., Culler, D.E.: The nesc language: A holistic approach to networked embedded systems. In: *Cytron, R., Gupta, R. (eds.) PLDI*. pp. 1–11. ACM (2003)

20. central dispatch (GCD) reference., A.C.I.G.: [http://developer.apple.com/library / mac/#documentation/ performance/ reference/ gcd\\_libdispatch\\_ref/ reference/ reference.html](http://developer.apple.com/library/mac/#documentation/performance/reference/gcd_libdispatch_ref/reference.html)
21. Gibbons, P.B., Korach, E.: Testing shared memories. *SIAM J. Comput.* **26**(4), 1208–1244 (1997)
22. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Ph.D. thesis, University of Liège (1996). <https://doi.org/10.1007/3-540-60761-7>, <http://www.springer.com/gp/book/9783540607618>, also, volume 1032 of LNCS, Springer.
23. Godefroid, P.: Model checking for programming languages using verisoft. In: *POPL*. pp. 174–186. ACM Press (1997)
24. Godefroid, P.: Software model checking: The verisoft approach. *Formal Methods Syst. Des.* **26**(2), 77–101 (2005)
25. Godefroid, P., Hanmer, R.S., Jagadeesan, L.J.: Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using verisoft. In: *ISSTA*. pp. 124–133. ACM (1998)
26. Hill, J.L., Szewczyk, R., Woo, A., Hollar, S., Culler, D.E., Pister, K.S.J.: System architecture directions for networked sensors. In: *ASPLOS*. pp. 93–104. ACM Press (2000)
27. Jensen, C.S., Möller, A., Raychev, V., Dimitrov, D., Vechev, M.T.: Stateless model checking of event-driven applications. In: *OOPSLA*. pp. 57–73. ACM (2015)
28. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* **2**(POPL), 17:1–17:32 (2018)
29. Kokologiannakis, M., Marmanis, I., Gladstein, V., Vafeiadis, V.: Truly stateless, optimal dynamic partial order reduction. *Proc. ACM Program. Lang.* **6**(POPL), 1–28 (2022)
30. Kokologiannakis, M., Sagonas, K.: Stateless model checking of the linux kernel’s hierarchical read-copy-update (tree RCU). In: *SPIN*. pp. 172–181. ACM (2017)
31. Kokologiannakis, M., Vafeiadis, V.: GenMC: A model checker for weak memory models. In: *CAV* (1). *Lecture Notes in Computer Science*, vol. 12759, pp. 427–440. Springer (2021)
32. Kragl, B., Enea, C., Henzinger, T.A., Mutluergil, S.O., Qadeer, S.: Inductive sequentialization of asynchronous programs. In: *PLDI*. pp. 227–242. ACM (2020)
33. LIBASYNC: <http://pdos.csail.mit.edu/6.824-2004/async/>
34. LIBEVENT: [http://monkey.org/ provos/libevent/](http://monkey.org/provos/libevent/)
35. Lukkarinen, A., Malmi, L., Haaranen, L.: Event-driven programming in programming education: A mapping review. *ACM Trans. Comput. Educ.* **21**(1), 1:1–1:31 (2021)
36. Maiya, P., Gupta, R., Kanade, A., Majumdar, R.: Partial order reduction for event-driven multi-threaded programs. In: *TACAS*. *Lecture Notes in Computer Science*, vol. 9636, pp. 680–697. Springer (2016)
37. Maiya, P., Kanade, A., Majumdar, R.: Droidracer tested apps repository. [https://bitbucket.org/iiscseal/droidracer-related-files/src/master/ pldi-2014-tested-apps/](https://bitbucket.org/iiscseal/droidracer-related-files/src/master/pldi-2014-tested-apps/) (2014), artefact accompanying the PLDI 2014 paper "Race Detection for Android Applications"
38. Maiya, P., Kanade, A., Majumdar, R.: Race detection for android applications. In: *PLDI*. pp. 316–325. ACM (2014)
39. Mazières, D.: A toolkit for user-level file systems. In: Park, Y. (ed.) *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. pp.

- 261–274. USENIX (Jun 2001), <http://www.usenix.org/publications/library/proceedings/usenix01/mazieres.html>
40. Mednieks, Z., Dornin, L., Meike, G.B., Nakamura, M.: Programming Android. "O'Reilly Media, Inc." (2012)
  41. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
  42. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI. pp. 267–280. USENIX Association (2008)
  43. Norris, B., Demsky, B.: A practical approach for model checking C/C++11 code. *ACM Trans. Program. Lang. Syst.* **38**(3), 10:1–10:51 (2016)
  44. Peled, D.A.: All from one, one for all: on model checking using representatives. In: CAV. Lecture Notes in Computer Science, vol. 697, pp. 409–423. Springer (1993)
  45. Project, T.M.: <http://mace.ucsd.edu>
  46. Raychev, V., Vechev, M.T., Sridharan, M.: Effective race detection for event-driven programs. In: OOPSLA. pp. 151–166. ACM (2013)
  47. Tovey, C.A.: A simplified np-complete satisfiability problem. *Discret. Appl. Math.* **8**(1), 85–89 (1984)
  48. Tunç, H.C., Abdulla, P.A., Chakraborty, S., Krishna, S., Mathur, U., Pavlogiannis, A.: Optimal reads-from consistency checking for c11-style memory models. *Proc. ACM Program. Lang.* **7**(PLDI), 761–785 (2023)
  49. Valmari, A.: Stubborn sets for reduced state space generation. In: Applications and Theory of Petri Nets. Lecture Notes in Computer Science, vol. 483, pp. 491–515. Springer (1989)
  50. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: PLDI. pp. 250–259. ACM (2015)

## A Appendix for Section 2

A transition occurs on either the execution of an instruction or a **get** which corresponds to receiving a message. The rules dictating these transitions are shown in Fig. 6. We write  $\alpha \xrightarrow{a} \alpha'$  to denote a transition.

### Event-transitions

$$\begin{array}{c}
 \text{WRITE} \\
 \hline
 \text{inst}(\alpha.s_h.\text{line}) = l_i : x = a \wedge \alpha.s_h.\text{val}(a) = v \\
 \alpha \xrightarrow{\langle h, \text{write}, x, v \rangle} \alpha(\nu \leftarrow \nu(x \leftarrow v), s_h.\text{line} \leftarrow \text{succ}(s_h.\text{line})) \\
 \\
 \text{POST} \\
 \hline
 \alpha.s_h.\text{line} = l : \text{post}(h', m) \quad \alpha.s_{h'}.\beta \xrightarrow{\text{post}, (m, \text{newmid})} \beta' \quad \text{newmid} = (h, \alpha.s_h.mcount) \\
 \alpha \xrightarrow{\langle h, \text{post}, h', \text{newmid} \rangle} \alpha(s_{h'}.\beta \leftarrow \beta', s_h.mcount \leftarrow s_h.mcount + 1, s_h.\text{line} \leftarrow \text{succ}(s_h.\text{line})) \\
 \\
 \text{READ} \\
 \hline
 \text{inst}(\alpha.s_h.\text{line}) = l : a = x \\
 \alpha \xrightarrow{\langle h, \text{read}, x \rangle} \alpha(s_h.\text{val} \leftarrow s_h.\text{val}(a \leftarrow \nu(x)), s_h.\text{line} \leftarrow \text{succ}(s_h.\text{line})) \\
 \\
 \text{GET} \\
 \hline
 \alpha.s_h.\text{line} = l : \text{last} \quad \alpha.s_h.\beta_h \xrightarrow{\text{get}, (m, \text{mid})} \text{MB } \beta' \\
 \alpha \xrightarrow{\langle h, \text{get}, \text{mid} \rangle} \alpha(s_h.\beta \leftarrow \beta', s_h.\text{mid} \leftarrow \text{mid}, s_h.\text{line} \leftarrow m.\text{init})
 \end{array}$$

### Local-transitions

$$\begin{array}{c}
 \text{INTWRITE} \\
 \hline
 \text{inst}(\alpha.s_h.\text{line}) = l : a = \text{exp} \\
 \alpha \xrightarrow{\varepsilon} \alpha(s_h.\text{val}(a) \leftarrow \text{val}(\text{exp}), s_h.\text{line} \leftarrow \text{succ}(s_h.\text{line})) \\
 \\
 \text{IFCOND} \\
 \hline
 \text{inst}(\alpha.s_h.\text{line}) = l : \text{if } \text{cond} \text{ goto } l' \quad \alpha.s_h.\text{val}(\text{cond}) = \text{true} \\
 \alpha \xrightarrow{\varepsilon} \alpha(s_h.\text{line} \leftarrow l') \\
 \\
 \text{IFNOTCOND} \\
 \hline
 \text{inst}(\alpha.s_h.\text{line}) = l : \text{if } \text{cond} \text{ goto } l' \quad \alpha.s_h.\text{val}(\text{cond}) = \text{false} \\
 \alpha \xrightarrow{\varepsilon} \alpha(s_h.\text{line} \leftarrow \text{succ}(l)) \\
 \\
 \text{GOTO} \\
 \hline
 \text{inst}(\alpha.s_h.\text{line}) = l : \text{goto } l' \\
 \alpha \xrightarrow{\varepsilon} \alpha(s_h.\text{line} \leftarrow l')
 \end{array}$$

Fig. 6: Transition rules of programs

## B Equivalence of Operational and Axiomatic Semantics

We recall that we work with the set of relation  $\text{rels} = \{\text{rf}, \text{co}, \text{po}, \text{eo}, \text{pb}, \text{mo}\}$ . A trace is a directed graph  $\tau = (E, \Delta)$  where  $E$  is a finite set of events,  $\Delta \subseteq E \times \text{rels} \times E$  is a set of edges on  $E$  with labels from  $\text{rels}$ . A *partial trace*  $\tau' = (E', \Delta')$  is said to *extend* a trace  $\tau = (E, \Delta)$  if  $E \subseteq E', \Delta \subseteq \Delta'$ . A *linearisation*  $\pi = (E, \leq_\pi)$  of a trace  $\tau = (E, \Delta)$  is a total ordering  $\leq_\pi$  satisfying  $\delta \in \Delta \Rightarrow \delta \in \leq_\pi$ .

**Remark.** Note that we also sometimes have to deal with traces that are not well-formed, i.e., where not every **post** event has a corresponding **get** event. We will need these notions in the proofs below. However, the traces we consider as input for consistency checking will always be well-formed.

Further, given a program  $\mathcal{P}$  and its execution  $\rho$ , recall that the event set  $E_\rho$  along with the total order  $\leq_\rho$  derived from the run induces a trace  $\tau(\rho)$ .

**Theorem 4.** *A trace  $\tau$  is axiomatically consistent iff there exists an event-driven program  $\mathcal{P}$  and an execution  $\rho$  of  $\mathcal{P}$  such that  $\tau = \tau(\rho)$ .*

*Proof.* For the forward direction, suppose that a trace  $\tau$  is axiomatically consistent. We need to show that there exists a program  $\mathcal{P}$  such that it has a run  $\rho$  that satisfies  $\tau(\rho) = \tau$ .

Let  $\tau = (E, \Delta)$ , where  $E$  is the set of events and  $\Delta$  is the set of edges. Since the trace  $\tau$  is axiomatically consistent, we know that  $\Delta$  is acyclic. Consider a linearisation  $\sigma$  of the trace  $\tau$  of the form  $e_1 \cdot e_2 \cdot e_3 \cdots e_n$ . Let  $\sigma_i$  denote the prefix of  $\sigma$  containing the first  $i$  events and let  $\tau_i$  denote the projection of  $\tau$  to the first  $i$  events of  $\sigma$  i.e.,  $\tau_i = \tau \downarrow_{E_i}$ .

We will produce a program  $\mathcal{P}$  and a run  $\rho$  of  $\mathcal{P}$  such that  $\tau(\rho) = \tau$ , where

$$\rho := C_0 \xrightarrow{\bar{e}_1} C_1 \xrightarrow{\bar{e}_2} C_2 \xrightarrow{\bar{e}_3} C_3 \cdots$$

where  $\bar{e}_i$ 's are event transitions. To show that  $\tau(\rho) = \tau$ , we need to show that

- The set of events of  $\tau(\rho)$  is precisely the set  $E$  of events of  $\tau$ . For this, it suffices to show that  $\bar{e}_i = e_i$  for all  $1 \leq i \leq n$ .
- The set of relations in  $\tau(\rho)$  are in agreement with the the relations in  $\tau$ , i.e.,

$$e_1 R e_2 \iff \bar{e}_1 R \bar{e}_2$$

where  $R$  is a relation in  $\text{rels}$ .

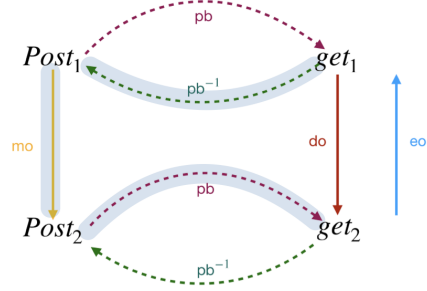


Fig. 7: Depiction of **qo** edge for the mailbox of a handler. Here, the **eo** edges that violates the queue semantics is depicted in blue, and the violating cycle caused by this violation may be observed.



- Next, we will produce the program  $\mathcal{P}$  by defining its building blocks as follows.
- The set of handlers  $H$  is given by the set  $\{h \mid \langle h, post, h' \rangle \text{ is an event in } E\}$ . In other words, the set of handlers are determined by the set of post events in  $\tau$ , as each message uniquely identifies its handler.
  - The set of variables are given by the set of variables  $x$  such that either  $\langle h, write, x, v \rangle$  or  $\langle h, read, x \rangle$  is an event in  $\tau$ .
  - The set of registers is given by  $\{a_h \mid h \in H\}$ , i.e., each handler has a register. Note that we need just one register per handler, as all the reads can be read to this register.
  - The set of messages of the handler is given the set of **get** events of a handler. For a handler  $h$ , consider the projection of  $\tau$  to the events of  $h$ . Then, the set of events reachable by a sequence of **po** edges from a **get** event constitute the instructions of the message  $m$ 
    - For a read event of the form  $\langle h, read, x \rangle$ , we add the instruction  $a_h = x$  to the current message of handler  $h$ .
    - For a write event of the form  $\langle h, write, x, v \rangle$ , we add the instruction  $x = v$  to the current message of handler  $h$ .
    - For a post event of the form  $\langle h, post, h' \rangle$ , we instantiate a post instruction of the message  $\text{post}(h, m)$  in the current active message of the handler  $h$ .

Eventually, this **po** path will lead to an *last* event, which indicates the end of the message, at which point, we stop.
  - Note that an exception to this is the initial messages in each handler, which do not start with a **get** event. Consequently, any trace  $\tau$  has  $n$  minimal events which do not have any incoming edges, one for each handler. To address this, for each handler, we construct the initial message of the handler with the minimal event in the trace  $\tau$  corresponding to that handler, and keep adding the set of events reachable by a sequence of **po** edges from this event. As mentioned above, the end of these messages will be indicated by an *last* event.

Now that we have constructed the program  $\mathcal{P}$ , we will show that it has a run  $\rho$  that satisfies  $\tau(\rho) = \tau$ , i.e., the event-driven program relations in  $\tau(\rho)$  agree with the respective relations imposed by  $\tau$ . We will show this inductively. Recall that  $\tau_i$  denotes the projection of  $\tau$  to the first  $i$  events of  $\sigma$ . We will denote by  $\rho_i$  the prefix of  $\rho$  containing the first  $i$  events. Then, we will show that

$$\tau(\rho_i) = \tau_i$$

Note that  $\tau_i$  is not a well-formed trace. The proof follows by induction on the number of events in  $\tau$ .

For the base case, consider the empty trace  $\tau_0$ . This corresponds to the initial configuration  $C_0$  of the program, where the mailboxes are empty and all the variables and registers have the initial value, and all the mailboxes are empty.

Let  $C_i$  be the configuration reached by the execution  $\rho_i$ , i.e., by executing the sequence of events  $e_1 \cdot e_2 \cdot e_3 \cdots e_i$ . Note that in  $C_i$ , all the shared variables and registers will have the value written by the latest write instruction involving

that variable (respectively register). Further, let  $C_i \xrightarrow{e_{i+1}} C_{i+1}$ . We will show that  $\tau(\rho_{i+1}) = \tau_{i+1}$ .

We will consider four cases, depending on the type of the event  $e_{i+1}$ .

- **Read event.** Suppose  $e_{i+1} = \langle h, read, x \rangle$ . Due to sequential consistency, the value read by  $a_h$  will be the value written by the last write event on  $x$  in the sequence  $\rho_i$ , say  $e_k$  where  $k < i$ , introducing an **rf** edge from  $e_k$  to  $e_{i+1}$  in  $\tau(\rho_{i+1})$ . Since  $\sigma$  is a linearisation of  $\tau$ , the **rf** edge to  $e_{i+1}$  in  $\tau$  should also agree with this.
- **Write event.** Suppose  $e_{i+1} = \langle h, write, x, v \rangle$ . This will introduce **co** edges from all the write events on  $x$  in  $\rho_i$  to  $e_{i+1}$  in  $\tau(\rho_{i+1})$ . Once again, since  $\sigma$  is a linearisation of  $\tau$ , these **co** edges are consistent with the **co** edges in  $\tau_{i+1}$ . Additionally, these two events also induce **po** edges from the preceding event  $e_i$  of  $\rho_i$ , whose consistency also follows by the same argument.
- **Post event.** Suppose  $e_{i+1} = \langle h, post, h' \rangle$ . This will introduce **mo** edges to  $e_{i+1}$  from all the post events in  $\rho_i$  that have posted to the same handler. The consistency of these edges follows from the fact that  $\sigma$  is a linearization of  $\tau$ .
- **Get event.** Suppose  $e_{i+1} = \langle h, get \rangle$ . This will introduce
  - a **pb** edge from the post event  $\langle h', post, h \rangle$  corresponding to this message.
  - **eo** edges from the get events  $\langle h, get \rangle$  seen before in the sequence  $\rho_i$ .

Thus, for all the possibilities of  $e_{i+1}$ , we have shown that  $\tau(\rho_{i+1}) = \tau_{i+1}$ .

For the reverse direction, suppose that a program  $\mathcal{P}$  has an execution  $\rho$  that induces the trace  $\tau(\rho)$ . We need to show that  $\tau(\rho)$  is axiomatically consistent. From the definition of the trace of a program, we know that  $\tau(\rho)$  is of the form  $\tau = (E, \leq)$ , where

- $E = E_\rho$  is the set of events in  $\rho$  and
- $\leq$  is a total order on  $E_\rho$  defined as  $e_i \leq_\rho e_j$  iff  $i \leq j$ , where  $e_i, e_j \in E$ .

From Definition 1, we know that  $\tau$  is said to be axiomatically **MB**-consistent if the relation  $(\mathbf{po} \cup \mathbf{rf} \cup \mathbf{fr} \cup \mathbf{co} \cup \mathbf{pb} \cup \mathbf{mo} \cup \mathbf{eo}^\dagger \cup \mathbf{qo})$  is acyclic.

It is easy to see that no cycles are created by **po**, **rf**, **co**, **pb**, **mo** and **eo** edges, as this follows from the equivalence of axiomatic and operational models for sequentially consistent programs. Further, since **fr** is derived from **rf** and **co** edges, addition of **fr** edges do not create cycles.

It remains to show that adding the **eo**<sup>†</sup> and **qo** edges also do not create cycles.

- **eo**<sup>†</sup> edges: Note that **eo**<sup>†</sup> edges only exists between events of a message  $m$  and a message  $m'$  of the same handler, such that there is an **eo** edge between the **get** event of  $m$  and **get** event of  $m'$ . Thus, they only add edges between events of a handler. Since all the events of a handler are totally ordered, and the **eo**<sup>†</sup> respects this total order (as it is inherited from **eo**), these edges respect the order given by  $\rho$ .
- **qo** edges: Before discussing the **qo** edges that are induced, we recall that the order of insertion and deletion of messages into the mailbox are governed by the **mo** and **eo** edges. Recall that the **mo** relation orders all the post events involving messages posted to the same handler, and the **eo** relation orders all the get events of the messages of a handler.

Further, recall that  $qo = pb^{-1}.mo.pb$ . Since the  $mo$  relation orders events that post messages to the mailbox of a handler, it is easy to see that the new  $qo$  edges that are induced between  $get$  events of the messages posted to the mailbox of the same handler. It suffices to show that the new edges introduced agree with the  $eo$  edges between these  $get$  events.

Note that the operations of each handler in the run  $\rho$  follows the queue semantics, i.e., the  $get$  instructions should be processed in first-in-first-out manner. Further, the relation  $qo = pb^{-1}.mo.pb$  orders the  $get$  events of the messages posted to the mailbox of any given handler in the order in which  $post$  events are ordered in  $\tau$ . Therefore, if any two  $get$  events, say  $get_1$  and  $get_2$ , violate the queue semantics, then the  $eo$  edge between these events will cause a cycle - there will be a cycle of the form  $get_1 qo get_2 eo get_1$ , as depicted in Figure 7.

Formally, suppose there is a cycle involving  $qo$  edges. Then, there is a  $qo$  cycle involving two  $get$  events, say  $get_1$  and  $get_2$ . The cycle is of the form  $get_1 qo get_2 hb get_1$ . However, the existence of the  $qo$  edge from  $get_1$  to  $get_2$  implies that the message corresponding to  $get_1$  was added to the queue before the message corresponding to  $get_2$ . Further, the  $hb$  edge from  $get_2$  to  $get_1$  implies that  $get_2$  is processed before  $get_1$ , which means that the execution of messages corresponding to  $get_1$  and  $get_2$  violate the queue semantics. This is a contradiction to the assumption that  $\sigma$  was a consistent execution.

## C Formal Proof of NP-Completeness of ED-Consistency

In this section, we provide a formal proof of Theorem 2, which states that the ED-consistency problem is NP-complete even when the number of handlers is bounded.

### Formal Construction

The proof is done by reduction from 3-BI-3SAT. Let  $\phi$  be a 3-BI-3SAT instance with variables  $x_1, x_2, \dots, x_n$  and clauses  $C_1, C_2, \dots, C_m$ . We will construct a partial ED trace  $\tau = (E, \Delta)$ , with  $\Delta \subseteq E \times rels' \times E$ , such that  $\tau$  can be extended to a axiomatically consistent trace  $\tau' = (E, \Delta')$ , with  $\Delta' \cap (E \times rels' \times E) = \Delta \cap (E \times rels' \times E)$  iff  $\phi$  is satisfiable.

**High level structure.** The construction of the trace  $\tau$  is divided into two stages, which we call Stage 1 and Stage 2 respectively. There are 8 handlers in Stage 1 and 5 handlers in Stage 2. One handler, namely  $h_W$  is common to both stages, hence totally there are 12 handlers. If a satisfying assignment exists for  $\phi$ , then there is a program which can execute the events in Stage 1 followed by those in Stage 2, i.e.,  $\tau$  is consistent. If  $\phi$  is unsatisfiable then there is no witnessing execution possible which executes both stages and  $\tau$  is inconsistent.

Stage 1 corresponds to the selection of a satisfying assignment  $f$  for  $\phi$ . We can encode the information of whether a variable  $x_i$  is assigned true or false using the

order of execution of two messages  $m_{i,1}$  and  $m_{i,0}$  on the same handler, where  $x_i$  is assigned true (resp. false) if  $m_{i,1}$  (resp.  $m_{i,0}$ ) is executed later. Unfortunately, this will not work due to technical difficulties faced in clause verification which we explain when describing Stage 2. This necessitates our extremely technical reduction which makes use of the structure of the 3-BI-3SAT instance where each variable occurs in *at most* three clauses and the variables occurring in a clause are all different. We have to create (at most) 3 copies of the messages, one for each clause in which  $x_i$  occurs and find a way to synchronise the assignment between these three copies.

Hence the messages for  $x_i$  are actually of the form  $m_{i,j,b}$  where  $j$  refers to clause  $C_j$  and  $b \in \{0, 1\}$ . Using the technique of *nested postings* (explained later in full construction), we post the set  $M$  of  $m_{i,j,b}$  messages in the queue of  $h_W$  in some order  $\sigma$ . The remaining 7 handlers of Stage 1 are used to shuffle the messages with certain restrictions on the order  $\sigma$  of messages.

The set  $S$  of all the possible orders  $\sigma$  is such that, every  $\sigma$  is constrained to be *consistent* (we explain later in full construction) with some particular assignment  $f$  of variables of  $\phi$ . There are no other constraints on the order  $\sigma$ . At the end of Stage 1, the queues of all other Stage 1 handlers is empty and the queue of  $h_W$  is populated in some order  $\sigma \in S$  consistent with some assignment  $f$ . Note that there are multiple  $\sigma$  which are consistent with a particular  $f$ , this fact will be important later.

Stage 2 Let us fix  $\sigma$  and  $f$  from Stage 1. Then Stage 2 verifies that  $f$  indeed satisfies all the clauses of  $\phi$ . For this, we build a clause gadget  $G_j$  corresponding to each clause  $C_j$ . The set  $E_G$  of events of these clause gadgets occupy the 4 non- $h_W$  handlers of Stage 2. The  $E_G$  events belong to an initial message of each of the 4 handlers, and consist purely of read and write events. Recall that the queue of  $h_W$  is populated at this point with messages  $M$ . The information regarding the assignment is encoded in the order of the messages in  $h_W$ . This information is transferred to the other 4 non- $h_W$  handlers via a technique we call *sandwiching* (explained in the full construction). There are now two possibilities:

- (1) If  $f$  is not a satisfying assignment, then some clause  $C_j$  is not satisfied by  $f$ . In this case, any order  $\sigma$  of messages consistent with  $f$  will induce a **hb** (happens before) cycle in the corresponding gadget  $G_j$  via the sandwiching. Therefore Stage 2 cannot be executed by any witnessing execution. If there are no satisfying assignments, then  $\phi$  is unsatisfiable and hence  $\tau$  is not consistent.
- (2) If  $f$  is a satisfying assignment then there is some order  $\sigma$  of the messages in  $M$  which is consistent with  $f$  such that there is a witnessing execution. The clause gadgets are executed interleaved with the messages in  $M$  due to the sandwiching. The execution happens sequentially i.e.  $G_1$  is executed, then  $G_2$  etc. This implies  $\tau$  is consistent.

### Full Construction

For now we assume that at the end of Stage 1, all of the messages in  $M$  have been posted to  $h_W$  in some order  $\sigma$  consistent with some assignment  $f$  to the variables of  $\phi$ . We describe how we can check the satisfaction of clauses in Stage 2 before describing Stage 1.

**Stage 2: Clause checking.** Stage 2 events occur in the 5 handlers  $h_{C_a}, h_{C_b}, h_{C_c}, h_{C_d}, h_W$ . For each clause  $C_j$  we create a clause gadget  $G_j$  which consists of 14 events in handlers  $h_{C_a}, h_{C_b}, h_{C_c}, h_{C_d}$ . Pick a clause, say  $C_2 = x_1 \vee x_2 \vee \overline{x_n}$ . Figure 8 shows the clause gadget  $G_2$  for clause  $C_2$  together with two of the messages  $m_{i,j,b}$  in  $h_W$  which are posted by Stage 1. For reasons of space we use  $W$  and  $R$  for *write* and *read* in the description of events. We will use program variables of two forms: (F1)  $l_{i,k}^j$  and  $\overline{l_{i,k}^j}$ , and (F2)  $z_k$ . The F2 variables do not correspond in any way to the formula, while the F1 variables do.

Clause satisfaction. First let us focus on the dotted boxes  $b_1$  and  $b_2$  in the figure. Each box consists of a read followed in **po** by a write event. Focusing on  $b_1$ , the program variable  $\overline{l_{n,1}^2}$  in  $e_{10}$  has the information: superscript 2 for clause  $C_2$ , first subscript  $n$  indicating the literal  $\overline{x_n}$  and second subscript 1 indicating it is the first event in the box. Note that since we have made (up to) 3 different copies of each variable in our reduction of regular 3SAT to 3-BI-3SAT, the variables  $l_{i,k}^j$  and  $\overline{l_{i,k}^j}$  occur exactly once. The events in  $b_1$  are linked to the read and write events in the message  $m_{n,2,0}$  via **rf** arrows. The direction of the arrows implies that the events in box  $b_1$  have to be executed after event  $e_{15}$  and before  $e_{16}$  which are both in message  $m_{n,2,0}$ . This is the technique we call *sandwiching*. Similarly  $b_2$  has to be executed during the execution of  $m_{n,2,1}$ . Suppose  $m_{n,2,0}$  is executed before  $m_{n,2,1}$  as indicated by the **eo**, this means that  $x_n$  is assigned the value **true**. The sandwiching induces the red **hb** relation shown between  $e_{11}$  and  $e_{12}$ , *copying* the value of the variable from handler  $h_W$  to the clause gadget by ensuring that the read and write events of the  $\overline{l_{n,1}^2}$  and  $l_{n,2}^2$  program variables occur before the events on the  $l_{n,1}^2$  and  $l_{n,2}^2$  variables, which again reflects that the  $x_n$  has been set to true in  $C_2$ .

Notice that similar boxes can be drawn around events  $e_2, e_3$  and  $e_4, e_5$  corresponding to copying the assignment to variable  $x_1$  and for  $e_6, e_7$  and  $e_8, e_9$  for variable  $x_2$ . Further note that the F1 program variables occur in a certain order when we traverse  $h_{C_a}, h_{C_b}, h_{C_c}, h_{C_d}$  from top to down and from  $a$  to  $d$ . Let us skip the second subscript (which is just used to denote two copies) and see the order in this example:  $l_1^2, \overline{l_1^2}, l_2^2, \overline{l_2^2}, \overline{l_n^2}, l_n^2$ . The fact that  $l_1^2$  occurs before  $\overline{l_1^2}$  indicates that  $x_1$  is present in positive form (as also  $l_2^2$  before  $\overline{l_2^2}$  representing the occurrence of  $x_2$  in positive form). Whereas  $\overline{l_n^2}$  before  $l_n^2$  program variables indicates that  $\overline{x_n}$  is present in  $C_2$ . In this way, the clause gadget captures the structure of the clause.

Each of the other boxes around events  $e_2, e_3$  and  $e_4, e_5$  has similar sandwiching **rf** relations to messages  $m_{i,j,b}$  in  $h_W$  which are not shown in the figure. The three red **hb** arrows correspond to setting each of the three variables in  $C_2$  to a value that falsifies the corresponding literal in  $C_2$ . The events  $e_1$  and  $e_{14}$  use a variable  $z_2$  (where the subscript refers to the clause  $C_2$ ) and are connected by an **rf**. Thus if  $m_{1,2,1}$  **eo**  $m_{1,2,0}$ ,  $m_{2,2,1}$  **eo**  $m_{2,2,0}$  and  $m_{n,2,0}$  **eo**  $m_{n,2,1}$  all hold, a cycle is formed and the clause gadget cannot be executed. On the other hand, if even one of the red arrows is flipped (indicating that a literal of  $C_2$  is set to **true**), then the arrows form a partial order allowing execution of the clause gadget  $G_2$ . Lastly, note that a variable e.g.  $x_2$  in  $C_2$  also occurs in  $C_1$  and  $C_3$ . In Stage 1 we

explain how we can select an assignment for  $x_2$  in a consistent way for all three clauses  $C_1, C_2, C_3$ .

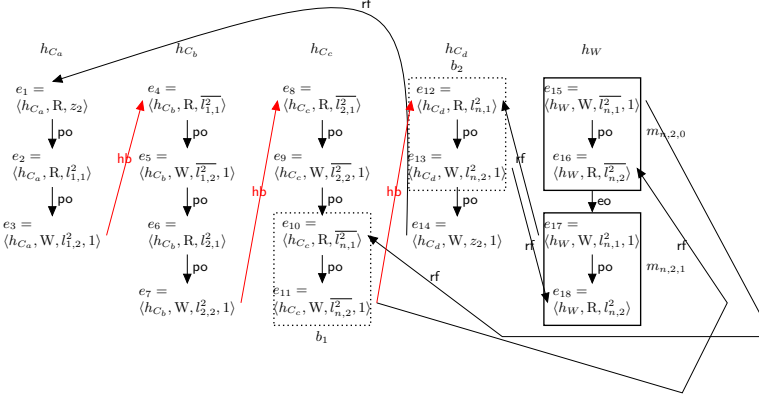


Fig. 8: The structure of clause gadget  $G_2$  of  $C_2$  for the example in Figure 9.

The complete set of events in the handlers  $h_{C_a}, h_{C_b}, h_{C_c}, h_{C_d}$  is made up of the union of the events in the clause gadgets  $G_j$ . The clause gadgets  $G_1, G_1, \dots, G_m$  are placed in that order in the handlers  $h_{C_a}, h_{C_b}, h_{C_c}, h_{C_d}$  and connected by  $po$  arrows. For example,  $G_j.e_3$  will be  $po$  before  $G_{j+1}.e_1$  in  $h_{C_a}$ ,  $G_j.e_7$  will be  $po$  before  $G_{j+1}.e_4$  in  $h_{C_b}$ , etc. In other words, the events of each of these four handlers can be assumed to be in an initial message in the respective handlers. There is no posting of events either from or to these 4 handlers.

**Stage 1: Variable assignment.** In this Stage, we use the handlers  $h_V, h_{t_1}, h_{t_2}, h_{t_3}, h_{t_4}, h_{t_5}, h_{t_6}$  in order to post messages to  $h_W$ . We stated that an assignment to variable  $x_i$  in clause  $C_j$  can be encoded as the order between two messages  $m_{i,j,0}, m_{i,j,1}$ .

**Challenge 1:** How can we ensure that two messages  $m_{i,j,0}, m_{i,j,1}$  can be posted in any order to  $h_W$ ?

In order to solve this, we use *nested posting*.  $h_V$  can post messages  $m'_{i,j,0}$  to  $h_{t_1}$  and  $m'_{i,j,1}$  to  $h_{t_2}$ . Then  $m'_{i,j,0}$  (resp.  $m'_{i,j,1}$ ) posts  $m_{i,j,0}$  (resp.  $m_{i,j,1}$ ) to  $h_W$ . Since  $m'_{i,j,0}$  and  $m'_{i,j,1}$  are on different handlers, they can be executed in any order, thus ensuring that  $m_{i,j,0}, m_{i,j,1}$  can be posted in any order to  $h_W$ . Next we take up the reason for using multiple messages for each variable.

Consider the sandwiching technique that we presented in Stage 2 in order to copy the assignment of a variable to the clause gadget. Suppose we were to use a single pair of messages  $m_{i,0}, m_{i,1}$  in  $h_W$  for a variable  $x_i$  from which this value was copied to the different clauses in which  $x_i$  occurs. This means that any handler in which a clause gadget is being executed would be blocked from running till all of the clauses containing  $x_i$  are able to finish executing the boxes corresponding to  $x_i$ . This leads to a cascading set of blocked handlers, requiring an unbounded number of handlers to execute the clause gadgets. In order to overcome this difficulty, we have to use three copies of the two messages

as mentioned before. But this leads to a different challenge:

Challenge 2: How can we ensure that the different copies of the messages corresponding to the same variable encode the same value?

To address this, we further extend the nesting of posts. In order to understand how this is done, we have to look into the structure of the formula  $\phi$ . Figure 9 represents the occurrence of the  $n$  variables in the  $m$  clauses (along with which literal occurs by use of a bar). Note that each row contains 3 marked cells and each column contains 4 or 6 marked cells as per the restriction on 3-BI-3SAT. A *post sequence* is a partial trace of the form

$$\langle h_0, \text{post}, h_1 \rangle \xrightarrow{\text{pb}} \langle h_1, \text{get} \rangle \xrightarrow{\text{po}} \langle h_1, \text{post}, h_2 \rangle \xrightarrow{\text{pb}} \dots \xrightarrow{\text{po}} \langle h_{n-1}, \text{post}, h_n \rangle$$

	$C_1$	$C_2$	$C_3$	$C_4$	$\dots$	$C_8$	$\dots$	$C_m$
$x_1$		(1, 1)		(1, 2)	$\dots$	(1, 3)	$\dots$	
$\bar{x}_1$		(1, 1)		(1, 2)	$\dots$	(1, 3)	$\dots$	
$x_2$	(1, 1)	(2, 2)	(3, 1)		$\dots$			
$\bar{x}_2$	(1, 1)	(2, 2)	(3, 1)		$\dots$			
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$x_n$		(1, 3)			$\dots$			
$\bar{x}_n$		(1, 3)			$\dots$			

Fig. 9: Relationship between variables and clauses dictating the nesting of posts. Empty cell means variable does not occur in clause(not all nonempty cells are shown in figure). A cell marked  $(u, v)$  or  $(\bar{u}, v)$  indicates that it is the  $u$ -th occurrence of the variable in a clause and is the  $v$ -th variable of the clause. The bars on the tuple indicate the polarity of the variable occurrence.  $C_2 = x_1 \vee x_2 \vee \bar{x}_n$ ,  $x_1$  occurs in  $C_2, C_8$  and  $\bar{x}_1$  occurs in  $C_4$ .

We will simply write this as  $p = \langle h_1, \text{post}, h_2, \text{post}, \dots, \text{post}, h_n \rangle$ . In case  $h_i = h_{i+1} = \dots = h_j$  we will further shorten this to  $\langle h_1, \text{post}, h_2, \text{post}, \dots, h_{i-1}, \text{post}^{j-i}, h_j, \text{post}, h_{j+1}, \text{post}, \dots, \text{post}, h_n \rangle$ .

Before explaining how we use this in our construction, let us look at a simple (and unrelated to the construction) example which shows what nested posting can achieve in Figure 10. The figure shows the run of a program with three handlers  $h_1, h_2, h_3$ . Initially, in configuration  $c_1$  (assume that some other handler has made these posts to  $h_1$ ), the queue of  $h_1$  contains 3 messages while the other two handlers have empty queues. For space reasons, the messages are written in short. The string  $p_2(p_3(p_1(m_1)))$  is short for a message containing a single instruction  $\text{post}(h_2, m'_1)$ , which is at the head of the  $h_1$  queue. Note that first  $p_2$  indicates that the post instruction posts the message  $m'_1$  to  $h_2$ . Here  $m'_1$  is itself a message with a single instruction  $\text{post}(h_3, m''_1)$  where  $m''_1 = \text{post}(h_1, m_1)$  for the message  $m_1$ . Our goal is to show how the ‘inner’ messages  $m_1, m_2$  and  $m_3$  can be put into the queue of  $h_1$  in certain orders but not in certain other orders.

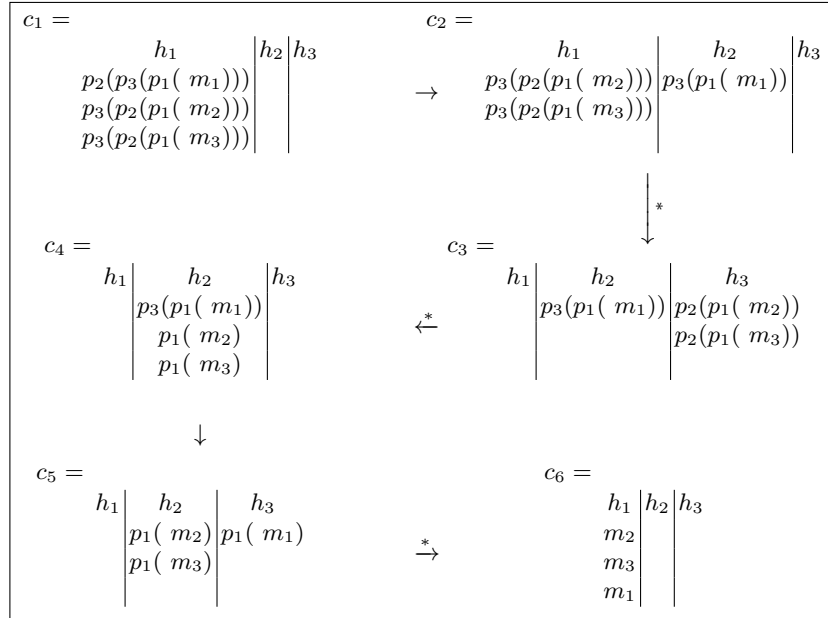


Fig. 10: Sorting messages via nested posts.

When  $p_2(p_3(p_1(m_1)))$  is dequeued and executed, then  $m'_1$  is posted and shows up in configuration  $c_2$  as  $p_3(p_1(m_1))$  in the queue of  $h_2$ . The other two messages remain in the queue of  $h_1$  with the head of the queue now being  $p_3(p_2(p_1(m_2)))$ . The rightarrow indicates that it is a one step transition from  $c_1$  to  $c_2$ . The down arrow with a  $*$  indicates that in multiple steps we go to  $c_3$ , the posting of the two messages in  $h_1$  one by one to  $h_3$ . Focus on the inner most messages  $m_2$  and  $m_3$ . Note that they are now ‘present’ in  $h_3$  (wrapped up in posts) in the same order as they were originally in  $h_1$ . Since the post sequence  $p_2, p_3, p_1$  is the same for  $m_2$  and  $m_3$ , they will always pass through different handlers in the same order as they were originally present in  $h_1$ . However, the post sequences are different for  $h_1$ . Skipping ahead to  $c_5$ , we see that  $m_1$  is in  $h_3$  while  $m_2, m_3$  are in  $h_2$ . At this point, we have shown a sequence of transitions where  $m_2$  and  $m_3$  are posted first to  $h_1$  followed by  $m_1$ . This results in the inner message order of  $m_2, m_3, m_1$  in the queue of  $h_1$ . However, at configuration  $c_5$ , by executing  $p_1(m_2)$ , then  $p_1(m_1)$  and then  $p_1(m_3)$  we can instead obtain a configuration  $c_7$  which results in  $h_1$  being populated in the order  $m_2, m_1, m_3$ . To summarise,  $m_1$  can be shuffled with  $m_2$  and  $m_3$  in all possible ways, but  $m_2$  must always be before  $m_3$ . In particular, this means we can never get  $m_3, m_2, m_1$  in  $h_1$ . While this explanation has been with respect to program execution, the same logic can be lifted to traces.

Now let us see how the idea is used in our construction. For each row of the grid labelled by a literal  $l$ , we create a post sequence  $p^l$ . Consider the row labelled by  $\bar{x}_1$  in the Figure 9. The post sequence is the concatenation of 7 post sequences



$p^{\overline{x_1}} = p_1^{\overline{x_1}} p_2^{\overline{x_1}} p_3^{\overline{x_1}} p_4^{\overline{x_1}} p_5^{\overline{x_1}} p_6^{\overline{x_1}} p_7^{\overline{x_1}}$  where  $p_2^{\overline{x_1}}, p_4^{\overline{x_1}}, p_6^{\overline{x_1}}$  correspond to the cells marked  $(1, 1), (1, 2)$  and  $(1, 3)$  respectively, while the others correspond to the part of the row consisting of unmarked cells, with  $p_1^{\overline{x_1}}$  for the part from the beginning till the first marked cell, etc. Note that in concatenation of post sequences, we have to add a get event in between appropriately. Each of the post sequences  $p_1^{\overline{x_1}}, p_3^{\overline{x_1}}, p_5^{\overline{x_1}}, p_7^{\overline{x_1}}$  consists of a long sequence of posts to  $h_V$  of length the number of unmarked cells in the segment they correspond to. For example  $p_1^{\overline{x_1}} = p_3^{\overline{x_1}} = \langle h_V, \text{post}, h_V \rangle$  while  $p_5^{\overline{x_1}} = \langle h_V, \text{post}^3, h_V \rangle$  since there are 3 empty cells in between (in the figure they are not explicitly shown, but rather by  $\dots$ , but once can infer that the boxes corresponding to  $C_5, C_6, C_7$  are empty along this row). The overall idea is that the post sequences are executed column by column. The post sequences of the empty cells simply ‘send to the back of queue’ of  $h_V$  while the marked cells are responsible for shuffling the messages in the  $h_{t_k}$  handlers which populate  $h_W$  with an appropriate sequence of messages as we explain below.

We now describe the post sequences made in the marked cells. Consider the 6 marked cells corresponding to column  $C_2$ . Top to bottom, these are  $p_2^{x_1}, p_2^{\overline{x_1}}, p_4^{x_2}, p_4^{\overline{x_2}}, p_2^{x_n}, p_2^{\overline{x_n}}$ . Let us consider the post sequence for a cell labelled  $(u, v)$  (resp.  $(\overline{u}, \overline{v})$ ), indicating that it is the  $u$ -th occurrence of the variable in a clause and is the  $v$ -th variable of the clause, with the bar indicating whether the variable or its negation occurs in the clause. Suppose  $u \neq 1$ , then the post sequence is  $\langle h_V, \text{post}, h_{t_v}, \text{post}, h_V \rangle$  for both  $(u, v)$  as well as  $(\overline{u}, \overline{v})$ . If  $u = 1$  then the post sequence is  $\langle h_V, \text{post}^2, h_{t_v}, \text{post}, h_V \rangle$  for  $(u, v)$  but it is  $\langle h_V, \text{post}, h_{t_{v+3}}, \text{post}, h_{t_v}, \text{post}, h_V \rangle$  for  $(\overline{u}, \overline{v})$ . For example,  $p_2^{x_n}$  which is marked  $(1, 3)$  has the post sequence  $\langle h_V, \text{post}, h_{t_6}, \text{post}, h_{t_3}, \text{post}, h_V \rangle$ . Intuitively, the post sequences of the variable and its negation move to different handlers  $h_{t_k}$  before coming back to the same handler iff a variable is occurring for the first time in a clause i.e., if  $u = 1$ .

We modify each post sequence of a marked cell to post a message  $m_{i,j,b}$  (corresponding to the occurrence of  $x_i$  in  $C_j$  in positive or negative form based on the value of the bit  $b$ ) to  $h_W$  just before its return to  $h_V$ . This is what we called *message insertion*. For example, in  $p_2^{x_n}$  we insert the events  $e_2, e_3, e_4, e_5$  between the events  $e_1$  and  $e_6$  which are part of  $p_2^{x_n}$  as follows:

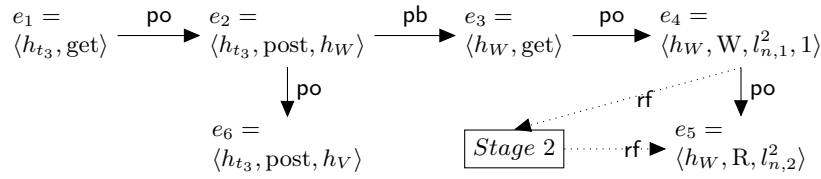


Fig. 11: Inserting message into post sequence.

Consider the set  $M_6$  of six messages posted to  $h_W$  corresponding to  $C_2$ . The assignment to  $x_1$  and  $x_n$  are chosen by using different choices of  $k$  of handlers  $h_{t_k}$

for them, but the assignment to  $x_2$  was already chosen when executing the post sequence for  $C_1$ . Hence  $p_4^{x_2}, p_4^{\bar{x}_2}$  will both contain a single post to  $h_{t_2}$  and the corresponding messages will be posted to  $h_W$  in the order already chosen during the  $C_1$  part. Note that two identical post sequences  $p_1 = p_2$  which start in some order in the queue of some handler  $h$  will occupy the queue of subsequent handlers  $h'$  of the post sequence in the same order due to queue semantics. Crucially, we prevent orderings of  $M_6$  messages in  $h_W$ 's queue which do not correspond to consistent assignment of variables. However we allow all other possible reorderings of  $M_6$  messages and this is essential for the verification in Stage 2, where only some of these reorderings may be allowed based on the partial order of events in a satisfiable clause i.e. one where not all red **hb** arrows are present (see Figure 8).

We now show the correctness of this construction.

### Correctness of the Construction

**Lemma 2.** *The 3-BI-3SAT formula  $\phi$  is satisfiable if and only if there exists a witnessing execution consistent with  $\tau$ .*

*Proof.* Note that in order to obtain a consistent trace, we will have to extend our constructed trace by specifying the **eo** and **mo** relations. These will induce a happens-before relation **hb** which needs to be acyclic.

$\phi$  is unsatisfiable. For this direction, it suffices to look at the **eo** edges. Since  $\phi$  is unsatisfiable, every assignment made to the variables must set some clause  $C_j$  to false. In our example, consider the setting when  $C_2 = x_1 \vee x_2 \vee \bar{x}_n$  is false i.e. each of  $x_1, x_2$  is set to false and  $x_n$  is set to true. Recall that setting  $x_n$  to true means that the message  $\overline{m_{n,2,0}}$  has an **eo** edge leading to  $m_{n,2,1}$ . We have the following chain of relations:  $e_{15} \text{ rf } e_{10} \text{ po } e_{11} \text{ rf } e_{16} \text{ eo } e_{17} \text{ rf } e_{12}$ . This implies  $e_{11} \xrightarrow{\text{hb}} e_{12}$ . Similarly we also have  $e_7 \xrightarrow{\text{hb}} e_8$  and  $e_3 \xrightarrow{\text{hb}} e_4$ . Together with the existing relations, this creates a **hb** cycle, implying that this assignment of **eo** edges is inconsistent. Since we can find such a cycle for every assignment, we conclude that our constructed trace  $\tau$  is inconsistent.

$\phi$  is satisfiable. Instead of specifying **eo** and **mo** separately, it is easier to specify a total order **hb'** on the set of all posts. This clearly induces both the **eo** and the **mo** relations in a unique way. Intuitively, the **hb'** order executes parts of the post sequences in phases corresponding to the columns, with phase  $m$  corresponding to the parts in column  $m$ . We complete all the posts in phase  $m$  before moving on to phase  $m + 1$ .

Let  $P_\ell$  be the set of all posts made by handler  $h_\ell$ . We will assign the **hb'** edges in the following way: First we consider  $P_V$ . Moving column wise through the post sequences as in Figure 9 with  $2n$  rows and  $m$  columns, let  $\langle p, q \rangle$  refer to the cell in the  $p$ -th row and  $q$ -th column. Then we have the following total order on cells:  $\langle p, q \rangle < \langle p', q' \rangle$  iff either  $q < q'$  or  $(q = q' \text{ and } p < p')$ . Together with the nesting order of posts inside each cell, this is the **hb'** order on posts in  $P_V$ . The posts which remain are those in  $P' = \bigcup_{k=1}^6 P_{t_k}$  which are all contained in the marked cells (note that  $h_W$  does not make any posts and hence  $P_W = \emptyset$ ). The **hb'** order between posts in  $P_V$  and  $P'$  is given by the nesting order. Looking at

figure 11, an example of a post in  $P'$  is  $e_2$ . Let  $e, e'$  be posts in  $P_V$  immediately before  $e_1$  and immediately after  $e_6$  in the post sequence  $p^{x_n}$ . Then we have  $e \text{ hb}' e_1 \text{ hb}' e_6 \text{ hb}' e'$ . It remains to choose the order between two posts made by different  $h_{t_k}$  in  $P'$  when choosing the assignment of a variable  $x_i$ . Note that there are 6 in total, we denote this set of posts  $P_{x_i}$ .

In order to determine the order between two posts in  $P_{x_i}$  we will look at what happens in Stage 2. We explain using the example in Figure 8.

Let  $f$  be a satisfying assignment for  $\phi$ . Since  $f$  satisfies  $C_2$ , this means that it sets one of the literals in  $C_2$  to true. Let us consider the case where  $f(x_2) = f(x_n) = \text{true}$  and  $f(x_1) = \text{false}$ . This induces  $e_{11} \xrightarrow{\text{hb}} e_{12}$ ,  $e_7 \xrightarrow{\text{hb}} e_8$  and  $e_4 \xrightarrow{\text{hb}} e_3$ . Overall we then get the following happens-before relation on the partial trace:

$e_4 \xrightarrow{\text{hb}} e_5 \xrightarrow{\text{hb}} e_6 \xrightarrow{\text{hb}} e_7 \xrightarrow{\text{hb}} e_8 \xrightarrow{\text{hb}} e_9 \xrightarrow{\text{hb}} e_{10} \xrightarrow{\text{hb}} e_{11} \xrightarrow{\text{hb}} e_{12} \xrightarrow{\text{hb}} e_{13} \xrightarrow{\text{hb}} e_{14} \xrightarrow{\text{hb}} e_1 \xrightarrow{\text{hb}} e_2 \xrightarrow{\text{hb}} e_3$ . In this case, we already have a total ordering on the events. In the case where more than one literal of a clause is set to true by  $f$ , we will get a partial order for the hb relation. In all cases, we can choose a total order  $<_2$  which extends this partial order.

Let us consider our example clause  $C_2$  with assignment  $f$ . Since  $x_1 = \text{true}$  and this is the first occurrence of  $x_1$  in any clause, the post sequences containing  $m_{1,2,0}$  and  $m_{1,2,1}$  will first be sent to  $h_{t_1}$  and  $h_{t_4}$  respectively. After this, they are both sent to  $h_{t_1}$  from where they post the messages to  $h_W$ . This means that depending on which of the two post sequences is posted to  $h_{t_1}$  just before they post to  $h_W$ , the value of  $x_1$  is assigned. In the example since  $x_1 = \text{true}$ , this means that  $m_{1,2,1}$  will be posted later than  $m_{1,2,0}$ . After this, they are both sent back to  $h_V$  in the same order in which they appeared in  $h_{t_1}$  for the last time. The message sequences corresponding to  $x_1$  and  $\bar{x}_1$  still have two more messages to be posted to  $h_W$ . Since the remainder of the post sequences  $p^{x_1}$  and  $p^{\bar{x}_1}$  is identical, the order chosen between them is retained and the assignment to  $x_1$  by the 4 other messages is consistent with the original choice. For example,  $x_2$  is occurring for the second time when it appears in  $C_2$ . This means that the order between the messages  $m_{2,2,0}$  and  $m_{2,2,1}$  has already been chosen during the posting of  $m_{2,1,0}$  and  $m_{2,1,1}$  executed previously. When  $m_{2,2,0}$  and  $m_{2,2,1}$  are to be posted to  $h_W$ , the post sequence sends both to  $h_{t_2}$ , maintaining the prior order.

In general, the posts within  $\bigcup_k h_{t_k}$  are completed before sending back to  $h_V$  for each marked cell, and the execution continues with the sending of the next row with unmarked cell to the back of the queue in  $h_V$ . In this manner, all posts in phase  $m$  are completed and we proceed to phase  $m + 1$ .

When all post events have been executed for all phases, the configuration consists of empty queues in all but the handler  $h_W$ , where the messages  $m_{i,j,b}$  are placed in the order  $<_2$  dictated by  $f$ . We can now execute the clause checking in Stage 2 i.e. the events in the handlers  $h_{C_a}, h_{C_b}, h_{C_c}, h_{C_d}$  which are sandwiched with the writes of the messages in  $h_W$ . This completes the proof of correctness.

## D Proof of Theorem 3

Consider a trace  $\tau$  with  $k$  handlers and with no nesting of posts. This implies that there is an initial message  $m_i$  in each handler  $h_i$   $1 \leq i \leq k$  such that all **post** events posted by  $h_i$  occur in  $m_i$ . The set of all post instructions from a handler is totally ordered by the **po** relation since they are in the initial message. This implies a total **mo** order  $\text{mo}_{i,j}$  on the set  $P(i,j)$  of all posts made by  $h_i$  to  $h_j$ . Hence the trace already specifies  $k$  many total orders on the set of all posts made to a handler  $h_j$ . Due to queue semantics, this translates to  $k$  many total orders on the messages corresponding to these posts. Note that the initial message  $m_j$  occurs before each of the posted messages in  $h_j$ . Let  $M_{i,j} = m_{i,j,1} \text{ eo } m_{i,j,2} \text{ eo } \dots m_{i,j,l}$  be the set of messages corresponding to the posts  $P_{i,j}$ . Note that each message consists of sequence of events  $e_1 \text{ po } \dots \text{ po } e_o$ .

We define a configuration  $C$  as containing for each handler  $h_j$ :

- (1) A pointer  $s_{i,j}$  denoting an element of  $M_{i,j}$  (or "start" if the first message in  $M_{i,j}$  has not yet started, or "end" if all messages in  $M_{i,j}$  have completed executing) for each  $i$ , and
- (2) A pointer  $r_j$  to an event which is either in the initial message or one of the messages in  $M_{i,j}$  or "term".

The pointers in (1) indicate which messages have been executed, while (2) indicates the program pointer of the currently executing message in  $h_j$ . Note that if  $r_j$  points to an event in the initial message  $m_j$ , then the pointer in (1) is set to "start".

Each pointer can be stored in space  $O(\log(n))$ , hence the total amount of space required is  $O(k^2 \log(n))$ . This implies that the total number of configurations is polynomial in  $n$  and exponential in  $k$ . Let  $\mathcal{C}$  denote the set of all configurations. We create a graph  $G = (\mathcal{C}, \mathcal{E})$  where the set of edges  $\mathcal{E}$  is determined as follows: Consider two configurations  $C = (\{s_{i,j} \mid 1 \leq i, j \leq k\}, s)$  and  $C' = (\{s'_{i,j} \mid 1 \leq i, j \leq k\}, s')$ . There is an edge between  $C$  and  $C'$  iff one of the following holds: Either  $s'_{i,j} = s_{i,j}$  for all  $j$ ,  $r_j = r'_j$  for all  $j$  except for a unique handler  $h_q$  for which  $r'_j$  points to the event which is the successor of  $r_j$  under the **po** order, or there exists  $j_0$  such that  $r_j$  points to the last event of a message,  $s'_{i,j} = s_{i,j}$  for all  $j \neq j_0$ , and  $r'_j$  points to the first event of the successor message of  $s_{i,j}$  under the **eo** order (or to "end" if all messages in  $h_j$  have been executed). In the case where the initial message is finishing execution, the pointers for  $s_{i,j}$  are all set to the first message in each of the  $M_{i,j}$ .

The initial node  $v_0$  of the graph sets all pointers  $s_{i,j}$  to "start" and  $r_j$  to the first event of every initial message. The last node  $v_f$  sets all pointers  $s_{i,j}$  to "end" and  $r_j$  to "term". The trace is consistent iff there is a path from  $v_0$  to  $v_f$  in the graph.

## E Appendix for Section 6

## Experimental Results for Traces generated via NIDHUGG

We used the open-source model checker NIDHUGG to generate traces from event-driven programs. While NIDHUGG supports an event-driven execution model, it currently interprets asynchronous semantics using multisets rather than FIFO queues. As a result, some of the generated traces may be inconsistent under queue semantics. For each benchmark program, we randomly sampled traces from Nidhugg’s output. These traces are guaranteed to satisfy multiset semantics, but may violate stricter queue-based consistency.

Benchmark	# E	# M	# H	# T	Algorithm 1			Algorithm 2		
					# Consistent traces	# T/O traces	Time in sec.	# Consistent traces	# T/O traces	Time in sec.
Buyers (2)	88	6	2	2	2	0	0.0071	2	0	0.0295
Buyers (4)	166	10	2	10	10	0	14.3448	10	0	0.0358
Buyers (8)	322	18	2	10	0	10	-	10	0	0.0603
ChangRoberts (2)	207	11	3	2	2	0	0.0930	2	0	0.0356
ChangRoberts (4)	353	17	5	10	4	6	36.1441	10	0	0.0432
ChangRoberts (8)	737	33	9	10	0	10	-	10	0	0.0687
Consensus (2)	221	9	3	4	2	0	0.1624	2	0	0.0338
Consensus (4)	677	25	5	10	0	10	-	1	0	0.0702
Consensus (8)	2333	81	9	10	0	10	-	1	0	2.3321
Counting (2)	129	9	3	4	3	0	0.0269	4	0	0.0329
Counting (4)	443	25	5	10	0	10	-	10	0	0.0596
Counting (8)	1647	81	9	10	0	10	-	10	0	1.1833
MessageLoop (2)	268	23	3	10	1	0	0.3115	1	0	0.1172
MessageLoop (4)	954	73	5	10	1	0	72.2307	1	0	4.7470
MessageLoop (8)	3670	269	9	10	0	10	-	0	10	-
SparseMat (2)	231	7	3	7	7	0	0.1507	7	0	0.0375
SparseMat (4)	427	11	3	10	6	4	64.3954	10	0	0.0485
SparseMat (8)	819	19	3	10	0	10	-	10	0	0.1141

Table 2: Experimental results for benchmark programs collected from droidracer. The field # T denotes the number of traces. The traces can differ in size (events # E), messages # M, handlers # H), and the field contains the maximum of its traces. The field # Consistent traces denotes the number of these traces for which the implementation reports the existence of a satisfying execution. The field # T/O traces denotes the number of traces for which our tool timed out (with a timeout of 120s). For any remaining traces, the tool concludes inconsistency. The time fields represent the average runtime for the traces that did not time out. A value of - indicates that the corresponding algorithm timed out on every trace.

## Benchmarks

We consider standard benchmark programs from prior works on event-driven programs [32]. To demonstrate the subtleties of our setting, we have modified these examples to allow for multiple messages being posted to the same handler.

- **Consensus** is an example taken from Kragl et al. [32]. Here, the protocol outlines a straightforward broadcast consensus approach where  $n$  nodes aim to reach agreement on a single value. Each node, labeled  $i$ , spawns two threads: one responsible for broadcasting, which sends the node’s value to all

other nodes, and another serving as an event handler that runs a collection process. This collection process gathers  $n$  values and decides on the maximum, storing it as the node's final decision. As each node receives inputs from every other node, by the protocol's conclusion, all nodes converge on the same agreed-upon value.

- **Buyer.** This example involves  $n$  buyer processes working together to purchase an item from a seller. Initially, one buyer obtains a price quote from the seller. The buyers then coordinate their individual contributions, and if their combined contributions meet or exceed the cost of the item, an order is placed. This benchmark is adapted from [10].
- **Sparse-matrix** is a program that determines the count of non-zero elements in a sparse matrix with dimensions  $m \times n$ . The computation is carried out by splitting it into  $n$  separate tasks, each sent as a message to different handlers. These handlers process their respective portions and then aggregate the results, ultimately combining the outputs from each task to produce the final count.
- **Message-loop** is a synthetic benchmark in which there are  $n$  handlers. Whenever the  $k$ th handler receives a message, it increments a global message counter, and sends a new message to the  $k + 1$ th handler, looping back from  $n$  to 1. Each such chain passes every handler  $n$  times (for a total of  $n^2$  messages). Initially, two chains are started; each at the first handler.
- **Counting** is a synthetic benchmark. Initially, each of  $n$  handlers send a message to each other handler, and finally a message to itself. When a handler handles a message from a different handler  $j$ , it writes to a variable that it most recently received a message from handler  $j$ . This gets overwritten if it then handles a message from handler  $k$ . If a handler receives the message from itself, it reads the value of the shared variable, to see from which handler it has gotten the current value.