Science of Computer Programming 240 (2025) 103212

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

VAMOS: Middleware for best-effort third-party monitoring

Marek Chalupa [a,*], Fabian Muehlboeck [a,b], Stefanie Muroya Lei [a], Thomas A. Henzinger [a]

[a] Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria
[b] Australian National University, Canberra, Australia

ARTICLE INFO

Keywords:
Runtime monitoring
Runtime verification
Parallel computing
Third-party
Best-effort
Black-box

ABSTRACT

As the complexity and criticality of software increase every year, so does the importance of runtime monitoring. Third-party and best-effort monitoring are especially valuable, yet under-explored areas of runtime monitoring. In this context, third-party monitoring means monitoring with a limited knowledge of the monitored software (as it has been developed by a third party). Best-effort monitoring keeps pace with the monitored software at the cost of possibly imprecise verdicts when keeping up with the monitored software would not be feasible. Most existing monitoring frameworks do not support the combination of third-party and best-effort monitoring because they either require the full access to the monitored code or the ability to process all observable events, or both.

We present a middleware framework, VAMOS, for the runtime monitoring of software. VAMOS is explicitly designed to support third-party and best-effort scenarios. The design goals of VAMOS are (i) efficiency (tracing events with low overhead), (ii) flexibility (the ability to monitor a variety of different event channels, and to connect to a wide range of monitors), and (iii) ease-of-use. To achieve its goals, VAMOS combines aspects of event broker and event recognition systems with aspects of stream processing systems.

We implemented a prototype toolchain for VAMOS and conducted a set of experiments demonstrating the usability of the scheme. The results indicate that VAMOS enables writing useful yet efficient monitors, and simplifies key aspects of setting up a monitoring system from scratch.

1. Introduction

Runtime monitoring—the checking of a formal specification over a concrete run of a system—is a lightweight verification technique for deployed software. Writing monitors is especially challenging if it is *third-party* and *real-time*. In third-party monitoring, the monitored software and the monitoring software are written independently, in order to increase trust in the monitor. In the extreme case, the monitor has very limited knowledge of and access to the monitored software, as in black-box monitoring. In real-time monitoring, the monitor must not slow down the monitored software while also following its execution close in time. Best-effort monitoring is real-time monitoring where we allow to check the specification only approximately if the monitor is not be able to timely process the influx of all observed events.

* Corresponding author.
E-mail address: marek.chalupa@ist.ac.at (M. Chalupa).

https://doi.org/10.1016/j.scico.2024.103212
Received 6 November 2023; Received in revised form 16 September 2024; Accepted 17 September 2024

Available online 23 September 2024

We present a middleware framework, called VAMOS ("Vigilant Algorithmic Monitoring of Software"), that facilitates the addition of best-effort third-party monitors to deployed software. The primary goals of our middleware are

(i) performance – keeping pace with the monitored system while incurring only a low overhead in the system,
(ii) flexibility – compatibility with a wide range of heterogeneous event sources that connect the monitor with the monitored software, and with a wide range of formal specification languages that can be compiled into VAMOS specifications, and
(iii) ease-of-use – the middleware aims to relieve the designer of the monitor of system and code instrumentation concerns.

All of these goals are fairly standard, but VAMOS' particular design tradeoffs center around making it as easy as possible to create a best-effort third-party monitor of actual software without investing much time into low-level details of instrumentation or load management. In practice, instrumentation—enriching the monitored system with code that is gathering observations on whose basis the monitor generates verdicts—is a key part of writing a monitoring system and highly affects the performance characteristics of the monitoring setup [1]. These considerations become even more important in third-party monitoring, where the limited knowledge of and access to the monitored software may force the monitor to spend more computational effort to re-derive information that it could not observe, or combine it from smaller pieces obtained from more (and different) sources. By contrast, current implementations of monitor specification languages mostly offer either very targeted instrumentation support for particular systems or some general-purpose API to receive events, or both, but little to organize multiple heterogeneous event streams, or to help with the kinds of best-effort performance considerations that we are concerned with. Thus, VAMOS fills a gap left open by existing tools.

Our vision for VAMOS is that users writing a best-effort third-party monitor start by selecting configurable instrumentation tools from a rich collection. This collection includes tools that periodically query webservices, generate events for relevant system calls, observe the interactions of web servers with clients, and of course standard code instrumentation tools. The configuration effort for each such *event source* largely consists of specifying patterns to look for and what events to generate for them. VAMOS then offers a simple specification language for filtering and altering events coming from the event sources, and simple yet expressive event recognition rules that produce a single, global event stream by combining events from a (possibly dynamically changing) number of event sources. This global event stream is forwarded into an actual monitor to generate verdicts about the run of the monitored system. This monitor could be written manually or generated from formal specifications like LTL [2], or stream verification languages [3] such as TeSSLa [4].

VAMOS thus represents middleware between the monitored system and higher-level monitoring code, abstracting away many low-level details about the interaction between the two.

In the setups that we consider, it is not unlikely that the monitored system and the monitor each run with different speeds. To decouple the performance of higher-level monitoring code from the performance of the monitored system, we provide a simple load-shedding mechanism that we call *autodrop buffers*. Autodrop buffers are buffers that drop events when the monitoring code cannot keep up with the rate of incoming events, while maintaining summarization data about the dropped events. This summarization data can later be used by our event recognition and broker system, that we call the *arbiter*; note that some standard monitoring systems can handle such *holes* in the event streams automatically [5–7], and for those the arbiter does not have to do anything special. Apart from event recognition, the arbiter allows dynamical grouping and ordering of the buffers to prioritize or rotate within variable sets of similar event sources, and specifying patterns over multiple events and buffers, to extract and combine the necessary information for a single global event stream.

Data from event sources is transferred to the monitor using efficient lock-free cache-friendly buffers in shared memory. These buffers can transfer over one million events per second per event source on a standard desktop computer. Together with autodrop buffers, this satisfies our performance goal while keeping the specification effort low. As such, VAMOS resembles a single-consumer version of an event broker [8–13] specialized to runtime monitoring.

The core features we built VAMOS around are not novel on their own, but to the best of our knowledge, their combination and application to simplify best-effort third-party monitoring setups is. Thus, we make the following contributions:

- We built middleware to connect higher-level monitors with event sources, addressing challenges of best-effort third-party monitoring (Section 2). The middleware mixes efficient inter-process communication and simple facilities for load management (Sections 3.2 and 3.3), with event recognition techniques (Section 3.4).
- We implemented a compiler for VAMOS specifications, a number of event sources, and a connector to TeSSLa [4] monitors (Section 4).
- We conducted some stress-test experiments using our implementation, as well as two slightly bigger case studies in which we implemented a monitor looking for data races and UI event inconsistencies, providing evidence of the feasibility of low-overhead third-party monitoring with simple specifications (Section 5).

1.1. Extensions to the conference paper

This paper extends our previous work presented at the *26th International Conference on Fundamental Approaches to Software Engineering (FASE'23)* [14]. In particular, we have

- extended the specification language and its compiler implementation to allow for sorting buffers in buffer groups based on event data,

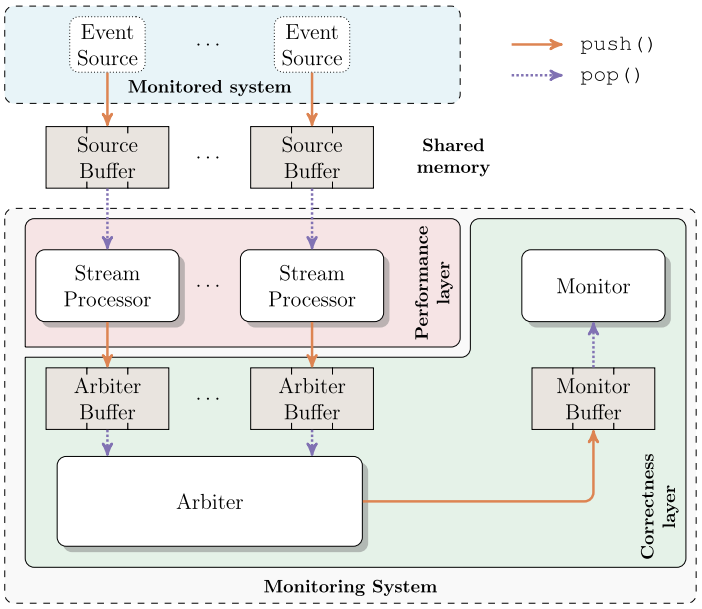


Fig. 1. The components of a VAMOS setup and the flow of events: edges represent that the edge's source component pushes events into the buffer, and edges represent that the edge's target component pops events from the buffer.

- substantially rewritten some parts of the paper and added a running example to better explain VAMOS specification language,
- added new experiments with monitoring Wayland [15] connections with VAMOS, and
- updated and extended the sections on related and future work (sections 6 and 7).

2. Overview of VAMOS

The architecture of VAMOS is shown in Fig. 1. On the top, we assume an arbitrary number of distinct event sources that generate events inside the monitored system(s). The ability to handle multiple event sources is particularly important in third-party monitoring, as information may need to be collected from multiple different sources instead of just a single program, but can be also useful, e.g., for multi-threaded programs. The component at the very end of the flow of events is the *monitor*. As middleware, VAMOS connects events sources with the monitor, providing abstractions for common issues that monitors' writers would otherwise have to address.

The core of VAMOS consists of the *performance layer* and the *arbiter*; together with the already-mentioned *monitor*, the *arbiter* constitutes the *correctness layer*. This division directly reflects the requirement for the separation of higher-level monitoring code from the monitored system mentioned in Section 1. Components in the layers are connected via event buffers that are used to pass events between them as shown in the figure.

Once an event is generated by an event source, the event source pushes it into its associated *source buffer*. Source buffers reside in shared memory and are used to transfer events from the monitored system to the monitoring system. From the source buffer, the event gets into the performance layer when it is read and consumed by the *stream processor*. Every stream of input events is handled by its own stream processor running in its own thread. A stream processor sees all incoming events on its event stream and for each event it decides whether to filter it out or forward it, possibly modified, to the correctness layer. If an event is forwarded, the stream processor pushes it into its associated *arbiter buffer*, where events wait until they are handled by the arbiter.

As the name suggests, the performance layer should be as fast as possible so that the monitored system does not need to wait for space in source buffers while pushing new events. Ideally, the code in this layer can keep up with the rate at which the monitored systems generate events. If it is the case, the instrumentation in the monitored system never has to wait for free space in the source buffer, which reduces the overhead of monitoring. Therefore, the early filtering and modification of events in stream processors is an important part of the design of VAMOS. In third-party monitoring, observing coarse-grained event types like system calls may yield many uninteresting events. For example, all calls to the `read` system call may be instrumented, but only certain arguments make them interesting. Filtering out the uninteresting events reduces pressure and computational load on the correctness layer and so increases the monitoring system's ability to observe as much as possible while staying real-time with the monitored system.

Complex computations should occur in the correctness layer, whose performance can be decoupled from the rest of the monitoring system. This decoupling is achieved by marking the arbiter buffers as *autodrop buffers*. As mentioned in Section 1, an autodrop buffer drops events when it is full, but maintains a summarization of the dropped events in the form of special *hole* events. Using autodrop buffers ensures that both layers can run independently at their own speed at the cost of a possible partial loss of information. In the correctness layer, the arbiter – the event recognition and broker system – sees the contents of *all* arbiter buffers and generates a single stream of higher-level events.

```
1  stream type Observation { Op(arg: int, ret: int); }
2  event source Program: Observation to autodrop(16, 4)
3  arbiter: Observation {
4      on Program: Op(arg, ret) | yield;
5      on Program: hole(n) | ;
6  }
7  monitor(2) { on Op(arg, reg) $$ CheckOp(arg, ret); $$ }
```

Listing 1: A basic asynchronous best-effort monitor.

Variable/Field	X	$\in$	some set of variable names
Value	v	$\subset$	$n \mid \texttt{true} \mid \texttt{false} \mid \ldots$
Expression	ξ	$\subset$	$X \mid v \mid \xi + \xi \mid \ldots$
Statement	σ	$\subset$	$\{\overline{\sigma}\} \mid X = \xi \mid \xi \mid \tau X = \xi \mid \texttt{if}(\xi)\ \sigma\ \texttt{else}\ \sigma \mid \ldots$
Type	τ	$\subset$	$\texttt{int} \mid \texttt{bool} \mid \ldots$
Exp Eval Ctx	$\mathbb{E}$	$\subseteq$	$\cdot \mid v + \mathbb{E} \mid \mathbb{E} + \xi \mid \ldots$
Stmt Eval Ctx	$\mathbb{S}$	$\subseteq$	$\cdot \mid \{\mathbb{S} +\!\!+ \overline{\sigma}\} \mid X = \mathbb{E} \mid \tau X = \mathbb{E} \mid \texttt{if}(\mathbb{E})\ \sigma\ \texttt{else}\ \sigma \ldots$
Base State	Π		**opaque**

Fig. 2. The base language structure.

The performance and correctness layers are only conceptual abstractions – our implementation generates C code for the monitoring system without any layers. The code defines functions for each stream processor, for the arbiter and the monitor, and runs these functions in threads (after necessary initial setup of input event streams). The only communication between the threads is via event buffers that connect them.

The final component in the scheme in Fig. 1 is the monitor, that receives the single global stream of events from the arbiter via the *monitor buffer*. In the scheme, the monitor is shown as a part of the correctness layer, but it can also be an external monitor that receives the single stream of events via, e.g., again a shared-memory buffer.

3. Specification language

We created a specification language to define the whole VAMOS' pipeline. Before going into details, we show a simple example specification that just forwards events into a monitor which processes them with a custom C function. This specification is in Listing 1.

The specification first defines a *stream type* `Observation` to define the kinds of events that can appear in the source buffer of an event source `Program`. In our example, streams of type `Observation` contain only one possible event named `Op` with two fields of type `int`. Every event source is associated with a stream type and a stream processor; the specification of the stream processor is missing for the event source `Program`, which means the default one that simply forwards all events to the arbiter is used. However, the specification of this event source has attached the information that the arbiter buffer where the events should be forwarded should be an autodrop buffer that can hold up to 16 events, and when it is full it keeps dropping the events until there is again space for at least 4 new events. The rest of the specification describes the arbiter and the monitor. The arbiter specification consists of two rules describing that `Op` events coming from the event source `Program` should be forwarded (yielded) to the monitor (line 4), and that `hole` events should be just ignored (line 5). The symbol '|' separates events that are consumed from the arbiter buffer if a rule is successfully matched, and those that serve only as a lookahead (which is empty in our example). Matching rules is described in a greater detail in Subsection 3.4. Lastly, the monitor specification has a single rule (line 7) that invokes C code specified in between `$$` escape characters. This code calls the function `CheckOp`, which can be an arbitrary C function. The number in parentheses after the keyword `monitor` specifies the size of the buffer for the global stream of events where arbiter pushes events to be processed by the monitor. This buffer is always blocking (not autodrop) and in the example has size 2.

In the next few sections, we explain more details about the kinds of VAMOS specifications one can write, including a semi-formal description of the specification language. In particular, in sections 3.1 and 3.2, we discuss the general traits of the specification language. In Section 3.3, we give details about specifying event sources and the performance layer. Finally, in Section 3.4, we discuss the specification of the correctness layer, most importantly specifications of the arbiter.

3.1. Base language

The VAMOS specification language is designed to be an extension of some existing programming language, referred to as the *base language*. Our implementation works as a sort of preprocessor on C code, but could be instantiated for other languages too. Fig. 2 sketches the core parts that we assume are in the base language. We assume that a reduction relation for this base language exists.

As a matter of notation, grammatical constructs whose description is in *italics* are for identifiers of various constructs. In our abstract grammar, the sets of relevant identifiers are distinguished, and assumed to be countably infinite with decidable equality. In our implementation, there is only one kind of identifier, and the compiler distinguishes them based on their location and environment—this is again a standard behavior. In the grammar, we use $\overline{N}$ for the list of forms N, and $[N]$ for N being optional, i.e., M $[N]$ M' is a shortcut for $M\ M' \mid M\ N\ M'$ (and analogously for multiple optional forms).

Event	E	$\in$	some set of event names
Stream Type	T	$\in$	some set of stream type names
Aggregation Op.	O	$\in$	(e.g. MIN/MAX/COUNT/SUM)
Event Defn	$\hat{E}$	::=	$E(\Gamma)$ [`creates` T]
Stream Type	$\hat{T}$	::=	`stream type` $T(\Gamma)$ [`shared`(Γ)] [`aggregates`($\overline{\hat{X}}$)] [`extends` T] { $\overline{\hat{E}}$ }
Aggregate Defn	$\hat{X}$	::=	$X = O(\overline{E.X}) \mid X = O(\overline{X})$
Type Env	Γ	::=	$\overline{X : \tau}$

Fig. 3. Event and stream type definitions.

```
1  stream type ThreadEvent
2    shared(timestamp: uint64_t)
3  {
4      read(addr: intptr_t);
5      write(addr: intptr_t);
6      lock(addr: intptr_t);
7      unlock(addr: intptr_t);
8      fork(newthreadid: uint64_t) creates ThreadEvent;
9      join(threadid: uint64_t);
10 }
11
12 stream type MThreadEvent(threadid: int, hole_ts: uint64_t)
13   extends ThreadEvent
14   aggregates(maxts: uint64_t = MAX(timestamp))
15 {
16     te_hole(n: uint64_t, lastts: uint64_t);
17 }
```

Listing 2: More featureful stream type definitions.

3.2. Event and stream types

Fig. 3 presents the key definitions related to events and stream types. As discussed in Section 2, events in VAMOS are passed between components via various buffers and each such a buffer is associated with a stream type that describes the events it may hold. Each event has a name E and some typed fields specified by a type environment Γ that assigns field names to types. Additionally, some events may specify that they indicate the creation of a new stream of type T. Stream types, in turn, have a name T, some *stream fields* Γ, and, a list of event declarations $\overline{\hat{E}}$. Optionally, stream types can specify special *shared fields*, *stream aggregates* $\overline{\hat{X}}$, and the specification of extending some other stream. We discuss the components of stream types on an example in Listing 2.

In the example, which shows a part of the (simplified) specification for data race detection from our experiments, we define two stream types. The stream type `ThreadEvent` describes events coming from an event source observing memory accesses and synchronization events of some program. All events have a shared field `timestamp` (line 2). Shared fields provide a simple form of inheritance and thus simplify writing a stream specification (by avoiding repetition), and allow abstraction over events in cases where the type of an event does not matter, but the values of the shared fields do (e.g., for ordering purposes). Of course, all events can carry also individual data, and the `fork` event on line 8 indicates that when it is received, a new stream with the same stream type is created (because this event is received when a new thread in the monitored program is spawned). We discuss the details of dynamic streams creation later.

Our example also contains a second stream type definition. On line 12, we define the stream type `MThreadEvent` that *extends* `ThreadEvent`. An extension of a stream type carries over all stream fields, aggregates, and event definitions from the stream type it extends, and cannot adjust the shared fields. This is purely for convenience, and does not create any subtyping relationships.

Except for adding an additional event `te_hole` that is not defined for `ThreadEvent`, the stream type `MThreadEvent` specifies also stream fields `threadid` and `hole_ts`. Stream fields are data that are associated with a particular stream, not its events. In this case, we want one event stream per thread in the observed program, and therefore each instance of `MThreadEvent` stream keeps track of the thread ID it was created for. In addition, the stream field `hole_ts` stores some bookkeeping data (the timestamp of the last hole event on the stream) that are not important at this moment.

Finally, `MThreadEvent` specifies also the aggregate field `maxts`. Aggregate fields are similar to stream fields in that they are data associated with a particular stream. The key difference is that their value is computed automatically, based on the events that come in on the stream. In this case, the `maxts` field represents the maximum `timestamp` of events that has been transmitted on a particular

Event Source	S	$\in$	some set of event source names
Stream Processor	P	$\in$	some set of stream processor names
Event Source	$\hat{S}$	::=	`event source` $S(\Gamma)$: T
			`process using` $P(\overline{\xi})$ `to` k
Stream Processor	$\hat{P}$	::=	`stream processor` $P(\Gamma)$: $T(\overline{\xi}) \rightarrow T(\overline{\xi})$
			$\{\overline{p}\ ;\ \hat{H}\}$
SP Expression	e	::=	`forward` $E(\overline{\xi})$ | `drop` | `if` ξ `then` e `else` e
SP Rule	p	::=	`on` $E(\overline{X})$ e
		|	`on` $E(\overline{X})$ `creates` [`at most` n] T
			`process using` $P(\overline{\xi})$
			`to` k `include in` $\overline{B}$ e
Hole Defn	$\hat{H}$	::=	`hole` E { $\overline{\hat{X}}$ }
Buffer Kind	k	::=	`autodrop`(n,n) | `infinite` | `blocking`(n)

Fig. 4. Performance layer specification.

stream. The definition for aggregate fields expresses their calculation as application of some aggregate operator O over a list of (event, field name) pairs, or just field names if the field is shared (as in our example).

Stream fields and stream aggregates are only used for arbiter buffers. In our running example, `ThreadEvent` is the stream type used for source buffers, while `MThreadEvent` is used for arbiter buffers. Later in this section, we will see how arbiter buffers make use of the additional features.

3.3. The performance layer

In this subsection, we discuss the parts of the specification language that are related to the performance layer. Fig. 4 shows the relevant parts of the formal specification, and we will extend our running example further below.

3.3.1. Event sources and stream processors

The key components of the performance layer specification are event sources and stream processors. At least one event source needs to be specified by name (the rest can be dynamically created). Event source specification can contain constructor arguments that are forwarded to the constructor of its stream processor. It also specifies a stream type to describe the events that it expects coming from the outside, as well as an arbiter buffer kind (described further below) and its stream processor.

Stream processors are relatively simple matching functions, defined by a name (so their implementation can be shared by multiple event sources) and with potential arguments that are used to initialize the stream fields of their output buffers. Stream processors define a matching rule for each event in the incoming stream type. Each rule is a tree of if-then-else expressions that end in either creating an event in the outgoing arbiter buffer (using **`forward`**) or dropping the incoming event. Additionally, stream processors define a *hole* event, used for `autodrop` buffers discussed further below.

We saw earlier that some events signal the creation of a new event source. This is mirrored in the stream processor rules for those events, which specify what to do with such new event sources. In particular, a new stream processor is instantiated for them, using a particular buffer kind. Since such event sources are dynamically created, they lack a user-defined name. Buffer groups, discussed further below, provide a way to deal with this. The **`include in`** part of a **`creates`** stream processor rule refers to one or more buffer groups that newly created event sources should be added to. Our implementation allows limiting the number of concurrent sub-event-sources created this way to avoid overloading the monitor (using the `at most` n parameter).

In Listing 3, we see an example specification of a performance layer, continuing our running example. We start by declaring a single event source named `Program` that has the stream type `ThreadEvent`. The compiled monitor will expect command line arguments declaring how to connect to a specific running instance of it.

`Program` uses the stream processor `TEProc`, and is by default included in the buffer group `Threads`. `TEProc` is a stream processor for a given thread in the observed program. In our implementation, each stream processor also runs in its own thread, aiming to process incoming events as quickly as possible. By default, events for which there is no stream processor rule are forwarded as-is; this is supported by `MThreadEvent` being an extension of `ThreadEvent`. The only event that needs special handling is the `fork` event, which signals the creation of a new thread, and its associated new event source. To avoid overloading the monitor, once 100 threads are reached, the system immediately closes additional event sources until some of the existing ones shut down.

3.3.2. Autodrop buffers

The specification of each event source contains an arbiter buffer kind. Two of these are standard: an `infinite`-size buffer that expands on demand, and a fixed-size buffer that blocks when its full (`blocking`). To truly decouple the performance of the monitoring system from the monitored system, `autodrop` buffers are fixed-size buffers that simply drop events when they are full. The two arguments for an `autodrop` buffer are the size of the overall buffer, and the minimum number of free space in the buffer before it stops dropping events after it has started doing so.

Dropping events that would otherwise be forwarded to the arbiter clearly loses information. To mitigate this loss at least somewhat, dropped events are summarized in a special **`hole`** event, which is the last part of a stream processor specification. By default, every arbiter buffer has a special **`hole`** event that simply counts the number of events that were dropped before the buffer resumed accepting

```
1  event source Program: ThreadEvent
2    process using TEProc(0)
3    include in Threads
4
5  stream processor TEProc(thrdid: int):
6    ThreadEvent -> autodrop(512, 256) MThreadEvent(thrdid, 0)
7  {
8    on fork(ts, ntid)
9      creates at most 100 ThreadEvent
10       process using TEProc(ntid)
11       include in Threads
12     if(ntid>0) then forward fork(ts, ntid) else drop;
13
14   hole te_hole {
15     n = COUNT(*);
16     timestamp = MIN(timestamp);
17     lastts = MAX(timestamp);
18   }
19 }
20 buffer group Threads [...]
```

Listing 3: Performance layer example.

Buffer Group	B	$\in$	some set of buffer group names
Buffer Group	$\hat{B}$	::=	`buffer group` B : T `order by` o $\{\overline{S}\}$
Order Expr	o	::=	`round robin` | X | `shared` X ω
Aggregate Expr	λ	::=	$O\ X$ | X | n | $\lambda + \lambda$ | $\lambda - \lambda$ | $\lambda * \lambda$
Missing Mode	ω	::=	`wait` | `assume` λ | `ignore`

Fig. 5. Buffer group specification.

events. However, user-defined `hole` events, such as `te_hole` in line 14, are possible. These need to be events specified by the stream processor's outgoing stream type, and a `hole` definition needs to specify an aggregate formula for each field of the specified event. In the case of our example, we still ask for the count of all dropped events `n`, and the minimum and maximum `timestamp` of the dropped events.

This aggregate summarization data can be updated each time an event is dropped, and still provide some (and in some cases, all relevant) information about the events on a given event stream.

3.4. The correctness layer

The key part of our correctness layer is the arbiter, a rule-based event recognition and transformation system that generates a single stream of events for the monitor from combinations of events on different arbiter buffers (that correspond to different event sources). VAMOS arbiter specifications are flexible enough to support many ways of ordering and merging events into a single global stream.

3.4.1. Buffer groups

The arbiter matches rules against events in arbiter buffers. Arbiter buffers that correspond to named event sources can be referred to via the same name. However, we saw that source buffers, and therefore corresponding arbiter buffers, may also be created dynamically. And sometimes, there may be a set of distinct named event sources that nevertheless should be treated uniformly. Buffer groups are the key mechanism to uniformly operate on arbitrary sets of arbiter buffers, and to be able to use dynamically created event sources.

Fig. 5 shows the grammar for the specification related to buffer groups. Each buffer group has a name B, and specifies a stream type, which is the type of buffers it can contain. As the arbiter looks for a match of one of its rules, it tries the members of a buffer group one by one. The order in which it does this is often critical, and so a buffer group specifies an ordering in which it lists its members. Finally, a buffer group can list an initial set of members by naming the corresponding event sources.

To order buffer groups, we provide three options. The first option, `round robin`, ensures a fairness principle that prioritizes each member equally often. The second option specifies buffers in the buffer group should be ordered by a stream field X. Here we assume that there is a total order (typically $<$) on the stream field's type. Finally, we allow ordering buffer groups by the first event in their queue, using one of the shared fields in the given stream type, and similarly assuming that the type of that share field has a total order that we can use for sorting. All three options are reasonably efficient to implement: the first option induces no additional cost, and for the other two, it is easy to detect when a stream or shared field is updated and thus the buffers should be re-ordered, because only arbiter specification can update stream or shared fields.

$$
\begin{array}{lll}
\text{Arbiter} & \hat{A} & ::= \texttt{arbiter} : T\ \{\overline{\hat{a}}\} \\
\text{Arbiter Rule} & \hat{a} & ::= \texttt{on}\ \overline{b}\ [\texttt{where}\ \tilde{\xi}]\ \{\ \tilde{\sigma}\ \} \\
 & & \mid\ \texttt{choose}\ \overline{S}\ \texttt{from}\ B\ \{\overline{\hat{a}}\} \\
 & & \mid\ \texttt{choose}\ \overline{S}\ \texttt{from first}\ n\ B\ \{\overline{\hat{a}}\} \\
 & & \mid\ \texttt{choose}\ \overline{S}\ \texttt{from last}\ n\ B\ \{\overline{\hat{a}}\} \\
\text{Match Expr} & b & ::= S : \texttt{done} \mid S : \texttt{nothing} \mid S : n \mid S : \overline{E(\overline{X})}
\end{array}
$$

$$
\begin{array}{lll}
\text{Expression} & \tilde{\xi}\ \textbf{extends} & \xi ::= \ldots \mid \$S.X \\
\text{Statement} & \tilde{\sigma}\ \textbf{extends} & \sigma/\tilde{\xi} ::= \ldots \mid \$S.X = \tilde{\xi} \\
 & & \mid\ \texttt{drop}\ n\ \texttt{from}\ S \mid \texttt{yield}\ E(\overline{\tilde{\xi}}) \\
 & & \mid\ \texttt{add}\ S\ \texttt{to}\ B \mid \texttt{remove}\ S\ \texttt{from}\ B
\end{array}
$$

Fig. 6. Specification of arbiter rules and extended expressions/statements.

Ordering a buffer group based on the top event might require additional specification what to do if there is no top event because a buffer is empty. Our formalization allows three answers. First, `wait` means that the arbiter will always stop and wait for all buffers in a buffer group to have at least one event once it gets to a point where it tries to select a buffer from a buffer group. Second, `assume` uses an expression λ that can be a constant, or it can be a simple operation on stream fields/aggregates, or an aggregation over all stream fields/aggregates in the buffer group. The value to which this expression evaluates is used instead of the value of the shared field of the missing top event. Finally, `ignore` means that empty buffers are not considered while iterating over the buffers in the buffer group.

3.4.2. Arbiter rules

Fig. 6 shows the grammar for specifying the arbiter. Arbiter rules specify the code of the base language $\tilde{\sigma}$ (recall that this is e.g., the C programming language) that should be executed if a rule is matched. This code, however, is allowed to contain certain special expressions and statements that are shown in the lower half of Fig. 6.

An arbiter specification is assigned a stream type T that describes the unified event stream generated by the arbiter for the monitor, and otherwise consists of a number of arbiter rule definitions $\hat{a}$. These rules either directly match events (`on`), or deal with buffer groups (`choose`) and then have further arbiter rules inside of them.

A `on` rule describes the state of arbiter buffer(s) that the arbiter matches against using a list of buffer match expressions b. The basic expression is to match that the first n events form a given pattern, where the pattern is a list of event types. The variables in the pattern will be instantiated by the corresponding event field values if the rule matched. Additionally, the pattern can contain shared fields specified by the keyword `shared` that match any event and instantiate the values of the given shared fields. Except for patterns, other options that we provide are to check that the buffer has currently no events (`nothing`), that there are no events and no more events can come in the future (`done`), or that at least a given number n of events are waiting in the buffer.

There can be arbitrary buffer match expressions on multiple arbiter buffers. The rule applies if all of the expressions match *and* the boolean expression in the `where` clause is satisfied (if given). For example, such a `where` expression can require that the values of certain fields in certain events have the same value. A `where` expression can also refer to stream field or stream aggregate values in any available buffer, via the special expression $\$S.X$.

Once a rule matches, the arbiter executes its code, which is a list of extended statements $\tilde{\sigma}$. These statements are statements from the base language, or statements that assigning new values to stream fields, drop the first n events from an arbiter buffer (`drop`), yield new events to the monitor (`yield`), and add and remove buffers to and from buffer groups (`add` and `remove`, resp.).

A `choose` rule goes through a buffer group in its order. The modifier `from first` n allows limiting the choices to consider only the first n buffers in a buffer group (in their order). The modifier `from last` is similar, but it inverts the order for the particular `choose` rule. While a `choose` rule chooses from a single buffer group B, it can choose tuples of *distinct* buffers from it. In this case, it enumerates the possible choices by lifting the order of the buffer group to a lexicographic ordering of the appropriate n-tuple. Within the scope of the `choose` rule, each selected buffer is treated as a named event source based on the given names $\overline{S}$. Matching proceeds by trying the contained rules in order – if no rule matches, we either try the next tuple in order, or, once we have exhausted all options, the rule fails to match, and we proceed trying to match the next rule on the same level.

3.4.3. Arbiter example

We continue our running example in Listing 4. It starts with the specification of the buffer group `Threads` whose definition we skipped in Listing 3. The buffer group contains buffers whose stream type is `MThreadEvent`. It also keeps its buffers in the order specified as the maximum of the `timestamp` of their first events, or, for empty buffers, we assume the maximum timestamp seen on any buffer minus 100. What this effectively means is that we assume that any buffer's processing is delayed by at most the time it takes to process 100 other events on the other buffers, so the timestamp of any event that might eventually show up on that buffer is at least that. As more events enter the system, the buffers with no events will move further down in the order of the buffer group.

Line 4 specifies that the output stream from the arbiter to the monitor is `GlobalEvent`. We omit the specification of this type here; it is very similar to `MThreadEvent`, except that all events also contain the thread ID (on account of there being only one global event stream from here on), and two special events to inform the monitor about timespans where our autodrop buffers dropped events.

```
1  buffer group Threads : MThreadEvent
2    order by shared timestamp assume MAX(maxts) - 100
3
4  arbiter: GlobalEvent  {
5    choose T from Threads
6    {
7        on T : done $$
8            $yield done_($T.threadid;, lastemitts);
9            $remove T from Threads;
10       $$
11   }
12   choose first 1 T from Threads {
13     on T : nothing $$  $$
14     on T: | shared(ts) where $$ ts > minholeend $$
15     $$
16         while(holequeue != 0 && holequeue.ts < ts) {
17           $yield skipend(holequeue.tid, holequeue.ts);
18           holequeue = holequeue_pop(holequeue);
19         }
20         if(holequeue == 0) {
21           minholeend = UINT64_MAX;
22         } else {
23           minholeend = holequeue.ts;
24         }
25         lastemitts = ts;
26     $$
27     on T: write(ts, addr) | $$
28       $yield write($T.threadid;, ts, addr);
29     $$
30     [...]
31     on T: te_hole(ts, n, lastts) | $$
32       int tid = $T.threadid;;
33       $yield skipstart(tid, ts);
34       holequeue = holequeue_insert(tid, lastts);
35       minholeend = holequeue.ts;
36     $$
37   }
38 }
```

Listing 4: Buffer groups and the arbiter example.

From then on, the specification of the arbiter is relatively straightforward. First, on lines 5–11, we look for any buffers in the buffer group that are done. If there are some, we inform the monitor that a particular thread is done, and remove its buffer from the buffer group. The variable `lastemitts`, just as the variables `minholeend` and `holequeue` are global C variables that are defined elsewhere, and we use them to track some global state. Here, `lastemitts` just captures the timestamp of the last emitted event, as the `done` state is not associated with any particular event, and therefore any particular timestamp.

If no buffer is done, we move on to the second `choose` block (line 12), which always only examines the first buffer in the buffer group's ordering. If that buffer has no events (line 13), we just continue with the next iteration, waiting for new events to arrive. As explained above, this buffer will eventually move back in the order as new events arrive on any buffer.

The second rule (line 14) matches any event and just looks at its timestamp. The `minholeend` variable tracks the timestamp of the earliest end of any seen and as of yet unfinished hole event, and the `holequeue` variable is a priority queue of such events. The rule processes all entries in the queue whose timestamp is lower than the timestamp of the next event to be processed, and yields a `skipend` events that notifies the monitor about the end timestamp of a `hole`. Once that is done, `minholeend` is not less than the current timestamp any more, and therefore, the rule does not match, and we will actually process the event in the next iterations. The '|' character in the match expression list is important: events before this character are consumed, while events after it stay in the buffer. Therefore, this rule does not consume any events when it is matched, which is what we want here.

The last rule we describe here matches a hole event (line 31). This rule yields a `skipstart` event to the monitor, and updates our global variables to capture the end of the hole accordingly.

Our example code does not show all rules, because most of them are similar to the one we show for `write` (line 27): we simply yield a corresponding event to the monitor that contains all the original event's fields plus the current buffer's thread-id. Our actual specification contains also a few extra rules to handle cases where events show up later than expected. In this case, we do not forward the event to the monitor, but instead we simulate a hole using the `skipstart` and `skipend` events so that the monitor knows that something went missing.

$$\begin{array}{lll} \text{Monitor Rule} & \hat{m} & ::= \texttt{on}\ E(\overline{X})\ \{\ \sigma\ \} \\ \text{Monitor} & \hat{M} & ::= \texttt{monitor}(n)\ \{\overline{\hat{m}}\} \end{array}$$

Fig. 7. Monitor specification.

3.4.4. Extended arbiter rules

As we described the grammar of arbiter rules earlier, we noted that one can match against multiple events on a buffer, and multiple buffers simultaneously. For example, one rule in the example about monitoring a banking application that we use in our evaluation in Section 5 looks as follows:

```
on Out : transfer(t2, src, tgt) transferSuccess(t4) |,
   In  : numIn(t0, act) numIn(t1, acc) numIn(t3, amnt) |
   where $$ t2 == t0 + 4 $$
$$ $yield SawTransfer(src, tgt, amnt); ... $$
```

This rule matches multiple events on two different buffers (`In` and `Out`), describing a series of user input and program output events that together form a single higher-level event `SawTransfer`, which is forwarded to the monitor. The pattern must be matched on all mentioned buffers.

In our implementation, rules can also be grouped into *rule sets* that the arbiter can explicitly switch between in the style of a finite state machine (rule sets were omitted from the grammar for simplicity). Within a rule set (similarly as within a `choose` block), the arbiter will execute the code for the first rule whose pattern and `where`-condition match, and then start again from the top (possibly in a different rule set). A `choose` block counts as having matched if any of the rules within it was executed.

3.4.5. Choosing tuples from buffer groups

In our running example, we only ever instantiate a single buffer when choosing from a buffer group. As we said earlier, VAMOS' `choose` rules are more flexible, allowing to select tuples of buffers. If no match is found for the current tuple, the choose rule will try the next tuples until it hits an optionally specified limit. Once a match is found, however, the whole rule set is done for the current iteration. Consider the following code adapted from our example about differential monitoring primes generators from Section 5:

```
choose F,S from Both {
  on F : Prime(n,p) | where $$ $F.pos; < $S.pos; $$
  $$ ... $$
  on F : hole(n) |
  $$ $F.pos; += n; $$
}
```

Here, the `pos` stream field keeps track of which of the two buffers received less events than the other, and is used to order the buffer group. The rule goes through all tuples in the lexicographic extension of the order specified for the buffer group. In this example, there are only two buffers at any time, so there are only two possibilities, and only one of them allows to match the first rule. However, if the buffer that saw less events has no events at all, then any hole event waiting on the other buffer can still be processed right away.

3.4.6. The monitor

The final part of the correctness layer is the monitor, which consumes the single global event stream produced by the arbiter. It's grammar is shown in Fig. 7. A monitor specification lists a number of rules, one per event kind in the stream type that the arbiter produces. Its rules are much closer to that of a stream processor: exactly one event is processed at a time, and automatically consumed at the end of the matching rule.

In our running example, the monitor's specification is rather straightforward:

```
monitor(3) {
    on read(tid, timestamp, addr)
    $$
        monitor_handle_read(tid, timestamp, addr);
    $$
    on write(tid, timestamp, addr)
    $$
        monitor_handle_write(tid, timestamp, addr);
    $$
    ...
}

```

In this case, the events that the monitor receives are handled by external C functions that correspond one-to-one to the kinds of events the monitor might see, though one could of course write arbitrary C code in these rules. The relative simplicity of the monitor's interface also makes it possible to replace it wholesale with monitors generated by other monitoring frameworks. We demonstrate this last point by implementing a hook to TeSSLa [16] monitors that can be used instead of specifying a monitor like the one above—see Section 5.2.2.

4. Implementation

The two main parts of the implementation are the VAMOS compiler, and the supporting VAMOS libraries that contain the implementation of event buffers and basic stream abstractions over them – for both the event source side and the monitoring system side. More details follow in the rest of this section.

4.1. VAMOS libraries

VAMOS libraries implement mainly event buffers and stream abstractions over them. The types of implemented buffers are the autodrop and monitor buffers, and the source buffers residing in shared memory.

The source buffers are concurrent single-producer single-consumer lock-free cache-friendly ring buffers that allow low-overhead interprocess communication between a monitored and monitoring system. The libraries allow setting up an arbitrary number of source buffers with unique names. Further, each source buffer can inform the connected monitor about new dynamically created source buffers.

Source buffers support checking for binary compatibility of the reader and writer, i.e., checking that both sides assume the same kind of events with same arguments. Also, the reader side can specify that it is interested only in a subset of events that the writer can produce. Entries in the buffers have a fixed size, which corresponds to the size of the largest event that the writer can produce. To transport variable-sized data (e.g., strings) with fixed-size events, we transport them in a separately managed shared memory and represent them in the events by 64-bit identifiers.

Arbiter and monitor buffers are also concurrent single-producer single-consumer lock-free buffer, and share a large part of the implementation with source buffers. The main difference is that they are not created in shared-memory and the arbiter buffers implement the autodrop functionality.

4.1.1. VAMOS event sources

Along with the libraries, VAMOS ships also a set of pre-defined event sources. At the time of writing the paper, these include:

- Several generic event sources that intercept the `read` and `write` (or any other) system calls. These tools allow specifying an arbitrary number of regular expressions that are matched against the traced data and turn them into events. They are implemented with the help of either *DynamoRIO* [17] (a dynamic instrumentation framework) or the *eBPF* subsystem of the Linux Kernel [18–20].
- Event sources that read data from file descriptors and parse the data into events using regular expressions, as in the previous point.
- An event source that captures hardware input events (mouse, keyboard) using the library *Libinput* [21].
- An event source that serves as a transparent proxy to trace communication between *Wayland* [15] graphical server and its clients.
- An LLVM [22] based instrumentation tool to implement data race detection monitor in our experiments (Section 5.3.2).

Example uses of these tools are included in our artifact [23]. Users can implement new event sources with the help of VAMOS libraries that provide a straightforward API to create and fill shared-memory buffers.

4.2. The Vamos compiler

The compiler takes a VAMOS specification described in the previous sections and turns it into a C program implementing the monitoring system, i.e., it generates the code for all components shown in Fig. 1. It does some minimal checking, for example whether events used in parts of the program correspond to the expected stream types, but otherwise defers type-checking to the C compiler. The generated program expects a command-line argument for each specified event source, providing the name of the source buffer created by whatever actual event source is used. Event sources signal when they are finished, and the monitor stops once all event sources are finished and all events have been processed.

Finally, the compiler generates also a script that compiles the generated C file into an executable file and links it against the VAMOS libraries.

4.2.1. The TeSSLa connector

A recent version of TeSSLa [16] can generate monitors in the form of Rust code. This code exposes an interface to send events to the monitor and drive the stream processing. Our compiler can generate the necessary bridging code and replace the *monitor* component in VAMOS with such a TeSSLa Rust monitor. We used TeSSLa as a representative of higher-level monitoring specification tools; in principle, one could similarly use other standard monitor specification languages, thus making it easier to connect them to arbitrary event sources.

5. Evaluation

We conducted a set of experiments with our implementation of VAMOS that aim to answer these research questions:

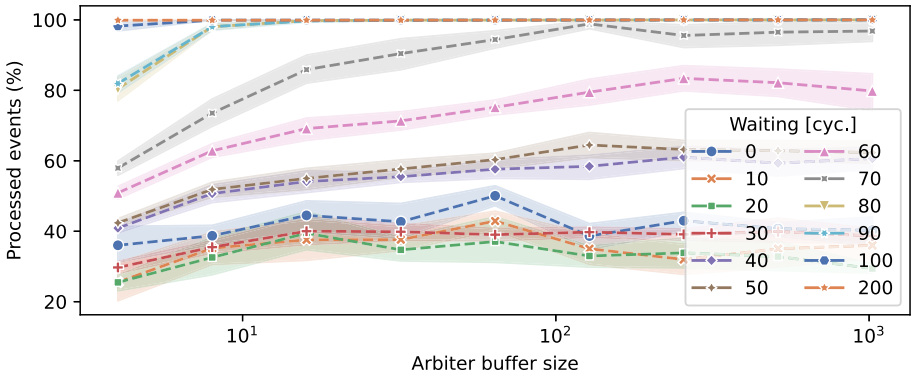


Fig. 8. The percentage of events that reached the final stage of the monitor in a stress test where the source sends events rapidly. Parameters are different arbiter buffer sizes (x-axis) and the delay (*Waiting*) of how many empty cycles the source waits between sending individual events. The shading around lines shows the 95% confidence interval around the mean of the measured value. The source buffer was 8 pages large, which corresponds to a bit over 1 300 events.

RQ1: How efficient is the implementation of VAMOS?
RQ2: Does VAMOS facilitate best-effort monitoring?
RQ3: Is VAMOS applicable in different, real-world scenarios?

We answer the research questions in the following subsections.

Experimental setup All experiments were run on a common personal computer with 16 *GB* of RAM and an *Intel(R) Core(TM) i7-8700* CPU with 6 physical cores running on 3.20 *GHz* frequency. Hyper-Threading was enabled and dynamic frequency scaling disabled. The operating system was Ubuntu 20.04. All provided numbers are based on at least 10 runs of the relevant experiments and the measured time is wall time in seconds.

5.1. RQ1: the performance of VAMOS

To answer **RQ1**, we consider two benchmarks: one is an evaluation of throughput where we send events rapidly from an event source to the monitor. In this benchmark, we measure how long it takes to transfer an event through the whole pipeline all the way to the monitor, and how many events get dropped (for different runtime parameters). The other benchmark is differential monitoring of two primes generators that generate a lot of events in a short time. Here we measure the overhead of our instrumentation.

5.1.1. Measuring throughput

In this experiment, an event source sends 10 million events carrying a single 64-bit number (plus 128 bits of metadata), waiting for some number of cycles between each event. The performance layer forwards the events to autodrop buffers of a certain size, the arbiter retrieves the events, including holes, and forwards them to the monitor, which keeps track of how many events it saw and how many were dropped. We varied the number of cycles and the arbiter buffer sizes to see how many events get dropped because the arbiter cannot process them fast enough—the results can be seen in Fig. 8.

At about 70 cycles of waiting time, almost all events could be processed even with very small arbiter buffer sizes (4 and up). In our test environment, this corresponds to a delay of roughly 700 *ns* between events, which means that VAMOS is able to transmit approximately 1.4 million of events per second all the way from the event source through all the buffers to the monitor.

5.1.2. Differential monitoring of primes generators

In this benchmark, the monitoring system compares two parallel runs of a program that generates streams of primes and prints them to the standard output, simulating a form of differential monitoring [24]. The task of the monitor is to compare the output of the programs and alert the user whenever the two programs generate different data. Each output line is of the form `#n: p`, indicating that `p` is the `n`th prime. This is easy to parse using regular expressions, and our DynamoRIO-based instrumentation tool simply yields events with two 32-bit integer data fields (`n` and `p`).

While being started at roughly the same time, the programs as event sources run independently of each other, and scheduling differences can cause them to run out of sync quickly. To account for this, a VAMOS specification needs to allocate large enough buffers to keep enough events to make up for possible scheduling differences. The arbiter uses the event field for the index variable *n* to line up events from both streams, exploiting the buffer group ordering functionality described in Section 3.4.5 to preferentially look at the buffer that is "behind", but also allowing the faster buffer to cache a limited number of events while waiting for events to show up on the other one. Once it has both results for the same index, the arbiter forwards a single pair event to the monitor to compare them.

Fig. 9 shows results of running this benchmark for different number of generated primes (from 10 000 to 40 000) and for arbiter buffer sizes ranging between 128 and 2024 events. The left plot compares the running time of prime generators with and without

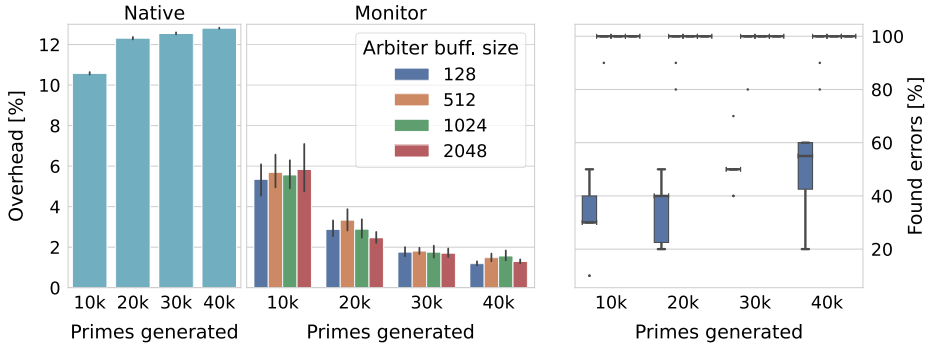


Fig. 9. Overheads (left) and percentage of found errors (right) in the primes benchmark for various numbers of primes and arbiter buffer sizes relative to DynamoRIO-optimized but not instrumented runs. DynamoRIO was able to optimize the program so much that the native binary runs slower than the instrumented one. (For colored versions of the figures, the reader is referred to the web version of this article.)

monitoring (*Native* and *Monitor*, resp). Because our instrumentation uses DynamoRIO, we use the time of running the prime generators with DynamoRIO without any instrumentation as the baseline. DynamoRIO is primarily a runtime optimization tool, and in these experiments, the generators run slower without DynamoRIO than with DynamoRIO, which results in the overhead around 10–12% shown in the *Native* barplot. Instrumented and moniored prime generators executed under DynamoRIO (barplot *Monitor*) have smaller overhead than *Native* (that runs without DynamoRIO). This overhead does not differ significantly between different arbiter buffer sizes and longer runs amortize the initial cost of dynamic instrumentation – for them, the overhead of instrumentation settles on around 2%.

Note that this benchmark gives us also information about the overall performance of VAMOS, not only about instrumentation: if the arbiter or monitor cannot process events fast enough, it increases the overhead of instrumentation because it has to wait for space in source buffers which are blocking.

5.2. RQ2: best-effort monitoring with VAMOS

To answer **RQ2**, we extend the benchmark about differential monitoring prime generators from the previous section and measure how many errors is the (best-effort) monitor that compares the two streams able to find for different setups of the experiment.

5.2.1. Differential monitoring of primes generators

Here we continue the differential monitoring benchmark from the previous section. Except for measuring the instrumentation overhead, we also created a setting where one of the programs generates a faulty prime about once every 10 events and we measured how many of these discrepancies the monitor can find (which depends on how many events are dropped). Results of this part of the experiments are shown in the right plot of Fig. 9. Unsurprisingly, larger buffer sizes are better at balancing out the scheduling differences that let the programs get out of sync. The plot suggests that, in these experiments, the arbiter buffer size that is enough to counter the desynchronization of programs is around 512 elements – for this arbiter buffer size, the monitor is able to catch most (if not all) the errors.

5.2.2. Differential monitoring of primes generators with a TeSSLa monitor

We experimented with a variation of the benchmark that uses a very simple TeSSLa [25,4] specification receiving two streams for each prime generator (i.e., four streams in total): one stream of indexed primes as in the original experiment, and the other with hole events. The specification expects the streams to be perfectly lined up and checks that, whenever the last-seen pairs on both streams have the same index, they also contain the same prime (and ignores non-aligned pairs of primes). We wrote three variants of an arbiter to go in front of that TeSSLa monitor:

1. the *forward* arbiter just forwards events as they come; it is equivalent to writing a script that parses output of generators and (atomically) feeds events into a pipe from which TeSSLa reads events;
2. the *alternate* arbiter always forwards the event from the stream where we have seen fewer events so far; if streams happen to be aligned (that is, contain no or only equally-sized *hole* events), the events will perfectly alternate;
3. the *align* arbiter is the one we used in our original implementation to intelligently align both streams.

Fig. 10 shows the impact of these different arbiter designs on how well the monitor is able to do its task, and that indeed more active arbiters yield better results—without them, the streams are perfectly aligned less than 1% of the time. While one could write similar functionality to align unsynchronized streams in TeSSLa directly, the specification would become complicated. Streams in TeSSLa are defined through (possibly mutually recursive) stream equations. To correct misalignments, especially ones that may change dynamically, requires assembling a stream from dynamically changing offsets into other streams. VAMOS allows to outsource the alignment of events into a pre-processing phase which in turn enables simple specifications in a higher-level monitoring language.

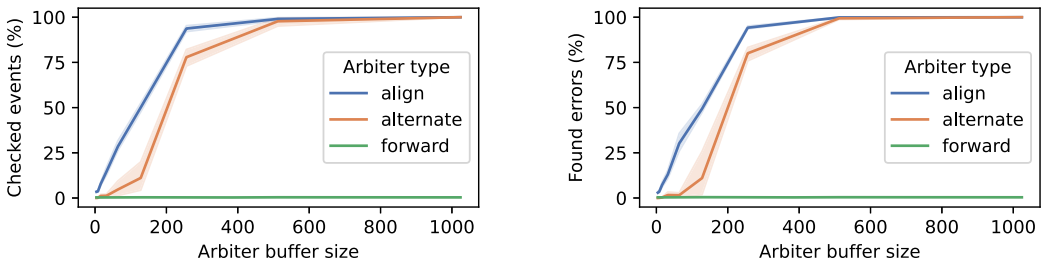


Fig. 10. The percentage of primes checked and errors found (of 40 000 events in total) by the TeSSLa monitor for different arbiter specifications and arbiter buffer sizes.

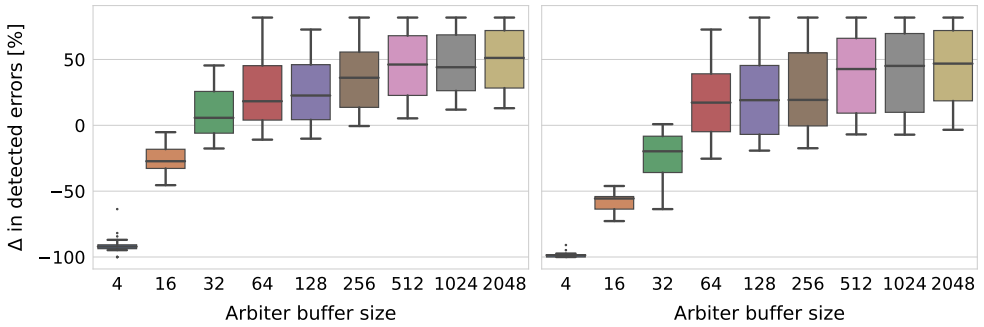


Fig. 11. Results of monitoring a simple banking simulator with VAMOS monitor (left) and TeSSLa monitor (right). Boxplots show the difference in the number of reported errors versus the number of errors the application made, in percent.

5.3. RQ3: different usage scenarios of VAMOS

In this section, we show several experiments with different usage scenarios for VAMOS. Although we measure some statistics that we later use to answer all the research questions, the main point of this section, which should most importantly answer **RQ3**, is to show the variation of the use cases where VAMOS can be applied.

5.3.1. Monitoring a simplified banking application

In this classical runtime monitoring scenario, we wrote an interactive console application simulating a banking interface. Users can check bank account balances, and deposit, withdraw, or transfer money to and from various accounts. The condition we want to check is that no operations should be permitted that would allow an account balance to end up below 0.

We use an event source that employs DynamoRIO [17] to dynamically instrument the program and captures its inputs and outputs. The captured data is parsed and relevant information is turned in events that are sent to the monitor. The monitor starts with no knowledge about any of the account balances (and resets any gathered knowledge when hole events indicate that some information was lost), but discovers them through some of the observations it makes: the result of a check balance operation gives precise knowledge about an account's balance, while the success or failure of the deposit/withdraw/transfer operations provides lower and upper bounds on the potential balances. For example, if a withdrawal of some amount fails, this amount provides an upper bound on an account's balance, and any higher successive withdrawal attempt must surely fail too.

In the spirit of third-party monitoring, however, the stateful interface does not necessarily make it easy to derive these higher level events. For example, there is no individual confirmation that says that the withdrawal of some amount from some account was successful or not. Instead, the user selects an account, then the withdraw action, is then prompted which amount they would like to withdraw from said account, and after entering said amount, the system only displays a message that the withdrawal failed or was successful. The event source parses each individual step and provides them on two separate streams, one for the inputs and one for the outputs. This is where VAMOS' higher-level event recognition capabilities (see also the example in Section 3.4.4) allow the arbiter to recognize the higher-level events to forward to the monitor, which itself is therefore again much easier to specify.

To conduct measurements, we randomly generated 10 000 (well-formed) inputs and fed them to the banking application as fast as possible. We also let the application generate erroneous outputs (wrong balances, swapping success and failure messages) at random and measured how many of those our best-effort third-party monitor was able to detect. The size of the source buffer was 128 events and we varied the size of arbiter buffers from 4 to 2048.

The heavy-weight instrumentation we used in this scenario caused the banking application to run through its script about 40% slower than without instrumentation for all sizes of the arbiter buffer, which is more than in our other benchmarks. We could still optimize this overhead, but for a real user interacting with the application, this overhead becomes invisible because the application waits for the user inputs most of the time. The interesting metric here is how many errors the monitor actually detects. Fig. 11 shows this for both the monitor we described above and a TeSSLa variant that only considers exact knowledge about account balances

(no upper or lower bounds) and thus finds fewer errors, demonstrating both an alternate monitor design and the use of our TeSSLa connector. The results vary quite a bit with arbiter buffer sizes and between runs, and the monitor may report more errors than were inserted into the run. This is because, first, especially with smaller buffer sizes, the autodrop buffers may drop a significant portion (up to 60% at arbiter buffer size 4 and 5% at size 256) of the events, but the monitor needs to see a contiguous chunk of inputs and outputs to be able to gather enough information to find inconsistencies. Second, some errors cause multiple inconsistencies: when a transfer between accounts is misreported as successful or failed when the opposite is true, the balances (or bounds) of two accounts are wrong. Overall, both versions of the monitor were able to find errors with even smaller sizes of arbiter buffers, and increasing numbers improved the results steadily, matching the expected properties of a best-effort third-party monitor.

5.3.2. Data race detection

While our other benchmarks were written artificially, we also used VAMOS to develop a best-effort data race monitor. Most tools for dynamic data race detection use some variation of the *Eraser* algorithm [26]: obtain a single global sequence of synchronization operations and memory accesses, and use the former to establish happens-before relationships whenever two threads access the same memory location in a potentially conflicting way. This entails keeping track of the last accessing threads for each location, as well as of the ways in which any two threads might have synchronized since those last accesses. Implemented naïvely, every memory access causes the monitor to pause the thread and atomically update the global synchronization state. Over a decade of engineering efforts directed at tools like ThreadSanitizer [27] and Helgrind [28] have reduced the resulting overhead, but it can still be substantial.

We implemented a version of the Goldilocks [29] algorithm (a variant of Eraser [26]) that we connect to programs using VAMOS. We use VAMOS not to only connect to programs, but also to pre-process the events, because Goldilocks algorithm requires as input a single global stream of events.

To build our event sources, we used ThreadSanitizer's source-code-based instrumentation[1] to instrument relevant code locations. Based on our facilities for dynamically creating event sources, each thread creates its own event source to which it sends events. Every event is assigned a global timestamp obtained by atomically increasing a counter.

In the correctness layer, the arbiter builds the single global stream of events based on the timestamps that is forwarded to the Goldilocks monitor. The biggest complication here is deciding when to abandon looking for the next event to put to the output stream if it may have been dropped.

To avoid slowing down the analyzed program more than necessary, we use autodrop arbiter buffers. When an autodrop buffer drops some events of a thread, we only report data races that the algorithm finds if all involved events were generated after the last time that events were dropped. This means that our tool may not find some races. However, it still found many races in our experiments, and other approaches to detecting data races in best-effort ways have similar restrictions [30].

Our implementation (contained in our artifact [23]) consists of:

- a straightforward translation of the pseudocode in [29], using the C++ standard library `set` and `map` data structures, with extensions to handle holes;
- VAMOS specification to retrieve events from the variable number of event streams in order of their timestamps to forward to the monitor;
- an LLVM [22] instrumentation pass post-processing ThreadSanitizer's instrumentation to produce an event source compatible with VAMOS.

Altogether, we were able to build a reasonable best-effort data-race monitor that uses VAMOS with relatively little effort. To evaluate its performance, we tested it on 391 SV-COMP [31] concurrency test cases supported by our implementation, and compared it to two state-of-the-art dynamic data race detection tools, ThreadSanitizer [27] and Helgrind [28]. The timeout was set to 20 s. Fig. 12 shows that the monitor that uses VAMOS in most cases caused less overhead than both ThreadSanitizer and Helgrind in terms of time while producing largely the same verdicts.

On the side of running time, the monitor implemented with VAMOS has the advantage that it is parallelized, because the monitoring system is a separate process distinct from the monitored program. Even though, there are cases when ThreadSanitizer was faster and it is no surprise – although setting up the VAMOS' shared-memory buffers and connecting to them from the monitoring system is very fast, it still takes some time during which the program is blocked. For programs that execute really quickly, this startup overhead dominates the running time. Helgrind was never faster than VAMOS, which is also not that surprising, because it disassembles, instruments, and then recompiles the executed code *dynamically* at runtime. It also has to perform expensive lookups of the code that it has already recompiled.

As for the verdicts, all the tools gave very similar results. Helgrind reached timeout in 3 cases. If we let it run longer, it was able to decide those programs (2 were racy and one not). Then there were two racy programs where only our monitor found a race. For these, Helgrind was not able to find the race and ThreadSanitizer gave inconsistent results – in all trials in our experiments, it did not find the race. When we executed the programs later manually under ThreadSanitizer, it was able to find the race eventually after many trials. This can happen, because ThreadSanitizer is also a best-effort tool and might not report all data races everytime.

[1] This decision was entirely to reduce our development effort; a dynamic instrumentation source could be swapped in without other changes.

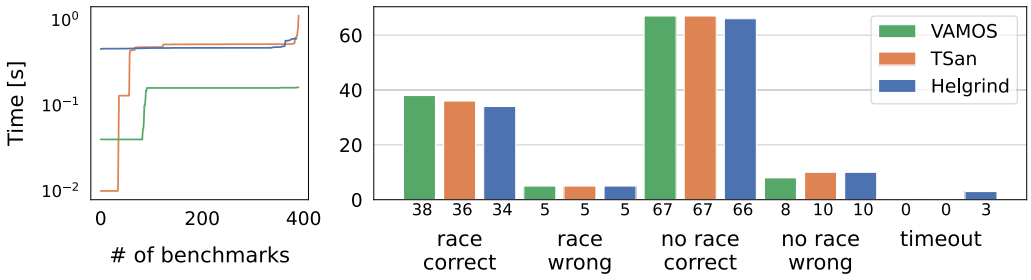
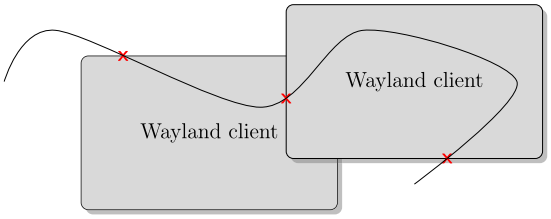


Fig. 12. Comparing running times of the three tools on all 391 benchmarks (left) and the correctness of their verdicts on the subset of 118 benchmarks for which it was possible to determine the ground truth (right). *Race* vs. *no race* indicates whether the tool found at least one data race in some trial, *correct* vs. *wrong* indicates whether that verdict matches the ground truth. For benchmarks with unknown ground truth, the three tools agreed on the existence of data races more than 99% of the time.



Client 1: 12.0:(50, 0), 12.2:(51, 1), 12.3:(52, 1), 12.5:(53, 2) ...
Client 2: 58.3:(0, 130), 58.5:(1, 131), 58.8:(2, 132), 58.9:(3, 133) ...
Libinput: 7.2:(+0.3, -0.1) ... 12.1:(+0.2, -0.2), 12.2:(+0.2, -0.3) ...

Fig. 13. The pointer integrity monitor checks that whenever a client is notified about a change of the cursor position, there is a sequence of Libinput events that can lead to this change. Blue are the timestamps; Libinput events carry the deltas of the global cursor position while Wayland events carry coordinates relative to the client's window and are received on the corresponding stream only when the cursor is above the client.

5.3.3. Monitoring Wayland connections

Wayland [15] is a graphical protocol meant as a replacement of X Window System protocol which is almost 40 years old at the time of writing this text. It uses a server-client architecture where the server notifies clients about events occurring in the system (for example that a key was pressed while the client was focused). In turn, clients can send requests to the server to draw things on the screen.

In this experiment, we use VAMOS to do pointer motion integrity checking. Our monitor traces the information about pointer motion from the hardware and matches it with information that is sent by the Wayland server to the clients. If the integrity check is violated, it might mean that there is a bug in the Wayland server, or it can be also a result of some malware corrupting the system. In contrast to our other experiments, this is a case where a monitor connects to truly heterogeneous event streams.

We use two kinds of event sources: one that traces the communication between Wayland clients and Wayland server, and one that traces pointer motion events generated by the Linux kernel. The event source that listens for the communication between Wayland clients and the server works as a transparent proxy: it intercepts any message between a Wayland client and the server and copies the interesting ones to VAMOS. The stream of events generated by the kernel is based on the Libinput library [21].

There is one (dynamically created) event stream per Wayland client and a single stream of events from Libinput. The underlying systems do not use a common set of identifiers we could use to match up events, so our event sources tag events with millisecond-precise timestamps, which allows us to order and match them approximately. Obtaining a total order is impossible, though, since the resolution of the timestamps is not fine-grained enough and the timestamps can be distorted by layers of code running around generating the events. We therefore have to deal with what essentially amounts to *clock skew*, and we assume the skew is limited by some value τ, an assumption also used in other work [32].

Although all our event streams contain information about pointer motion, the streams of Wayland events are fundamentally different from the Libinput stream. A Wayland client gets notified when the pointer cursor enters and leaves its window, and the coordinates of any cursor movement events it receives in between those times are relative to the upper left corner of the window. When the cursor is not over a Wayland client, this client has no means to find out where the cursor is, and the client itself has no information where it is on the screen. In contrast, Libinput supplies us with information that the user moved the mouse/touchpad a certain number of pixels in a certain direction. That same information is given to the Wayland server, indicating that it should move the cursor on the screen relative to the current position of the cursor. Multiple Libinput events may be aggregated into one Wayland motion event. The situation is depicted in Fig. 13.

Our monitoring system works as follows. The monitor orders Wayland buffers by the timestamp of their top event. Because the cursor can only be in one window at a time, the events are naturally grouped into segments belonging to a particular buffer (the

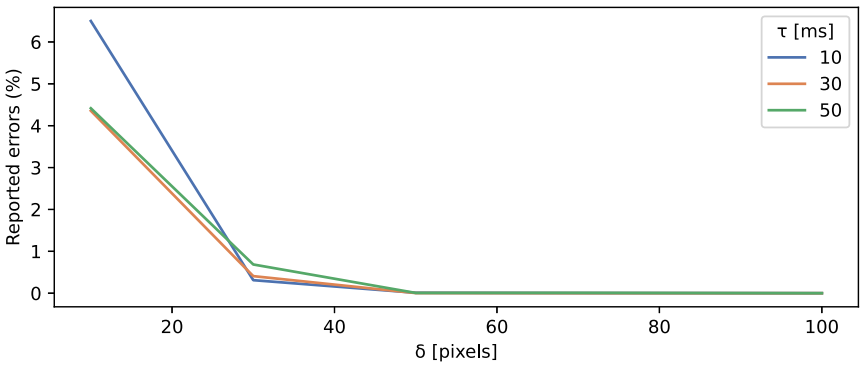


Fig. 14. Results of monitoring integrity of pointer motion information sent to Wayland clients. τ is the clock skew parameter and δ is the allowed imprecision in estimated pointer movement.

boundaries of those segments are depicted by red crosses in Fig. 13). The arbiter forwards all Wayland events that constitute a segment to the monitor. While doing that, it also forwards all Libinput events whose timestamp differs at most by τ from the last sent Wayland event. For simplicity, there is no buffering or double-counting, so Libinput events may be dropped because they arrived too late or may not be associated to a segment because they have been already associated to the previous one; this is still in line with a best-effort approach. Any hole event is simply forwarded to the monitor and causes it to stop checking and wait for the next segment where it can start anew.

The monitor processes every segment where it considers every two consecutive Wayland motion events, i.e., every two consecutive positions of the cursor inside the Wayland client, and checks if it can get from the first position to the second position using a sequence of Libinput motion events. Because there might be relevant Libinput motion events missing in the segment and the server may also do some rounding of coordinates, we allow for imprecision: we say that the integrity holds if we can get from the first position to a position that is not further than δ pixels from the target position, where δ is a parameter of the monitor.

We conducted experiments with the Weston compositor [15] (Wayland reference server). After starting Weston, clients, and the monitor, we replayed a pre-recorded sequence of pointer motions. This replay is completely transparent to Weston and the monitor – they see the events coming from the operating system kernel as if they were not pre-recorded.

The frequency of events on the streams is in this use-case much lower than in the other experiments. Although we set the arbiter buffer size to 3, the possible minimum accepted by the implementation, holes occurred only sporadically. Fig. 14 shows how many integrity violations are reported by the monitor depending on the parameters τ and δ. While τ has basically no effect on the results, δ, the allowed imprecision in checking the transition from one position to another, is important. The number of reported violations (which we assume are false positives here) is under 1% of the number of all position-to-position comparisons even for small values of δ. Increasing it to around 30 pixels, we already get very close to 0 reported violations. For finding subtle bugs, this monitor may not be good enough, but it is a reasonable start when checking the integrity of the pointer for security reasons, because "hijacking" the cursor in order to perform an abusive action would very likely lead to a diversion of the cursor by more than 30 pixels.

5.4. Discussion

Based on the results presented in this section, we believe that the answers to our research questions are all positive:

RQ1: how efficient is the implementation of VAMOS The experiments with throughput and primes generators show that the implementation of VAMOS is efficient and it can process high thousands of events per second. Our instrumentation based on DynamoRIO was very efficient in experiments with prime generators and reasonably efficient also in the experiments with the simulated banking application.

RQ2: does VAMOSfacilitate best-effort monitoring? The experiments with prime generators suggest that VAMOS facilitates best-effort monitoring: the best-effort monitors created using VAMOS are able to detect most of the errors with the right setup of the parameters. Moreover, VAMOS can be used to prepare events in order to use simpler monitor specifications as shown in the experiments with primes generators and TeSSLa monitor. Finally, the experiments with the banking application, data races, and monitoring Wayland clients all use best-effort monitors that were able to detect reasonable amount (in the case of the banking application) or most of (in the case of the other two benchmarks) errors. The use of VAMOS in these benchmarks also simplifies the monitors of the system by pre-processing the events (e.g., doing event recognition) – at the cost of turning the monitors into best-effort monitors (because they have to handle possibly dropped events).

RQ3: is VAMOSapplicable in different, real-world scenarios? Experiments with banking application, data race detection and Wayland connection monitoring suggest that VAMOS is applicable in different, even real-world, scenarios.

Getting back to our goals stated in the introduction: performance, flexibility, and ease-of-use, we believe that the experiments suggest that the first two goals were reached.

As for the ease-of-use, there is no good metric to measure this. Our specification language is still rather low-level in that it gives a lot of control to the user. The specifications written in this language are therefore not necessarily short. However, they are much shorter than the C code generated from them. In our experiments, the size of specifications ranged from 12% to 21% of the size of the generated code (measured in lines of code, without the code in VAMOS libraries). The time of setting up the monitoring system with VAMOS should be decreased simply by the fact that VAMOS provides components that are repeatedly implemented in every monitoring setup: tracing and preparing events for the monitor. Of course, this argument gets weaker if VAMOS does not have a suitable pre-defined event source to be used. But even then, writing a new event source using VAMOS' libraries should be faster than writing it from the scratch – again because VAMOS provides the necessary components for this.

5.4.1. Threats to validity

There are many aspects that can influence the performance of parallel software like VAMOS on modern computers. Our implementation explicitly aligns memory where suitable to avoid problems like false sharing, but our solutions may not work on all architectures and in all environments. Our experiments that assess the performance of VAMOS do not consider all possible scenarios and the results may not transfer to situation where, e.g., the resources are constrained (for example, when there is not enough physical CPU cores to run truly in parallel).

Also, none of our experiments considers situations with a huge number of event streams. In the current design of VAMOS, we have one thread and two buffers per an event stream, which means that a huge number of event streams will consume a lot of resources and the number of threads could pose a problem to the system. Therefore, we can expect that our results do not carry over to such scenarios.

Our experiments with data race detection were conducted only on a relatively small set of benchmarks. Although these were the best benchmarks that we could find, the results still may be biased by this selection. Similarly for the experiments with Wayland clients, the results are based on pointer motions pre-recorded by one of the authors and one might argue that this is not a representative sample.

Finally, there is always a chance of a bug in our implementation. We have extensively tested the implementation of buffers (including bounded verification with the bounded model checker CBMC [33]), but the compiler for the specification language has undergone only manual testing on hand-written specifications.

6. Related work

As mentioned before, VAMOS' design features a combination of ideas from works in runtime monitoring and related fields, which we review in this section.

Event brokers/event recognition A large number of event broker systems with facilities for event recognition [8,9,11–13] already exist. These typically allow arbitrary event sources to connect and submit events, and arbitrarily many observers to subscribe to various event feeds. Mansouri-Samani and Sloman [34] outlined the features of such systems, including filtering and combining events, merging multiple monitoring traces into a global one, and using a database to store (parts of) traces and additional information for the longer term. Modern industrial implementations of this concept, like Apache Flink [13], are built for massively parallel stream processing in distributed systems, supporting arbitrary applications but providing no special abstractions for monitoring, in contrast to more runtime-monitoring-focused implementations like ReMinds [9].

ReMinds [9], compared to VAMOS, is a heavy-weight solution implemented in Java that focuses on industrial systems, possibly distributed over network. It is designed to process events from multiple heterogeneous sources, aggregate and store them to a persistent storage, and distribute them to different analysis and visualization components. VAMOS, on the other hand, is designed to run locally on a personal computer (although event sources could be scattered across a network, too), and the use case scenarios we have in mind are rather monitoring application(s) used by a common computer user. Note that this does not remove the requirement on heterogeneous event sources as today's applications often consist of many different processes and interact with outside world. As a result, VAMOS is designed to handle much faster event flows – to compare, VAMOS assumes scenarios with hundreds of thousands of events per second, while the experimental evaluation of ReMinds uses at most 1000 events per second [9].

Complex event recognition systems also sometimes provide capabilities for load-shedding [35], of which autodrop buffers are the simplest version.

Stream run-time verification LOLA [36,37], TeSSLa [4], and Striver [38] are stream runtime verification [3] systems that allow expressing a monitor as a series of mutually recursive data streams that compute their current values based on each other's values. This requires some global notion of time, as the streams are updated with new values at time ticks and refer to values in other streams relative to the current tick, which is not necessarily available in a heterogeneous setting.

Stream runtime verification systems also do not commonly support handling variable numbers of event sources. LOLA2.0 [37] has the abilities to spawn and close additional streams in response to certain events, but due to the synchronized nature of LOLA, this feature is more a logical abstraction than a true separation of both logical and computational concerns.

Some systems allow for dynamically instantiating sub-monitors for parts of the event stream [39,40,37,41] in a technique called *parametric trace slicing* [42]. This is used for logically splitting the events on a given stream into separate streams, making them easier to reason about, and can be exploited for parallelizing the monitor's work. These additional streams are internal to the monitoring

logic. In contrast, VAMOS' ability to dynamically add new event sources relates to new sources of data appearing inside the monitored system (like a new thread), and the arbiter unifies the events coming in from all such sources into one global stream.

Finally, BeepBeep 3 [43], combines many of the traditional stream runtime verification features with those of complex event processing systems. Its key difference to VAMOS is that BeepBeep's focus is almost exclusively on *expressiveness*, providing a Java-based modular framework to combine complex event processing stages with more classical RV-style reasoning about the resulting traces. It furthermore requires that values on two incoming streams are matched synchronously. While this does technically get rid of timestamping issues, it essentially timestamps all events sequentially. As such, it would be less well-suited for heavy-load online tasks like our data race monitoring example. VAMOS, in contrast, is designed as middleware rather than a higher-level specification language, and focuses on efficiently connecting event sources to a higher-level monitor, which in turn could be a BeepBeep 3 monitor. VAMOS itself neither relies on timestamps nor on any sort of synchronicity or cardinality assumptions; individual specifications, however, may do so. For example, our data race detection case study (Section 5.3.2) relies on timestamps generated by the event sources to establish the order of events arriving from different threads. In this scenario, there effectively exists a global clock.

Instrumentation The two key questions in instrumentation revolve around the technical side of how a monitor accesses a monitored system as well as the behavioral side of what effects these accesses can have. On the technical side, static instrumentation can be either applied to source code [44–47,22,48] or compiled binaries [49,50], while dynamic instrumentation, like DyanmoRIO, is applied to running programs [51,52,17]. Alternatively, monitored systems or the platforms they run on may have specific interfaces for monitors already, such as PTrace and DTrace [18–20] in the Linux kernel. Any of these can be used to create an instrumentation tool for VAMOS.

On the behavioral side, Cassar et al. surveyed various forms of instrumentation between completely synchronous and offline [1]. Many of the systems surveyed [53–56] use a form of static instrumentation that can either do the necessary monitoring work while interrupting the program's current thread whenever an event is generated, or offer the alternative of using the interruption to export the necessary data to a log to be processed asynchronously or offline. A mixed form called *Asynchronous Monitoring with Checkpoints* allows stopping the monitored system at certain points to let the monitor catch up [57]. Our autodrop buffers instead trade precision for avoiding this kind of overhead. Aside from the survey, some systems (like TeSSLa [4]) incrementalize their default offline behavior to provide a monitor that may eventually significantly lag behind the monitored system.

Executing monitoring code or even just writing event data to a file or sending it over the network is costly in terms of overhead, even more so if multiple threads need to synchronize on the relevant code. Ha et al. proposed Cache-Friendly Asymmetric Buffering [58] to run low-overhead runtime analyses on multicore platforms. They only transfer 64-bit values, which suffices for some analyses, but not for general-purpose event data.

There are instrumentation approaches, especially in real-time systems, that dynamically adjust the overhead of instrumentation by deactivating some of its parts according to some runtime metrics, inserting hole events for phases when instrumentation is deactivated [59–61]. In contrast, the goal of holes generated by VAMOS' autodrop buffers is to ensure that the monitor is working with reasonably up-to-date events while not forcing the monitored system to wait. For many monitors, the two approaches could easily be combined.

Monitorability, uncertainty, and missing events Monitorability [62,2] studies the ability of runtime monitors to produce verdicts about the monitored system. The possibility of missing events on an event stream significantly reduces the number of monitorable properties [63]. The *autodrop* buffers of VAMOS insert *hole* information, which some monitors can automatically use to yield useful verdicts even when events are missing [5–7,64].

Uncertainty may also affect the data within events, even if they are not lost. The most common case of this are imprecise timestamps, which means one cannot be sure about the real order of the incoming events. Some work has suggested using SMT solvers to work with many plausible orderings at once [65,32]. This is typically expensive, though VAMOS's decoupling of expensive monitoring computations from the performance layer may enable more use of SMT solvers in runtime monitoring, particularly where best-effort approaches are reasonable.

7. Future work

Except for applying VAMOS to more diverse application scenarios, we would like to integrate VAMOS with higher-level monitoring languages/specifications. There are also new features to be considered.

Multiple arbiters/monitors Having a single arbiter and a single monitor may be a bottleneck. In some scenarios, different kinds of events might be processed in parallel, plausibly with multiple arbiters and monitors. For example, in our banking scenario, most events on individual accounts can be processed independently by a monitor for that specific account (the exception being transfers between two accounts). The challenge in adding multiple arbiters and monitors is to keep consistency between event streams or to efficiently duplicate them, and to synchronize efficiently between arbiters and monitors in those cases where they actually do affect shared state.

Enforcement So far, VAMOS only supports observing events and producing verdicts about them. Enforcement [66] is another feature of runtime monitors, giving the monitor power to influence the monitored system. The possibility of enforcement of course always depends on the event source—a bank's API to receive transaction information will typically not allow arbitrarily changing this

Event	E	Stream Type	T	Event Source	S
Buffer Group	B	Stream Processor	P	Variable/Field	X
		Aggregation Operation	O		

Fig. A.15. Identifiers.

transaction history. The key challenge is that for enforcement, the monitor must typically run synchronously with the monitored system, which goes against the asynchronous design of VAMOS. If the monitor runs asynchronously with the monitored system, an enforcement action may be issued long after the system got into the state that made the monitor want to change its behavior, and the state of the system may have changed since then.

8. Conclusion

We have presented VAMOS, which we designed as a middleware for best-effort third-party runtime monitoring that should simplify connecting monitors to sources of events, particularly for best-effort third-party runtime monitoring, which may often need some significant pre-processing of the traced information, potentially collected from multiple heterogeneous sources. We have presented several experiments that suggest that the way we built VAMOS can handle large numbers of events and lets us use VAMOS in a variety of use cases. In future work, we plan to apply VAMOS to more diverse application scenarios, such as multithreaded webservers processing many requests in parallel, or embedded software, and to integrate our tools with higher-level monitoring languages. If a system's behavior conforms to the expectation of a third party, this is generally recognized as inspiring a higher level of trust than if that monitor was written by the system's developers. We hope that our design can help making best-effort third-party runtime monitoring more common.

CRediT authorship contribution statement

Marek Chalupa: Writing – review & editing, Writing – original draft, Visualization, Software, Investigation, Conceptualization. **Fabian Muehlboeck:** Writing – review & editing, Writing – original draft, Software, Investigation, Conceptualization. **Stefanie Muroya Lei:** Visualization, Software. **Thomas A. Henzinger:** Writing – original draft, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was supported in part by the ERC-2020-AdG 101020093. The authors would like to thank the STTT reviewers for their valuable feedback and suggestions.

Appendix A. Language formalization

This section formalizes the core[2] of VAMOS in terms of an extension of some base language. Our rough notational convention is that Greek letters refer to either the base language or structures of the operational semantics. Patched versions of base language constructions are marked with $\tilde{\cdot}$. Fig. A.15 shows different kinds of identifiers we will use - for simplicity, we assume that all sets of identifiers are distinct and no shadowing occurs. Grammatical non-terminals that define an identifier use $\hat{\cdot}$, e.g. $\hat{S}$ is the non-terminal for an event source definition. Evaluation contexts use blackboard bold, e.g. $\mathbb{E}$, and certain reducible non-terminals that may represent one or more other non-terminals plus a special value $\circ$ use the marker $\dot{\cdot}$, as in $\dot{a}$. We use $\bar{\cdot}$ to mark lists; the operator $+\!\!+$ works both as cons, snoc, and concatenate operation, and [] marks an empty list, which in some cases may also be simply elided.

Our actual language provides additional syntactic shorthands, as well as the option to limit how many dynamically created event sources can be accepted to run at the same time, and a notion of *rule sets* for the arbiter that can be switched between with a special statement; this serves to be easily able to activate and deactivate some rules in one fell swoop (see also the examples in our artifact [23]).

A.1. Base language

We assume some base language with standard imperative features (in our implementation: C), i.e. standard notions of values ν, Expressions ξ, Statements σ, and so on, as in Fig. A.16.

[2] The formalization omits a few high-level constructs like constraining the number of dynamically created streams that can exists at any point in time.

$$
\begin{array}{lll}
\text{Value} & v & \subset\ n \mid \texttt{true} \mid \texttt{false} \mid \ldots \\
\text{Expression} & \xi & \subset\ X \mid v \mid \xi + \xi \mid \ldots \\
\text{Statement} & \sigma & \subset\ \{\overline{\sigma}\} \mid X = \xi \mid \xi \mid \tau X = \xi \mid \texttt{if}(\xi)\ \sigma\ \texttt{else}\ \sigma \mid \ldots \\
\text{Type} & \tau & \subset\ \texttt{int} \mid \texttt{bool} \mid \ldots \\
\text{Exp Eval Ctx} & \mathbb{E} & \subseteq\ \cdot \mid v + \mathbb{E} \mid \mathbb{E} + \xi \mid \ldots \\
\text{Stmt Eval Ctx} & \mathbb{S} & \subseteq\ \cdot \mid \{\mathbb{S} +\!\!+ \overline{\sigma}\} \mid X = \mathbb{E} \mid \tau X = \mathbb{E} \mid \texttt{if}(\mathbb{E})\ \sigma\ \texttt{else}\ \sigma \ldots \\
\text{Base State} & \Pi & \textbf{opaque}
\end{array}
$$

Fig. A.16. Assumed base language structure.

$$
\begin{array}{lllll}
\text{Expression} & \tilde{\xi} & \textbf{extends}\ \xi & ::= & \ldots \mid \$S.X \\
\text{Statement} & \tilde{\sigma} & \textbf{extends}\ \sigma/\tilde{\xi} & ::= & \ldots \mid \$S.X = \tilde{\xi} \\
 & & & \mid & \texttt{drop}\ n\ \texttt{from}\ S \mid \texttt{yield}\ E(\overline{\tilde{\xi}}) \\
 & & & \mid & \texttt{add}\ S\ \texttt{to}\ B \mid \texttt{remove}\ S\ \texttt{from}\ B \\
\text{Stmt Eval Ctx} & \tilde{\mathbb{S}} & \textbf{extends}\ \mathbb{S} & ::= & \ldots \mid \texttt{yield}\ E(\overline{v} +\!\!+ \tilde{\mathbb{E}} +\!\!+ \overline{\tilde{\xi}})
\end{array}
$$

Fig. A.17. Extended language.

$$
\begin{array}{lll}
\text{Event Defn} & \hat{E} & ::= E(\Gamma) \mid E(\Gamma)\ \texttt{creates}\ \overline{T} \\
\text{Stream Type} & \hat{T} & ::= \texttt{stream type}\ T(\Gamma\ ;\ \overline{\hat{X}})\ \texttt{shared}\ E(\Gamma)\ \{\ \overline{\hat{E}}\ \} \\
\text{SP Expression} & e & ::= \texttt{forward}\ E(\overline{\xi}) \mid \texttt{drop} \mid \texttt{if}\ \xi\ \texttt{then}\ e\ \texttt{else}\ e \\
\text{SP Rule} & p & ::= \texttt{on}\ E(\overline{X})\ e \\
 & & \mid\ \texttt{on}\ E(\overline{X})\ \texttt{process using}\ P(\overline{\xi}) \\
 & & \quad \texttt{to}\ k\ \texttt{include in}\ \overline{B}\ e \\
\text{Stream Processor} & \hat{P} & ::= \texttt{stream processor}\ P(\Gamma)\ :\ T(\overline{\xi}) \rightarrow T(\overline{\xi}) \\
 & & \quad \{\overline{p}\ ;\ \hat{H}\} \\
\text{Hole Defn} & \hat{H} & ::= \texttt{hole}\ E\ \{\ \overline{\hat{X}}\ \} \\
\text{Aggregate Defn} & \hat{X} & ::= X\ =\ O(\overline{E.X}) \\
\text{Buffer Kind} & k & ::= \texttt{autodrop}(n,n) \mid \texttt{infinite} \mid \texttt{blocking}(n) \\
\text{Event Source} & \hat{S} & ::= \texttt{event source}\ S(\Gamma)\ :\ T \\
 & & \quad \texttt{process using}\ P(\overline{\xi})\ \texttt{to}\ k \\
\text{Type Env} & \Gamma & ::= \overline{X : \tau}
\end{array}
$$

Fig. A.18. Performance layer.

We extend this language with a few extra forms in some places, shown in Fig. A.17. In particular, for arbiter rules, we add an expression for referring to source fields $\$S.X$, and several kinds of statements: the first can assign a new value to a source field, `drop` drops a number of events from a given buffer between the arbiter and the performance layer, `yield` forwards and event from the arbiter to a monitor, and `add`/`remove` manage membership in buffer groups. Of these, only `yield` makes a difference to evaluation contexts as we compute the data associated with an event that is going to be forwarded to the monitor.

A.2. The performance layer

Fig. A.18 shows the parts of the performance layer. Stream types define a list of events, each of which has a number of fields with name X and type τ, and some of which may signal the creation of an additional event source of some stream type. They also have

1. *Shared fields* (after the `shared` keyword), which are fields that every event of a stream type shares. For pattern-matching purposes, the shared fields precede each individual event's own fields.
2. *Stream fields* (the Γ immediately after the stream's name), which serve to store information about every particular event source. We will discuss those more in the correctness layer.
3. *Aggregate fields* (a special kind of stream field $\hat{X}$) which represent aggregate values over all events that have been seen on a particular event source. These are again to be discussed in the context of the correctness layer.

A stream processor $\hat{P}$ may specify some arguments Γ to use in instantiating the fields of the stream types it uses to describe the events it assumes to receive and produce. This is mostly important for the output stream, as there is currently no way for the stream processor itself to access those fields. The stream processor then provides a list of rule definitions p, which in turn provide a pattern to match any given event against, and a stream processor expression e that is to be evaluated on a match. These expressions are limited to nested branches on base expressions ξ that eventually make a decision whether to drop the current event or forward something to arbiter. A variant of these rules is for those events that signal the existence of a new stream source, in which case the rule contains similar parameters as the specification of event sources (see next). A stream processor also specifies what summarization data to keep for holes, and what event name to use for the eventual hole event. This event's fields are specified similar to events in stream type declarations, except that they also define the summarization operation (e.g. count, sum, max, ...) and which fields of which events

$$
\begin{array}{lll}
\text{Aggregate Expr} & \lambda & ::= O\ X \mid X \mid n \mid \lambda + \lambda \mid \lambda - \lambda \mid \lambda * \lambda \\
\text{Missing Mode} & \omega & ::= \texttt{wait} \mid \texttt{assume}\ \lambda \\
\text{Order Expr} & o & ::= \texttt{round robin} \mid X \mid \texttt{shared}\ X\ \omega \\
\text{Buffer Group} & \hat{B} & ::= \texttt{buffer group}\ B : T\ \texttt{order by}\ o\ \{\overline{S}\} \\
\text{Match Expr} & b & ::= S : \texttt{done} \mid S : \texttt{nothing} \mid S : n \mid S : \overline{E(\overline{X})} \\
\text{Arbiter Rule} & \hat{a} & ::= \texttt{on}\ \overline{b}\ \texttt{where}\ \tilde{\xi}\ \{\ \tilde{\sigma}\ \} \\
 & & \quad \mid \texttt{choose}\ \overline{S}\ \texttt{from first}\ n\ B\ \{\overline{\hat{a}}\} \\
 & & \quad \mid \texttt{choose}\ \overline{S}\ \texttt{from last}\ n\ B\ \{\overline{\hat{a}}\} \\
\text{Arbiter} & \hat{A} & ::= \texttt{arbiter} : T\ \{\overline{\hat{a}}\} \\
\text{Monitor Rule} & \hat{m} & ::= \texttt{on}\ \overline{E(\overline{X})}\ \{\ \sigma\ \} \\
\text{Monitor} & \hat{M} & ::= \texttt{monitor}(n)\ \{\overline{\hat{m}}\}
\end{array}
$$

Fig. A.19. Correctness layer.

are to be used for summarization calculations. By default, every stream type contains one special event "hole" with one 64-bit integer field, and the standard definition for holes in stream processors uses that event to summarize all possible events with the `count` operator (i.e. it counts the dropped events). Custom hole events need to first be specified in the stream processor's output stream type.

Finally, event sources specify the type of events coming in from some source, the processor used to process them, and a buffer kind specifying how much space there is in the buffer between the performance layer and the arbiter for events forwarded by the event processor, and what to do if the buffer is full. Event sources also potentially specify arguments they can be instantiated with, which in turn can be used to instantiate the stream processor and thus the stream fields. Our actual implementation accepts event source definitions using array syntax to specify some fixed number of distinct, but similar event sources, or also a special keyword to allow an arbitrary number of event sources of the same type, depending on the run time parameters of the monitor. We elide the specifics of this in this core formalization.

A.3. The correctness layer

The various components of the correctness layer are shown in Fig. A.19. The correctness layer allows grouping various event source buffers into buffer groups. A definition $\hat{B}$ of such a buffer group specifies the name B of the buffer group, the stream type of the buffers it contains, an order expression o and a list of buffers it contains from the start referred to by their event source name S. The order expression o specifies the order in which event sources should be drawn from the buffer group in arbiter rules, see below.

Event sources can either be fairly ordered by a simple round-robin mechanism, or use a stream field X or a shared field X of the top event (if present) in each buffer. For the latter case, a *missing mode* ω specifies what to do when there is no event in any given buffer. The two options formalized here are to `wait` until such an event arrives, essentially stopping the execution of the arbiter buffer there, or specifying a way to replace a missing value using `assume` with an aggregation expression λ. These aggregation expressions can refer to stream fields and stream aggregation fields, thereby, for example, getting the maximum timestamp that any stream in the buffer group has seen so far.

The arbiter definition $\hat{A}$ is the key part of the correctness layer. It specifies the type of events it might forward to the monitor and a number of rules that might do so. These arbiter rules $\hat{a}$ form arbitrarily branching tree of choose expressions whose leaves are always final matching rules (`on ... where`). The `choose` expressions try instantiating their buffer variables with distinct members of the given buffer group in the order specified by the buffer group; the two different versions of `choose` expressions indicate whether to do this in ascending (`first` or descending `last` order, and whether to limit the instantiation attempts to the first few members (otherwise, one can just provide a large enough n). The final step of rule matching depends largely on buffer match expressions b. Each of them references some event source id S, which is either a defined event source or a variable bound in a `choose` expression. The buffer corresponding to that source can then be constrained to be either `done`, that is, there are no events on that buffer and the event source has indicated no new events will arrive, or `nothing`, which just means there are currently no events in the buffer, or some number n, which means that there need to be at least n events waiting on the buffer, or finally, a pattern match of potentially multiple events that need to match the first few events currently in the buffer. In addition to the buffer match expressions, a `where` condition $\tilde{\xi}$ further restricts when a rule can match. If all conditions are satisfied, the extended statements in $\tilde{\sigma}$ are executed.

In contrast to the arbiter, the monitor definition $\hat{M}$ consists of just an indication of the size of its event queue n and a list of pattern-matching rules $\hat{m}$ that can run arbitrary code, but does not have access to our extended expressions or statements.

A.4. Operational semantics

The (small-step) operational semantics of VAMOS is non-deterministic to reflect the asynchronous nature of the various parts of the program. In particular, the arbiter, the monitor, and every stream processor work on their own threads. Some additional non-determinism in the order in which buffer match expressions are evaluated reflects an openness to certain optimizations, as discussed below. The semantics is initialized with every event that eventually shows up on each event source and as such models the possible behaviors of the system up to some event in the case of an infinite event stream.

$$
\begin{array}{lrcl}
\text{Program} & \Omega & ::= & \langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}}, \hat{A}, \hat{M} \rangle \\
\text{Event} & \epsilon & ::= & E(\overline{v} +\!\!+ \Sigma) \\
\text{Maybe Event} & \dot{\epsilon} & ::= & \circ \mid \epsilon \\
\text{Event List} & \Sigma & ::= & \overline{\epsilon} \\
\text{Choice State} & \psi & ::= & \langle S, \overline{S}, \overline{S} \rangle \\
\text{Choice Stack} & \phi & ::= & \circ \mid \langle \overline{\psi}, \hat{a}, \overline{\hat{a}} \rangle ; \phi \\
\text{Processor Redex} & \dot{e} & ::= & \circ \mid e \\
\text{Processor State} & \rho & ::= & \langle S, P, \Delta, \Sigma, \dot{e}, k, \dot{\epsilon}, \Sigma \rangle \mid \langle S, P(\overline{\xi}), \Sigma, k, \overline{X \mapsto \xi}, \overline{B} \rangle \\
\text{Arbiter Redex} & \dot{a} & ::= & \circ \mid \hat{a} \mid \overline{\tilde{\sigma}} \\
\text{Arbiter Local State} & \varsigma & ::= & \langle \overline{B}, \overline{\rho}, \overline{\hat{B}} \rangle \\
\text{Arbiter State} & \alpha & ::= & \langle \dot{a}, \overline{\hat{a}}, \phi, \varsigma \rangle \\
\text{Monitor State} & \mu & ::= & \sigma \\
\text{VAMOS State} & \Theta & ::= & \langle \overline{\rho}, \alpha, \mu \rangle \\
\text{Processor Eval Ctx} & \mathbb{P} & ::= & \cdot \mid \texttt{forward}\ E(\overline{v} +\!\!+ \mathbb{E} +\!\!+ \overline{\xi}) \mid \texttt{if}\ \mathbb{E}\ \texttt{then}\ e\ \texttt{else}\ e \\
\text{Source State E-Ctx} & \mathbb{B} & ::= & \cdot \mid \langle S, P, \Delta, \Sigma, \mathbb{P}, k, \dot{\epsilon}, \Sigma \rangle \mid \langle S, P(\overline{v} +\!\!+ \mathbb{E} +\!\!+ \overline{\xi}), \Sigma, k, [], \overline{B} \rangle \\
 & & \mid & \langle S, P(\overline{v}), \Sigma, k, \overline{X \mapsto v} +\!\!+ X \mapsto \mathbb{E} +\!\!+ \overline{X \mapsto \xi}, \overline{B} \rangle \\
\text{Arbiter Eval Ctx} & \mathbb{A} & ::= & \cdot \mid \langle \overline{\mathbb{S}}, \overline{\hat{a}}, \phi, \varsigma \rangle \mid \langle \texttt{on}\ \overline{b}\ \texttt{where}\ \overline{\mathbb{E}}\ \{\tilde{\sigma}\}, \overline{\hat{a}}, \phi, \varsigma \rangle \\
\text{Monitor Eval Ctx} & \mathbb{M} & ::= & \overline{\mathbb{S}} \\
\text{Eval Context} & \mathbb{X} & ::= & \langle \overline{\rho} +\!\!+ \mathbb{B} +\!\!+ \overline{\rho}, \alpha, \mu \rangle \mid \langle \overline{\rho}, \mathbb{A}, \mu \rangle \mid \langle \overline{\rho}, \alpha, \mathbb{M} \rangle
\end{array}
$$

Fig. A.20. Operational semantics grammar.

$$
\boxed{\Sigma \mid \Omega \mid \Pi \vdash \Theta \to \Theta \dashv \Sigma \mid \Omega \mid \Pi}
$$

$$
\frac{\Pi \mid [] \vdash \tilde{\xi} \to \tilde{\xi}' \dashv \Pi}{\Sigma \mid \Omega \mid \Pi \vdash \mathbb{X}[\tilde{\xi}] \to \mathbb{X}[\tilde{\xi}'] \dashv \Sigma \mid \Omega \mid \Pi} \qquad \frac{\Sigma \mid \Omega \mid \Pi \mid [] \vdash \tilde{\sigma} \to \tilde{\sigma}' \dashv \Sigma' \mid \Omega' \mid \Pi' \mid []}{\Sigma \mid \Omega \mid \Pi \vdash \mathbb{X}[\tilde{\sigma}] \to \mathbb{X}[\tilde{\sigma}'] \dashv \Sigma' \mid \Omega' \mid \Pi'}
$$

$$
\frac{\Omega \mid \Pi \vdash \rho \to \overline{\rho'} \dashv \Omega'}{\Sigma \mid \Omega \mid \Pi \vdash \mathbb{X}[\rho] \to \mathbb{X}[\overline{\rho'}] \dashv \Sigma \mid \Omega' \mid \Pi} \qquad \frac{\Sigma \mid \Omega \mid \Pi \mid \overline{\rho} \vdash \alpha \to \alpha' \dashv \Sigma' \mid \Omega' \mid \Pi' \mid \overline{\rho'}}{\Sigma \mid \Omega \mid \Pi \vdash \langle \overline{\rho}, \alpha, \mu \rangle \to \langle \overline{\rho'}, \alpha', \mu \rangle \dashv \Sigma' \mid \Omega' \mid \Pi'}
$$

$$
\frac{\Omega = \langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}}, \texttt{arbiter}\ :\ T\ \{\overline{\hat{a}}\}, \hat{M} \rangle \qquad \texttt{on}\ E(\overline{X})\ \{\sigma\} \in \hat{M} \qquad \Omega \vdash \sigma[E(\overline{X \mapsto v}) : T] \to \sigma'}{E(\overline{v}) +\!\!+ \Sigma \mid \Omega \mid \Pi \vdash \langle \overline{\rho}, \alpha, \{\} \rangle \to \langle \overline{\rho}, \alpha, \sigma' \rangle \dashv \Sigma \mid \Omega \mid \Pi}
$$

Fig. A.21. Operational semantics—top level.

All of this is reflected in the data structures we use to track the program state—shown in Fig. A.20—and the top-level semantics shown in Fig. A.21. The state of the program is split to make it easy to access sub-parts of it in many cases. The signature of the core stepping relation in Fig. A.21 is

$$\Sigma \mid \Omega \mid \Pi \vdash \Theta \to \Theta \dashv \Sigma \mid \Omega \mid \Pi$$

modeling a transition of the core state Θ and the auxiliary state with its three components: Σ is a list of events representing the event queue between the arbiter and the monitor, Ω mostly contains the code of the program as written, except that it also keeps track of membership in buffer groups (otherwise, it is just used for reference and the proof of type safety), and Π is the (opaque) store data structure of the base language—we do not manipulate this on our own, but thread it through for use by the base transition relation, whose signature we assume to be:

$$\Pi \vdash \sigma \dashv \Pi$$

The core VAMOS state data structure as shown in Fig. A.20 essentially keeps track of the threads of the monitoring system—some number of threads for the stream processors, and one each for the arbiter and the monitor. The state of the monitor is simple, as it is just the current (block of) statement(s) that is being executed—we assume that the base step relation uses the following rule to step in blocks of code:

$$\frac{}{\Pi \vdash \{\{\} +\!\!+ \overline{\sigma}\} \to \{\overline{\sigma}\} \dashv \Pi}$$

In turn, we also assume that statements that have finished their work turn themselves into $\{\}$ in the stepping relation. This means that the initial state of the monitor and the state it ends up in after executing a rule can be $\{\}$.

The state of a stream processor is more complex, as it keeps track of the source S it belongs to, the name P of the processor template it was instantiated from, a map Δ of stream fields, a list Σ of events that may eventually show up on the event source (as discussed above), a redex $\dot{e}$ describing the state of executing the stream processor's code ($\circ$ means it is ready to process another event), the buffer kind k of the buffer the stream processor forwards events to, the next hole event $\dot{\epsilon}$ under construction, and another list Σ' of events in said buffer. The second version of a stream processor state keeps track of stream processors as they are initializing when a stream is opened dynamically.

$\boxed{\Omega \mid \Pi \vdash \rho \to \overline{\rho} \dashv \Omega}$

$$\frac{\Omega = \langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}}, \hat{A}, \hat{M} \rangle \qquad \texttt{stream processor } P(\ldots) : \ldots \{ \overline{p} +\!\!+ p' +\!\!+ \overline{p''}; \hat{H} \} \in \overline{\hat{P}} \qquad p' \vdash \epsilon \to e \dashv \overline{p}}{\Omega \mid \Pi \vdash \langle S, P, \Delta, \epsilon +\!\!+ \Sigma, \circ, k, n, \Sigma' \rangle \to \langle S, P, \Delta, \Sigma, e, k, n, \Sigma' \rangle +\!\!+ \overline{\rho} \dashv \Omega}$$

$$\frac{\Omega = \langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}}, \hat{A}, \hat{M} \rangle \qquad k \mid \hat{H} \mid \overline{\hat{E}} \mid \dot{e} \mid \Sigma' \mid \Delta \mid \overline{\hat{X}} \mid E(\Gamma) \vdash e \to e' \dashv \dot{e}' \mid \Sigma'' \mid \Delta' \qquad \texttt{stream processor } P(\overline{X : \tau}) : T(\overline{\xi}) \to T'(\overline{\xi'}) \{ \overline{p}; \hat{H} \} \in \overline{\hat{P}} \qquad \texttt{stream type } T'(\overline{X' : \tau'}; \overline{\hat{X}}) \texttt{ shared } E(\Gamma) \{ \overline{\hat{E}} \} \in \overline{\hat{T}}}{\Omega \mid \Pi \vdash \langle S, P, \Delta, \Sigma, e, k, \dot{e}, \Sigma' \rangle \to [\langle S, P, \Delta', \Sigma, e', k, \dot{e}', \Sigma'' \rangle] \dashv \Omega}$$

$$\frac{k = \texttt{autodrop}(n', n'') \qquad |\Sigma'| < n' - n''}{\Omega \mid \Pi \vdash \langle S, P, \Delta, \Sigma, e, k, \epsilon, \Sigma' \rangle \to [\langle S, P, \Delta, \Sigma, e, k, \circ, \Sigma' +\!\!+ \epsilon \rangle] \dashv \Omega}$$

$$\frac{\Omega = \langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}}, \hat{A}, \hat{M} \rangle \qquad \texttt{stream processor } P(\overline{X : \tau}) : T(\overline{\xi}) \to T'(\overline{\xi'}) \{ \overline{p}; \hat{H} \} \in \overline{\hat{P}} \qquad \texttt{stream type } T'(\overline{X' : \tau'}; \overline{\hat{X}}) \texttt{ shared } E(\Gamma) \{ \overline{\hat{E}} \} \in \overline{\hat{T}} \qquad \overline{\hat{X} \vdash X'' \mapsto \nu''}}{\Omega \mid \Pi \vdash \langle S, P(\overline{\nu}), \Sigma, k, [], \overline{B} \rangle \to [\langle S, P(\overline{\nu}), \Sigma, k, \overline{X' \mapsto \xi'[\overline{X \mapsto \nu}]} +\!\!+ \overline{X'' \mapsto \nu''}, \overline{B} \rangle] \dashv \Omega}$$

$$\frac{\Omega \vdash \overline{B} + S \dashv \Omega'}{\Omega \mid \Pi \vdash \langle S, P(\overline{\nu'}), \Sigma, k, \overline{X \mapsto \nu}, \overline{B} \rangle \to [\langle S, P, \overline{X \mapsto \nu}, \Sigma, \circ, k, \circ, [] \rangle] \dashv \Omega'}$$

Fig. A.22. Operational semantics—performance layer (high-level).

$\boxed{p \vdash \epsilon \to e \dashv \overline{\rho}}$

$$\frac{}{\texttt{on } E(\overline{X})\ e \ \vdash\ E(\overline{\nu}) \to e[\overline{X \mapsto \nu}] \dashv []}$$

$$\frac{p = \texttt{on } E(\overline{X}) \texttt{ process using } P(\overline{\xi}) \texttt{ to } k \texttt{ include in } \overline{B}\ e}{p \vdash E(\overline{\nu} +\!\!+ \Sigma) \to e[\overline{X \mapsto \nu}] \dashv [\langle \mathit{fresh}_S, P(\overline{\xi[\overline{X \mapsto \nu}]}), \Sigma, k, [], \overline{B} \rangle]}$$

$\boxed{\Omega \vdash \overline{B} + S \dashv \Omega}$

$$\frac{}{\Omega \vdash [] + S \dashv \Omega}$$

$$\frac{\Omega = \langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}_l} +\!\!+ \hat{B} +\!\!+ \overline{\hat{B}_r}, \hat{A}, \hat{M} \rangle \qquad \Omega' = \langle \overline{\hat{P}}, \overline{\hat{B}_l} +\!\!+ \hat{B}' +\!\!+ \overline{\hat{B}_r}, \hat{A}, \hat{M} \rangle \qquad \hat{B} = \texttt{buffer group } B : T \texttt{ order by } o \{ \overline{S'} \} \qquad \overline{S''} = \overline{S'} \cup S \qquad \hat{B}' = \texttt{buffer group } B : T \texttt{ order by } o \{ \overline{S''} \} \qquad \Omega' \vdash \overline{B'} + S \dashv \Omega''}{\Omega \vdash \overline{B'} +\!\!+ B + S \dashv \Omega''}$$

Fig. A.23. Operational semantics—performance layer (high-level).

Finally, the arbiter state keeps track of a redex $\dot{a}$ as well, but combines this with continuations $\overline{\hat{a}}$ for the next rules to be tried if a match fails or a stack ϕ of backtracking options for choosing different kinds of event sources. The stack ϕ of backtracking options in turn consists of entries which keep track of an individual permutation state ψ for each variable of a `choose` expression, the list of arbiter rules $\overline{\hat{a}}$ to be tried within that `choose` expression, and the continuation $\overline{\hat{a}'}$ of sibling rules of the `choose` expression we move on to if all potential choices are exhausted without finding a match. The individual permutation state ψ consists of the name of the corresponding variable, a list $\overline{S}$ of sources in the buffer group we have already tried in this permutation attempt, and a list $\overline{S'}$ of sources in the buffer group that are still to be tried. Each round of checking arbiter rules keeps track of some local state ς, consisting of three elements. For implementing the round-robin order of buffer groups, we keep track of the list of buffer group ids $\overline{B}$ we have explored in this round. Furthermore, in order to stay consistent about the events in a particular buffer and the buffers in a particular buffer group, the first check that references either of them makes a local copy for the particular round, to be used later. This avoids first checking a rule that asks for an event in a particular buffer while that buffer is empty but matching on a rule that asks for two events on that buffer later when more events have concurrently appeared, and similar for checks if a buffer has a member that matches certain conditions.

The top-level rules of the stepping relation then start with general-purpose rules to evaluate expressions and statements. The empty list as the last part of the context on both sides is an argument that in more specialized cases contains the list of stream processor threads $\overline{\rho}$. This only applies in the arbiter, however, so in general, expressions and statements have to make progress without access to that information. In addition, the expression rule here requires that the expressions do not affect the global state of the program (this is only allowed for expressions contained in statements).

The next two rules refer to the individual stepping relations for stream processors and the arbiter, respectively. The final rule starts each cycle of the monitor processing an event by retrieving the first waiting event from the queue between the arbiter and the monitor (if any), generating appropriate local variable assignments for the pattern match variables, and prepending them to the code associated with the matched rule. All other work by the monitor is covered by the first two rules.

A.5. Performance layer semantics

The specific stepping rules for the performance layer are shown in Figs. A.22 to A.25. The first rule resets the current redex to process the next event when the last one has been processed, potentially also initializing the creation of a new event source and

$$\boxed{k \mid \hat{H} \mid \overline{\hat{E}} \mid \dot{\epsilon} \mid \Sigma \mid \Delta \mid \overline{\hat{X}} \mid E(\Gamma) \vdash e \to e \dashv \dot{\epsilon} \mid \Sigma \mid \Delta}$$

$$\frac{}{k \mid \hat{H} \mid \overline{\hat{E}} \mid \dot{\epsilon} \mid \Sigma \mid \Delta \mid \overline{\hat{X}} \mid E(\Gamma) \vdash \texttt{drop} \to \circ \dashv \dot{\epsilon} \mid \Sigma \mid \Delta}$$

$$\frac{k \mid \hat{H} \mid \overline{\hat{E}} \mid \dot{\epsilon} \mid E'(\Gamma) \vdash \Sigma + E(\overline{v}) \to \Sigma' \dashv \dot{\epsilon}' \qquad \overline{\hat{X}} \mid \overline{\hat{E}} \mid E'(\Gamma) \vdash \Delta + E(\overline{v}) \to \Delta'}{k \mid \hat{H} \mid \overline{\hat{E}} \mid \dot{\epsilon} \mid \Sigma \mid \Delta \mid \overline{\hat{X}} \mid E'(\Gamma) \vdash \texttt{forward}\ E(\overline{v}) \to \circ \dashv \dot{\epsilon}' \mid \Sigma' \mid \Delta'}$$

$$\frac{}{k \mid \hat{H} \mid \overline{\hat{E}} \mid \dot{\epsilon} \mid \Sigma \mid \Delta \mid \overline{\hat{X}} \mid E(\Gamma) \vdash \texttt{if true then}\ e_1\ \texttt{else}\ e_2 \to e_1 \dashv \dot{\epsilon} \mid \Sigma \mid \Delta}$$

$$\frac{}{k \mid \hat{H} \mid \overline{\hat{E}} \mid \dot{\epsilon} \mid \Sigma \mid \Delta \mid \overline{\hat{X}} \mid E(\Gamma) \vdash \texttt{if false then}\ e_1\ \texttt{else}\ e_2 \to e_2 \dashv \dot{\epsilon} \mid \Sigma \mid \Delta}$$

$$\boxed{k \mid \hat{H} \mid \overline{\hat{E}} \mid \dot{\epsilon} \mid E(\Gamma) \vdash \Sigma + \epsilon \to \Sigma \dashv \dot{\epsilon}}$$

$$\frac{}{\texttt{infinite} \mid \hat{H} \mid \overline{\hat{E}} \mid \circ \mid E(\Gamma) \vdash \Sigma + \epsilon \to \Sigma +\!\!+ \epsilon \dashv \circ}$$

$$\frac{k = \texttt{blocking}(n) \qquad |\Sigma| < n}{k \mid \hat{H} \mid \overline{\hat{E}} \mid \circ \mid E(\Gamma) \vdash \Sigma + \epsilon \to \Sigma +\!\!+ \epsilon \dashv \circ}$$

$$\frac{|\Sigma| \geq n \vee (\dot{\epsilon}' \neq \circ \wedge |\Sigma| \geq n - n') \qquad \hat{H} \mid \overline{\hat{E}} \mid E(\Gamma) \vdash \dot{\epsilon} + \epsilon = \dot{\epsilon}'}{\texttt{autodrop}(n, n') \mid \hat{H} \mid \overline{\hat{E}} \mid \dot{\epsilon} \mid E(\Gamma) \vdash \Sigma + \epsilon \to \Sigma \dashv \dot{\epsilon}'}$$

$$\frac{k = \texttt{autodrop}(n, n') \qquad |\Sigma| < n}{k \mid \hat{H} \mid \overline{\hat{E}} \mid E(\Gamma) \mid \circ \vdash \Sigma + \epsilon \to \Sigma +\!\!+ \epsilon \dashv \circ}$$

Fig. A.24. Operational semantics—performance layer (event processing).

$$\boxed{\hat{H} \mid \overline{\hat{E}} \mid E(\Gamma) \vdash \dot{\epsilon} + \epsilon = \dot{\epsilon}}$$

$$\frac{\overline{\hat{E}} \mid E''(\Gamma) \vdash E' : \overline{X : \tau} \qquad \overline{\hat{X} \vdash v + E'(\overline{X = v'}) = v''}}{\texttt{hole}\ E\ \{\ \overline{\hat{X}}\ \} \mid \overline{\hat{E}} \mid E''(\Gamma) \vdash E(\overline{v}) + E'(\overline{v'}) = E(\overline{v''})}$$

$$\frac{\overline{\hat{E}} \mid E''(\Gamma) \vdash E' : \overline{X : \tau} \qquad \overline{\hat{X} \vdash X' \mapsto v} \qquad \overline{\hat{X} \vdash v + E'(\overline{X = v'}) = v''}}{\texttt{hole}\ E\ \{\ \overline{\hat{X}}\ \} \mid \overline{\hat{E}} \mid E''(\Gamma) \vdash \circ + E'(\overline{v'}) = E(\overline{v''})}$$

$$\boxed{\overline{\hat{X}} \mid \overline{\hat{E}} \mid E(\Gamma) \vdash \Delta + \epsilon \to \Delta}$$

$$\frac{\overline{\hat{E}} \mid E''(\Gamma) \vdash E' : \overline{X'' : \tau} \qquad \overline{X = O(\overline{E.X'}) \vdash \Delta(X) + E'(\overline{X'' = v}) = v'}}{\overline{X = O(\overline{E.X'})} \mid \overline{\hat{E}} \mid E''(\Gamma) \vdash \Delta + E'(\overline{v}) \to \Delta[\overline{X \mapsto v'}]}$$

$$\boxed{\hat{X} \vdash v + E(\overline{X = v}) = v}$$

$$\frac{}{X = O([]) \vdash v + E(\overline{X' = v'}) = v}$$

$$\frac{E'' \neq E \qquad X = O(\overline{E'''.X'''}) \vdash v + E(\overline{X' = v'}) = v''}{X = O(\overline{E'''.X'''} +\!\!+ [E''.X'']) \vdash v + E(\overline{X' = v'}) = v''}$$

$$\frac{X'' = v_X \in \overline{X' = v'} \qquad v''' = O(v'', v_X) \qquad X = O(\overline{E'''.X'''}) \vdash v + E(\overline{X' = v'}) = v''}{X = O(\overline{E''.X'''} +\!\!+ [E.X'']) \vdash v + E(\overline{X' = v'}) = v'''}$$

Fig. A.25. Operational semantics—performance layer (aggregation).

stream processor using the first helper relation. The second rule takes a step in evaluating the current redex using the last two helper relations below. The third rule applies only to event sources that use `autodrop` buffers—in that case, if a hole event that collects summarization information has been generated and there is enough space again to enqueue new events in the arbiter buffer, we emit that hole event and reset its slot. This also leaves space for at least one regular event, so at no point there are two consecutive `hole` events. The fourth rule takes a step in initializing a dynamically added event source, substituting the arguments to the stream processor in the arguments for the arbiter buffer, and calculating the starting values of its aggregation expressions - once the arguments for the stream processor are evaluated, we can start evaluating the arguments to the stream type that contains the relevant stream fields.

$\boxed{\Sigma \mid \Omega \mid \Pi \mid \overline{\rho} \vdash \alpha \to \alpha \dashv \Sigma \mid \Omega \mid \Pi \mid \overline{\rho}}$

$$\frac{\Sigma \mid \Omega \mid \Pi \mid \overline{\rho} \vdash \tilde{\sigma} \to \tilde{\sigma}' \dashv \Sigma' \mid \Omega' \mid \Pi' \mid \overline{\rho'}}{\Sigma \mid \Omega \mid \Pi \mid \overline{\rho} \vdash \mathbb{A}[\tilde{\sigma}] \to \mathbb{A}[\tilde{\sigma}'] \dashv \Sigma' \mid \Omega' \mid \Pi' \mid \overline{\rho'}}$$

$$\frac{\Omega \mid \Pi \mid \overline{\rho} \vdash \alpha \to \alpha'}{\Sigma \mid \Omega \mid \Pi \mid \overline{\rho} \vdash \alpha \to \alpha' \dashv \Sigma \mid \Omega \mid \Pi \mid \overline{\rho}}$$

$$\frac{\Omega = \langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}}, \hat{A}, \hat{M} \rangle \qquad \vdash \overline{\rho} \textbf{ not done} \qquad \Omega \mid \overline{\rho} \mid \overline{B} \vdash \overline{\hat{B}} \to \overline{\hat{B}'} \qquad \hat{A} = \texttt{arbiter} : T\ \{\overline{\hat{a}}\} \qquad \Omega' = \langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}'}, \hat{A}, \hat{M} \rangle}{\Sigma \mid \Omega \mid \Pi \mid \overline{\rho} \vdash \langle \circ, [], \circ, \langle \overline{B}, \overline{\rho'}, \overline{\hat{B}''} \rangle \rangle \to \langle \circ, \overline{\hat{a}}, \circ, \langle [], [], [] \rangle \rangle \dashv \Sigma \mid \Omega' \mid \Pi \mid \overline{\rho}}$$

Fig. A.26. Operational semantics—arbiter.

$\boxed{\Omega \mid \Pi \mid \overline{\rho} \vdash \alpha \to \alpha}$

$$\frac{}{\Omega \mid \Pi \mid \overline{\rho} \vdash \langle \{\}, \overline{\hat{a}}, \phi, \varsigma \rangle \to \langle \circ, [], \circ, \varsigma \rangle}$$

$$\frac{\phi \vdash \overline{\hat{a}} \dashv \phi'}{\Omega \mid \Pi \mid \overline{\rho} \vdash \langle \circ, [], \phi, \varsigma \rangle \to \langle \circ, \overline{\hat{a}}, \phi', \varsigma \rangle} \qquad \frac{}{\Omega \mid \Pi \mid \overline{\rho} \vdash \langle \circ, \hat{a} +\!\!+ \overline{\hat{a}'}, \phi, \varsigma \rangle \to \langle \hat{a}, \overline{\hat{a}'}, \phi, \varsigma \rangle}$$

$$\frac{\Omega \mid \varsigma \vdash \texttt{buffer group}\ B : T\ \texttt{order by}\ o\ \{\overline{S'}\} \dashv \varsigma' \qquad \overline{S'} = \overline{S''} +\!\!+ \overline{S'''} \qquad |\overline{S''}| = min(n, |\overline{S'}|) \qquad \overline{\psi} = \overline{\langle S, [], \overline{S''} \rangle}}{\Omega \mid \Pi \mid \overline{\rho} \vdash \langle \texttt{choose}\ \overline{S}\ \texttt{from first}\ n\ B\ \{\overline{\hat{a}}\}, \overline{\hat{a}'}, \phi, \varsigma \rangle \to \langle \circ, [], \langle \overline{\psi}, \overline{\hat{a}}, \overline{\hat{a}'} \rangle; \phi, \varsigma' \rangle}$$

$$\frac{\Omega \mid \varsigma \vdash \texttt{buffer group}\ B : T\ \texttt{order by}\ o\ \{\overline{S'}\} \dashv \varsigma' \qquad \overline{S'} = \overline{S'''} +\!\!+ \overline{S''} \qquad |\overline{S''}| = min(n, |\overline{S'}|) \qquad \overline{\psi} = \overline{\langle S, [], reverse(\overline{S''}) \rangle}}{\Omega \mid \Pi \mid \overline{\rho} \vdash \langle \texttt{choose}\ \overline{S}\ \texttt{from last}\ n\ B\ \{\overline{\hat{a}}\}, \overline{\hat{a}'}, \phi, \varsigma \rangle \to \langle \circ, [], \langle \overline{\psi}, \overline{\hat{a}}, \overline{\hat{a}'} \rangle; \phi, \varsigma' \rangle}$$

$$\frac{}{\Omega \mid \Pi \mid \overline{\rho} \vdash \langle \texttt{on}\ []\ \texttt{where true}\ \{\ \tilde{\sigma}\ \}, \overline{\hat{a}'}, \phi, \varsigma \rangle \to \langle \tilde{\sigma}, \overline{\hat{a}'}, \phi, \varsigma \rangle}$$

$$\frac{}{\Omega \mid \Pi \mid \overline{\rho} \vdash \langle \texttt{on}\ \overline{b}\ \texttt{where false}\ \{\ \tilde{\sigma}\ \}, \overline{\hat{a}'}, \phi, \varsigma \rangle \to \langle \circ, \overline{\hat{a}'}, \phi, \varsigma \rangle}$$

$$\frac{\Omega \mid \overline{\rho} \mid \varsigma \mid b \vdash \langle \tilde{\xi}, \tilde{\sigma} \rangle \to \langle \tilde{\xi}', \tilde{\sigma}' \rangle \dashv \varsigma'}{\Omega \mid \Pi \mid \overline{\rho} \vdash \langle \texttt{on}\ \overline{b} +\!\!+ b +\!\!+ \overline{b'}\ \texttt{where}\ \tilde{\xi}\ \{\ \tilde{\sigma}\ \}, \overline{\hat{a}'}, \phi, \varsigma \rangle \to \langle \texttt{on}\ \overline{b} +\!\!+ \overline{b'}\ \texttt{where}\ \tilde{\xi}'\ \{\ \tilde{\sigma}'\ \}, \overline{\hat{a}'}, \phi, \varsigma' \rangle}$$

$$\frac{\Pi \mid \overline{\rho} \vdash \tilde{\xi} \to \tilde{\xi}' \dashv \Pi}{\Omega \mid \Pi \mid \overline{\rho} \vdash \langle \texttt{on}\ \overline{b}\ \texttt{where}\ \tilde{\xi}\ \{\ \tilde{\sigma}\ \}, \overline{\hat{a}'}, \phi, \varsigma \rangle \to \langle \texttt{on}\ \overline{b}\ \texttt{where}\ \tilde{\xi}'\ \{\ \tilde{\sigma}\ \}, \overline{\hat{a}'}, \phi, \varsigma \rangle}$$

Fig. A.27. Operational semantics—arbiter / matching process.

$\boxed{\phi \vdash \overline{\hat{a}} \dashv \phi}$

$$\frac{}{\langle \langle S, \overline{S'}, [] \rangle +\!\!+ \overline{\psi}, \overline{\hat{a}}, \overline{\hat{a}'} \rangle; \phi \vdash \overline{\hat{a}'} \dashv \phi}$$

$$\frac{\psi_p = \langle S_p, \overline{S'_p}, S''_p +\!\!+ \overline{S'''_p} \rangle \qquad \psi'_p = \langle S_p, \overline{S'_p} +\!\!+ S''_p, \overline{S'''_p} \rangle \qquad \psi_e = \langle S_e, \overline{S'_e}, [] \rangle \qquad \psi'_e = \langle S_e, [], \overline{S'_e} \rangle}{\langle \overline{\psi} +\!\!+ \psi_p +\!\!+ \psi_e +\!\!+ \overline{\psi'}, \hat{a}, \overline{\hat{a}'} \rangle; \phi \vdash [] \dashv \langle \overline{\psi} +\!\!+ \psi'_p +\!\!+ \psi'_e +\!\!+ \overline{\psi'}, \hat{a}, \overline{\hat{a}'} \rangle; \phi}$$

$$\frac{\forall \overline{S_q}.\ \overline{S_q} \neq [] \qquad \overline{\langle S_v, \overline{S_s}, \overline{S_q} \rangle} +\!\!+ \langle S'_v, \overline{S'_s}, S_n +\!\!+ \overline{S'_q} \rangle \mid [] \vdash \overline{\hat{a}} \to \overline{\hat{a}''}}{\overline{\langle \langle S_v, \overline{S_s}, \overline{S_q} \rangle} +\!\!+ \langle S'_v, \overline{S'_s}, S_n +\!\!+ \overline{S'_q} \rangle, \overline{\hat{a}}, \overline{\hat{a}'} \rangle; \phi \vdash \overline{\hat{a}''} \dashv \overline{\langle \langle S_v, \overline{S_s}, \overline{S_q} \rangle} +\!\!+ \langle S'_v, \overline{S'_s} +\!\!+ S_n, \overline{S'_q} \rangle, \overline{\hat{a}}, \overline{\hat{a}'} \rangle; \phi}$$

$\boxed{\overline{\psi} \mid \overline{S} \vdash \overline{\hat{a}} \to \overline{\hat{a}}}$

$$\frac{}{[] \mid \overline{S} \vdash \overline{\hat{a}} \to \overline{\hat{a}}} \qquad \frac{}{\langle S_v, \overline{S_p}, [] \rangle +\!\!+ \overline{\psi} \mid \overline{S_s} \vdash \overline{\hat{a}} \to []} \qquad \frac{S_n \in \overline{S_s} \qquad \langle S_v, \overline{S_p}, \overline{S'} \rangle +\!\!+ \overline{\psi} \mid \overline{S_s} \vdash \overline{\hat{a}} \to \overline{\hat{a}'}}{\langle S_v, \overline{S_p}, S_n +\!\!+ \overline{S'} \rangle +\!\!+ \overline{\psi} \mid \overline{S_s} \vdash \overline{\hat{a}} \to \overline{\hat{a}'}}$$

$$\frac{S_n \notin \overline{S_s} \qquad \overline{\psi} \mid S_n +\!\!+ \overline{S_s} \vdash \overline{\hat{a}} \to \overline{\hat{a}'}}{\langle S_v, \overline{S_p}, S_n +\!\!+ \overline{S'} \rangle +\!\!+ \overline{\psi} \mid \overline{S_s} \vdash \overline{\hat{a}} \to \overline{\hat{a}'[S_v \mapsto S_n]}}$$

Fig. A.28. Arbiter rule-matching (backtracking).

$$\boxed{\Omega \mid \overline{\rho} \mid \varsigma \mid b \vdash \langle \tilde{\xi}, \tilde{\sigma} \rangle \to \langle \tilde{\xi}, \tilde{\sigma} \rangle \dashv \varsigma}$$

$$\frac{\overline{\rho} \mid \varsigma \vdash \mathit{stream}(b) \mapsto \rho' \dashv \varsigma' \qquad \Omega \mid \rho' \mid b \vdash \langle \tilde{\xi}, \tilde{\sigma} \rangle \to \langle \tilde{\xi}', \tilde{\sigma}' \rangle}{\Omega \mid \overline{\rho} \mid \varsigma \mid b \vdash \langle \tilde{\xi}, \tilde{\sigma} \rangle \to \langle \tilde{\xi}', \tilde{\sigma}' \rangle \dashv \varsigma'}$$

$$\boxed{\Omega \mid \rho \mid b \vdash \langle \tilde{\xi}, \tilde{\sigma} \rangle \to \langle \tilde{\xi}, \tilde{\sigma} \rangle}$$

$$\frac{\rho = \langle S, P, \Delta, [], \dot{e}, k, [] \rangle}{\Omega \mid \rho \mid S : \texttt{done} \vdash \langle \tilde{\xi}, \tilde{\sigma} \rangle \to \langle \tilde{\xi}, \tilde{\sigma} \rangle} \qquad \frac{\rho = \langle S, P, \Delta, \Sigma, \dot{e}, k, \Sigma' \rangle \qquad \Sigma +\!\!+ \Sigma' \neq []}{\Omega \mid \rho \mid S : \texttt{done} \vdash \langle \tilde{\xi}, \tilde{\sigma} \rangle \to \langle \texttt{false}, \tilde{\sigma} \rangle}$$

$$\frac{\rho = \langle S, P, \Delta, \Sigma, \dot{e}, k, [] \rangle}{\Omega \mid \rho \mid S : \texttt{nothing} \vdash \langle \tilde{\xi}, \tilde{\sigma} \rangle \to \langle \tilde{\xi}, \tilde{\sigma} \rangle} \qquad \frac{\rho = \langle S, P, \Delta, \Sigma, \dot{e}, k, \Sigma' \rangle \qquad \Sigma' \neq []}{\Omega \mid \rho \mid S : \texttt{nothing} \vdash \langle \tilde{\xi}, \tilde{\sigma} \rangle \to \langle \texttt{false}, \tilde{\sigma} \rangle}$$

$$\frac{\rho = \langle S, P, \Delta, \Sigma, \dot{e}, k, \Sigma' \rangle \qquad |\Sigma'| \geq n}{\Omega \mid \rho \mid S : n \vdash \langle \tilde{\xi}, \tilde{\sigma} \rangle \to \langle \tilde{\xi}, \tilde{\sigma} \rangle} \qquad \frac{\rho = \langle S, P, \Delta, \Sigma, \dot{e}, k, \Sigma' \rangle \qquad |\Sigma'| < n}{\Omega \mid \rho \mid S : n \vdash \langle \tilde{\xi}, \tilde{\sigma} \rangle \to \langle \texttt{false}, \tilde{\sigma} \rangle}$$

$$\frac{\Omega \vdash P : T' \to T \qquad \Omega \vdash \tilde{\sigma}[E/E'(\overline{X} \mapsto \overline{v}) : T] \to \tilde{\sigma}' \qquad \rho = \langle S, P, \Delta, \Sigma, \dot{e}, k, \overline{E'(\overline{v})} +\!\!+ \Sigma' \rangle}{\Omega \mid \rho \mid S : \overline{E(\overline{X})} \vdash \langle \tilde{\xi}, \tilde{\sigma} \rangle \to \langle \tilde{\xi}[\overline{X \mapsto v}], \tilde{\sigma}' \rangle} \qquad \frac{\rho = \langle S, P, \Delta, \Sigma, \dot{e}, k, \Sigma' \rangle \qquad \Omega \vdash P : T' \to T \qquad \Sigma' \not\leq^T \overline{E}}{\Omega \mid \rho \mid S : \overline{E(\overline{X})} \vdash \langle \tilde{\xi}, \tilde{\sigma} \rangle \to \langle \texttt{false}, \tilde{\sigma} \rangle}$$

Fig. A.29. Buffer match expression evaluation.

$$\boxed{\Sigma \mid \Omega \mid \Pi \mid \overline{\rho} \vdash \tilde{\sigma} \to \tilde{\sigma} \dashv \Sigma \mid \Omega \mid \Pi \mid \overline{\rho}}$$

$$\frac{\Pi \vdash \tilde{\sigma} \to \tilde{\sigma}' \dashv \Pi'}{\Sigma \mid \Omega \mid \Pi \vdash \tilde{\sigma} \to \tilde{\sigma}' \dashv \Sigma \mid \Omega \mid \Pi'} \qquad \frac{\Pi \mid \overline{\rho} \vdash \tilde{\xi} \to \tilde{\xi}' \dashv \Pi'}{\Sigma \mid \Omega \mid \Pi \mid \overline{\rho} \vdash \mathbb{S}[\tilde{\xi}] \to \mathbb{S}[\tilde{\xi}'] \dashv \Sigma \mid \Omega \mid \Pi' \mid \overline{\rho}}$$

$$\frac{\overline{\rho} = \overline{\rho_l} +\!\!+ \langle S, P, \Delta, \Sigma, \dot{e}, k, \Sigma' \rangle +\!\!+ \overline{\rho_r} \qquad \overline{\rho'} = \overline{\rho_l} +\!\!+ \langle S, P, \Delta[X \mapsto v], \Sigma, \dot{e}, k, \Sigma' \rangle +\!\!+ \overline{\rho_r}}{\Sigma \mid \Omega \mid \Pi \mid \overline{\rho} \vdash \$S.X = v \to \{\} \dashv \Sigma \mid \Omega \mid \Pi \mid \overline{\rho'}}$$

$$\frac{\overline{\rho} = \overline{\rho_l} +\!\!+ \langle S, P, \Delta, \Sigma, \dot{e}, k, \Sigma' +\!\!+ \Sigma'' \rangle +\!\!+ \overline{\rho_r} \qquad |\Sigma'| = n \qquad \overline{\rho'} = \overline{\rho_l} +\!\!+ \langle S, P, \Delta, \Sigma, \dot{e}, k, \Sigma'' \rangle +\!\!+ \overline{\rho_r}}{\Sigma \mid \Omega \mid \Pi \mid \overline{\rho} \vdash \texttt{drop}\ n\ \texttt{from}\ S \to \{\} \dashv \Sigma \mid \Omega \mid \Pi \mid \overline{\rho'}}$$

$$\frac{}{\Sigma \mid \Omega \mid \Pi \mid \overline{\rho} \vdash \texttt{yield}\ E(\overline{v}) \to \{\} \dashv \Sigma +\!\!+ E(\overline{v}) \mid \Omega \mid \Pi \mid \overline{\rho}}$$

$$\frac{\hat{B} = \texttt{buffer group}\ B\ :\ T\ \texttt{order by}\ o\ \{\overline{S'}\} \qquad \overline{S''} = \overline{S'} \cup S \qquad \hat{B}' = \texttt{buffer group}\ B\ :\ T\ \texttt{order by}\ o\ \{\overline{S''}\} \qquad \Omega = \langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}_l} +\!\!+ \hat{B} +\!\!+ \overline{\hat{B}_r}, \hat{A}, \hat{M} \rangle \qquad \Omega' = \langle \overline{\hat{P}}, \overline{\hat{B}_l} +\!\!+ \hat{B}' +\!\!+ \overline{\hat{B}_r}, \hat{A}, \hat{M} \rangle}{\Sigma \mid \Omega \mid \Pi \mid \overline{\rho} \vdash \texttt{add}\ S\ \texttt{to}\ B \to \{\} \dashv \Sigma \mid \Omega' \mid \Pi \mid \overline{\rho}}$$

$$\frac{\hat{B} = \texttt{buffer group}\ B\ :\ T\ \texttt{order by}\ o\ \{\overline{S'}\} \qquad \overline{S''} = \overline{S'} \setminus S \qquad \hat{B}' = \texttt{buffer group}\ B\ :\ T\ \texttt{order by}\ o\ \{\overline{S''}\} \qquad \Omega = \langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}_l} +\!\!+ \hat{B} +\!\!+ \overline{\hat{B}_r}, \hat{A}, \hat{M} \rangle \qquad \Omega' = \langle \overline{\hat{P}}, \overline{\hat{B}_l} +\!\!+ \hat{B}' +\!\!+ \overline{\hat{B}_r}, \hat{A}, \hat{M} \rangle}{\Sigma \mid \Omega \mid \Pi \mid \overline{\rho} \vdash \texttt{remove}\ S\ \texttt{from}\ B \to \{\} \dashv \Sigma \mid \Omega' \mid \Pi \mid \overline{\rho}}$$

Fig. A.30. Operational semantics—statements.

$$\boxed{\Pi \mid \overline{\rho} \vdash \tilde{\xi} \to \tilde{\xi} \dashv \Pi}$$

$$\frac{\Pi \vdash \tilde{\xi} \to \tilde{\xi}' \dashv \Pi'}{\Pi \mid \overline{\rho} \vdash \tilde{\xi} \to \tilde{\xi}' \dashv \Pi'} \qquad \frac{\langle S, P, \Delta, \Sigma', \dot{e}, k, \Sigma \rangle \in \overline{\rho} \qquad \Delta(X) = v}{\Pi \mid \overline{\rho} \vdash \$S.X \to v}$$

Fig. A.31. Operational semantics—expressions.

The fifth rule finishes this initialization process once all arguments to the stream type have been evaluated. In that step, the second helper relation takes care of adding the event source to the specified list of buffer groups.

The first four rules in Fig. A.24 deal with the stream processor redexes; the only interesting case is when an event is forwarded, in which case the first helper relation interprets the buffer kind k to determine whether and how we can proceed. The important cases here concern the autodrop buffer: hole event construction (using the final two helper relations) only starts when the arbiter buffer is full, and only continues while not enough space is free. Conversely, events are only forwarded when no hole event is being constructed (if one is, the rule that emits and resets it is in Fig. A.23 above). The second helper relation updates the stream's aggregation fields, whether an event is dropped or not. Hole events use the same mechanism to compute summarization data on the events that were dropped.

$$\boxed{\Omega \mid \overline{\rho} \mid \overline{B} \vdash \overline{\hat{B}} \to \overline{\hat{B}}}$$

$$\frac{}{\Omega \mid \overline{\rho} \mid \overline{B} \vdash [] \to []}$$

$$\frac{\texttt{stream type } T(\Gamma\,;\,\overline{\hat{X}}) \texttt{ shared } E_S(\Gamma_S)\ \{\ \overline{\hat{E}}\ \} \in \overline{\hat{T}} \quad \hat{B} = \texttt{buffer group } B\ :\ T \texttt{ order by } o\ \{\overline{S}\} \quad \hat{B}'' = \texttt{buffer group } B\ :\ T \texttt{ order by } o\ \{\overline{S'}\} \quad \overline{\rho} \mid \overline{B'} \vdash \overline{\hat{B}'} \to \overline{\hat{B}'''} \quad \Gamma_S \mid \overline{\rho} \mid \overline{B'} \vdash^{o}_{B} \overline{S} \to \overline{S'}}{\langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}_\Omega}, \hat{A}, \hat{M} \rangle \mid \overline{\rho} \mid \overline{B'} \vdash \hat{B} +\!\!+ \overline{\hat{B}'} \to \hat{B}'' +\!\!+ \overline{\hat{B}'''}}$$

$$\boxed{\Gamma \mid \overline{\rho} \mid \overline{B} \vdash^{o}_{B} \overline{S} \to \overline{S}}$$

$$\frac{}{\Gamma \mid \overline{\rho} \mid \overline{B'} \vdash^{o}_{B} [] \to []} \qquad \frac{B \in \overline{B'}}{\Gamma \mid \overline{\rho} \mid \overline{B'} \vdash^{\texttt{round robin}}_{B} S +\!\!+ \overline{S'} \to \overline{S'} +\!\!+ S}$$

$$\frac{\Gamma \mid \overline{\rho} \mid \overline{S} \vdash^{o} \overline{S} \to \overline{S'}}{\Gamma \mid \overline{\rho} \mid \overline{B'} \vdash^{o}_{B} \overline{S} \to \overline{S'}} \qquad \frac{B \notin \overline{B'}}{\Gamma \mid \overline{\rho} \mid \overline{B'} \vdash^{\texttt{round robin}}_{B} \overline{S} \to \overline{S}}$$

$$\boxed{\Gamma \mid \overline{\rho} \mid \overline{S} \vdash^{o} \overline{S} \to \overline{S}}$$

$$\frac{}{\Gamma \mid \overline{\rho} \mid \overline{S} \vdash^{o} [] \to []} \qquad \frac{\overline{\rho} \mid \overline{S_B} \vdash^{o} \overline{S'} \to \overline{S''} +\!\!+ \overline{S'''} \quad \forall S''.\, \Gamma \mid \overline{\rho} \mid \overline{S_B} \vdash^{o} S'' \le S \quad \forall S'''.\, \Gamma \mid \overline{\rho} \mid \overline{S_B} \vdash^{o} S \le S'''}{\Gamma \mid \overline{\rho} \mid \overline{S_B} \vdash^{o} S +\!\!+ \overline{S'} \to \overline{S''} +\!\!+ S +\!\!+ \overline{S'''}}$$

$$\boxed{\Gamma \mid \overline{\rho} \mid \overline{S} \vdash^{o} S \le S}$$

$$\frac{\langle S, P, \Delta, \Sigma, \dot{e}, k, \Sigma' \rangle \in \overline{\rho} \quad \langle S', P', \Delta', \Sigma'', \dot{e}', k', \Sigma''' \rangle \in \overline{\rho} \quad \Delta(X) \le \Delta'(X)}{\Gamma \mid \overline{\rho} \mid \overline{S''} \vdash^{X} S \le S'}$$

$$\frac{\langle S, P, \Delta, \Sigma, \dot{e}, k, \Sigma' \rangle \in \overline{\rho} \quad \langle S', P', \Delta', \Sigma'', \dot{e}', k', \Sigma''' \rangle \in \overline{\rho} \quad \overline{\langle S_S, P_S, \Delta_S, \Sigma_S, \dot{e_S}, k_S, \Sigma'_S \rangle} \subseteq \overline{\rho} \quad \Gamma \mid \overline{\Delta_S} \vdash (\Delta, \Sigma') \le^{X}_{\omega} (\Delta', \Sigma''')}{\Gamma \mid \overline{\rho} \mid \overline{S_S} \vdash^{\texttt{shared } X\ \omega} S \le S'}$$

Fig. A.32. Operational semantics—buffer group ordering, Part 1.

A.6. Arbiter semantics

The rules for the arbiter in Fig. A.26 start by allowing statements to step with the additional information about the stream processors, in particular, this is where the stream fields are stored. The second rule refers to the more specific rules for running the matching process below, while the last rule resets the arbiter state to look for the next rule to apply. This latter rule also checks if there are any events left to process anyway before proceeding, and also adjusts the order of event sources in each buffer group according to its specification.

The first rule of the rule matching process in Fig. A.27 considers the state when a rule has matched and its statements have been executed to completion. In that case, we also clear both kinds of continuations: the rules that we might have tried next if the current one would not have matched, and the stack of potential other permutations we would try if none of those latter rules apply, either. The second rule handles that latter case, retrieving a new list of rules to try and an updated stack, while the third rule handles the former case of simply retrieving the next rule.

The next two rules are very similar and only differ in the way they apply the ordering of the buffer group, setting up the evaluation of a choice expression by pushing the initial setup for the given permutation to the stack, also storing the current arbiter rules continuation there.

The final four rules in Fig. A.27 deal with the top-level aspects of matching a concrete rule. This is the one case of non-determinism in this semantics that is not due to multithreading, but rather for optimization: while the arbiter needs to try to apply the rules in order, the evaluation of the components of a particular rule can happen in any order. This allows the compiler to try to re-use existing information efficiently to fail fast, and makes little semantic difference so long as the expressions in the `where` clause do not affect the state of the program (which the semantics requires anyway). Of these rules, the first deals with when a rule applies: all buffer match expressions must have been evaluated, and the `where` clause must be true. Conversely, if the `where` clause ever turns out to be false, the rule does not match, and we reset the redex to $\circ$. The penultimate rule refers to the rules for evaluating buffer match expressions belows; the key thing here is that evaluating a buffer match expression may affect both the `where` clause and the statements to be executed when the arbiter rule applies, mainly by essentially substituting variables in matched events, but also by setting the `where` clause to false if a buffer match expression does not match. Finally, the arbiter allows expressions in the where clause to access the stream fields of the given streams, but not to affect the state in terms of the base language.

A.6.1. Choose expressions and backtracking

The rules in Fig. A.28 deal with generating the k-permutations of event sources in buffer groups for `choose` expressions. An element of the choice stack ϕ consists of a list of choice states $\overline{\psi}$, the arbiter rules $\overline{\hat{a}}$ to be tried within the current `choose` expression, and the arbiter rules $\overline{\hat{a}'}$ forming the continuation of the `choose` expression. A choice state ψ corresponds to a buffer variable S in

$$\boxed{\Gamma \mid \overline{\Delta} \vdash (\Delta, \Sigma) \leq^X_\omega (\Delta, \Sigma)}$$

$$\frac{\Gamma \mid \overline{\Delta} \vdash (\Delta', \Sigma') \to^X_\omega \nu' \qquad \Gamma \mid \overline{\Delta} \vdash (\Delta'', \Sigma'') \to^X_\omega \nu'' \qquad \nu' \leq \nu''}{\Gamma \mid \overline{\Delta} \vdash (\Delta', \Sigma') \leq^X_\omega (\Delta'', \Sigma'')}$$

$$\boxed{\Gamma \mid \overline{\Delta} \vdash (\Delta, \Sigma) \to^X_\omega \nu}$$

$$\frac{|\Gamma| = |\overline{\nu}|}{\Gamma +\!\!+ X : \tau +\!\!+ \Gamma' \mid \overline{\Delta} \vdash (\Delta, E(\overline{\nu} +\!\!+ \nu' +\!\!+ \overline{\nu''}) +\!\!+ \Sigma) \to^X_\omega \nu'}$$

$$\frac{\overline{\Delta} \mid \Delta \vdash \lambda \to \nu}{\Gamma \mid \overline{\Delta} \vdash (\Delta, []) \to^X_{\texttt{assume}\ \lambda} \nu}$$

$$\boxed{\overline{\Delta} \mid \Delta \vdash \lambda \to \nu}$$

$$\frac{X = O([]) \vdash X \mapsto \nu}{[] \mid \Delta \vdash OX \to \nu} \qquad \frac{\overline{\Delta'} \mid \Delta'' \vdash OX \to \nu}{\Delta +\!\!+ \overline{\Delta'} \mid \Delta'' \vdash OX \to O(\nu, \Delta(X))}$$

$$\frac{}{\overline{\Delta'} \mid \Delta \vdash X \to \Delta(X)} \qquad \frac{}{\overline{\Delta'} \mid \Delta \vdash \nu \to \nu}$$

$$\frac{\overline{\Delta'} \mid \Delta \vdash \lambda_l \to \nu_l \qquad \overline{\Delta'} \mid \Delta \vdash \lambda_r \to \nu_r \qquad \nu_l + \nu_r = \nu}{\overline{\Delta'} \mid \Delta \vdash \lambda_l + \lambda_r \to \nu} \qquad \frac{\overline{\Delta'} \mid \Delta \vdash \lambda_l \to \nu_l \qquad \overline{\Delta'} \mid \Delta \vdash \lambda_r \to \nu_r \qquad \nu_l - \nu_r = \nu}{\overline{\Delta'} \mid \Delta \vdash \lambda_l - \lambda_r \to \nu} \qquad \frac{\overline{\Delta'} \mid \Delta \vdash \lambda_l \to \nu_l \qquad \overline{\Delta'} \mid \Delta \vdash \lambda_r \to \nu_r \qquad \nu_l * \nu_r = \nu}{\overline{\Delta'} \mid \Delta \vdash \lambda_l * \lambda_r \to \nu}$$

$$\boxed{\Omega \mid \varsigma \vdash \hat{B} \dashv \varsigma}$$

$$\frac{\hat{B}' = \texttt{buffer group}\ B' : T\ \texttt{order by}\ o\ \{\overline{S'}\} \qquad \hat{B}' \in \overline{\hat{B}}}{\Omega \mid \langle \overline{B}, \overline{\rho}, \overline{\hat{B}} \rangle \vdash \hat{B}' \dashv \langle \overline{B} \cup B', \overline{\rho}, \overline{\hat{B}} \rangle}$$

$$\frac{\hat{B}'' = \texttt{buffer group}\ B' : T\ \texttt{order by}\ o\ \{\overline{S'}\} \qquad \hat{B}'' \notin \overline{\hat{B}'} \qquad \hat{B}'' \in \overline{\hat{B}}}{\langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}}, \hat{A}, \hat{M} \rangle \mid \langle \overline{B}, \overline{\rho}, \overline{\hat{B}'} \rangle \vdash \hat{B}'' \dashv \langle \overline{B} \cup B', \overline{\rho}, \overline{\hat{B}'} +\!\!+ \hat{B}'' \rangle}$$

Fig. A.33. Operational semantics—buffer group ordering, Part 2.

`choose` expression and tracks the current state of its permutation. Essentially, the last (right-most) variable in that list makes a step every time, moving a variable from the to-be-used list on the right to the already-used list on the left. Once a variable has exhausted that list, it resets and lets the variable to its left take a step. The first rule in Fig. A.28 deals with the case when the first (left-most) variable has exhausted all its options: in this case, we have examined all permutations without success, so we return the continuation arbiter rules and pop the current entry from the choice stack. The second rule covers the left-propagation of making steps when all options are exhausted, while the last rule tries to generate an instantiation of the current `choose` expression's arbiter rules, and makes the last variable take a step as discussed above. Because each variable has been initialized with the same list of potential event sources, this would by default generate a k-tuple, so a helper relation makes sure we skip variables that have already been used. This may end up with some variable being out of options, in which case we just generate an empty list of arbiter rules to move on to the next attempt at generating a permutation.

A.6.2. Buffer match expressions

Fig. A.29 shows the rules for evaluating buffer match expressions. The top rule makes sure to either retrieve the local copy of some buffer state for the current round, or makes that copy if it does not exist. Then, for each form of buffer match expressions, there are two rules: one for the case where the expression matches the current state of the referred-to buffer, and one for the case where it does not. In the second case, we simply adjust the `where` expression of the rule to be false. Otherwise, the first three cases leave the expression and statement unchanged, while the last case substitutes the pattern-match variables in the expression and adds definition statements for them in the statement (code block). Note that each stream type has a name for `shared` event whose fields precede the fields of each of the other events. A buffer match pattern can just refer to this shared event to match against *any* event in the buffer, just referring to the fields shared by all.

A.7. Statements

The semantics for statements are shown in Fig. A.30. The first rule refers to the semantics of statements in the base language, while the second rule allows expressions nested in a statement to affect the base language program state. The other rules are similarly straightforward: adjusting stream fields, dropping events from buffers, forwarding events to the monitor, and adjusting membership in buffer groups work just as one would expect.

$\boxed{\Omega \vdash \tilde{\sigma}[E/E(\overline{X} \mapsto \overline{v}) : T] \to \tilde{\sigma}}$

$$\frac{\texttt{stream type}\ T(\Gamma\ ;\ \overline{\hat{X}})\ \texttt{shared}\ E_S(\Gamma_S)\ \{\ \overline{E'(\Gamma')} +\!\!+ E(\overline{\Gamma}) +\!\!+ \overline{E''(\Gamma'')}\ \} \in \overline{\hat{T}} \qquad \Gamma_S +\!\!+ \Gamma = \overline{X : \tau}}{\langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}}, \hat{A}, \hat{M} \rangle \vdash \tilde{\sigma}[E/E(\overline{X} \mapsto \overline{v}) : T] \to \{\overline{\tau\ X\ =\ v} +\!\!+ \tilde{\sigma}\}}$$

$$\frac{\texttt{stream type}\ T(\Gamma\ ;\ \overline{\hat{X}})\ \texttt{shared}\ E(\overline{X : \tau})\ \{\ \overline{E''(\Gamma')}\ \}}{\langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}}, \hat{A}, \hat{M} \rangle \vdash \tilde{\sigma}[E/E'(\overline{X} \mapsto \overline{v} +\!\!+ \overline{v'}) : T] \to \{\overline{\tau\ X\ =\ v} +\!\!+ \tilde{\sigma}\}}$$

$\boxed{\Omega \vdash \Sigma \not\leq^T \overline{E}}$

$$\frac{}{\overline{\Omega \vdash [] \not\leq^T \overline{E}}}$$

$$\frac{\texttt{stream type}\ T(\Gamma\ ;\ \overline{\hat{X}})\ \texttt{shared}\ E_S(\Gamma_S)\ \{\ \overline{E''(\Gamma'')}\ \} \in \overline{\hat{T}} \qquad \langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}}, \hat{A}, \hat{M} \rangle \vdash \Sigma \not\leq^T \overline{E'}}{\langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}}, \hat{A}, \hat{M} \rangle \vdash E(\overline{v}) +\!\!+ \Sigma \not\leq^T E''' +\!\!+ \overline{E'}}$$

$$\frac{\texttt{stream type}\ T(\Gamma\ ;\ \overline{\hat{X}})\ \texttt{shared}\ E_S(\Gamma_S)\ \{\ \overline{E''(\Gamma'')}\ \} \in \overline{\hat{T}} \qquad E''' \neq E_S \qquad E''' \neq E}{\langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}}, \hat{A}, \hat{M} \rangle \vdash E(\overline{v}) +\!\!+ \Sigma \not\leq^T E''' +\!\!+ \overline{E'}}$$

$\boxed{\vdash \overline{p}\ \textbf{not done}}$

$$\frac{\Sigma +\!\!+ \Sigma' \neq []}{\vdash \langle S, P, \Delta, \Sigma, \dot{e}, k, \Sigma' \rangle +\!\!+ \overline{p}\ \textbf{not done}} \qquad \frac{\vdash \overline{p}\ \textbf{not done}}{\vdash \langle S, P, \Delta, [], \dot{e}, k, [] \rangle +\!\!+ \overline{p}\ \textbf{not done}}$$

$\boxed{\Omega \vdash P : T \to T}$

$$\frac{\texttt{stream processor}\ P(\Gamma)\ :\ T(\overline{\xi})\ \to\ T'(\overline{\xi'})\ \{\overline{p}; \hat{H}\} \in \overline{\hat{P}}}{\langle \overline{\hat{T}}, \overline{\hat{S}}, \overline{\hat{P}}, \overline{\hat{B}}, \hat{A}, \hat{M} \rangle \vdash P : T \to T'}$$

Fig. A.34. Operational semantics—helpers, Part 1.

$\boxed{\overline{\rho} \mid \varsigma \vdash S \mapsto \rho \dashv \varsigma}$

$$\frac{\varsigma = \langle \overline{B}, \overline{\rho''}, \overline{\hat{B}} \rangle \qquad \rho' \notin \overline{\rho''} \qquad \rho' \in \overline{\rho} \qquad \rho' = \langle S, P, \Delta, \Sigma, \dot{e}, k, \Sigma' \rangle}{\overline{\rho} \mid \varsigma \vdash S \mapsto \rho' \dashv \varsigma} \qquad \frac{\rho' \notin \overline{\rho''} \qquad \rho' \in \overline{\rho} \qquad \rho' = \langle S, P, \Delta, \Sigma, \dot{e}, k, \Sigma' \rangle}{\overline{\rho} \mid \langle \overline{B}, \overline{\rho''}, \hat{B} \rangle \vdash S \mapsto \rho' \dashv \langle \overline{B}, \overline{\rho''} +\!\!+ \rho', \overline{\hat{B}} \rangle}$$

$\boxed{\mathit{stream}(b) \mapsto S}$

b	$\mathit{stream}(b)$
S : done	S
S : nothing	S
S : n	S
S : $\overline{E(\overline{X})}$	S

$\boxed{\overline{\hat{E}} \mid E(\Gamma) \vdash E : \Gamma}$

$$\frac{}{\overline{\hat{E}} \mid E(\Gamma) \vdash E : \Gamma} \qquad \frac{E(\Gamma) \in \overline{\hat{E}}}{\overline{\hat{E}} \mid E'(\Gamma') \vdash E : \Gamma' +\!\!+ \Gamma} \qquad \frac{E(\Gamma)\ \texttt{creates}\ T \in \overline{\hat{E}}}{\overline{\hat{E}} \mid E'(\Gamma') \vdash E : \Gamma' +\!\!+ \Gamma}$$

$\boxed{\hat{X} \vdash X \mapsto v}$

$$\frac{}{X = \mathit{count}(\overline{E.X}) \vdash X \mapsto 0} \qquad \frac{}{X = \mathit{sum}(\overline{E.X}) \vdash X \mapsto 0}$$

$$\frac{}{X = \mathit{prod}(\overline{E.X}) \vdash X \mapsto 1}$$

$$\frac{}{X = \mathit{max}(\overline{E.X}) \vdash X \mapsto -\infty} \qquad \frac{}{X = \mathit{min}(\overline{E.X}) \vdash X \mapsto \infty}$$

Fig. A.35. Operational semantics—helpers, Part 2.

A.8. Expressions and helper relations

Fig. A.31 deals with extended expressions, simply referring to the base language semantics for most except accessing stream fields, which is straightforward.

Figs. A.32 and A.33 define the rules to adjust the ordering of buffers in buffer groups. The first set of rules just apply this sorting to all buffer groups, retrieving the new order of streams within each group. The second set of rules either applies the `round-robin` sorting procedure directly, or uses the third set of rules to do an insertion sort based on the specified order expression. For pairwise comparisons, the fourth set of rules either compares stream fields, or tries to compare a shared field in the first event of each buffer, replacing that value based on an aggregation expression λ if a buffer is empty. A.33 contains some helper relations to do all this.

Figs. A.34 and A.35 define various helper relations used in other rules. The first set of rules is used to insert variable definitions into statements for variables used in event pattern matches. The second defines when a given buffer does not match an event pattern, taking into account the shared event of a stream type. The third checks whether the arbiter can expect to see any more events. Then, some predicates process the lookup of current stream and buffer group states if they are not already cached for a given round of checking arbiter rules, while the final three are basic lookup functions for rules used earlier.

References

[1] I. Cassar, A. Francalanza, L. Aceto, A. Ingólfsdóttir, A survey of runtime monitoring instrumentation techniques, in: PrePost@iFM 2017, in: EPTCS, vol. 254, 2017, pp. 15–28.
[2] A. Pnueli, A. Zaks, PSL model checking and run-time verification via testers, in: FM 2006, 2006, pp. 573–586.
[3] L. Bozzelli, C. Sánchez, Foundations of boolean stream runtime verification, Theor. Comp. Sci. 631 (2016) 118–138, https://doi.org/10.1016/j.tcs.2016.04.019.
[4] M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, A. Schramm, TeSSLa: runtime verification of non-synchronized real-time streams, in: SAC 2018, 2018, pp. 1925–1933.
[5] Y. Joshi, G.M. Tchamgoue, S. Fischmeister, Runtime verification of LTL on lossy traces, in: SAC 2017, 2017, pp. 1379–1386.
[6] M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, D. Thoma, Runtime verification for timed event streams with partial information, in: RV 2019, 2019, pp. 273–291.
[7] R. Taleb, R. Khoury, S. Hallé, Runtime verification under access restrictions, in: FormaliSE@ICSE 2021, 2021, pp. 31–41.
[8] G. Cugola, A. Margara, Processing flows of information: from data stream to complex event processing, ACM Comput. Surv. 44 (3) (2012) 15:1–15:62, https://doi.org/10.1145/2187671.2187677.
[9] M. Vierhauser, R. Rabiser, P. Grünbacher, K. Seyerlehner, S. Wallner, H. Zeisel, ReMinds: a flexible runtime monitoring framework for systems of systems, J. Syst. Softw. 112 (2016) 123–136, https://doi.org/10.1016/j.jss.2015.07.008.
[10] R. Rabiser, S. Guinea, M. Vierhauser, L. Baresi, P. Grünbacher, A comparison framework for runtime monitoring approaches, J. Syst. Softw. 125 (2017) 309–321, https://doi.org/10.1016/j.jss.2016.12.034.
[11] K. Tawsif, J. Hossen, J.E. Raja, M.Z.H. Jesmeen, E.M.H. Arif, A review on complex event processing systems for big data, in: CAMP 2018, 2018, pp. 1–6.
[12] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, M. Garofalakis, Complex event recognition in the big data era: a survey, VLDB J. 29 (1) (2019) 313–352, https://doi.org/10.1007/s00778-019-00557-w.
[13] Apache Software Foundation, Apache Flink, https://flink.apache.org/, 2023.
[14] M. Chalupa, F. Muehlboeck, S.M. Lei, T.A. Henzinger, Vamos: middleware for best-effort third-party monitoring, in: L. Lambers, S. Uchitel (Eds.), Fundamental Approaches to Software Engineering, Springer Nature, Switzerland, Cham, 2023, pp. 260–281.
[15] freedesktop org Wayland, visited 31-10-2023. URL, https://wayland.freedesktop.org/.
[16] H. Kallwies, M. Leucker, M. Schmitz, A. Schulz, D. Thoma, A. Weiss, TeSSLa - an ecosystem for runtime verification, in: RV 2022, 2022, pp. 314–324.
[17] D. Bruening, Q. Zhao, S. Amarasinghe, Transparent dynamic instrumentation, in: VEE 2012, 2012, pp. 133–144.
[18] B. Cantrill, M.W. Shapiro, A.H. Leventhal, Dynamic instrumentation of production systems, in: USENIX 2004, 2004, pp. 15–28, http://www.usenix.org/publications/library/proceedings/usenix04/tech/general/cantrill.html.
[19] B. Gregg, DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD, Prentice Hall, 2011.
[20] C.M. Rosenberg, M. Steffen, V. Stolz, Leveraging DTrace for runtime verification, in: RV 2016, 2016, pp. 318–332.
[21] freedesktop org Libinput, visited 31-10-2023. URL https://gitlab.freedesktop.org/libinput/libinput.
[22] C. Lattner, V.S. Adve, LLVM: a compilation framework for lifelong program analysis & transformation, in: CGO 2004, 2004, pp. 75–88.
[23] M. Chalupa, F. Muehlboeck, S. Muroya Lei, T.A. Henzinger, VAMOS: middleware for best-effort third-party monitoring, artifact, https://doi.org/10.5281/zenodo.7574687, 2024.
[24] F. Muehlboeck, T.A. Henzinger, Differential monitoring, in: RV 2021, 2021, pp. 231–243.
[25] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, D. Thoma, TeSSLa: temporal stream-based specification language, in: SBMF 2018, 2018, pp. 144–162.
[26] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, Eraser: a dynamic data race detector for multithreaded programs, ACM Trans. Comput. Syst. 15 (4) (1997) 391–411, https://doi.org/10.1145/265924.265927.
[27] K. Serebryany, T. Iskhodzhanov, ThreadSanitizer: data race detection in practice, in: WBIA 2009, 2009, pp. 62–71.
[28] Valgrind, Helgrind, https://valgrind.org/docs/manual/hg-manual.html, 2023.
[29] T. Elmas, S. Qadeer, S. Tasiran, Goldilocks: a race and transaction-aware Java runtime, in: PLDI 2007, 2007, pp. 245–255.
[30] M.A. Thokair, M. Zhang, U. Mathur, M. Viswanathan, Dynamic race detection with O(1) samples, PACMPL 7 (POPL), https://doi.org/10.1145/3571238, January 2023.
[31] D. Beyer, Progress on software verification: SV-COMP 2022, in: TACAS 2022, 2022, pp. 375–402.
[32] R. Ganguly, A. Momtaz, B. Bonakdarpour, Distributed runtime verification under partial synchrony, in: Q. Bramas, R. Oshman, P. Romano (Eds.), 24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France, Virtual Conference, in: LIPIcs, vol. 184, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 20:1–20:17.
[33] D. Kroening, M. Tautschnig, CBMC - C bounded model checker - (competition contribution), in: TACAS 2014, in: Lecture Notes in Computer Science, vol. 8413, Springer, 2014, pp. 389–391.
[34] M. Mansouri-Samani, M. Sloman, Monitoring distributed systems, IEEE Netw. 7 (6) (1993) 20–30, https://doi.org/10.1109/65.244791.
[35] B. Zhao, N.Q. Viet Hung, M. Weidlich, Load shedding for complex event processing: input-based and state-based techniques, in: ICDE 2020, 2020, pp. 1093–1104.
[36] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, Z. Manna, LOLA: runtime monitoring of synchronous systems, in: TIME 2005, 2005, pp. 166–174.
[37] P. Faymonville, B. Finkbeiner, S. Schirmer, H. Torfah, A stream-based specification language for network monitoring, in: RV 2016, 2016, pp. 152–168.
[38] F. Gorostiaga, C. Sánchez, Striver: stream runtime verification for real-time event-streams, in: RV 2018, 2018, pp. 282–298.

[39] H. Barringer, Y. Falcone, K. Havelund, G. Reger, D.E. Rydeheard, Quantified event automata: towards expressive and efficient runtime monitors, in: FM 2012, 2012, pp. 68–84.
[40] G. Reger, H.C. Cruz, D. Rydeheard, MarQ: monitoring at runtime with QEA, in: TACAS 2015, 2015, pp. 596–610.
[41] D. Basin, M. Gras, S. Krstić, J. Schneider, Scalable online monitoring of distributed systems, in: J. Deshmukh, D. Ničković (Eds.), Runtime Verification, Springer International Publishing, Cham, 2020, pp. 197–220.
[42] F. Chen, G. Rosu, Parametric trace slicing and monitoring, in: TACAS 2009, 2009, pp. 246–261.
[43] S. Hallé, When rv meets cep, in: Y. Falcone, C. Sánchez (Eds.), Runtime Verification, Springer International Publishing, Cham, 2016, pp. 68–91.
[44] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, O. Sokolsky, Formally specified monitoring of temporal properties, in: ECRTS 1999, 1999, pp. 114–122.
[45] K. Havelund, G. Rosu, Monitoring Java programs with Java pathexplorer, in: RV 2001, 2001, pp. 200–217.
[46] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, An overview of AspectJ, in: ECOOP 2001, 2001, pp. 327–353.
[47] M. Kim, S. Kannan, I. Lee, O. Sokolsky, M. Viswanathan, Java-MaC: a run-time assurance tool for Java programs, in: RV 2001, 2001, pp. 218–235.
[48] M. Karaorman, J. Freeman, jMonitor: Java runtime event specification and monitoring library, in: RV 2004, 2005, pp. 181–200.
[49] A. Eustace, A. Srivastava, ATOM: a flexible interface for building high performance program analysis tools, in: USENIX 1995, 1995, pp. 303–314, https://www.usenix.org/conference/usenix-1995-technical-conference/atom-flexible-interface-building-high-performance.
[50] B. De Bus, D. Chanet, B. De Sutter, L. Van Put, K. De Bosschere, The design and implementation of FIT: a flexible instrumentation toolkit, in: PASTE 2004, 2004, pp. 29–34.
[51] C. Luk, R.S. Cohn, R. Muth, H. Patil, A. Klauser, P.G. Lowney, S. Wallace, V.J. Reddi, K.M. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in: PLDI 2005, 2005, pp. 190–200.
[52] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, in: PLDI 2007, 2007, pp. 89–100.
[53] D. Drusinsky, Monitoring temporal rules combined with time series, in: CAV 2003, 2003, pp. 114–117.
[54] H. Barringer, A. Goldberg, K. Havelund, K. Sen, Rule-based runtime verification, in: VMCAI 2004, 2004, pp. 44–57.
[55] F. Chen, G. Roşu, Java-MOP: a monitoring oriented programming environment for Java, in: TACAS 2005, 2005, pp. 546–550.
[56] C. Colombo, G.J. Pace, G. Schneider, LARVA — safer monitoring of real-time Java programs (tool paper), in: SEFM 2009, 2009, pp. 33–37.
[57] A. Francalanza, A. Seychell, Synthesising correct concurrent runtime monitors, Form. Methods Syst. Des. 46 (3) (2015) 226–261, https://doi.org/10.1007/s10703-014-0217-9.
[58] J. Ha, M. Arnold, S.M. Blackburn, K.S. McKinley, A concurrent dynamic analysis framework for multicore hardware, in: OOPSLA 2009, 2009, pp. 155–174.
[59] E. Bartocci, R. Grosu, A. Karmarkar, S.A. Smolka, S.D. Stoller, E. Zadok, J. Seyster, Adaptive runtime verification, in: RV 2012, 2012, pp. 168–182.
[60] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S.A. Smolka, S.D. Stoller, E. Zadok, Software monitoring with controllable overhead, Int. J. Softw. Tools Technol. Transf. 14 (3) (2012) 327–347, https://doi.org/10.1007/s10009-010-0184-4.
[61] P. Arafa, H. Kashif, S. Fischmeister, Dime: time-aware dynamic binary instrumentation using rate-based resource allocation, in: EMSOFT 2013, 2013, pp. 1–10.
[62] M. Kim, S. Kannan, I. Lee, O. Sokolsky, M. Viswanathan, Computational analysis of run-time monitoring - fundamentals of Java-mac, in: RV 2002, 2002, pp. 80–94.
[63] S. Kauffman, K. Havelund, S. Fischmeister, What can we monitor over unreliable channels?, Int. J. Softw. Tools Technol. Transf. 23 (4) (2021) 579–600, https://doi.org/10.1007/s10009-021-00625-z.
[64] S.D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S.A. Smolka, E. Zadok, Runtime verification with state estimation, in: RV 2011, 2012, pp. 193–207.
[65] V.T. Valapil, S. Yingchareonthawornchai, S.S. Kulkarni, E. Torng, M. Demirbas, Monitoring partially synchronous distributed systems using SMT solvers, in: S.K. Lahiri, G. Reger (Eds.), Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings, in: Lecture Notes in Computer Science, vol. 10548, Springer, 2017, pp. 277–293.
[66] Y. Falcone, J. Fernandez, L. Mounier, What can you verify and enforce at runtime?, Int. J. Softw. Tools Technol. Transf. 14 (3) (2012) 349–382, https://doi.org/10.1007/s10009-011-0196-8.