Visualization of Execution Traces in the Colibri 2 SMT Solver

Christophe Junke[1], François Bobot[1]

[1]Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Abstract

SMT solvers are complex tools that can be hard to debug. We think that there is a need to add a layer of observability to gain a better understanding of how they work. We present our recent addition to Colibri2 to add structured traces in the TEF format, and its associated Vite/ReactJS vizualization front-end. We present the motivation behind this, the current design and some preliminary results.

Keywords

SMT, debugging, observability, user interface, vizualization, logging

1. Introduction

Unlike verification tools, SMT solvers are usually not designed to be interactive. Instead, they are used as backend tools to which proofs can be offloaded (Z3, Alt-Ergo as engines, versus Why3, Frama-C, Spark environments). As such, SMT solvers may offer a programmatic interface (such as *Smt-Switch* [1]) or a command-line interface, where typically inputs and outputs are expressed in the SMT-LIB language (in a terminal, this offers a simple REPL behavior).

From the point of view of both users and developers, it is important to build and maintain an understanding of how a solver works. This may be challenging due to the complex nature of this kind of software.

Colibri2[1] is an SMT solver written in OCaml. For debugging purposes it can emit log messages depending on the flags given to it in its options. For example it is possible to observe only the messages coming from a specific theory. This and the usual text filtering tools available to programmers is the main way we can analyze the behavior of the solver. Colibri2 also keep tracks of some internal metrics, emitted at the end of its execution.

Using logging for debugging and profiling has its limitations: it is hard to isolate how a particular node evolves in the system, given that it may be tied to various identifiers over time (union-find merge operations), and it is hard to vizualize some dynamic aspects of the solver, like the time spent in various parts of the system (e.g. to produce flamegraphs).

Complex systems like distributed large-scale web servers rely on continuous observation of their components. We develop a prototype user-interface for Colibri2 which relies on the structured tracing format TEF (Trace Event Format) from Google's Catapult Trace Viewer [2]. The program emits events that make up a trace file in JSON. This file is a summary of the program execution. It is intended to record all the important data from the program for a later analysis. For this reason it is also a useful artefact for users who want to report bugs.

Compared to a graphical interface that would be integrated in the solver, this approach is also less intrusive: any part of the solver where we need to keep track of some internal data can do it by emitting trace events. In return, this requires downstream tools to be able to interpret some events. The TEF format is such that that all events can be displayed in a generic way, but there is a reconstruction step that is necessary, for example, to take care of domain updates of nodes to build a graph of changes over time.

SMT 2025: 23rd International Workshop on Satisfiability Modulo Theories, August 10–11, 2025, Glasgow, UK

christophe.junke@cea.fr (C. Junke); francois.bobot@cea.fr (F. Bobot)

[1]https://colibri.frama-c.com/

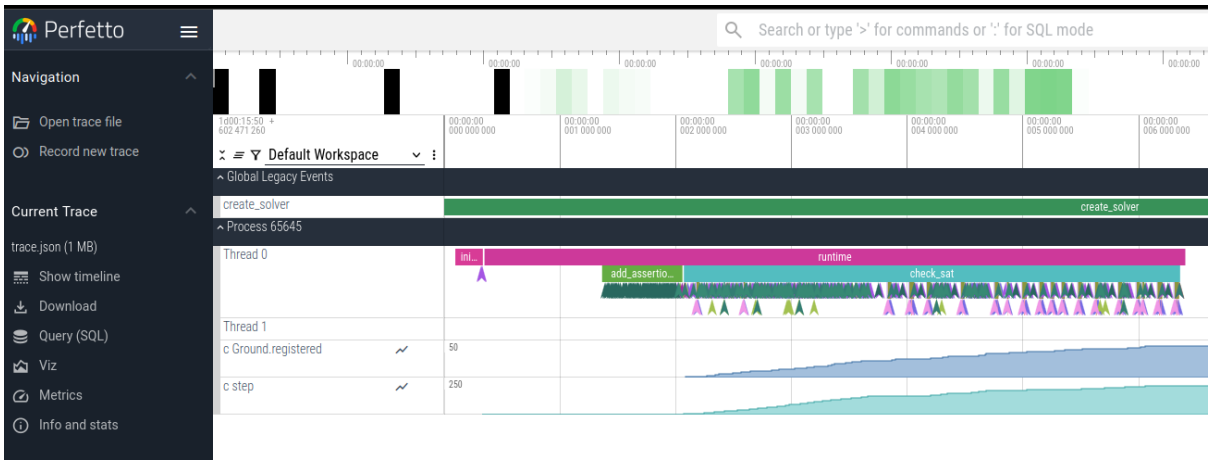


Figure 1: Screenshot from the Perfetto UI tool from Google after loading a trace generated by Colibri2

1.1. Related Work

There are existing efforts to add graphical user interfaces to SMT solvers. For example, `AltGr-Ergo` [3] is a GUI that displays internal information about Alt-Ergo (unsat cores, time spent in various computations, etc.) and offers more control, interactively, than the SMT-LIB format (sessions, selection mechanisms). The intent is to provide feedback about the internal state of the solver.

The Axiom Profiler [4] is an interface that analyzes log outputs from Z3 to focus on debugging quantifier instantiations (in particular, matching loops). Z3Hydrant [5] is a tool that converts detailed log files from Z3 to sound, by relying on the human ability to pick out patterns. These tools rely on the tracing format of Z3, which can be emitted in an extension of SMT-LIB2. Likewise, CVC5 emits diagnostic outputs in SMT-LIB2 and statistics in a separate format [6].

1.2. Outline

The rest of the paper is organized around two main parts. The first one focuses on the trace generation side, and in particular how we encode specific information from our tool using a general-purpose trace format. The second part in section 3 presents the general architecture of the graphical user-interface and the motivation behind it.

2. Tracing

For the purpose of understanding microservices and distributed systems, several tools have been developed in the last years around the concept of *observability* to help collect and analyze traces from systems [7]. We use the `ocaml-trace`[2] library to generate traces in the *Trace Event Format*, abbreviated TEF, from Google's Catapult Trace Viewer tool.

This approach requires to modify the existing code to emit traces at various points of the system, which is relatively easy to do. In return, the use of a publicly available format allows us to directly open the trace in existing tools, for example the Perfetto UI[3], as shown in figure 1. The TEF format can describe concepts such as objects or events spanning multiple processes and threads in distributed systems. We do not need the full range of events available in the language for our tool, and instead limit ourselves to a subset consisting of counter events, instant events and duration events[4].

[2] https://github.com/c-cube/ocaml-trace
[3] https://ui.perfetto.dev
[4] duration events, or spans, are described in our tool by a pair of *begin*/*end* events for the same key; by construction they nests according to function calls in our code, so that children spans are always contained in parent spans.

We emit messages that help us keep track of various changes inside our solver, which means that our events carry a meaning that is not directly interpretable by generic tracing tools.

2.1. Union-find node identifiers

We use a union-find datastructure [8, 9] to maintain a compact representation of terms in our solver. Each node in the graph is associated with an identifier, and merge operations can make two nodes equivalents. We generate trace events each time a new node is registered in the system, in order to track the identifier and its associated term. Likewise, node merges and domain updates of nodes are also traced.

When we need to print a node in a trace event, we print instead its identifier. In our user interface, we want to extract all the identifiers referenced in a trace payload to be able to show more information about each node (e.g. its current domain, or the last time it was modified, etc.)

Identifiers are prefixed by an *at* character: `@1`, `@2`, etc. The patterns of identifiers is however not regular, because some nodes use other names that give more information when debugging, like identifiers for numerical and logical constants: `@0R`, `@⊥`, etc. Moreover, these identifiers may be pretty-printed in contexts where they can be ambiguous to parse back, like in domain values, which are currently only represented as text and not structured values. Unless developers are really careful about it, a pretty-printer may introduce characters that render the parsing ambiguous. For example, given an arbitrary domain value for which the printed representation is `@0R`, we cannot tell if the string represents identifier 0 followed by `R` or identifier `0R`.

In order to extract these identifiers from events, we changed our pretty-printer for identifiers to add semantic tags [10] around them, and we change our formatters in the context of a trace event so that identifiers are surrounded by the non-printable characters STX and ETX (resp. ASCII codes 2 and 3).

This is the current workaround to be able to correctly parse node identifiers from messages that rely on our already existing pretty-printers. An alternative approach would be to ensure that each datastructure we want to trace in our solver has an associated JSON representation. This is currently not the case and if we do so we will also have to encode these JSON representations inside our events. In that case it may be interesting to rely on *object events* as defined by TEF (ie. memory addresses for which we store snapshots at various points of time): contrary to instant messages which only allow primitive values, snapshots of objects may be arbitrary JSON values.

2.2. Scopes

Our SMT solver undergoes different phases of computation over the course of its execution. Some are captured with `with_span` constructs from `ocaml-trace`, which encapsulate a function call by emitting a duration event trace. Other scopes are handled explicitly, because they do not directly fit the extent of a function call. For example, multiple node merge operations can be executed in the same phase, delimited by `start_merge` and `finalize_merge` events.

2.3. Breadcrumbs

The Colibri2 solver is organized around a central scheduler that manages multiple queues of events. Each subsystem can register callbacks to be executed when some future internal event is fired (e.g. domain change of a given node). Between the time that this subsystem decides to delay an action and the time that action is executed, the scheduler may have interleaved a number of other tasks.

For debugging purposes we need to keep track of the context of the delayed action. There is some information that each module may decide to add in its traces, but we would like to have a generic mechanism to explain at minimum when the task being currently run was originally delayed.

The approach we implemented consists in emitting a couple of events when an event enters a queue and when it leaves it. They are respectively named `breadcrumbs_push` and `breadcrumb_pop`, and requires two parameters: a unique identifier for the current breadcrumb, and the name of the queue.

For example, in the following simplified trace, two nodes are merged, and the corresponding code delays two actions in the `merge_dom` queue. This is represented by the two `breadcumb_push` events that follow `start_merge`:


```
{"name":"start_merge","args": {"other":"@42","repr":"@0"}},
{"name":"breadcrumb_push","args": {"id":1,"queue":"merge_dom"}},
{"name":"breadcrumb_push","args": {"id":2,"queue":"merge_dom"}},
```


Later, we can see that there is a `breadcrumb_pop` operation that retrieves the action whose `id` is 1 from the same queue. This event is followed by the `set_dom` event:


```
{"name":"breadcrumb_pop","args": {"id":1,"queue":"merge_dom"}},
{"name":"set_dom","args": {"dom":"Ground.tys","node":"@0","val":"Prop,"}},
```

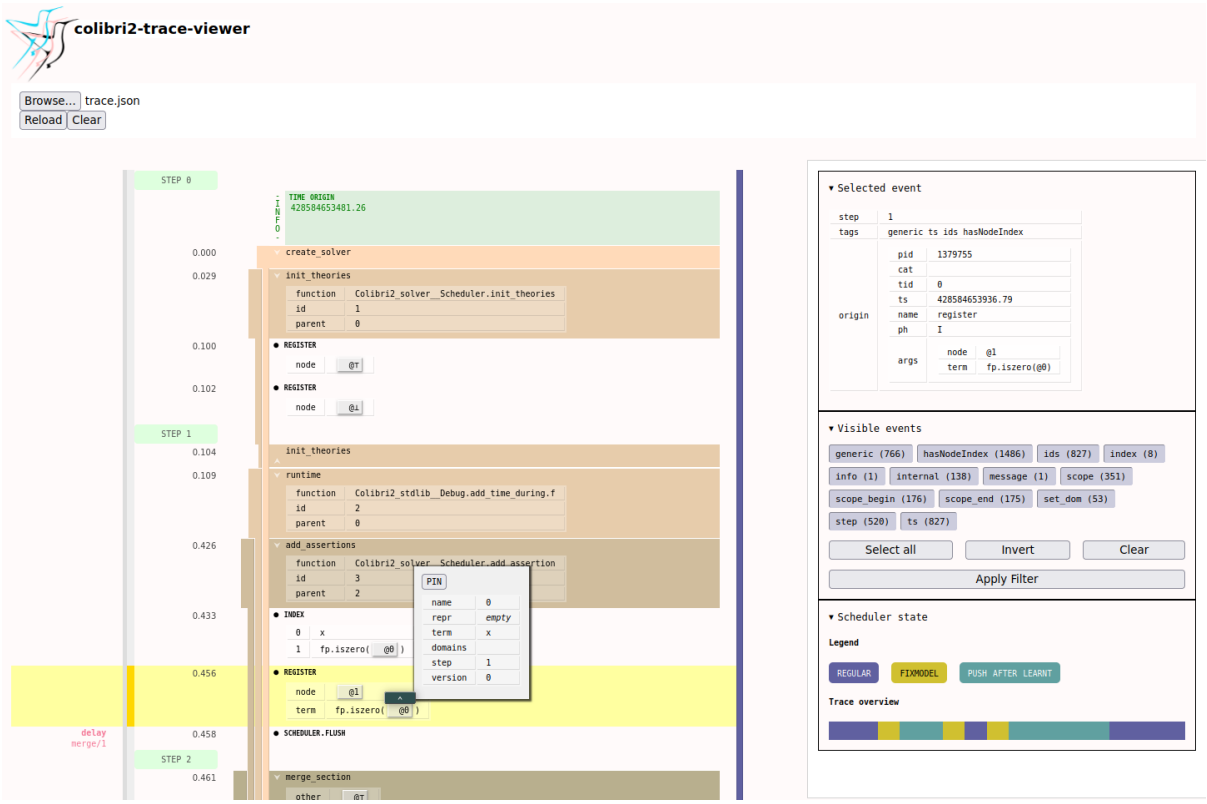

This sequence of events is enough to visually tie `set_dom` to `start_merge` in the interface by adding bidirectional links between their rows.

2.4. Counters

We maintain a set of counters in the solver, for which every change of value is traced. Notable counters are the `step` counter, which marks the increasing tick at which the scheduler is being run. We also keep track of the internal time spent in various sections of the code. Currently we make no use of these counters in our interface but they are already visible in the Perfetto UI.

3. Graphical interface

The trace viewer is a web interface based on ReactJS[5] and the Vite build tool[6]. It displays a trace as a table, where each row corresponds to an event.



[5] https://react.dev
[6] https://vite.dev

3.1. Design goals

One of our objectives with the interface is to be robust to changes in the trace format in face of updates to the software. The reason for this is that there might be traces generated by older or newer versions of the solver, which are not necessarily tied to the version of the interface (e.g. between developers or between users and developers). For example, by default we want to be able to display any trace event in a generic way, without entering an error state.

The second main objective is to be modular enough that adding support for particular type of events can be done by adding one specific module that attaches itself to various parts of the interface. We currently support adding independant side-panels from such modules.

3.2. Architecture

There are three main phases that occur when loading a trace file. The first one is a preprocessing step where all events are filtered, transformed and aggregated into an in-memory graph. The second phase is our rendering step, where the previous graph is used to assemble UI elements and event handlers in structures called *properties*. Finally, the third step is the React rendering step, where properties are turned into *reactive* HTML DOM[7] elements: typically, events generated through user interaction eventually reach the event handlers generated at step 2, which compute a modified set of properties that will be used to update the interface in-place. For example, hovering over a node identifier creates a new popup window: all the necessary information for the popup was computed once during the filtering pass, but during step 2 we generate components that react on mouse events to create or destroy popup elements.

The remaining of this section first looks at the rendering step and how row properties are represented. Then we go backward to the preprocessing step to describe how the properties are generated by using a pipeline of filters. We describe two general-purpose filters and finally present an example of how a particular type of trace event is transformed and rendered in this architecture.

3.2.1. React properties for rows

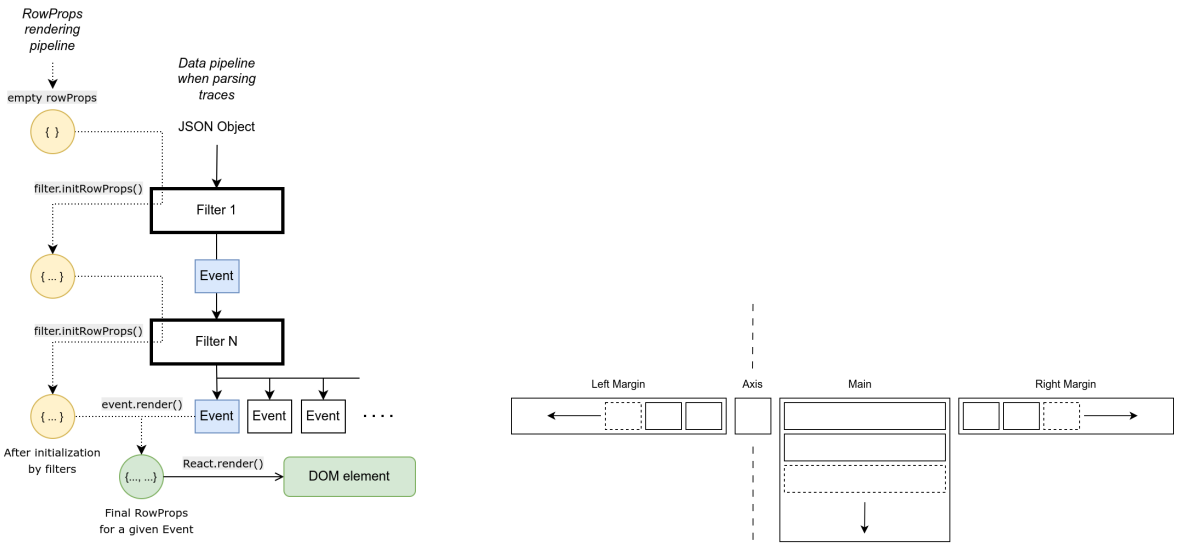
In order to build and update DOM elements, the functional API of React expect the programmer to define and instantiate *Components*, which are functions that transform *properties*, regular Javascript objects, into React objects that represents DOM elements (in our cases, using JSX). By using React *hooks*, the input properties can be updated in a way that React can observe and act upon, in order to update the generated DOM elements.

We focus here on properties called *RowProps*, which are properties that describe one row of the trace view. Figure 2b shows the basic structure of one *RowProps*, which acts like a blueprint for the final rendering of a table row. There are various fields like the central axis where we display either the current step or the current timestamp, the main content field, and extensible areas on the left and the right of the axis. It contains also various properties like a list of CSS *classes* attached to the row, possible DOM event callbacks, etc.

We rely on the data processing step to produce a sequence of *RowProps* values by successively transforming events through filters, and finally *rendering* all the final events as *RowProps*. Figure 2a describes this processing: initially the sequence of events is extracted from a stream of JSON entries to produce a sequence of events. Later, the rendering phase produces, for each *Event*, the *RowProps* instance required to render a *Row* component in React.

From a performance perspective, we are careful about not re-rendering the whole table of rows. For example, when the current row selection changes, the last row is unselected and the new one selected. A naive approach would recompute the set of all row properties for the other rows.

[7] Document Object Model



(a) Transformation pipeline from JSON to Events through filters during data processing, and rendering of events as rowProps: each rowProp structure is first initialized by all filters, then processed by the event's rendering method. The final structure is used to render a React Component for this Row.

(b) A RowProps structure represents the combination of different parts of the final layout: all rows are aligned on a central axis cell, a main cell which is a cell of one or more components, and two extensible margins on the left and right sides of the axis. Each filter can register a new index in a margin, so that events that are emitted by this filter can render some extra information there, without having to coordinate with other filters.

Figure 2: Simplified view of data-structures and their associated transformations.

3.2.2. Data preprocessing step

Events

Events are regular Javascript objects that are initially decoded from JSON as either `CounterEvent` or `GenericEvent` objects. During processing, they may be mutated or transformed into more specific classes, but the rest of the API only expects them to have a `render()` method that produces a `RowProps` value. The processing of trace events is architectured around the parsing of events through a series of *filters* executed in sequence.

Filters

Filters are Javascript objects built from a constructor that accepts only one input, the shared *environment* object. This object is where they can accumulate and share global data. We rely on the *immutable.js* library to use purely-functional datastructures, because events need to have access to different versions of this state over time. We also define a basic registration mechanism to document and check how resources are shared through filters. Filters also contribute to the rendering of row properties and possibly other parts of the interface. For example, the environment allows a filter to add a custom panel on the right-side of the screen, making filters the main extension point of the architecture.

Filters are analogous to POSIX pipes: each filter may produce zero, one or more events for the next filter in the pipeline using its `step` function, and may update its internal state while doing so. This is done only for events that match a `test` predicate (otherwise the event bypasses the current filter unmodified).

For example, message events in JSON are transformed into objects of type *GenericEvent*. Below we will see that counter events named `step` are transformed in instances of a `StepEvent` class. In object-oriented fashion, specialized objects have dedicated methods to be rendered as UI components, whereas `GenericEvent` objects only render themselves as a table of key/value pairs.

Each filter has also the opportunity to initially produce a set of events from nothing (`init` method).

Conversely, when there is no more inputs, a filter can also purge its internal state by emitting a last sequence of events (`done` method): for example, it is possible to sort all events inside a filter, by accumulating all incoming events internally and emitting them in a different order.

Finally, to accomodate for the fact that filters may have an internal state to manage, they may also define `reset` and `exit` methods, which are only used for side-effects at the beginning and end of the processing.

During the rendering phase, filters contribute to each row by initializing the current row, using `initRowProps`. By default it is the identity function, but in some cases the filter can decorate rows with custom values.

Filters need not define all the above functions, all of them are optional because we first normalize filters to give them default behaviors:

```
return {
  ...filter,
  env     : filter.env     || env,
  test    : filter.test    || ((t) => true),
  init    : filter.init    || (( ) => []),
  reset   : filter.reset   || (( ) => undefined),
  exit    : filter.exit    || (( ) => undefined),
  step    : filter.step    || ((t) => [t]),
  done    : filter.done    || (( ) => []),
  initRowProps : filter.initRowProps || ((l) => l)
}
```

The main pipeline of our tool consists currently of 14 filters, which focus on distinct types of events or processing. By doing manual tests we try to deactivate or reorder them to identify how much they depend on each other, and if so, we try to reduce this coupling when possible.

3.2.3. Composite filters

We define two high-level filters that work an an arbitrary array of filters, namely `Chain` and `Select`.

Chain The chain filter enforces the lifecycle described above when a series of filters are applied in sequence. For example, the `init` method of the `Chain` filter iterates over all child filters: each filter f_i may produce events from its `init` method, and they must be first given to the `test` and `step` functions of the next filters f_{i+1}, f_{i+2}, etc. Only once these events a processed, we may call the `init` method of f_{i+1}. Another detail is that `exit` methods are called in the reverse order of filters, to correctly undo previous calls to `reset` with respect to resources shared among filters.

Select The selection filter redirects events to exactly one filter among an array of filters: its constructor takes as a parameter a selector function which is called on *Configuration* events. When such an event is observed, the selection filter may deactivate the current filter and activate a different one. This is used to allow the definition of separate branches of filters based on the trace being read. The intended usage of this filter is to make it possible to handle different versions of the trace format, or to support different type of traces (generated by different tools). This filter takes care of properly cleaning the filter that is being deactivated (by calling `done` and `exit` methods), and reinitializes the new active filter (by calling `reset` and `init`).

3.2.4. StepEvent example

The code in figure 4 is the definition of the `Stepper` constructor function, which produces a filter that observes the `step` counter emitted by Colibri2 to produce intermediate `StepEvent` objects in the trace.

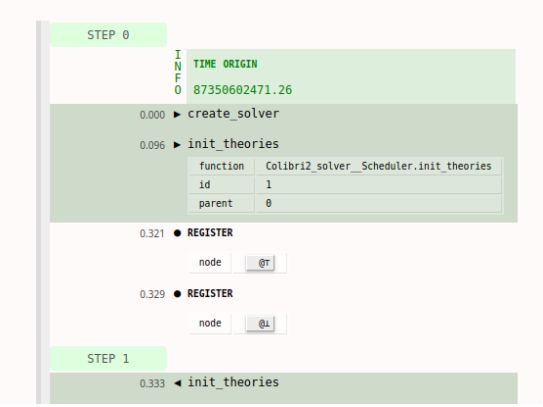
```
/* Transform rowProps to display event */
function render(event, rowProps) {
  return {
    /* inherit other fields */
    ...rowProps,
    /* add a CSS class to this row */
    classes: rowProps.classes.concat(["step"]),
    /* override the generic Axis component (timestamp) and show the step */
    axis: <span>{`step ${event.step}`}</span>
  }
}

export default function StepEvent(origin, step) {
  /* Return dedicated event for steps */
  return {
    tags: Set.of("step"),

    /* list of rendering functions applied lik
    a fold; we override the generic event
    renderer by using the function above */
    render: [render],

    step: step,
    origin: origin
  };
}
```

(a) Constructor for `StepEvent` events and the associated rendering function. Rendering is done by filling a structure representing a row that is iself the basis from which a `Row` component is rendered in React.



(b) Rendering of a test trace where `step` counter events have been transformed into `StepEvent` objects.

Figure 3: Step events and their rendering

The filter's `step` function is only called for events that match the `test` condition, namely counter events for which the associated name its `step`.

In that case, it produces two events: (i) the original event `e` (which may be useful for downstream filters), and (ii) a `StepEvent` object[8]. These events are used to visually distinguish each scheduler step over time, as shown in figure 3b, but also keep tracks of the last known `step` value. The shared environment `env` defines a mechanism to `register` and `request` access to shared state variables between filters: this establishes a dependency between filters, forcing filters that depends on `StepperState` to be executed after the one depicted here.

4. Conclusion

We presented a prototype implementation of a user-interface organized around interpreting traces of events emitted from our SMT solver Colibri2. The architecture of this interface can be adapted to accomodate other types of traces. The addition of event traces to a system feels less intrusive and costly than adding a graphical user-interface to an existing system and can help understand the behavior of a complex system. The chosen trace format was selected for its ability to be already understood by existing tools.

The filtering mechanism of the user-interface is designed to be modular and robust to changes. It can

[8] the `origin` field is used to remember the source JSON object that is responsible for this event.

```
import {and, isTagged, keyMatch} from '@/traces/FilterUtils';
import CounterEvent from '@events/CounterEvent.tsx';
import StepEvent from '@events/StepEvent.tsx';

export default function Filter (env) {
  var state
  return {
    description: "Inject step-changed events",
    test: and(isTagged("counter"), keyMatch("name", "step")),
    init: () => [ CounterEvent("step", 0), StepEvent(null, 0) ],
    reset: () => {
      state = env.register("StepperState")
      state.set("lastStep", 0)
    },
    exit: () => state.unregister(),
    step: (e) => {
      state.set("lastStep", e.value)
      return [e, StepEvent(e.origin, e.value)];
    }
  }
}
```

Figure 4: A `Stepper` filter that is introduces `StepEvent` events and keep track of the last known step value.

be used to analyse and process complex streams of events. For example, a current limitation is that the user interface renders the whole trace file, which can be impractical for large traces. We may however add a tool that splits traces into smaller traces, where each one starts with a header of events that build the required contextual state for each trace (e.g. reset node variable to their last known domains).

We currently use event traces to perform an offline analysis of the solver behavior, but this format does leave open the possibility of having a direct communication between our solver and a user-interface. To close the interaction loop we would however need to handle external events from our scheduler.

The use of web technology like React for the user-interface feels also beneficial in the context of this tool, as it allows for complex dynamic interactions with the data that is being analyzed while relying on a software platform that is now ubiquitous. A possible perspective in our case would be for example to rely on the generated trace to display the union-find graph over time, the evolution of variable domains using plots, or some internal state like simplex boundaries.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] M. Mann, A. Wilson, Y. Zohar, L. Stuntz, A. Irfan, K. Brown, C. Donovick, A. Guman, C. Tinelli, C. Barrett, Smt-switch: A solver-agnostic C++ API for SMT solving, in: C.-M. Li, F. Manyà (Eds.), Proceedings of the 24^{th} International Conference on Theory and Applications of Satisfiability Testing (SAT '21), volume 12831 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 377–386. URL: http://theory.stanford.edu/~barrett/pubs/MWZ+21.pdf. doi:10.1007/978-3-030-80223-3_26, barcelona, Spain.

[2] Google, Catapult trace viewer, 2015. URL: https://chromium.googlesource.com/catapult/+/HEAD/tracing/README.md.

[3] S. Conchon, M. Iguernelala, A. Mebsout, Altgr-ergo, a graphical user interface for the SMT solver alt-ergo, in: S. Autexier, P. Quaresma (Eds.), Proceedings of the 12th Workshop on User Interfaces

for Theorem Provers, UITP 2016, Coimbra, Portugal, 2nd July 2016, volume 239 of *EPTCS*, 2016, pp. 1–13. URL: https://doi.org/10.4204/EPTCS.239.1. doi:10.4204/EPTCS.239.1.

[4] N. Becker, P. Müller, A. J. Summers, The axiom profiler: Understanding and debugging smt quantifier instantiations, in: T. Vojnar, L. Zhang (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer International Publishing, Cham, 2019, pp. 99–116.

[5] F. Hackett, I. Beschastnikh, Listening to the firehose: Sonifying z3's behavior (????).

[6] A. Reynolds, Interfaces for understanding cvc5, 2024. URL: https://cvc5.github.io/blog/2024/04/15/interfaces-for-understanding-cvc5.html.

[7] A. Janes, X. Li, V. Lenarduzzi, Open tracing tools: Overview and critical comparison, Journal of Systems and Software 204 (2023) 111793. URL: https://www.sciencedirect.com/science/article/pii/S0164121223001887. doi:https://doi.org/10.1016/j.jss.2023.111793.

[8] C. G. Nelson, Techniques for program verification, Ph.D. thesis, Stanford, CA, USA, 1980. AAI8011683.

[9] L. de Moura, N. Bjørner, Efficient e-matching for smt solvers, in: F. Pfenning (Ed.), Automated Deduction – CADE-21, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 183–198.

[10] R. Bonichon, P. Weis, Format Unraveled, in: 28ièmes Journées Francophones des Langages Applicatifs, Gourette, France, 2017. URL: https://hal.science/hal-01503081.