International Journal on Software Tools for Technology Transfer (2024) 26:101–126
https://doi.org/10.1007/s10009-024-00736-3

GENERAL

Regular

Visualization, transformation, and analysis of execution traces with the eclipse TRACE4CPS trace tool

Martijn Hendriks[1] · Jacques Verriet[2] · Twan Basten[1,2]

Accepted: 28 November 2023 / Published online: 8 February 2024
© The Author(s) 2024

Abstract

An execution trace is a model of a single system behavior. Execution traces occur everywhere in the system's lifecycle as they can typically be produced by executable models, by prototypes of (sub)systems, and by the system itself during its operation. An execution trace can be visualized and analyzed with various techniques, providing insight into the dynamic behavior, performance, bottlenecks, etc., of the system. In this paper, we present the Trace tool of the Eclipse Trace4cps project for the visualization and analysis of execution traces. A prominent application is the trace-based performance engineering of embedded or cyber-physical systems. Performance is an important system quality, as it can give a competitive advantage. Reasoning about system-level performance in such systems, however, is hard due to its cross-cutting nature. We show how the Trace tool can support this by various examples. Performance engineering is not the only application of the Trace tool, however: it supports system analysis in a wide range of situations.

Keywords Embedded systems · Cyber-physical systems · Execution traces · Performance engineering · Performance analysis · Run-time verification · Trace-to-trace transformation · Bottleneck analysis · Trace comparison · Repetitive-structure analysis

1 Introduction

1.1 A tool for trace-based visualization and analysis

An embedded system is a computer system that is part of a larger mechanical or electronic system. A cyber-physical system integrates computation and physical processes. It may consist of several networked embedded subsystems that together monitor and control a physical process. The physical process affects the computation, and the computation affects the physical process [1]. Examples of embedded and cyber-physical systems from the high-tech industry include modern cars, wafer scanners, medical imaging systems, pick-and-place equipment, and production printers. We present the Trace tool of the Eclipse Trace4cps project [2] for trace-based visualization and analysis of embedded and cyber-physical systems. It is centered around the concept of an *execution trace*, which is a formal model of a single system behavior. Our execution traces consist of a discrete part (time-stamped *events* and *claims on resources* and *dependencies* between events and/or claims) and a continuous part (real-valued *signals* over time) and as such are tailored to the embedded and cyber-physical systems domain. The execution traces can be produced by executable models or prototypes of (sub)systems during system design or by an operational system. A prominent application of the tool is *trace-based performance engineering* of embedded and cyber-physical systems. We define system performance according to [3].

> **System performance**—the amount of useful work done by a system, measured in production speed of products of a predefined quality

Better system performance provides a competitive advange and is often a key attribute. Understanding, analyzing, and ultimately solving performance issues is, however, notoriously difficult. Image processing in software, for example, is complex due to pipelining, buffering, and resource sharing. Reasoning about the performance of such a software component is almost impossible without executable models that capture its behavior. Other software and electrical or mechanical components and networks that have to be considered in combination for the total system performance make matters worse: system performance is a cross-cutting concern. It is

✉ M. Hendriks
m.hendriks@tue.nl

[1] Eindhoven University of Technology, Eindhoven, The Netherlands

[2] TNO-ESI, Eindhoven, The Netherlands

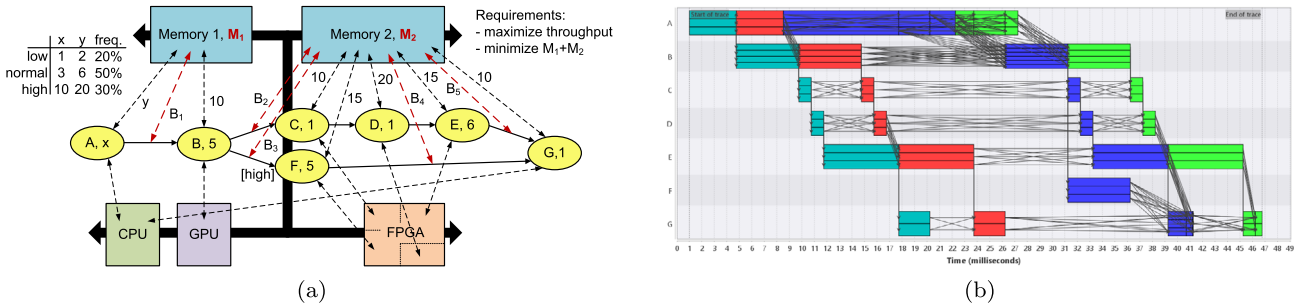


Fig. 1 Image-processing pipeline (a)[3] and a small execution trace (b)

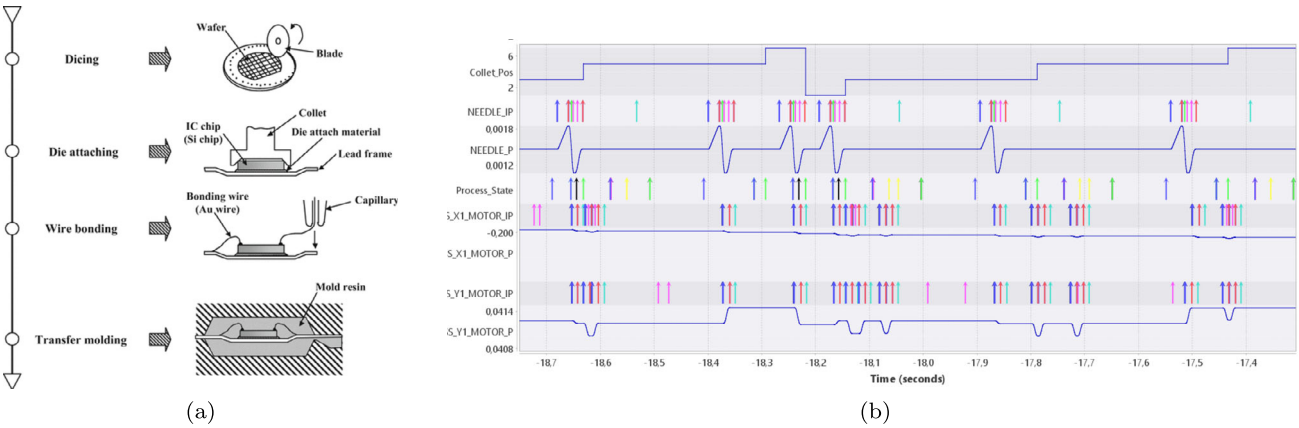


Fig. 2 Die-packaging process (a)[4] and a partial execution trace of a die bonder (b)

widely recognized that it is hard to reason about cross-cutting system-level properties of embedded and cyber-physical systems [4–6]. As a result, system performance often is *emerging* during the development and can only be dealt with in a reactive manner. But even then, understanding why the system performance is as it is often is hard. Throughout the paper we show with various examples how the Trace tool can aid the performance-engineering process.

Note that our tool is not concerned with design and implementation of embedded or cyber-physical systems. There is a plethora of work on this subject, and many approaches (e.g., simulation) have design-time methods that are capable of producing execution traces. Moreover, prototypes or operational systems typically are capable of generating execution traces, or can be redesigned to do so. Thus, our tool is complementary: it can be applied whenever execution traces are available. It helps to obtain insight into a system's behavior and supports detection and diagnosis of behavioral problems during many phases of the system's lifecycle.

The high-level method of application of the Trace tool that we propose is a simple iterative process consisting of three steps: acquire an execution trace, assess the trace, and then act on the results. This paper is focused on the second step: assessment of the trace by visualization, transformation, and analysis techniques. We illustrate the use of the Trace tool with two running examples.

Example 1

Fig. 1a shows a schematic representation of an artificial embedded image-processing pipeline. Although artificial, this example is inspired by the systems and problems that we encounter in our work with the high-tech industry (e.g., image processing in production printers) and contains many of their aspects. The system processes images using seven tasks (A–G). The tasks are mapped to a computational platform consisting of a CPU, a GPU, an FPGA with four functions, and two memories. The computational load of the tasks is specified inside the ellipses, and the storage requirements are indicated by the arrow to the memories, which are labeled with the required amount. Note that there are three kinds of images that impose different computational and memory loads on task A. These are given in the table (x and y values) for each of the image types (*low*, *normal*, or *high* workload). This table also specifies the frequency of occurrence of the various image types in the incoming image stream. Task F is

[3] Reprinted by permission from Springer Nature: [7], copyright 2010.

[4] Reprinted by permission from Springer Nature: [8], copyright 2017.

only executed for images of type *high*. We have a discrete-event simulation model of this system that gives execution traces with claims on resources and dependencies between these claims. Figure 1b shows an execution trace obtained with the simulation model for processing four objects. Time is shown on the horizontal axis. The colored rectangles are claims of resources. The cyan blocks in the upper-left corner, for example, model the claims of task A: one claim of the CPU, one claim of memory M1 for the internal processing (given by y), and one claim of memory M1 for data transfer to task B (always 10 units of memory). The claims are grouped according to the name of the task they belong to, i.e., the first horizontal swimlane contains all claims for task A, the second for task B, etc. The coloring of the claims is according to the image identifier that is being processed, i.e., all cyan claims are for processing image 0, all red claims are for image 1, etc. The durations of tasks A and G are sometimes longer than their nominal execution time as given in Fig. 1a because they share the CPU. When both are active at the same time, their execution speed is halved. Finally, the application dependencies that are shown in the schematic of Fig. 1a are also present in the trace.

Example 2
Fig. 2a shows the schematic outline of a die-packaging process. Dies are separated in the dicing step. In the following die-attach step, dies are picked from the wafer by a collet and attached to a lead frame. As part of the die-attach step, various inspections are performed that check for defects such as scratches on the dies. The wire-bonding step then makes connections between the die and the packaging. Finally, casting material is forced into a mold that contains the die. The casting material encapsulates the die and prevents it from being damaged. The die-attach step is realized by a die-bonder machine. Execution traces of the die bonder can be obtained by processing the logging of the system or via a simulation tool. These are execution traces with events and continuous signals. Figure 2b shows a part of an execution trace of the die bonder. Time is shown on the horizontal axis. The discrete events are shown as upward arrows, and the signals are shown as real-valued continuous functions of time. The signals in this example show the position of mechanical components or motors. For instance, the NEEDLE_P signal, shown in the third section from the top, gives the position of a needle that pushes a die from the wafer such that it can be picked up by the collet. The various NEEDLE_IP events in the second section from the top indicate that the needle has reached a certain position. The X1_MOTOR_P and Y1_MOTOR_P signals in the sixth and eighth sections show how the wafer moves between die pickups by the collet.

1.2 Related work

Trace-based performance engineering: Trace-based performance engineering covers the class of techniques that can be applied to systems and system models that can produce execution traces. As the Eclipse TRACE4CPS project, the Eclipse TRACE COMPASS project [9] also provides a tool for trace-based performance engineering. This tool, however, is mostly targeted at software and networking systems. The OpenTracing initiative provides vendor-neutral APIs and instrumentation for distributed tracing and its focus is on microservice architectures [10]. Further, there are many domain-specific and company-specific trace-based approaches for performance engineering. The ChronView tool provides detailed trace visualization and analysis for embedded systems and is tailored to the automotive domain [11]. Examples of tracing for internet-related services are Google's Dapper [12] and Facebook's Canopy [13]. The Concerto approach developed for ASML [14] is an example of company-specific tracing for a cyber-physical system. Our tool is targeted at embedded and cyber-physical systems and lies between generic approaches and company-specific approaches for tracing. The distinguishing feature of our approach is that it has a wide variety of analysis methods, which are founded on a sound and concise mathematical basis.

Critical-path analysis: Critical-path analysis originates from the project planning domain to minimize the duration of a project based on a graph representation of a project [15–17]. Extensions such as resource leveling [18] and the critical chain method [19] also take resources into account. Minimization of the makespan of a project with resource constraints, however, is known to be NP-hard [20]. In earlier work [21, 22], we have addressed the challenge of how to reconstruct a weighted directed acyclic graph of the system's execution from a single execution trace, which can then be used for critical-path analysis. The reconstruction is based on identifying *causal* dependencies. In this article, we generalize the approach to graphs with cycles and negatively weighted edges based on event networks [23]. Partial knowledge of the system's execution graph is assumed in most related work in the embedded and distributed systems domains, e.g., [24–30]. An exception is [31], which presents an informal approach similar to ours. Also related are techniques for process or workflow mining [32]. In [33], for instance, a Petri-net-based performance model is generated from a set of execution traces. This work uses the same intuition for deriving causal dependencies between tasks. Finally, our way to detect *resource bottlenecks* as part of the critical-path analysis is strongly related to the bottleneck analysis in Synchronous Dataflow as presented in [34].

Trace comparison: Quantitative similarity of timed systems has been studied in [35] and [36]. Both define a metric

on execution traces that requires that the events in both sequences are equal (i.e., only the timing of the events may differ). This is too strong for our setting. Execution traces can be seen as strings to which edit distances such as the Levenshtein distance [37] can be applied. This approach is not robust for execution traces in which the timestamps of equal events differ slightly. This has been addressed in [38] by a weighted edit distance. The complexity of the algorithm is, however, quadratic in the length of the input traces, which renders it expensive in many practical cases. Furthermore, a string-edit distance disregards the parallelism in the system as it assumes a total order on the events. In earlier work [22], we have taken parallelism into account by applying a graph-edit distance [39] on the graph representations of the execution traces. The computation of the graph-edit distance using insertion, deletion, and substitution of vertices is NP-hard in general [40]. In our case, however, substitution is not needed; omitting this option renders the computation of the graph-edit distance tractable. In addition to the earlier developed graph-edit distance, we introduce two new comparison techniques, one to verify that timing constraints specified by dependencies between claims and events in one trace are satisfied by another trace and one to compare the durations of claims of two traces.

Repetitive-structure analysis: Our analysis of repetitive structures is based on a graph representation of the execution trace (as is the case for critical-path analysis and the graph-edit distance for trace comparison). Graph-based anomaly detection is widely studied; see, e.g., [41] for a survey. The authors define the general graph anomaly detection problem as follows: "Given a (plain/attributed, static/dynamic) graph database, find the graph objects (nodes/edges/substructures) that are rare and that differ significantly from the majority of the reference objects in the graph." Our work falls in the class of static, attributed graphs and is related to structure-based methods. The essence of our technique is similar to the anomalous subgraph detection of [42]. We use domain information to partition the subgraph. Instead of using entropy-based graph regularity measures to compare the set of subgraphs, we apply the abovementioned graph-edit distance.

Little's-law-based analysis and resource-usage analysis: Resource usage, throughput, latency, and work-in-progress concepts are widely used and there are many tools that quantify them. It is often unclear how these values are exactly computed, i.e., there is a lack of formalization. There are exceptions, however. For instance, synchronous dataflow uses the maximum-cycle-mean concept to define throughput; see, e.g., [43]. Queueing theory [44] also has a mathematically defined notion of throughput, latency, and work-in-progress, and these are related through Little's law [45]. Little's law considers averages, and in our work we lift it to continuous real-valued signals. We define continuous throughput, latency, and work-in-progress signals for execution traces in an intuitive way.

Run-time verification: Following [46], we consider run-time verification to concern "verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property." A run of a system is captured in our execution-trace formalism, and we apply temporal-logic-based verification. Metric Temporal Logic (MTL) allows us to specify quantitative real-time properties of execution traces [47, 48]. Many MTL algorithms use tableau methods to construct an automaton of the formula that accepts exactly those execution traces that satisfy the formula [49–51]. Such an automaton can then be used for run-time monitoring of systems [52–57]. Our setting differs as we have offline access to the full execution trace which does not grow anymore. In earlier work [58, 59], we developed a dedicated checking algorithm that supports the notion of informative prefixes [57] that uses the truncated semantics of [60]. Our algorithm does not compute the truth value of all subformulas (such as the dynamic programming approaches that are suggested in [56, 57] do), but only the ones needed for a verdict. Signal Temporal Logic (STL) is a specification formalism for real-valued signals that was introduced in [61]. Later work extends this to a robustness estimate of the truth value [62, 63] and to efficient monitoring [64]. There are many tools that support temporal-logic specification and verification. The Breach toolbox of Matlab, for instance, enables specification and analysis of STL [65]. More recent work presents the AMT 2.0 tool, which supports STL extended with timed regular expressions [66], and RTAMT [67]. The work of [68] presents a tool with a domain-specific language (DSL) for testing CPS models that employs temporal logic. Dassault Stimulus [69] allows the user to express textual requirements in a readable formal language. Also, MathWorks supports expression and checking of temporal logic formulas via a templating language [70]. Our DSL to express temporal logic formulas is closely related and provides a lot of flexibility to end up with readable specifications. We have used the interface operator of [71] to mix MTL and STL specifications, which is very helpful in practice.

Trace transformation: Abstraction techniques for execution traces from software systems are presented, e.g., in [72, 73]. These works have in common that the abstraction techniques are hard coded. Our approach, however, makes the user responsible for a sound transformation by specification of the domain knowledge in a transformation recipe. This is related to ideas from the process-mining community that uses semantic structures such as taxonomies and ontologies to guide the abstraction process [74, 75]. Our work is also related to [76], in which a DSL is used to import packet-capturing data into the TraceCompass tool.

1.3 Contribution

The main contribution of our work is a combination of several visualization, transformation, and analysis techniques for execution traces. All techniques are based on the formal definition of the execution-trace concept and thus use a common, consistent, and mathematically precise foundation. Furthermore, our work has been consolidated in the open-source Trace tool of the Eclipse TRACE4CPS project [2] and thus is freely available (except for the trace-transformation techniques that are still being prototyped). Below, we detail our contributions.

- We combine and reconcile the contributions from earlier work on critical-path and distance analysis [21, 22]. We also generalize critical-path analysis and distance analysis by allowing weighted edges in the underlying graph representation of execution traces that is constructed by these analysis methods. This generalization is adapted from the event networks domain [23].
- We introduce two new techniques to compare execution traces: one based on the duration of resource claims and the other based on ordering constraints between discrete events.
- Our repetitive-structure analysis first partitions a graph representation of an execution trace into subgraphs and then partitions this set of subgraphs into blocks with equivalent behavior according to the graph-edit distance mentioned earlier. This partially follows the general idea stated in [42], but is applied in the domain of execution traces and performance analysis.
- We formalize the concept of throughput, latency, and work-in-progress by creating continuous signals from an execution trace, inspired by Little's law [45]. We also compute continuous resource-usage signals from execution traces. As an extension, we compute a parameterized sliding-window average of the signals using convolution. This is especially useful for the throughput as the resulting signal is often closer to human intuition than the raw throughput signal. Although these are widely used concepts, we have not found a similar precise treatment in the literature in the context of execution traces.
- We have used the interface operator of [71] to include STL subspecifications in the MTL specifications of [58]. Furthermore, we have created a DSL that aids the user in writing clear and manageable temporal-logic specifications.
- The main contribution of the transformation technique and DSL is the integration with the temporal-logic formalism to specify the rules for transformation. This allows the user to take the context into account, e.g., rules such as the following can be specified: "if an event A happens and within 5 milliseconds event B happens, then generate a start event for the high-level action *move*".

1.4 Outline

Section 2 presents the execution-trace formalism and the visualization technique. This is the basis for all other formalisms and techniques. Performance analysis based on Little's law is presented in Sect. 3. Then, Sect. 4 presents several temporal-logic formalisms, and a DSL for specifying temporal-logic properties. Section 5 introduces trace-to-trace transformations and a DSL for specifying such transformations. The constraint-graph formalism is presented in Sect. 6. This is a graph-based representation of an execution trace, and it lies at the basis of critical-path analysis, trace comparison, and repetitive-structure analysis. These techniques are also explained in this section. Finally, Sect. 7 concludes the paper.

2 Execution traces

In this section, we define the mathematical foundation of execution traces. Throughout the paper, we use $\mathbb{R}$ to denote the real numbers and $\overline{\mathbb{R}}$ to denote the extended real numbers (including $-\infty$ and $+\infty$). We use superscripts $^{\geq 0}$ and $^{>0}$ to indicate situations in which we only use the non-negative or strictly positive numbers, respectively. Furthermore, we let $\mathbb{B}$ denote the Boolean values.

2.1 Execution-trace formalism

Embedded and cyber-physical systems typically have both discrete and continuous aspects in their behavior, and our definition of execution traces covers both. The *event*, *claim-of-resource*, and *dependency* concepts cover the discrete aspects. The *signal* concept covers the continuous aspect. To model application-specific metadata that are associated with instances of these concepts, we define attribute–value mappings. Let $\mathbf{A}$ be a set of *attributes* and let $\mathbf{V}$ be a set of *attribute values*. We let $\mathbf{M}$ denote the set of all partial functions from $\mathbf{A}$ to $\mathbf{V}$. Each element of $\mathbf{M}$ is a value assignment to a subset of attributes. Assigning sensible attributes to events, claims, etc., is the responsibility of the user, and this is important as it serves as a basis for the visualization and the analysis techniques. Below, we formally define events, dependencies, resources, and claims.

Definition 1 (Event)
An *event* is specified by a tuple (t,m), where $t \in \mathbb{R}$ is the timestamp of the event and $m \in \mathbf{M}$ specifies the event's attributes.

Next, we introduce dependencies between events. Throughout this paper we loosely write about dependencies between claims and/or events. As we will see in Def. 7,

each claim is associated with a start event and an end event. A dependency involving a claim thus refers to a dependency as in the definition below involving the start or end event associated with the claim.

Definition 2 (Dependency)
Let E be a set of events. A *dependency* on E is a tuple (e_0, e_1, m), where $e_0, e_1 \in E$ are the source and destination, respectively, and $m \in \mathbf{M}$ specifies the dependency's attributes.

Definition 3 (Resource)
A *resource* is defined by a tuple (c, b, m), where $c \in \mathbb{R}^{\geq 0}$ is the capacity of the resource, $b \in \mathbb{B}$ indicates whether the resource uses an offset, and $m \in \mathbf{M}$ specifies the resource's attributes.

Definition 4 (Claim)
Let R be a set of resources. A *claim of a resource* (or just *claim*) is defined by a tuple (t_0, t_1, r, a, o, m), where $t_0, t_1 \in \mathbb{R}$ with $t_0 \leq t_1$ are the start and end time of the claim, $r = (c, b, m') \in R$ is the resource of the claim, $a \in \mathbb{R}^{\geq 0}$ is the claimed amount, $o \in \mathbb{R}^{\geq 0} \cup \{\perp\}$ is the offset of the claim ($\perp$ if and only if $b =$ **false**), and $m \in \mathbf{M}$ specifies the claim's attributes.

Resources and claims are not limited to the computational sphere as in Ex. 1. They can also be used to model, e.g., a portion of Euclidean space that needs to be claimed before a mechanical component can move to avoid collisions. The offset of the claim can be used to model, e.g., memory allocation in which a claim has an explicit area of the memory. A memory allocation of 2048 bytes starting at memory address 32768 can be modeled by a claim with offset 32768 and an amount of 2048. The special offset $\perp$ is used if and only if the resource does not support an offset. The amount that is claimed of a resource should not be larger than the capacity of the resource. Of course, multiple overlapping claims can use the same resource, and in that case the sum of their amounts should not exceed the resource capacity. Proper use of the offset and amount fields is the responsibility of the user.

The continuous part of an execution trace consists of signals, which are real-valued functions on $\mathbb{R}$. We limit ourselves to piecewise second-order polynomials. These are easy to define, have concise algebraic properties (e.g., the quadratic formula), and are powerful enough to express the basic kinematic equations, i.e., position as a function of time assuming constant acceleration. This can be useful to capture mechanical motion with just a few piecewise second-order polynomial fragments instead of many samples with constant or linear interpolation. Furthermore, this definition covers signals obtained through sampling and linear interpolation, which is natural for numerical simulation engines and systems. We believe, however, that all theory in this paper can be extended at least to signals represented by general piecewise polynomials. This, of course, implies more computationally involved methods.

Definition 5 (Signal fragment)
A *signal fragment* is a tuple $(t_0, t_1, a, b, c) \in \mathbb{R}^5$ with $t_0 < t_1$. It encodes the piecewise second-order polynomial $f : \mathbb{R} \to \mathbb{R}$ defined as follows:

$$f(t) = \begin{cases} a(t - t_0)^2 + b(t - t_0) + c & \text{if } t_0 \leq t < t_1, \\ 0 & \text{otherwise.} \end{cases}$$

We use the term signal fragment for both the tuple (t_0, t_1, a, b, c) and the function f that the tuple encodes. The domain of a signal fragment $f = (t_0, t_1, a, b, c)$, denoted by $dom(f)$, is the interval $[t_0, t_1)$.

Definition 6 (Signal)
A *signal* is defined by a tuple $s = (\{f_1, f_2, \ldots, f_n\}, m)$, where $f_1, f_2, \ldots, f_n$ are signal fragments and $m \in \mathbf{M}$ specifies the signal's attributes. The signal defines the function $f : \mathbb{R} \to \mathbb{R}$ defined as $f(t) = f_1(t) + f_2(t) + \cdots + f_n(t)$. We use the term signal for both the tuple s and the function f that the tuple encodes.

The domain of a signal s, denoted by $dom(s)$, is the union of the domain of its fragments. An execution trace consists of the concepts we have defined above.

Definition 7 (Execution trace)
An *execution trace* (or *trace*) is a tuple (E, D, R, C, S), where E is a set of events, D is a set of dependencies on E, R is a set of resources, C is a set of claims on R, and S is a set of signals. With each claim $c = (t_0, t_1, r, a, o, m)$ we associate two events: $start(c) = (t_0, m \cup \{event\_type \mapsto s\})$ and $end(c) = (t_1, m \cup \{event\_type \mapsto e\})$.[5] These start and end events are assumed to be part of E.

An execution trace has two types of events: regular events that are not associated with a claim and start and end events of claims. The claim-specific events can be distinguished from the regular events by the special *event_type* attribute.

Finally, we define the domains of the discrete and continuous data as follows:

$$dom(E) = [min\{t \mid (t, m) \in E\}, max\{t \mid (t, m) \in E\}],$$

[5] Here we assume that $event\_type \in \mathbf{A}$ and $s, e \in \mathbf{V}$, and also that this special *event_type* attribute is not used for the application-specific metadata defined by the user. Start and end events inherit the attributes of the parent claim.

$$dom(S) = \bigcap_{s \in S} dom(s).$$

Note that the domain of the events also covers the claims, because we assume that the start and end events of the claims are part of the events. The domain of the signals is the largest interval on which we have fragment information for all signals.

Example 3
Consider the image-processing system as in Ex. 1. The execution traces of this system have events, resources, claims, and dependencies. The events are only the claim-specific events as explained above. The resources only have a *name* attribute: CPU, GPU, M1, M2, F1, F2, F3, or F4. We use three attributes for the claims: (i) the *name* of the activity (A–G), (ii) the *id* of the image that is being processed (a natural number), and (iii) the *type* of image that is being processed. The dependencies do not have attributes. In the die-bonder example of Ex. 2 we have execution traces with events and signals. Both have a *name* attribute that specifies the event name and signal name, respectively. The events also have a *type* and *color* attribute.

2.2 Attributes and visualization

The attributes that are part of all elements of an execution trace are used heavily for the visualization and also for other analysis techniques. Assigning proper attributes is an important responsibility of the user. Below we define partitioning and filtering based on attributes.

Definition 8 (Attribute partition)
Given a set of attributes $A \subseteq \mathbf{A}$, we define an equivalence relation $\equiv_A$ on $\mathbf{M}$ as follows: $m \equiv_A m'$ if and only if $m(a) = m'(a)$ for all $a \in A$. This equivalence relation on M induces a partitioning of M, where elements in the same block have the same attribute values for A.

Definition 9 (Attribute filter)
A filter is a set $F \subseteq \mathbf{M}$. We say that $m \in \mathbf{M}$ satisfies F if and only if there is some $f \in F$ such that $f \subseteq m$.

These partitions and filters can be lifted to events, resources, claims, dependencies, and signals of an execution trace, as these all have an attribute mapping: they are *attribute aware*. If Z is a set of which the elements are attribute aware, then we let $Z_{\equiv_A}$ denote the partitioning of this set according to A. Furthermore, the application of a filter F to Z gives a subset of Z in which the attribute map of each element satisfies F.

We denote assignment (or overwriting) of an attribute a with value v in $m \in \mathbf{M}$ as follows: $m[a \leftarrow v]$. This is also lifted to a set Z with attribute aware elements, which is denoted by $Z[a \leftarrow v]$.

An execution trace is visualized with a Gantt chart as shown in Fig. 1b and Fig. 2b. Time is shown on the horizontal axis, and the vertical axis is divided into a number of *sections* (horizontal swimlanes). Each signal has its own section, the claims are distributed over a number of claim-specific sections, and the events are distributed over a number of event-specific sections. Dependencies are shown as arrows between claims and/or events. The visualization has a number of settings that can be used to configure the view:

- *Resource/activity view* (affects only claims): In *resource view*, there is a claim-specific section for each resource in the trace. In *activity view*, the claim-specific sections are determined by the grouping setting of the claims (see below).
- *Filtering*: Visual filters as in Def. 9 can be specified.
- *Grouping*: Assignment of claims/events to a certain section is done by partitioning them according to a set of *grouping* attributes.
- *Coloring*: The color of a claim/event or dependency is determined by a set of *coloring* attributes. Every element in a block of the coloring partition gets the same (randomized) color.

This kind of visualization of execution traces is rather straightforward, but it proves to be a valuable technique in practice. Showing the full concurrent behavior of a system in a single Gantt chart quickly gives an overview that is much easier to understand than, for instance, raw logging data.

2.3 Tool support

Figure 3 shows the user interface of the Trace tool. The tool supports a custom, human-readable file format that can be used to specify execution traces. The tool can read such files and process them further, e.g., by visualizing them or by applying analysis techniques via the user interface. The file format is designed for ease of use so that the threshold of creating suitable input for the tool is low. The tool also has a well-defined API that can be used to programatically create execution traces. Finally, there is a command-line executable to run temporal-logic verifications (also see Sect. 4.3).

3 Performance analysis

In this section, we present methods to create signals in the sense of Def. 6 for resource usage, latency, throughput, and work-in-progress from a given execution trace. Furthermore, we present two ways to process the signals. First, we apply a convolution with a rectangular window function to obtain a sliding-window average of the signals. This, for instance,

Fig. 3 Overview of the Trace user interface. The file explorer view (1) shows the trace files (Gantt-chart icon) and in this case a single specification file (see Sect. 4) with the green checkmark icon. The Gantt-chart viewer (2) shows trace files. Parts of a trace file – events, dependencies, claims, and signal fragments – can be selected and their properties including the attributes are then shown in the properties view (3). Each Gantt-chart viewer has a toolbar (4) with items to apply techniques, such as the filtering, coloring, and grouping visualization techniques. The various editors support syntax highlighting, etc. In (5), a property specification file is shown. The outline (6) shows the structure of the property in the highlighted line. A specification can be checked on a trace file (invoked from the toolbar (4)). Logging of the verification algorithm is written to the console view (7), and the results are shown in the verification view (8). Another type of analysis is the timing analysis, for which the results are shown in a timing-analysis view (9) and a timing-histogram view (10). There is also an Eclipse perspective (11) that gives a default layout for all relevant views. The layout shown is not the default; various views have been relocated (by drag-and-drop) to convey as much information as possible in a single screenshot. Finally, the help menu (12) gives access to documentation of the Trace tool

gives the signal for the average throughput of the last 10 seconds. Second, we apply a histogram analysis to the signals. This, for instance, gives the percentage of time that there are at least five tasks running on the four-core CPU. The signals that we obtain from traces and that result from the sliding-window processing are piecewise constant or piecewise linear functions, which enables us to treat them as part of the signals of an execution trace. All functionality (e.g., computing a throughput signal) is available via the toolbar (4 in Fig. 3).

3.1 Sliding-window average and histogram analysis

First, we define these two processing steps in general terms. The first one is to compute a *sliding-window average* of some function $f : \mathbb{R} \to \mathbb{R}$. Therefore, we define a parameterized rectangular window function $g_w : \mathbb{R} \to \mathbb{R}$ with width $w \in \mathbb{R}^{>0}$:

$$g_w(t) = \begin{cases} \frac{1}{w} & \text{if } 0 \le t \le w, \\ 0 & \text{otherwise.} \end{cases}$$

We use the Dirac delta function δ for the window with width zero, i.e., we define $g_0 = \delta$. The *w-average* of f then is defined as the convolution of f and g_w.

Definition 10 (*w*-average)
Given a function $f : \mathbb{R} \to \mathbb{R}$ and $w \in \mathbb{R}^{\ge 0}$. The w-average of f, denoted by avg_w^f, is defined as $f * g_w$.

The w-average of f at time t gives the average value of f over the last w time units, or in case $w = 0$, it gives f itself.

Proposition 1
If $w = 0$, *then* $avg_w^f(t) = f(t)$. *If* $w > 0$, *then* $avg_w^f(t) = \frac{1}{w} \int_{t-w}^{t} f(\tau) d\tau$.

The second processing step that we define for signals is to create a *time histogram*. A time histogram is based on a set of predicates that return either true or false for each possible value of the signal. A time histogram then maps each predicate to the total amount of time that the predicate was true.

Definition 11 (Time histogram)
Consider a set of predicates $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ with $\phi_i : \mathbb{R} \to \mathbb{B}$. The *time histogram* for Φ is a function $h : \Phi \to \mathbb{R}$ defined in two steps as follows. Let P_{ϕ_i} be a partition of $\mathbb{R}$ into disjoint blocks such that:

- each block can be written as an open, closed, or half open interval,
- for each block B holds $\forall_{t \in B} \phi_i(t)$ or $\forall_{t \in B} \neg \phi_i(t)$, and
- P_{ϕ_i} is finite.

Then $h(\phi_i)$, defined below, gives the value of ϕ_i in the time histogram:

$$h(\phi_i) = \sum_{B \in P_{\phi_i} \wedge \forall_{t \in B} \phi_i(t)} sup(B) - inf(B).$$

When the predicates are of the form $f = v$, where f is a signal and $v \in \mathbb{R}$, then computation of the time histogram is straightforward. Such a time histogram can be used, for instance, to show how much time a single-core CPU was occupied by more than one task. In that case, we need a signal f for the number of concurrent clients on the CPU and a set of predicates such that $\phi_i(t)$ is defined as $f(t) = i$.

3.2 Resource-usage signals

The instantaneous resource-usage signal is constructed from the individual contributions of the claims to the usage of the resource.

Definition 12 (Instantaneous resource usage)
Let (E, D, R, C, S) be an execution trace, let $r \in R$ be a resource, and let $v : C \to \mathbb{R}$ be a valuation function. The instantaneous resource usage of a claim $c = (t_0, t_1, r, a, o, m)$, denoted by ru_r^c, is defined as follows:

$$ru_r^c(t) = \begin{cases} v(c) & t_0 \le t < t_1, \\ 0 & \text{otherwise.} \end{cases}$$

The instantaneous resource usage of the trace, denoted by ru_r, is then defined as $\sum_{c \in C} ru_r^c$.

The valuation function v that we use can be used to get different interpretations of the resource usage. For instance, if we use the claimed amount, that is, $v(t_0, t_1, r, a, o, m) = a$, then $ru_r(t)$ models the total amount of the resource that is claimed at time t. If we use the constant 1, however, then $ru_r(t)$ models the number of different claims on the resource at time t.

Example 4
Consider the image-processing system from Ex. 1. Figure 4a shows the claims on the CPU and memory M2 (second and fourth section from the top). The first section shows the CPU resource-usage signal with a sliding-window average of 100 ms. The CPU usage is about 95%. The third section shows the instantaneous resource usage of M2. The resource is almost never completely used because of segmentation, which can be clearly seen in the fourth section. We further compute a time histogram for the CPU and M2 using the valuation function that counts the number of claims on the resource rather than the claimed amount. In terms of Def. 11, we use the predicates $f = i$ with $i = 0, 1, \ldots$, where f is the instantaneous-resource-usage signal from Def. 12 with valuation function $v(c) = 1$ for every claim c that uses the resource. The resulting histograms for the two resources are shown in Fig. 4b. Note that not the absolute time is given on the y-axis, but the percentage with respect to the total length of the underlying execution trace. This shows that for about 20% of the time, the single-core CPU has two claims on it.

3.3 Little's law for execution traces

Little's law stems from queueing theory and states that $L = \lambda W$, where L is the average number of items in the system, λ is the average arrival rate of items, and W is the average waiting time of an item in the system [45, 77]. In this section, we present an analogy in the context of execution traces of systems that process discrete items, based on signals for the system's work-in-progress, latency, and throughput.

In the following, we assume a system that works on discrete items. In our execution traces, the distinction between work on different items is made by the value of a certain attribute, assumed to be *id* from now on. Without loss of generality, we assume that every claim and event in the execution trace has an integer value for the *id* attribute and that the values $1, 2, \ldots, n$ occur. Furthermore, we assume that each item i has a specific value $v_i \in \mathbb{R}$. Examples of systems that typically satisfy this assumption are video-processing systems and manufacturing systems.

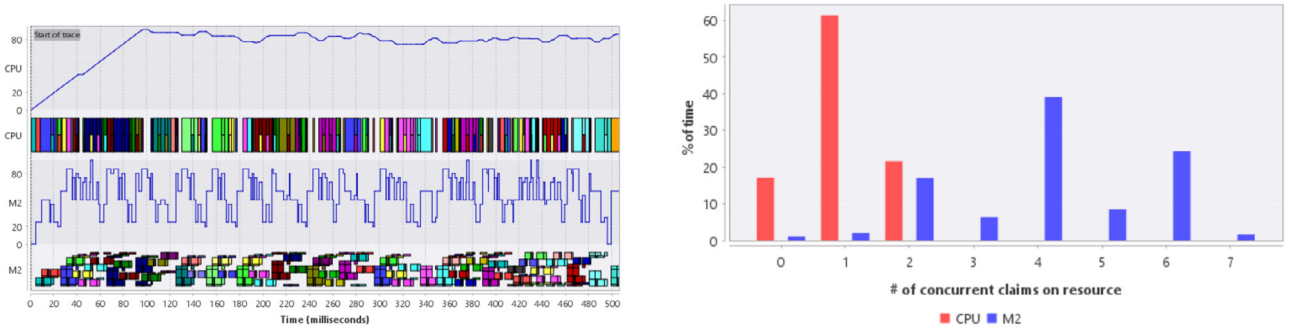
Let (E, D, R, C, S) be an execution trace. We define the start and end timestamps of item processing as follows:

$$T_{id=i} = \begin{array}{l} \{t \mid (t, m) \in E \wedge m(id) = i\} \cup \\ \{t_0, t_1 \mid (t_0, t_1, r, a, o, m) \in C \wedge m(id) = i\}, \end{array}$$

$$t^s_{id=i} = min(T_{id=i}),$$

$$t^e_{id=i} = max(T_{id=i}).$$

The start time of the work on item i is given by $t^s_{id=i}$ and the end time of the work on item i is given by $t^e_{id=i}$. We abbreviate $t^e_{id=i} - t^s_{id=i}$ with δ_i. We can compute these timestamps and δ_i values straightforwardly from an execution trace. This gives us the work-in-progress as follows.



(a) (b)

Fig. 4 Resource usage of the CPU and memory M2 (a) and a time histogram that shows the number of concurrent claims on the CPU and memory M2 (b)

Definition 13 (Work-in-progress)
For item i, the work-in-progress, denoted by wip_i, is defined as

$$wip_i(t) = \begin{cases} v_i & t^s_{id=i} \leq t < t^e_{id=i}, \\ 0 & \text{otherwise.} \end{cases}$$

The instantaneous work-in-progress, denoted by wip, then is defined as $wip = \sum_{i=1}^{n} wip_i$.

The value v_i can be used for counting the items (e.g., the number of images processed; $v_i = 1$ for all items) or for more specific accounting (e.g., the number of square meters printed; v_i then equals the number of square meters of sheet i).

System throughput is an important aspect for many systems. Throughput is "an amount of work, etc. done in a particular period of time" according to the Cambridge dictionary. We note, however, that generally applicable mathematical definitions of throughput are not readily available for our discrete setting. Often, throughput is calculated by counting a number of events and dividing that number by the total time over which was measured (the "particular period" in the quote above). This, however, gives only a single number, and at what point in time do we actually have that throughput? Below, we provide a rigourous definition of throughput. We start with a definition of the work that is done. We therefore assume that the value of an item is produced in a linear fashion.

Definition 14 (Cumulative work)
The work done for item i is the function $u_i : \mathbb{R} \to \mathbb{R}$. defined as

$$u_i(t) = \begin{cases} 0 & t < t^s_{id=i}, \\ \frac{v_i(t - t^s_{id=i})}{\delta_i} & t^s_{id=i} \leq t < t^e_{id=i}, \\ v_i & t \geq t^e_{id=i}. \end{cases}$$

The *cumulative work* done is $work(t) = \sum_{i=1}^{n} u_i(t)$.

The cumulative work as defined above is a piecewise linear function. Next, we define the *instantaneous throughput* as the rate at which the cumulative work that is done changes over time.

Definition 15 (Instantaneous throughput)
The *instantaneous throughput*, denoted by the function $tp : \mathbb{R} \to \mathbb{R}$, is defined as $\frac{d}{dt} work$.[6]

The area below the instantaneous throughput function on the interval $[a, b]$ is equal to the amount of work done in that interval: $work(b) - work(a)$.

Both wip and tp are thus piecewise constant polynomials that can easily be computed from our execution traces and fit Def. 6. The instantaneous throughput can be very irregular with many changes between extreme values. This is often hard to interpret. What does it mean that at time t the throughput is 200 images per second and at time $t + 1$ ms the throughput is 50 images per second? Using Def. 10, we can compute the average throughput over the last w time units. We say that the w-throughput at time t is the average of the instantaneous throughput over the last w time units:

$$tp_w = avg^{tp}_w.$$

From Prop. 1, it follows immediately that $tp_0 = tp$. Furthermore, the area under the tp_w graph on the interval $[a, b]$ is the average value of $work$ on $[b - w, w]$ minus the average value of $work$ on $[a - w, w]$. If the work is constant on $[b - w, w]$ and on $[a - w, w]$, then this reduces to $work(b) - work(a)$ just as for the instantaneous throughput. This notion of average throughput can be more intuitive, especially if w has a magnitude that is easily observable by humans.

[6] Note that $work$ is not differentiable at all points because it is piecewise linear, resulting in a finite number of holes in $work'$. We fill a hole at time t by taking the value $\lim_{x \to t^+} work'(x)$ and do this for every hole.

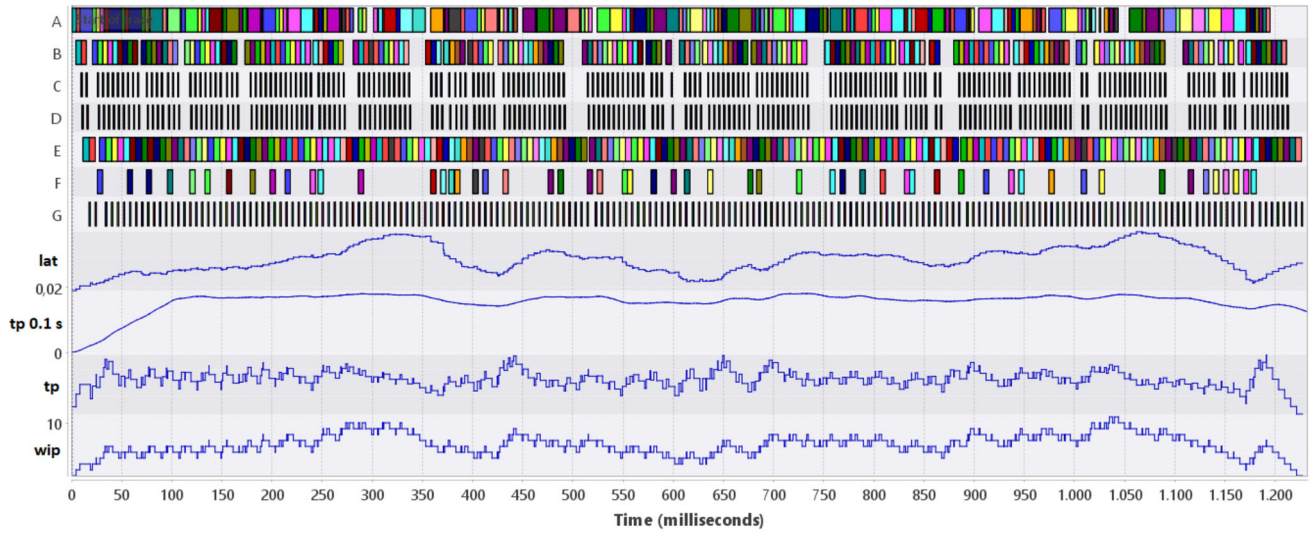


Fig. 5 Derived Litte's law signals

We have now covered two of the three ingredients of Little's law: work-in-progress and throughput. What remains is the latency. Unfortunately, we cannot give an intuitive compositional definition of latency, starting with the latency of a single item, as we have done for the work-in-progress and the throughput. Instead, we give the definition of latency by applying Little's law and we consequently give a property of the latency signal that matches our intuition.

Definition 16 (Latency by Little's law)
We define $lat(t)$ as $\frac{wip(t)}{tp(t)}$ for all $t \in \mathbb{R}$ such that $tp(t) \neq 0$.

When we consider only a single item i, the latency signal has value δ_i on the interval $[t^s_{id=i}, t^e_{id=i})$ and is undefined otherwise. This is rather intuitive. When we have two items that have no overlap, the latency signal is just the combination of the latency signals for the two individual items. This still could be written in the compositional style that we have used for the work-in-progress and the throughput. It becomes different, however, when we have items that have overlapping processing times. In that case, just summing the individual latencies is not appropriate. Instead, some kind of averaging method is needed. This is exactly what Little's law as applied in Def. 16 does. It gives the following property of the latency signal.

Proposition 2
We have $\delta_{min} \leq lat(t) \leq \delta_{max}$, *where* $\delta_{min} = min\{\delta_i \mid 1 \leq i \leq n\}$ *and* $\delta_{max} = max\{\delta_i \mid 1 \leq i \leq n\}$ *for all* t *such that* $lat(t)$ *is defined.*

The latency signal thus lies between the minimum and maximum of the latencies of the individual items.

Example 5
Consider the image-processing system from Ex. 1. The signal at the bottom in Fig. 5 shows the work-in-progress. Its value reaches up to 10, which means that at that point in time there are 10 images in the pipeline. The figure also contains the signal for the instantaneous throughput *tp* and its window average over 0.1 seconds. The instantaneous throughput varies with a high frequency between 0 and approximately 200 images per second. The window-average throughput smoothes this out; its value is approximately between 140 and 180 images per second. (These numbers are not visible in the figure.) Finally, the figure shows the latency signal, which has a value between 23 and 39 ms. The actual bounds on the latency of the individual images are 41 ms (upper bound) and 19 ms (lower bound).

4 Run-time verification

In this section, we first present three temporal-logic formalisms. We then describe a DSL that can be used to specify properties.

4.1 Temporal-logic formalisms

This section covers temporal logic, which is a formalism to specify properties of execution traces. Since we have both discrete and continuous data in our concept of execution traces, we use MTL for the discrete part, STL for the continuous part, and a mix of the two to express properties over both discrete and continuous data. Sections 4.1.1 and 4.1.2 recall existing theory on temporal logic and rephrase it slightly to fit our definition of an execution trace. Section 4.1.3 presents an adaptation of existing theory to integrate MTL and STL.

4.1.1 Metric temporal logic

MTL is a formalism that allows specification of quantitative real-time properties of sequences of discrete events [48]. The syntax of MTL formulas ranged over by ϕ is inductively defined as follows:

$$\phi := \mathbf{true} \mid m \mid \phi \wedge \phi \mid \neg\phi \mid \phi \mathbf{U}_I \phi.$$

Here $m \in \mathbf{M}$ is an *atomic proposition* and I is an interval (open, closed, or half open) on $\overline{\mathbb{R}}^{\geq 0}$. The temporal until operator $\phi_1 \mathbf{U}_I \phi_2$ informally expresses that ϕ_1 must hold until ϕ_2 holds somewhere within I time units from now. We use the following abbreviations for disjunction, implication, finally, and globally:

$$\phi_1 \vee \phi_2 = \neg(\neg\phi_1 \wedge \neg\phi_2),$$
$$\phi_1 \Rightarrow \phi_2 = \neg\phi_1 \vee \phi_2,$$
$$\mathbf{F}_I \phi = \mathbf{true} \mathbf{U}_I \phi,$$
$$\mathbf{G}_I \phi = \neg \mathbf{F}_I \neg\phi.$$

The satisfaction relation $\models$ for MTL is defined on finite sequences of events that are ordered by ascending timestamps. From now on, we assume that any event sequence that we refer to is ordered in this way.

Definition 17 (MTL semantics)
Consider a finite sequence of events $\sigma = (t_0, m_0), (t_1, m_1), \ldots, (t_n, m_n)$ that is ordered by ascending timestamps.[7] The satisfaction relation on the events $\sigma_i = (t_i, m_i)$ is defined inductively as follows:

$$\begin{array}{lll}
\sigma_i \models \mathbf{true}, & & \\
\sigma_i \models m & \text{iff} & m \subseteq m_i, \\
\sigma_i \models \phi_1 \wedge \phi_2 & \text{iff} & \sigma_i \models \phi_1 \wedge \sigma_i \models \phi_2, \\
\sigma_i \models \neg\phi & \text{iff} & \sigma_i \not\models \phi, \\
\sigma_i \models \phi_1 \mathbf{U}_I \phi_2 & \text{iff} & \exists_{n \geq j \geq i}\ \sigma_j \models \phi_2 \wedge t_j - t_i \in I \wedge \\
 & & \quad \forall_{i \leq k < j}\ \sigma_k \models \phi_1.
\end{array}$$

Note that formulas are interpreted relative to events and not to arbitrary moments in time. A sequence of events σ satisfies an MTL formula ϕ, denoted by $\sigma \models \phi$, if and only if $\sigma_0 \models \phi$. So we assume a strict semantics of finite until ϕ_2 must hold.

Example 6
Consider the image-processing pipeline of Ex. 1. All claims and events in an execution trace of this system have a *name* attribute for the current activity and an *id* attribute for the image that is being processed by the activity. We can use these attributes to specify lower and upper bounds on the total processing time of an image with the following MTL formula:

$$\mathbf{G}\big(\{name \mapsto \mathrm{A}, type \mapsto s, id \mapsto 1\} \Rightarrow$$
$$\mathbf{F}_{[0,50]}\{name \mapsto \mathrm{G}, type \mapsto e, id \mapsto 1\}\big).$$

Informally, this formula specifies that it always holds that if activity A starts for image 1, then within 50 time units activity G for that image ends.

In practice, we are always dealing with finite sequences taken from some system. These are often *prefixes* of ongoing behavior. Naively using the $\models$ relation thus brings the problem that an extension of the finite sequence that we are considering might invalidate the result of the $\models$ computation. To deal with this problem, we use the *informative prefix* concept of [78]. Let σ be a finite prefix of some event sequence and let ϕ be an MTL formula. If every extension of σ dissatisfies ϕ, then we call σ a *bad* prefix. Dually, if every extension of σ satisfies ϕ, then we call σ a *good* prefix. A prefix is not necessarily good or bad. We call a prefix that is neither a *neutral* or *non-informative* prefix. Intuitively, an *informative* prefix tells the whole story about the (dis)satisfaction of an MTL formula [78]. For a more formal discussion, we refer to [58].

4.1.2 Signal temporal logic

STL is a formalism that allows specification of quantitative real-time properties of real-valued continuous signals [61]. We start with a summary of definitions and results from [63, 64]. Let $S = \{s_1, s_2, \ldots\}$ be a set of signals. The syntax of STL formulas is inductively defined as follows:

$$\phi := \mathbf{true} \mid s_i \geq 0 \mid \neg\phi \mid \phi \wedge \phi \mid \phi \mathbf{U}_I \phi.$$

Here, I equals either a real-valued interval $[a, b]$ for some $0 \leq a < b$ or $[a, \infty)$ for some $a \geq 0$. For $I = [a, b]$ and $t \in \mathbb{R}$ we define $I + t = [t + a, t + b]$ and for $I = [a, \infty)$ we define $I + t$ as $[t + a, \infty)$. The atomic propositions are of the form $s_i \geq 0$ and specify whether signals exceed the threshold value 0. Extension of the syntax and semantics with atomic propositions of the form $s_i \leq x$ and $s_i \geq x$, where $x \in \mathbb{R}$, can be done by preprocessing [64]. We do not handle this explicitly in the following theory, but the Trace tool supports these constructions and we also use them in the examples.

Example 7
Consider the die bonder of Ex. 2. It is essential that the wafer table is not moving whenever the collet is extended beyond a certain threshold. Otherwise, physical contact between the

[7] Note that there are multiple such sequences for a given set of events, because multiple events can have the same timestamp. Our tooling picks an arbitrary sequence that satisfies the ordering constraint.

collet and the dies on the moving wafer might cause scratches on the dies. We can specify this property with STL. Execution traces for this system have a signal *wv* for the speed of a motor that moves the wafer table. They also have a signal *cp* that gives the position of the collet that is used for picking a die from the wafer. The STL specification then is the following:

$$\mathbf{G}\big((cp \geq 0.01 \vee cp \leq -0.01) \Rightarrow wv = 0\big).$$

Informally, this formula specifies that it always holds that if the pickup collet is extended at least 0.01 length units in either direction, then the wafer table is not moving (its speed equals zero).

The semantics of an STL formula ϕ for a set of signals S is defined over the time interval $dom(\phi, S)$ (we omit the definition for the sake of conciseness). The definition of the Boolean semantics $S,t \models \phi$ for an STL formula ϕ, a set of signals S and $t \in dom(\phi, S)$, is similar to the semantics of MTL. The difference is that MTL semantics is only defined for the timestamps at which events occur, whereas STL semantics is defined on all points in the domain of the formula. The *quantitative* semantics of STL, as defined below, has a specific interpretation in the sense that it gives an indication of the robustness of the result. A value farther away from zero indicates a "stronger" satisfaction. This quantitative semantics has a wide range of uses and as such is a core technique for the formal specification and verification of systems with discrete and continuous components.

Definition 18 (Quantitative STL semantics)
Let ϕ be an STL formula, let $S = \{s_1, s_2, \ldots\}$ be a set of signals, and let $t \in dom(\phi, S)$. The quantitative semantics is a value $\rho \in \overline{\mathbb{R}}$ and is defined as

$$\begin{aligned}
\rho(\mathbf{true}, S, t) &= \infty, \\
\rho(s_i \geq 0, S, t) &= f_i(t) \ (s_i \text{ is encoded by function } f_i), \\
\rho(\neg\phi, S, t) &= -\rho(\phi, S, t), \\
\rho(\phi_1 \wedge \phi_2, S, t) &= min\{\rho(\phi_1, S, t), \rho(\phi_2, S, t)\}, \\
\rho(\phi_1 \mathbf{U}_I \phi_2, S, t) &= \sup_{t' \in t+I} min\Big\{\rho(\phi_2, S, t'), \inf_{t'' \in [t, t']} \rho(\phi_1, S, t'')\Big\}.
\end{aligned}$$

The quantitative semantics has the following two fundamental properties. First, a ρ value greater than zero means that the formula is satisfied and a ρ value smaller than zero means that the formula is not satisfied. Second, ρ is a robustness estimate: if $S,t \models \phi$ and the signals in S and S' differ by at most $\rho(\phi, S, t)$, then $S', t \models \phi$.

There is also work that deals with trace prefixes in the STL context [79]. The authors define a robust satisfaction interval that captures lower and upper bounds on the robustness value of any completion of a partial signal. Many STL formulas with an unbounded time horizon (e.g., $\mathbf{G}(\varphi \Rightarrow \mathbf{F}\psi)$) have a trivial interval that is not useful because it does not give any information. They therefore introduce a nominal semantics that is very close to the robust satisfaction of Def. 18 that comes from [64].

4.1.3 Mixed temporal logic

We mix MTL and STL using the interface-operator idea of [71]. This enables us to specify properties that refer to both discrete and continuous parts of an execution trace. Let σ be the sequence of events and let S be the set of signals of an execution trace. The syntax of mixed-temporal-logic (mix-TL) formulas is inductively defined as follows:

$$\begin{aligned}
\phi &:= \mathbf{true} \mid m \mid \phi \wedge \phi \mid \neg\phi \mid \phi \, \mathbf{U}_{I_d} \, \phi \mid @^{\mathsf{cd}}(\phi_c), \\
\phi_c &:= \mathbf{true} \mid s_i \geq 0 \mid \neg\phi_c \mid \phi_c \wedge \phi_c \mid \phi_c \mathbf{U}_{I_c} \phi_c.
\end{aligned}$$

Here, $m \in \mathbf{M}$, $s_i \in S$ is a signal, I_d is a proper interval for MTL as defined above, and I_c is a proper interval for STL as defined above. Note that the defining rule for ϕ is the MTL rule extended with an interface-operator clause that allows the inclusion of STL formulas. At each moment that an event occurs, an STL formula can be evaluated. The cd superscript indicates a transformation from the continuous to the discrete domain. These STL formulas are produced by the ϕ_c rule, which is equivalent to the STL syntax defined above. The domain on which a mix-TL formula is defined depends on the structure of the formula as follows:

$$\begin{aligned}
dom(\mathbf{true}, \sigma, S) &= dom(S) \cap dom(\sigma), \\
dom(m, \sigma, S) &= dom(S) \cap dom(\sigma), \\
dom(\neg\phi, \sigma, S) &= dom(\phi, \sigma, S), \\
dom(\phi_1 \wedge \phi_2, \sigma, S) &= dom(\phi_1, \sigma, S) \cap dom(\phi_2, \sigma, S), \\
dom(\phi_1 \mathbf{U}_I \phi_2, \sigma, S) &= dom(\phi_1, \sigma, S) \cap dom(\phi_2, \sigma, S), \\
dom(@^{\mathsf{cd}}(\phi), \sigma, S) &= dom(\sigma) \cap dom(\phi, S).
\end{aligned}$$

This brings us to the semantics of mix-TL. We use the standard MTL semantics in combination with the quantitative STL semantics for STL subformulas.

Definition 19 (mix-TL semantics)
Let ϕ be a mix-TL formula, let σ be a sequence of events, let S be a set of signals, and let i be an index of σ such that $t_i \in dom(\phi, \sigma, S)$. The semantics is defined as

$$\begin{aligned}
&\sigma_i, S \models \mathbf{true}, \\
&\sigma_i, S \models m \quad \text{iff} \quad m \subseteq m_i, \\
&\sigma_i, S \models \phi_1 \wedge \phi_2 \quad \text{iff} \quad \sigma_i \models \phi_1 \wedge \sigma_i \models \phi_2, \\
&\sigma_i, S \models \neg\phi \quad \text{iff} \quad \sigma_i \not\models \phi, \\
&\sigma_i, S \models \phi_1 \mathbf{U}_I \phi_2 \quad \text{iff} \quad \exists_{j \geq i} \ \sigma_j \models \phi_2 \wedge t_j - t_i \in I \wedge \\
&\qquad\qquad\qquad\qquad \forall_{i \leq k < j} \ \sigma_k \models \phi_1, \\
&\sigma_i, S \models @^{\mathsf{cd}}(\phi_c) \quad \text{iff} \quad \rho(\phi_c, S, t_i) > 0.
\end{aligned}$$

We say that σ, S satisfies ϕ, denoted by $\sigma, S \models \phi$, if and only if $\sigma_j, S \models \phi$, where σ_j is the first event with timestamp

in $dom(\phi, \sigma, S)$. If the domains of the discrete and continuous parts do not overlap, then the satisfaction relation is undefined.

Note that the mix-TL semantics "samples" the continuous signals only at the moments when an event happens. This can give unexpected results. To make this more transparent to the user, our tool raises an information message for this kind of properties. Furthermore, it injects dummy events into the trace and reports the maximum time between events. This maximum time between events gives the user an indication of the resolution of the evaluation of the STL part. Ideally, the number of dummy events should depend on the degree of variation in the range of the signals.

Example 8
Consider the die bonder of Ex. 2. A property that the system must satisfy for correct operation is that a certain height measurement is done after each wafer change. Dies may only be picked from the new wafer after the height measurement. This can be expressed using a mix-TL property. Execution traces for the system have a signal *wc* that indicates the state of the wafer change process. A value of at least one means that a wafer change is in progress. Furthermore, the events have a *name* attribute that indicates the current activity. The presence of the *height* value for the *name* attribute means that a height measurement has been taken. The presence of the *pickup* value for the *name* attribute means that a die has been picked from the wafer. The following mix-TL property formalizes the requirement:

$$\mathbf{G}\big(@^{\mathsf{cd}}(wc \geq 1) \Rightarrow \neg\{name \mapsto pickup\}\mathbf{U}\{name \mapsto height\}\big).$$

The $wc \geq 1$ indicates that a wafer change is in progress. In that case, no pickup event happens until a height-measurement event happens.

4.2 Tool support for the temporal logic formalism

The Trace tool contains a DSL that can be used to specify MTL, STL, or mix-TL properties. The DSL has four important features. First, the logical operators are implemented by quasi-natural-language grammar elements. This makes the formulas easier to read and understand for non-specialists without loss of their precise mathematical semantics. Second, the language supports definitions of formulas that can consequently be used in any other formula by referencing. Such decomposition improves readability of complicated formulas. Furthermore, proper naming of the definitions helps to improve the readability of composed formulas. Third, a simple parameter mechanism is supported that helps the specification for systems that process discrete items. In the image-processing system of Ex. 1, for instance, every claim and every event has an *id* attribute whose value gives the identifier of the image that is being processed. Instead of making a separate specification for each image, a single specification can be made with the parameter mechanism that quantifies over all images. Fourth, the language supports specification of time units in the intervals of the until operators. The intervals are converted automatically to match the time unit of the execution trace.

The language also has syntax to refer to the derived resource-usage, throughput, work-in-progress, and latency signals that have been defined in the previous section. This is shown, for instance, in Ex. 10 below.

```
def processing_starts(i) : start {'name'='A', 'id'=i}
def processing_ends(i)   : end   {'name'='G', 'id'=i}

check latency_discrete :
    forall (i : 0 ... 199)
        globally
            if processing_starts(i) then
                within [0.0, 50.0) ms processing_ends(i)
```

Fig. 6 Bounds on latency

```
signal cp : {'name'='cp'}
signal wv : {'name'='wv'}

def the_wafer_table_is_not_moving : wv == 0.0
def the_collet_is_extended        : (cp >= 0.01 or cp <= -0.01)

check die_scratch:
    globally
        if the_collet_is_extended then the_wafer_table_is_not_moving
```

Fig. 7 Die-scratch property

Example 9
This example shows the DSL specifications of the formulas in Ex. 6, 7, and 8.

Figure 6 shows how we express bounds on latency of the processing of the individual images of the image-processing system in Ex. 6. Two parameterized definitions are given for the start and end of processing of image i, respectively. The formula to be verified, indicated by the `check` keyword, quantifies over 200 image identifiers and expresses the lower and upper bound on the processing latency.

Figure 7 shows how we express the die-scratch property of the die bonder in Ex. 7. The `signal` keyword is used to refer to signals in the execution trace based on the values of attributes. Note that these signals are used in the subsequent definitions and that the naming of these definitions is tailored to readability. The final formulation of the check then is straightforward.

Figure 8 shows the height-measurement property of the die bonder in Ex. 8. The names of the definitions are again tailored to readability. Note that the check is underlined with a warning that this is a mix-TL property that thus might suffer from sampling effects as explained above.

```
def height_measurement        : {'name'='height'}
def no_pickup_activity_occurs : not {'name'='pickup'}
signal wafer_change           : {'name'='wc'}

def wafer_change_in_progress : wafer_change >= 1.0

check proper_height_measurement :
    globally
        if wafer_change_in_progress then
            until height_measurement
            we have that no_pickup_activity_occurs
```

Fig. 8 Height-measurement property

```
signal tp_per_s : throughput of id per s over 10.0 ms

signal work_in_progress : wip of id

def work_done_within_100ms :
    during (100.0, Infty) ms work_in_progress == 0.0

def good_throughput: ( tp_per_s >= 100.0 or work_done_within_100ms )

check productivity: within [0.0, 100.0) ms
                    globally good_throughput
```

Fig. 9 Productivity specification

An MTL formula (a mix-TL formula without STL subformulas) is verified using the informative-prefix semantics (see [58]). The verdict can be that the property is satisfied (*good*), the property is not satisfied (*bad*), or that the trace is *non-informative*. The last verdict indicates that extensions of the trace exist that make the property satisfied or not satisfied. An STL formula (also an STL subformula of a mix-TL formula) is verified using the semantics in Def. 18, using an algorithm based on [64] that is adapted to piecewise second-order polynomial signals. This algorithm can currently only handle STL formulas without general until subformulas (i.e., only until operators with $\phi_1 = true$ are supported). This, however, still leaves a very usable subset of STL for the user. A mix-TL formula that has at least one STL subformula is verified using the semantics of Def. 19. Since this semantics uses the standard MTL semantics and a clear interface for STL subformulas, we can use the algorithm from [58], in combination with an algorithm for the robust semantics of STL. The underlying verification algorithms are motivated by pragmatic considerations to bring verification technology to the user of the Trace tool. There is room for improvement. The open-source nature of the tooling lowers the threshold for this. The Trace tool also has rudimentary support for explanation of the verification verdict, as illustrated in the following example. We intend to further refine this support with, for instance, the systematic approach presented in [80].

Example 10
Consider the image-processing pipeline of Ex. 1. The productivity of this system can be specified and verified using our signal-based equivalent of Little's law as introduced in Sect. 3. The DSL for specifying temporal-logic properties has support for the derived signals for resource usage, latency, throughput, and work-in-progress. Figure 9, for instance, shows a productivity specification for the image-processing system. The first signal declaration, `tp_per_s`, does not refer to a user-defined signal in the trace data, but to the tp_w signal as defined at the end of Sect. 3. The declaration specifies that the `id` attribute must be used to distinguish the processing activities for different images. The `per s` part specifies that the throughput is to be given in images per second, and `over 10.0 ms` specifies a window width of 10 milliseconds for the window-averaging operation. Similarly, the declaration of the `work_in_progress` signal refers to the *wip* signal of the previous section based on the *id* attribute. Using these signals, the `productivity` specification can be written that states that the throughput in the steady state is at least 100 images per second. We can use the Trace tool to verify it on an execution trace. Figure 10 shows a part of the Trace UI after we have checked this property on a trace. The result view on the right shows the verdict of the properties, grouped by labels that are inspired by the informative-prefix status. The result shows that the `productivity` property is not satisfied. Double-clicking this property then gives an explanation in terms of the definitions that have been used in the property. For instance, the section labeled with `good_throughput` shows on which part of the execution trace the property holds (green) and where it does not hold (red). Clearly, the steady-state throughput is not always as we required.

4.3 Application example: automated verification

Many embedded and cyber-physical systems rely on software for supervisory control that directs the operation. Such software typically is subjected to automated tests that are integrated with the version control system in a continuous integration environment. The Trace tool contains a command-line application to run verifications that can be used for automation. We can use it to create automated tests based on formal specifications of the behavior. Creating good specifications, however, is far from trivial and theferore we describe a top-down process for creating specifications in the DSL:

- Create a `check` in the DSL and choose one of the temporal operators (globally, finally, or until) that applies on the highest level.
- Depending on this choice, one or two subformulas need to be specified. For each of these subformulas create a `def` in the DSL with a proper name such that the formula reads naturally.
- Formalize the formula part of the `def` statements from the previous step. Here, any DSL construct can be used, such as conjunction, negation, etc. This again creates `def` statements with proper names that need to be formalized.

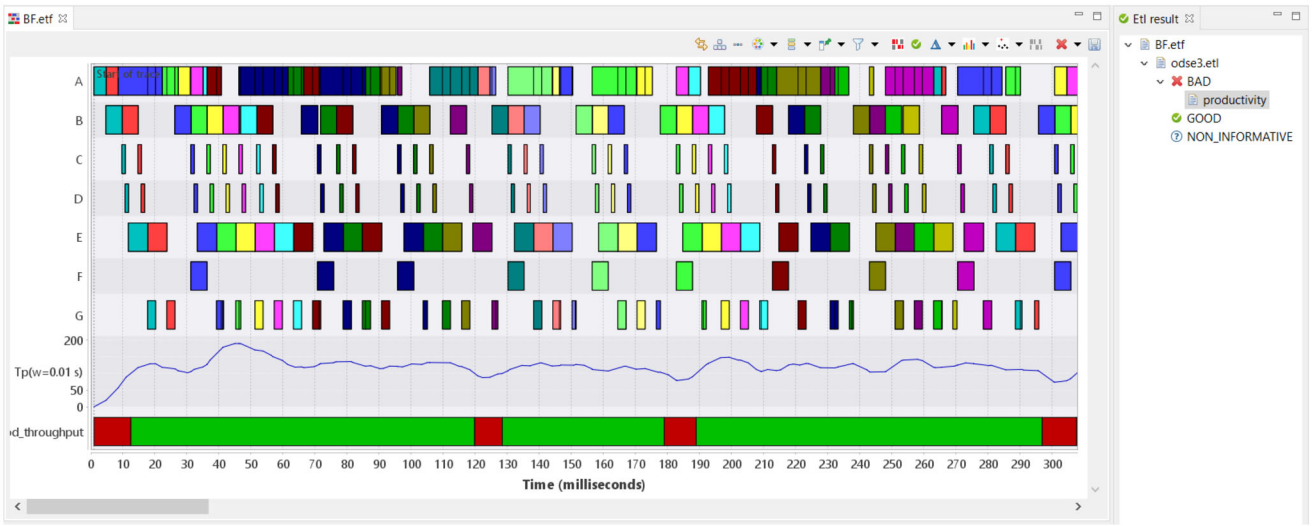


Fig. 10 Explanation of productivity violation

- Finish the recursive process by formalization of `def` statements with atomic propositions, i.e., either an attribute–value map or a signal comparison.
- The final step is to rewrite the definitions to improve the readability. Sometimes it helps to introduce additional definitions or to merge definitions. Naming the definitions in a way such that semi-natural language appears is a good way to increase readability and convey the ideas to others.

During this process, it often is useful to have a number of execution traces for validation, i.e., have a number of traces where the specification should hold and a number of traces where it should not hold. The default input format of the Trace tool is straightforward and human-readable. It is easy to create such test traces with a text editor.

Example 11
Consider the die bonder of Ex. 2. In certain scenarios, the action that picks a die from the wafer must be skipped. In practice, however, this sometimes failed. The underlying cause was an error in the supervisory control software which has been fixed. To prevent this error from being reintroduced by software evolution, we have created an automated regression test based on the temporal-logic DSL. The continuous integration environment uses Behavioral-Driven Development with the Python-based Behave tool [81]. We have scripted the scenario to be tested and the extraction of the execution trace. The command-line verification tool of Trace has been integrated in the Python environment. Furthermore, we have formalized the property that needs to be verified for traces that result from executing the scenario. This results in a formal, automated regression test for the issue, which nevertheless is easy to read and understand by non-specialists.

5 Trace-to-trace transformation

A trace-to-trace transformation specifies how an output trace can be generated from an input trace. This is often useful for, e.g., normalizing traces from different sources such as a trace from a simulation model and a trace from a system. This section first presents a brief overview of the structure of a transformation and then provides the realization of transformations in Trace through a DSL.

5.1 Transformation structure

A trace transformation consists of three main parts: a part for the generation of new events, a part for the generation of new claims, and a part for the generation of new signals. Dependencies and resources are handled automatically during the handling of events and claims.

An *event generator* consists of a mix-TL formula and an attribute map. Every event in the input trace that satisfies the formula generates an event in the output trace with the same timestamp but with the new attribute map.

A *claim generator* is similar to an event generator. There are two mix-TL formulas: one for the start of the claim and one for the end of the claim. If there are n_s events for the start of the claim and n_e events for the end of the claim, then at most $n_s \cdot n_e$ claims can be generated. Usually, however, we intend to match a single start event to a single end event. This is realized by an *event-matching heuristic*. This heuristic matches a start event with an earliest end event that happens after the start event.

A *signal generator* is built on the notion of a signal expression. Such an expression allows signal transformations such that their result still is a piecewise second-order polynomial. Examples include adding two signals, multiplying

two signals, and taking the minimum and maximum of two signals.

Next to these three main ingredients, there are four more parts: a *filtering* specification that filters the input trace before the transformation starts, a *trimming* specification to trim the output of the transformation to specific events/start and end times, a real number that specifies the *time shift* to normalize timestamps, and a Boolean that indicates whether the transformation *augments* the input trace or creates a new trace.

5.2 Tool support for transformation specifications

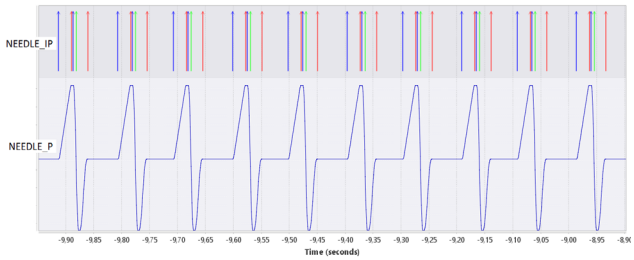
We developed a DSL plugin for the Trace tool that can be used to specify transformations. A trace-transformation specification contains a number of subtransformations. The subtransformations are executed in the order in which they are specified, and the result of a subtransformation is the input for the next subtransformation.

Example 12
Consider the die bonder in Ex. 2. An example trace transformation in the DSL is shown in Fig. 11a. The first line points to the input execution trace. Then, two subtransformations follow. The first subtransformation generates four types of claims, and the second subtransformation adds a throughput signal based on the generated claims. In the first subtransformation, we first limit the input trace to the interval of interest and apply filters to the events and signals. Only relevant data from this interval are considered when processing the transformation. Filtering data that are not relevant for the transformation can significantly improve the transformation speed. Next, four temporal-logic declarations follow (the transformation DSL uses the temporal-logic DSL). Then, four claim generators are specified. They each consist of three elements separated by a comma. The first two are the start and end mix-TL formulas, respectively, which use the signal specification and the definitions from above. The third element contains the attributes for the generated claim. In this example, discrete events are combined with values of continuous signals to define start and end events of claims. The last part of the first subtransformation specifies that the trace must be trimmed to the first occurrence of the `extend` claim and the 50th occurrence of the `retract2` claim (relative to the start of the trimming). The trimmed result is shifted in time such that the first event happens at time 0.

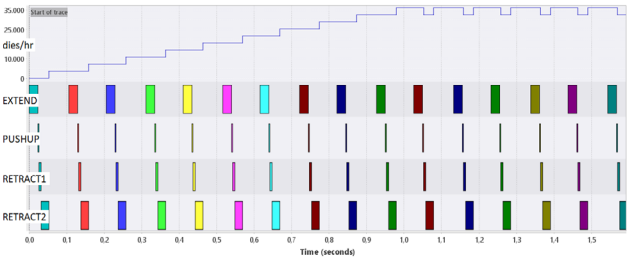
The second subtransformation adds a throughput signal to the result of the first subtransformation (because of the augmenting strategy; if it is not specified, then the default is used which only includes the generated events, claims, and signals). The `throughput of ...` construction is also part of the temporal-logic DSL and computes a signal using the

```
trace: 'ex.etf'

transformation create_needle_actions {

    interval : [-10.0, 0.0] s

    event-filter  : {'name'='NEEDLE_IP'}
    signal-filter : {'name'='NEEDLE_P'}

    signal pos    : {'name'='NEEDLE_P'}
    def half_blue : {'name'='NEEDLE_IP', 'type'='spike', 'color'='half_blue'}
    def red       : {'name'='NEEDLE_IP', 'type'='spike', 'color'='red'}
    def green     : {'name'='NEEDLE_IP', 'type'='spike', 'color'='green'}

    claim-spec extend:
        (half_blue and pos <= 0.0015), (red and pos >= 0.0015),
        {'resource'='PushupUnit', 'peripheral'='motor', 'action'='extend'}
    claim-spec pushup:
        (red and pos >= 0.0015), (half_blue and pos >= 0.0015),
        {'resource'='PushupUnit', 'peripheral'='motor', 'action'='pushup'}
    claim-spec retract1:
        (half_blue and pos >= 0.0015), (green and pos <= 0.0015),
        {'resource'='PushupUnit', 'peripheral'='motor', 'action'='retract1'}
    claim-spec retract2:
        (green and pos <= 0.0015), (red  and pos <= 0.0015),
        {'resource'='PushupUnit', 'peripheral'='motor', 'action'='retract2'}

    trim     : extend[0] to retract2[49]
    shift-to : 0.0 s
}

transformation add_throughput {

    strategy : augment

    signal tp : throughput of end {'action'='retract2'} per hr over 1.0 s

    signal-spec dies_per_hr : tp, {'name'='dies/hr'}

    save-with-suffix : transformed
}
```

(a)



(b)



(c)

Fig. 11 A transformation specification (a), (filtered) input trace (b), and generated output trace (c) for the die bonder

performance-analysis techniques based on Little's law. This is explained in Sect. 3. The final line of the second subtransformation then specifies a file-name suffix for writing the resulting trace.

Now consider the execution trace of the die bonder as shown in Fig. 11b. This trace contains the signal and events that are used in the transformation of Fig. 11a. The first subtransformation discretizes the movement of the needle

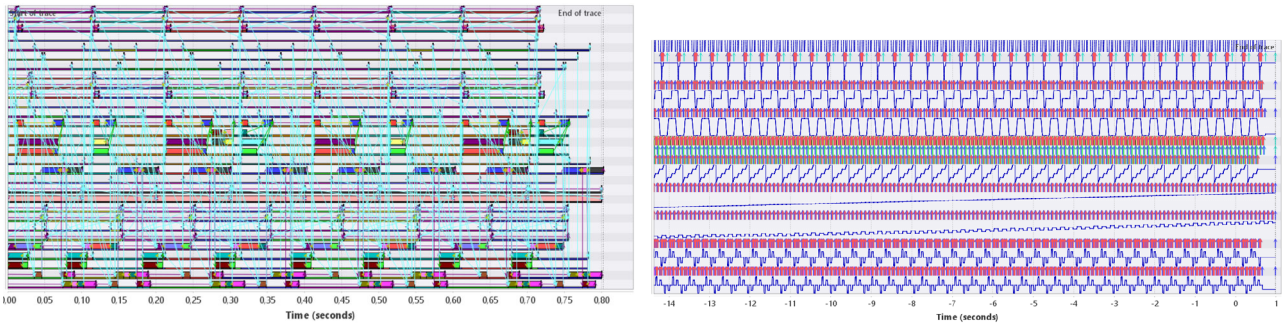


Fig. 12 LSAT trace of happy flow (a) and partial system trace of happy flow (b) for the die bonder

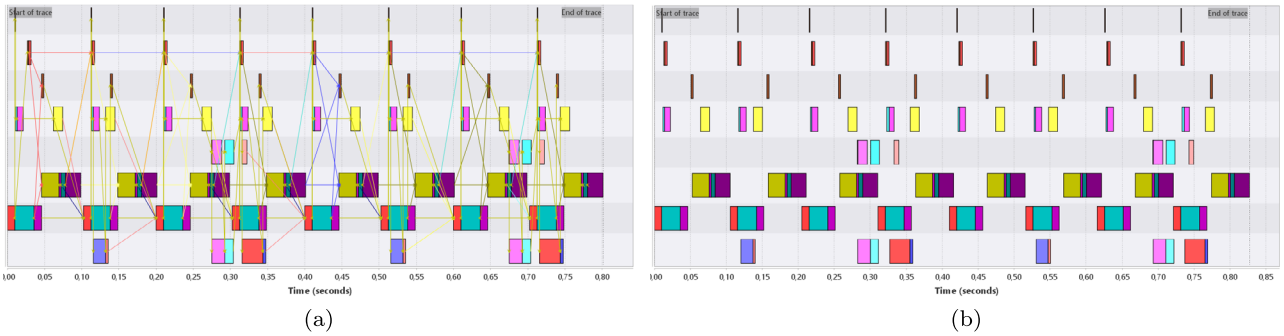


Fig. 13 Normalized LSAT trace (a) and normalized system trace (b) for the die bonder

in four phases using the information from both the events and the signal. Furthermore, the output trace is trimmed (not visible) and shifted in time. The second subtransformation adds a throughput signal. Figure 11c shows part of the output of the transformation with the four types of claims and the throughput signal.

Example 13
For another example of the use of transformations, consider again the die bonder of Ex. 2. We want to develop a new variant of the system with an additional inspection operation. We have used the LSAT tool [82–84] to model the existing system, and we need to validate the model. First, we acquire a model trace of the happy-flow execution using LSAT (see Fig. 12a) and a system trace of the happy-flow execution (see Fig. 12b). (The labels on the y-axis in Figs. 12 and 13 have been left out; they do not matter for this example.) Clearly, these traces are very different. The LSAT trace is on a high level of abstraction (mechatronic actions) and contains modeling artifacts such as elements that are solely used for ensuring mutual exclusion. The system trace is on a low level of abstraction and contains supervisory control events and signals for, e.g., motor positions. Furthermore, the timestamps are far from synchronized. Using the trace-transformation technique, we normalize both traces to contain claims for the relevant high-level actions (see Fig. 13a and Fig. 13b, respectively). The traces look similar, but clearly the model trace is a bit shorter than the system trace. Also note that the dependencies of the LSAT trace are preserved by the transformation. The normalization allows the traces to be compared, which is described in the next section.

6 Constraint graphs and their techniques

This section presents constraint graphs, as well as critical-path analysis, trace comparison, and repetitive-structure analysis, which all are based on constraint graphs.

6.1 Constraint-graph formalism

Constraint graphs are directed graphs with real-valued weights on the edges. They specify constraints on the relative timing of events.

Definition 20 (Constraint graph)
A weighted and directed graph $(\mathcal{V},\mathcal{E})$, where $\mathcal{V} \subseteq \overline{\mathbb{R}} \times \mathbf{M}$ is a set of events (using the extended real numbers) and $\mathcal{E} \subseteq \mathcal{V} \times \overline{\mathbb{R}} \times \mathcal{V}$ is a set of edges, is called a constraint graph.

Let $w \in \overline{\mathbb{R}}^{\geq 0}$. The edge $((t_i, m_i), w, (t_j, m_j))$ encodes the constraint $t_j \geq t_i + w$ and thus specifies a lower bound on the time between the events. Note that we can specify upper bounds by negative weights: the edge $((t_j, m_j), -w, (t_i, m_i))$ encodes the constraint $t_i \geq t_j + (-w)$, which can be written as $t_j \leq t_i + w$.

The constraint-graph representation of an execution trace is used for critical-path analysis, for comparison of traces, and for repetitive-structure analysis. Before we explain these techniques, we first explain the semantics of a constraint graph and various ways to construct constraint graphs from execution traces.

Definition 21 (Constraint-graph semantics)
Consider a trace $X = (E, D, R, C, S)$. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a constraint graph. We say that X satisfies $\mathcal{G}$, denoted by $X \models \mathcal{G}$, if and only if for every $((t_i, m_i), w, (t_j, m_j)) \in \mathcal{E}$ it holds that if $(t_i, m_i) \in E$ and $(t_j, m_j) \in E$, then $t_j \geq t_i + w$.

The default constraint graph for an execution trace $X = (E, D, R, C, S)$, denoted by $\mathcal{G}_X$, is the graph $(E, \emptyset)$. That is, the default constraint graph contains all events and no edges between the events. Clearly, $X \models \mathcal{G}_X$. The default constraint graph can be extended in the following ways:

- *Source and sink*: Add an *src* event $(-\infty, \emptyset)$ and an *snk* event $(\infty, \emptyset)$. It also adds constraints $(src, 0, v)$ and $(v, 0, snk)$ for all other events $v \in \mathcal{V}$.
- *Claim durations*: For each $c = (t_0, t_1, r, a, o, m) \in C$ with $\delta = t_1 - t_0$, constraints $(start(c), \delta, end(c))$ and $(end(c), -\delta, start(c))$ are added. These constraints encode the claim duration.
- *ϵ-end-start constraints*: Given parameter $\epsilon \in \mathbb{R}^{\geq 0}$, this heuristic adds a lower bound constraint with value 0 between the end of a claim c and the start of a claim c' if and only if c' starts within ϵ time units after c ends. More formally, for each claim $c = (t_0, t_1, r, a, o, m) \in C$ and $c' = (t'_0, t'_1, r', a', o', m') \in C$ such that $0 \leq t'_0 - t_1 \leq \epsilon$, a lower-bound constraint $(end(c), 0, start(c'))$ is added.
- *Dependency constraints*: Let $D' \subseteq D$. We add a constraint (e_0, w, e_1) for each $(e_0, e_1, m) \in D'$, where $w \in \mathbb{R}$ or w is given by some attribute $a \in \mathbf{A}$ such that $m(a)$ is interpretable as a real number.

With these extensions, we define three constraint graphs:

1. $\mathcal{G}_X^{\epsilon=d}$ starts from $\mathcal{G}_X$, adds claim durations, adds ϵ-end-start constraints for $\epsilon = d$, and adds source and sink.
2. $\mathcal{G}_X^{\infty}$ starts from $\mathcal{G}_X$ and adds ϵ-end-start constraints for $\epsilon = \infty$.
3. $\mathcal{G}_X^{D(w)}$ starts from $\mathcal{G}_X$, adds claim durations, adds dependency constraints for all $(e_0, e_1, m) \in D$ with weight w (in case $w \in \mathbb{R}$) or with a weight given by interpretation of $m(w)$ as a real number (in case $w \in \mathbf{A}$), and adds source and sink.

The first constraint graph captures execution dependencies: intra-claim (via the claim durations) and inter-claim (via the ϵ-end-start constraints). The second constraint graph gives a partial ordering on claims that orders two claims if and only if one starts not earlier than the other finishes. The third constraint graph interprets the dependencies of the trace as constraints. Note that $X \models \mathcal{G}_X^{\epsilon=d}$ and $X \models \mathcal{G}_X^{\infty}$ by definition. Whether $X \models \mathcal{G}_X^{D(w)}$ depends on the user-specified dependencies.

A path in a constraint graph is a finite sequence of edges $(v_1, w_1, v'_1), (v_2, w_2, v'_2), \ldots, (v_n, w_n, v'_n)$ such that for all $1 \leq i < n$ we have $v'_i = v_{i+1}$. The sum of all the weights of the edges in the path is the weight of the path. Cycles with a positive weight indicate inconsistent constraints. A constraint graph that has cycles with a positive weight has no trace that satisfies it.

6.2 Critical-path analysis

Critical-path analysis originates from the domain of project management and was originally used to optimize project duration by analysis of the project's dependency graph [15, 16]. Given a weighted directed and acyclic graph, the critical paths are the longest paths (in terms of the weight), and they show the parts of the system whose improvement can help to decrease the time needed to finish all the work. This concept has succesfully been applied to execution traces in [21, 22] by representing an execution trace as a weighted directed acyclic graph. This section on critical-path analysis summarizes and generalizes [22] by applying critical-path analysis to the constraint graph, which may contain cycles and negatively weighted edges that encode upper bounds on the time between events. This generalization is straightforward, as a constraint graph is an *event network* [23], for which critical-path analysis has already been studied.

Definition 22 (Critical path)
Let $(\mathcal{V}, \mathcal{E})$ be a constraint graph and let $v, v' \in \mathcal{V}$. The *critical paths* from v to v' are the longest paths (in terms of weight) from v to v'.

The critical paths in a constraint graph can be computed with the Bellman–Ford–Moore longest path algorithm as shown in [23]. Note that a constraint graph can contain cycles. If a constraint graph has a cycle with a positive weight, then we say that the constraint graph is infeasible, because in that case the constraints are inconsistent. This situation can be detected on-the-fly by the Bellman–Ford–Moore algorithm.

Our critical-path analysis technique first creates a constraint graph from an execution trace, then applies critical-path analysis, and finally visualizes the critical-path result by highlighting the claims and dependencies of the critical

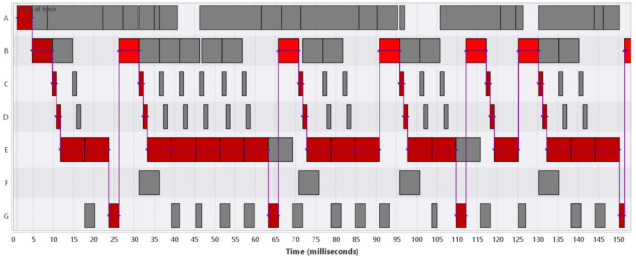


Fig. 14 Critical path with application information

path. We use two of the three types of constraint graphs that have been defined in Sect. 6.1:

- $\mathcal{G}_X^{D(0)}$,
- $\mathcal{G}_X^{\epsilon=d}$.

Both these constraint graphs have *src* and *snk* events, and the Bellman–Ford–Moore algorithm computes the critical paths from the *src* event (which happens before every event in the execution trace) to the *snk* event (which happens after every event in the execution trace).

The first graph uses the existing dependencies in the trace as lower-bound constraints in the constraint graph. The second constraint graph uses a heuristic to capture the execution dependencies in the trace. This is equivalent to the ϵ-approximation of [22]. Our critical-path analysis technique is therefore equivalent to the technique described in [22], and the theory from [22] carries over. Most notably, under certain assumptions the ϵ-end-start heuristic gives an over-approximation of the critical paths of the system. Note that the existing dependencies in the trace are not used for construction of this constraint graph. Similar to [22], we provide the option to interpret the existing dependencies in the trace as *application dependencies*. This information is combined with the critical-path information to identify dependencies in the critical path that are not due to application dependencies. Such critical dependencies may be of special interest, because they may be resolved in the implementation of the system (without affecting the application), potentially speeding up the system. Ex. 14 below shows an example.

Note that the critical-path analysis that uses $\mathcal{G}_X^{\epsilon=d}$ uses the ϵ-end-start heuristic that only works on claims. That is, there will be no dependencies involving regular events, and these will thus not be used for the critical-path analysis. Ultimately, it is the user's responsibility to specify the constraint graph on which to apply critical-path analysis.

Example 14
Consider the image-processing pipeline of Ex. 1. We apply critical-path analysis to $\mathcal{G}_X^{\epsilon=0}$. Note that the trace has dependencies and that these are exactly the dependencies that encode the application order. We therefore use the option to interpret the existing dependencies as application dependencies. This information is used for an extra highlight of claims on the critical path that have no critical predecessor that follows from the application dependencies. Figure 14 shows part of the critical path. The path follows the application dependencies, except for the G to B jumps. The B instances on the critical path that follow a G instance are therefore colored bright red. What happens is that the execution of task B is blocked due to lacking memory resources. As G ends, memory is freed and a waiting B can start. Increasing the amount of memory may therefore help to improve the performance.

6.3 Application example: bottleneck analysis

During performance engineering, one of the main questions often is to determine the *bottleneck* of a system. Therefore, we obtain one or more representative traces. Since the critical-path analysis technique is at the core of this method, it is essential that the execution traces are suitable for the construction of meaningful constraint graphs. Typically, execution traces in which the main system activities are modeled by claims or execution traces with dependencies can be used. We can then apply the following techniques:

- If necessary, process the trace with the transformation technique. Low-level events and signals such as in the traces of the die bonder of Ex. 2 can be abstracted into claims and/or dependencies. This makes the traces suitable for critical-path analysis.
- Apply critical-path analysis to obtain the critical paths of the system. The essence of critical-path analysis is a proper constraint-graph representation. The technique uses either $\mathcal{G}_X^{D(0)}$ or $\mathcal{G}_X^{\epsilon=d}$. The former lets the user define all dependencies (and thus constraints). This can be done, e.g., during construction of the trace or during application of the transformation technique. The latter is subtle and seems to work best on traces from models with $\epsilon = 0$, because models typically have an as-soon-as-possible execution semantics with "perfect" timing.
- Apply resource-usage analysis to discover resources with high load. This gives an indication of improvements with respect to resources if combined with the critical-path information.

Example 15
Consider the image-processing pipeline of Ex. 1. We have modeled the existing system and want to experiment with upgrades that improve its performance. Therefore, we need to know the bottlenecks of the system. Ex. 4 already showed that both the CPU and memory M2 have high loads. Since the execution trace is produced by a discrete-event simulator that we can tune, we do not need any preprocessing using the trace-transformation technique. The dependencies in the trace are dependencies of the application. No dependencies that result from interaction on the resources are present in the

execution trace because these are not statically known: they emerge from the simulation. That is why we rely on the $\epsilon = 0$ heuristic: if a claim starts at the same moment that another claim ends, then we add a dependency between them. This assumes that this kind of "perfect" timing behavior is due to the fact that the latter claim must wait for a resource used by the former claim. Ex. 14 gave us the result of the first application of the critical-path analysis. The interpretation of the dependencies as application dependencies gives us additional information, namely that activity B always waits for activity G to release memory. We can thus conclude that memory M2 is a bottleneck of the system. We act by increasing the amount of memory M2, and this indeed solves this bottleneck.

A second application of the critical-path analysis now shows that in the system with increased M2 memory, activity A is mostly on the critical path. Since both A and G run on the CPU, we create another resource-usage histogram of the number of concurrent clients on the CPU. This histogram shows that 34% of the time the two processes compete for the single-core CPU. Since G has higher priority, A is preempted. To resolve this bottleneck, we can either use a faster single-core CPU or use a dual-core CPU.

6.4 Trace comparison

We present three analysis techniques that can be used for comparing execution traces. Each of the techniques focuses on another aspect. First, the timing analysis focuses on the duration of claims. Second, the order analysis focuses on the ordering and overlap of claims. No quantitative timing aspects are taken into account. Third, constraint analysis focuses on dependencies as timing constraints. Currently, the supported trace-comparison techniques do not consider the signals of execution traces. These can be compared manually via proper visualization (possibly after transformation).

6.4.1 Timing analysis

Timing analysis computes the durations of a set of claims and shows statistics and a histogram of these durations. The set of claims that is used is determined by the *grouping* attributes for the visualization: all claims in a single section form a set. More formally, let $X = (E, D, R, C, S)$ be an execution trace and let $A \subseteq \mathbf{A}$ be the set of grouping attributes of the visualization. Then, we compute the partition of the set of claims according to A, which is $C_{\equiv_A} = \{C_1, C_2, \ldots, C_n\}$. Each subset of claims C_i is used to compute a claim-duration histogram and duration statistics. If we have a second trace, say $X' = (E', D', R', C', S')$, then we also compute the partition: $C'_{\equiv_A} = \{C'_1, C'_2, \ldots, C'_m\}$. We relate the blocks of the different partitions: C_i relates to C'_j if and only if for all $c \in C_i$ and $c' \in C'_j$ it holds that c and c' have the same values for the attributes in A. Histograms for such related blocks are visualized in the same graph to visually compare them. Note that the proper use of attributes in the traces, which might come from different sources, is essential.

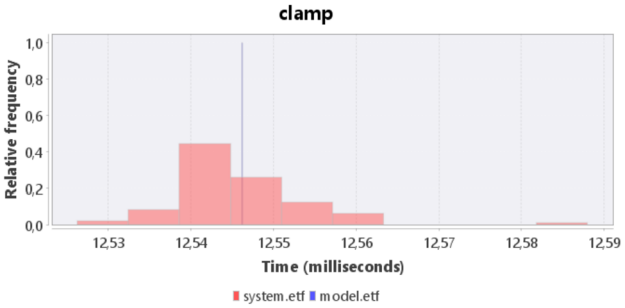


Fig. 15 Timing histogram for a model and system trace

Example 16
Consider the die bonder of Ex. 2. In Ex. 13, we have explained how we have applied trace transformation to a model and a system trace to go from low-level events, claims, and signals to abstract claims with normalized attributes. Figure 15 shows the histogram of the *clamp* claims for both the system and model traces. The distribution for the model trace is very narrow because the model uses only a single value for the timing of the *clamp* claims. Having such an overlay of the timing gives a good visual indication of how the model timing compares to the system timing.

6.4.2 Order analysis

This section is a condensed version of parts of earlier work [22]. We have discussed how we can create constraint graphs from execution traces. For distance analysis, we use the constraint graph $\mathcal{G}^{\infty}_{X}$. This constraint graph contains an edge from the end of claim c to the start of claim c' if and only if c does not end strictly later than c' starts. This graph is equivalent to the ∞-approximated task graph of [22], and the algorithm to compute it from an execution trace carries over to our setting. This graph then is used in the graph edit distance d_{ged} from [22]. This distance essentially captures the number of insertions and deletions of events and edges needed to transform one graph into the other. The distance between two execution traces X_1 and X_2 thus is defined as $d_{\text{ged}}(\mathcal{G}^{\infty}_{X_1}, \mathcal{G}^{\infty}_{X_2})$. In [22], it is shown that this is a pseudo-metric on execution traces. We use the scheme of [22] to assign weights to claims to indicate how much they contribute to the pseudo-metric's value. This weight is used for visualization. Note that we use the attribute mappings of the events to decide whether they are equivalent. The success of comparing graphs with this technique thus heavily depends on assigning proper attributes and values. Finally, note that

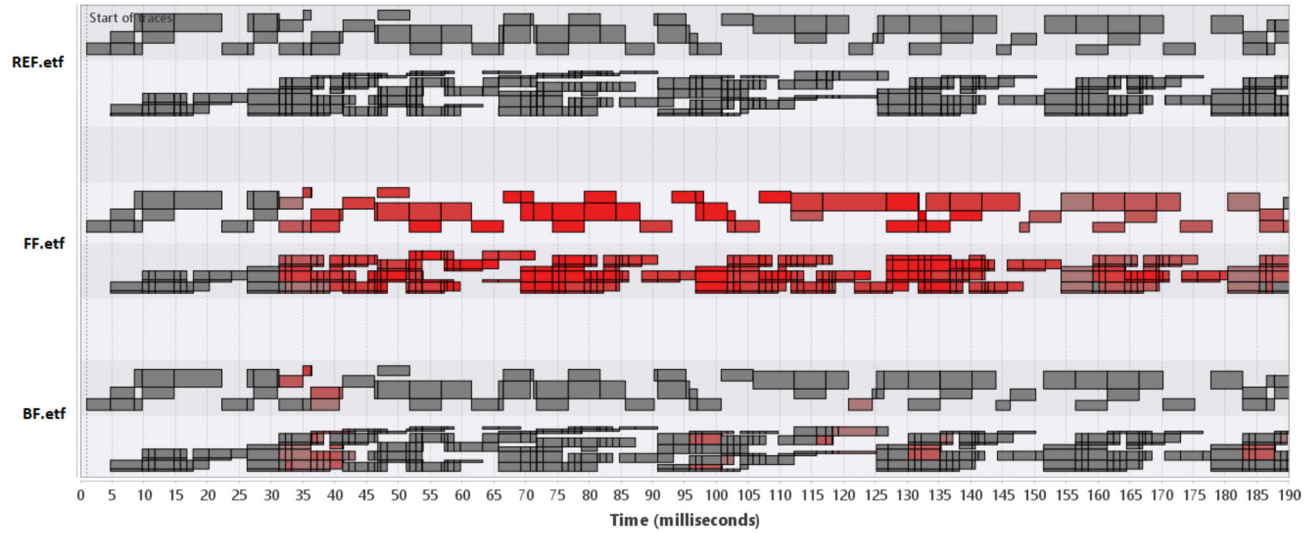


Fig. 16 Visualization of differences of memory allocation strategies (first-fit and best-fit) with respect to a reference

the timestamps in the execution trace are only used qualitatively: they determine the partial order of the claim-specific events but are not used in any other way in the distance computation.

Instead of using the attributes to decide equivalence of events from the traces that are compared, we could apply a more sophisticated matching algorithm such as presented in [85] for structural comparison of labeled transition structures. The scalability of this approach is not so good, however. What we have taken from this work, though, is the method to compute the similarity between traces as a number between 0 (no similarity at all between the traces) and 1 (the traces are identical). This is called the *F-measure* in [85] and its computation is a straightforward extension of the graph-edit-distance computation. It provides a normalized similarity value, whereas the raw graph edit distance may be hard to interpret.

Example 17
Consider the image-processing pipeline of Ex. 1. Suppose that, in addition to the discrete-event simulation model, we have a reference execution trace from the system. We have difficulty modeling the memory-allocation strategy that the system uses because we do not exactly know the algorithm that the system uses. We decide to approximate it with a best-fit allocation strategy and a first-fit allocation strategy. We create two execution traces and compare them to the reference trace with respect to memory allocation (see Fig. 16). The first two sections on the vertical axis contain the memory allocation pattern of the reference trace. The third and fourth section are for the first-fit strategy, and the fifth and sixth section are for the best-fit strategy. The difference algorithm visualizes the differences by shades of red. Clearly, the best-fit strategy is closest to the reference. In fact, the graph edit distance between the reference and the best-fit strategy is 8,282, and the graph edit distance between the reference and the first-fit strategy is 26,311. The F-measure for the best-fit strategy is 0.80 and the F-measure for the first-fit strategy is 0.36.

Note that the distance analysis uses $\mathcal{G}_X^\infty$ and thus applies the ϵ-end-start heuristic that works on claims. That is, there will be no dependencies involving regular events not being the start or end of a claim. These events are, however, part of the constraint graph and hence they do count in the graph edit distance through the vertex part.

6.4.3 Constraint-graph satisfaction

Consider the traces X_1 and X_2. We use the constraint-graph semantics of Def. 21 to compute whether the trace X_1 is a model of the constraints specified by $\mathcal{G}_{X_2}^{D_2(a)}$, where D_2 contains the dependencies of X_2 and $a \in \mathbf{A}$ is an attribute that is present on each dependency and is interpretable as a real number. The algorithm to compute $X_1 \models \mathcal{G}_{X_2}^{D_2(a)}$ follows directly from the semantics, and it is straightforward to compute the set of dependencies that are not respected by the execution trace.

Example 18
Consider the die bonder of Ex. 2. In Ex. 13, we have explained how we have applied trace transformation to a model and a system trace to go from low-level events, claims, and signals to more abstract claims with normalized attributes. Let X_1 be the normalized system trace shown in Fig. 13b and let X_2 be the normalized model trace shown in Fig. 13a. The

transformation has preserved the transitive dependencies and added the sum of the durations of claims between source and destination events of a generated dependency to the *weight* attribute. Next, we compute whether $X_1 \models \mathcal{G}_{X_2}^{D_2(weight)}$. This is indeed the case: all constraints specified by the model are satisfied by the system. This kind of comparison is useful for model or system validation.

6.5 Application example: model/system validation

Model and/or system validation can be done by comparing model traces and system traces [84]. Therefore, we need to obtain representative traces from the system and the model. Care needs to be taken that the traces that need to be compared execute the same scenario, e.g., the happy flow. We can then apply the following techniques:

- Normalize the traces using the transformation technique. This results in traces in which equivalent trace entities have the same attributes and values.
- Visualize the traces in a single view. If equivalent trace entities have the same attributes and values, then these can be used for grouping and coloring. This makes manual inspection of (small) traces much easier.
- Compare the traces using the timing analysis, graph-edit distance, and constraint-graph satisfaction techniques.

The appropriate action following the comparison depends on whether we do model validation or system validation. In the first case, the model is a *scientific model* in terms of Lee [86], which serves as a useful approximation of something that already exists in nature (the system). Any mismatches are thus caused by an inappropriate model and should therefore be corrected in the model. In the latter case, the model is an *engineering model*, which serves as a specification of something that has been created. Mismatches are thus caused by an inappropriate implementation, and should therefore be corrected there (of course, that might be expensive).

Example 19
Consider Ex. 13 about the die bonder. We have used the trace-transformation technique to normalize both the model and the system trace to contain claims for the relevant high-level actions (see Fig. 13a and Fig. 13b, respectively). The traces look similar, but the model trace is a bit shorter than the system trace. In addition to the visual inspection, we also apply the timing analysis and constraint-graph satisfaction techniques (see Ex. 16 and Ex. 18, respectively). The former shows that the system timing deviates from the model timing. The latter shows that all constraints in the model trace are respected by the system trace.

6.6 Repetitive-structure analysis

Our repetitive-structure analysis method is based on the assumption that execution traces originate from systems that process discrete items and that each event in the execution trace has an attribute that encodes an item identifier. This often is a natural assumption for, e.g., image-processing systems such as in Ex. 1 or manufacturing systems such as in Ex. 2. This item-identifier attribute, assumed to be *id*, can be used to split an execution trace $X = (E, D, R, C, S)$ into subtraces. Let $V = \{v_1, v_2, \ldots, v_n\}$ be the values of *id* in E and C. We define n subtraces $X_1, \ldots, X_n$ as follows: $X_i = (E_i, \emptyset, R, C_i, \emptyset)$, where

- $E_i = \{(t, m) \in E \mid m(id) = v_i\}$, and
- $C_i = \{(t_0, t_1, r, a, o, m) \in C \mid m(id) = v_i\}$.

In Fig. 1b, for instance, the colors mark the subtraces. The subtraces only consist of the events and claims; our approach leaves out the dependencies and signals of the trace. The reason is that we reuse the graph edit distance of the previous section, which uses a constraint graph based on events and claims only. We can now use the graph edit distance for an equivalence relation that says that two subtraces are equivalent if and only if their constraint graphs are identical modulo the value of the *id* attribute.

Definition 23 (Behavioral equivalence)
Consider the subtraces $X_1, \ldots, X_n$ for attribute *id*. We say that X_i and X_j are *behaviorally equivalent* for *id* if and only if $d_{\text{ged}}(\mathcal{G}^{\infty}_{X_i[id \leftarrow 0]}, \mathcal{G}^{\infty}_{X_j[\text{id} \leftarrow 0]}) = 0$. We denote this by $X_i \approx_{id} X_j$.

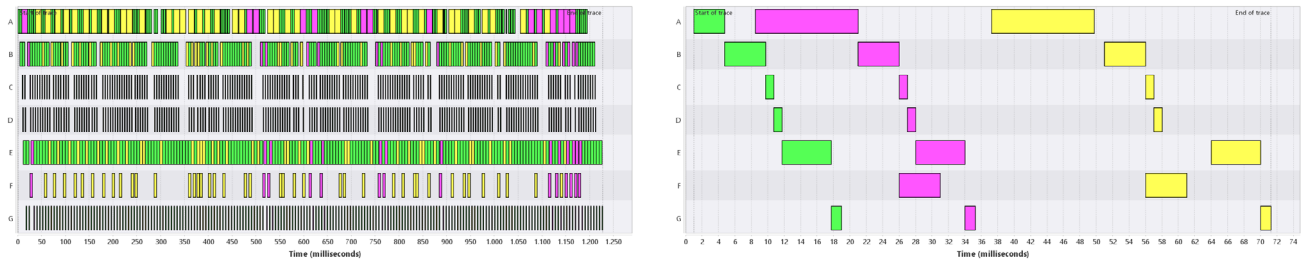
From the fact that the execution distance is a pseudo-metric, it follows straightforwardly that this is an equivalence relation on the set of subtraces. Therefore, we can partition the set of subtraces by this equivalence relation:

$$\left\{B^1 = \{X_1^1, \ldots, X_{n_1}^1\}, B^2 = \{X_1^2, \ldots, X_{n_2}^2\}, \ldots\right\}.$$

Each block B^i contains n_i subtraces X_j^i. Each subtrace contains the events and claims involved with processing a single item identifier, and all subtraces in a block have equivalent behavior. We use this partitioning for three kinds of analysis:

- showing a bar chart with the count of each behavior, i.e., $|B^i|$ for each i,
- visualizing the execution trace with a coloring according to behavior,
- visualizing representatives of each behavior.

Example 20
Consider the image-processing pipeline of Ex. 1. Note that task F is only executed for objects of type *high* and that tasks A and G share the CPU. We are wondering how many



(a) (b)

Fig. 17 Coloring according to the three behaviors (a) and representatives of each of the three behaviors (b)

different execution patterns exist with respect to the processing of a single image. There are three different behaviors, and Fig. 17a shows how they are distributed in the execution trace. Figure 17b shows a representative of each behavior. Note that two of the three behaviors are for processing images of type *high* using task F. These differ with respect to the ordering between E and F.

7 Conclusion

In this paper we have presented the Trace tool of the Eclipse Trace4cps project [2], which can support the trace-based performance engineering of embedded and cyber-physical systems. The core formalism is the *execution trace*, which models a single system behavior. Execution traces can be obtained from running systems (e.g., via the logging) and also from executable models (e.g., simulations). This makes the tool applicable in almost every phase of a system's lifecycle: from architecture, design, and realization (traces are produced by models and prototypes) to operation (traces are produced by the running system). Therefore, the tool fits well with model-based performance engineering methods.

Funding This work was partially supported by the TRANSACT EU project (https://transact-ecsel.eu/). TRANSACT has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No. 101007260. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Belgium, Denmark, Finland, Germany, Poland, the Netherlands, Norway, and Spain.

The research was carried out as part of the Bright program under the responsibility of ESI (TNO) with ITEC as the carrying industrial partner. The Bright research is supported by the Netherlands Organisation for Applied Scientific Research TNO.

The research was carried out as part of the Octo+ program under the responsibility of ESI (TNO) with Océ Technologies B.V. as the carrying industrial partner. The Octo+ research is supported by the Netherlands Organisation for Applied Scientific Research TNO.

This research was supported by the ARTEMIS joint undertaking under grant agreement No. 621439 (ALMARVI).

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Lee, E.A.: Cyber physical systems: design challenges. In: 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC) (2008)
2. Eclipse Foundation. Eclipse Trace4cps. https://www.eclipse.org/trace4cps/
3. van der Sanden, B., Li, Y., van den Aker, J., Akesson, B., Bijlsma, T., Hendriks, M., Triantafyllidis, K., Verriet, J., Voeten, J., Basten, T.: Model-driven system-performance engineering for cyber-physical systems. In: 2021 International Conference on Embedded Software (EMSOFT) (2021)
4. Heemels, M., Muller, G. (eds.): Boderc: Model-based design of high-tech systems. Embedded Systems Institute (2006)
5. Derler, P., Lee, E.A., Sangiovanni Vincentelli, A.: Modeling cyber–physical systems. Proc. IEEE **100**(1) (2012)
6. Fitzgerald, F., Larsen, P.G., Verhoef, M. (eds.): Collaborative Design for Embedded Systems Springer, Berlin (2014)
7. Basten, T., van Benthum, E., Geilen, M., Hendriks, M., Houben, F., Igna, G., Reckers, F., De Smet, S., Somers, L., Teeselink, E., Trčka, N., Vaandrager, F., Verriet, J., Voorhoeve, M., Yang, Y.: Model-driven design-space exploration for embedded systems: the octopus toolset. In: 4th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation. LNCS., vol. 6415. Springer, Berlin (2010)
8. Takeda, S., Masuko, T., Takano, N., Inada, T.: Die Attach Adhesives and Films. Springer, Berlin (2017)
9. Eclipse Foundation. Eclipse TraceCompass. https://www.eclipse.org/tracecompass/
10. OpenTracing. https://opentracing.io/
11. ChronView. https://www.inchron.com/chronview/
12. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Dapper, C.S.: Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010
13. Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K.W., Schaller, B., Shan, P., Viscomi, B., Venkataraman, V., Veeraraghavan, K., Canopy, Y.J.S.: An end-to-end performance

tracing and analysis system. In: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP'17. Association for Computing Machinery, (2017)
14. Roos, N.: Clearing the critical software path. *Bits&Chips*, 2021. https://bits-chips.nl/artikel/clearing-the-critical-software-path/
15. Kelley, J.E., Walker, M.R.: Critical-path planning and scheduling. In: Eastern Joint IRE-AIEE-ACM Computer Conference. ACM, New York (1959)
16. Lockyer, K.G.: Introduction to Critical Path Analysis. Pitman, London (1964)
17. Wiest, J.D.: Some properties of schedules for large projects with limited resources. Oper. Res. **12**(3) (1964)
18. Kastor, A., Sirakoulis, K.: The effectiveness of resource levelling tools for resource constraint project scheduling problem. Int. J. Proj. Manag. **27** (2009)
19. Goldratt, E.: Critical Chain. North River Press (1997)
20. Pinedo, M.: Scheduling: Theory, Algorithms and Systems, 2nd edn. Prentice Hall, New York (2002)
21. Hendriks, M., Vaandrager, F.W.: Reconstructing critical paths from execution traces. In: International Conference on Embedded and Ubiquitous Computing. IEEE Comput. Soc., Los Alamitos (2012)
22. Hendriks, M., Verriet, J., Basten, T., Theelen, B., Brassé, M., Somers, L.: Analyzing execution traces – critical-path analysis and distance analysis. Int. J. Softw. Tools Technol. Transf. **19**(4) (2017)
23. Elmaghraby, S.E., Kamburowski, J.: The analysis of activity networks under generalized precedence relations (GPRs). Manag. Sci. **38**(9) (1992)
24. Wu, M.Y., Hypertool, D.D.G.: A programming aid for message-passing systems. IEEE Trans. Parallel Distrib. Syst. **1**(3) (1990)
25. Bjorn-Jorgensen, P., Madsen, J.: Critical path driven cosynthesis for heterogeneous target architectures. In: 5th International Workshop on Hardware/Software Co-Design. IEEE, New York (1997)
26. Yang, C.-Q., Miller, B.P.: Critical path analysis for the execution of parallel and distributed programs. In: 8th International Conference on Distributed Computing Systems. IEEE, New York (1988)
27. Hollingsworth, J.K.: An online computation of critical path profiling. In: 1st ACM SIGMETRICS Symposium on Parallel and Distributed Tools. ACM, New York (1996)
28. Barford, P., Crovella, M.: Critical path analysis of TCP transactions. SIGCOMM Comput. Commun. Rev. **30**(4) (2000)
29. Schulz, M.: Extracting critical path graphs from MPI applications. In: International Conference on Cluster Computing. IEEE, New York (2005)
30. Bohme, D., Wolf, F., de Supinski, B.R., Schulz, M., Geimer, M.: Scalable critical-path based performance analysis. In: 26th International Parallel Distributed Processing Symposium. IEEE, New York (2012)
31. Luo, J., Jha, N.K.: Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems. In: ASP-DAC. IEEE, New York (2002)
32. van der Aalst, W.M.P., van Dongen, B.F., Herbst, J., Maruster, L., Schimm, G., Weijters, A.J.M.M.: Workflow mining: a survey of issues and approaches. Data Knowl. Eng. **47** (2003)
33. van der Aalst, W.M.P., van Dongen, B.F.: Discovering workflow performance models from timed logs. In: 1st International Conference on Engineering and Deployment of Cooperative Information Systems. Springer, Berlin (2002)
34. Yang, Y., Geilen, M., Basten, T., Stuijk, S., Corporaal, H.: Automated bottleneck-driven design-space exploration of media processing systems. In: Design, Automation Test in Europe Conference Exhibition. IEEE, New York (2010)
35. Henzinger, T.A., Majumdar, R., Prabhu, V.S.: Quantifying similarities between timed systems. In: Formal Modeling and Analysis of Timed Systems, vol. 3829. Springer, Berlin (2005)
36. Huang, J., Voeten, J., Geilen, M.: Real-time property preservation in concurrent real-time systems. In: 10th International Conference on Real-Time and Embedded Computing Systems and Applications (2004)
37. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. Sov. Phys. Dokl. **10** (1966)
38. Mannila, H., Ronkainen, P.: Similarity of event sequences. In: 4th International Workshop on Temporal Representation and Reasoning. IEEE Comput. Soc., Los Alamitos (1997)
39. Gao, X., Xiao, B., Tao, D., Li, X.: A survey of graph edit distance. Pattern Anal. Appl. **13**(1) (2010)
40. Zeng, Z., Tung, A.K.H., Wang, J., Feng, J., Zhou, L.: Comparing stars: on approximating graph edit distance. Proc. VLDB Endow. **2**(1) (2009)
41. Akoglu, L., Tong, H., Koutra, D.: Graph based anomaly detection and description: a survey. Data Min. Knowl. Discov. **29**(3) (2015)
42. Noble, C.C., Cook, D.J.: Graph-based anomaly detection. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Association for Computing Machinery, New York (2003)
43. Ghamarian, A.H., Geilen, M.C.W., Stuijk, S., Basten, T., Theelen, B.D., Mousavi, M.R., Moonen, A.J.M., Bekooij, M.J.G.: Throughput analysis of synchronous data flow graphs. In: Proceedings of the Sixth International Conference on Application of Concurrency to System Design. IEEE Comput. Soc., Los Alamitos (2006)
44. Kleinrock, L.: Queueing Systems, Vol I: Theory. Wiley, New York (1975)
45. Little, J.D.C.: A proof for the queueing formula: $l = \lambda w$. Oper. Res. **9**(3) (1961)
46. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebraic Program. **78**(5) (2009)
47. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. **2**(4) (1990)
48. Alur, R., Henzinger, T.A.: Real-time logics: complexity and expressiveness. Inf. Comput. **104** (1993)
49. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. J. ACM **43**(1) (1996)
50. Geilen, M.: An improved on-the-fly tableau construction for a real-time temporal logic. In: Computer Aided Verification, vol. 2725. Springer, Berlin (2003)
51. Ničković, D., Piterman, N.: From MTL to deterministic timed automata. In: Proceedings of the 8th International Conference on Formal Modeling and Analysis of Timed Systems, vol. 6246. Springer, Berlin (2010)
52. Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime assurance based on formal specifications. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (1999)
53. Drusinsky, D.: The temporal rover and the ATG rover. In: Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification. Springer, Berlin (2000)
54. Havelund, K., Roşu, G.: Testing linear temporal logic formulae on finite execution traces. Technical report (2001)
55. Thati, P., Roşu, G.: Monitoring algorithms for metric temporal logic specifications. Electron. Notes Theor. Comput. Sci. **113** (2005)
56. Markey, N., Raskin, J.-F.: Model checking restricted sets of timed paths. Theor. Comput. Sci. **358**(2) (2006)
57. Ho, H.M., Ouaknine, J., Worrell, J.: Online monitoring of metric temporal logic. In: Runtime Verification, vol. 8734. Springer, Berlin (2014)
58. Hendriks, M., Geilen, M., Behrouzian, A.R.B., Basten, T., Alizadeh, H., Goswami, D.: Checking metric temporal logic with trace. In: 16th International Conference on Application of Concurrency to System Design (2016)

59. Hendriks, M., Geilen, M., Behrouzian, A.R.B., Basten, T., Alizadeh, H., Goswami, D.: Checking metric temporal logic with Trace. Technical Report ESR-2016-01, (2016)
60. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Van Campenhout, D.: Reasoning with temporal logic on truncated paths. In: Computer Aided Verification, vol. 2725. Springer, Berlin (2003)
61. Maler, O., Ničković, D.: Monitoring temporal properties of continuous signals. In: Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, vol. 3253. Springer, Berlin (2004)
62. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theor. Comput. Sci. **410**(42) (2009)
63. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Proceedings of the 8th International Conference on Formal Modeling and Analysis of Timed Systems. LNCS., vol. 6246. Springer, Berlin (2010)
64. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: Proceedings of the 25th International Conference on Computer Aided Verification. LNCS., vol. 8044. Springer, Berlin (2013)
65. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Computer Aided Verification, vol. 6174. Springer, Berlin (2010)
66. Ničković, D., Lebeltel, O., Maler, O., Ferrère, T., Ulus, D.: AMT 2.0: qualitative and quantitative trace analysis with extended signal temporal logic. In: Tools and Algorithms for the Construction and Analysis of Systems, vol. 10806. Springer, Berlin (2018)
67. Ničković, D., Yamaguchi, T.: RTAMT: Online robustness monitors from STL. In: Automated Technology for Verification and Analysis, Volume 12302 of LNPSE. Springer, Berlin (2020)
68. Broenink, T., Jansen, B., Broenink, J.: Tooling for automated testing of cyber-physical system models. In: IEEE Conference on Industrial Cyberphysical Systems, vol. 1 (2020)
69. Dassault Systemes. Stimulus. https://www.3ds.com/products-services/catia/products/stimulus/
70. MathWorks. Assess temporal logic by using temporal assessments. https://nl.mathworks.com/help/sltest/ug/temporal-assessments.html
71. Ferrère, T., Maler, O., Ničković, D.: Mixed-time signal temporal logic. In: Formal Modeling and Analysis of Timed Systems, vol. 11750. Springer, Berlin (2019)
72. Noda, K., Kobayashi, T., Agusa, K.: Execution trace abstraction based on meta patterns usage. In: 20th Working Conference on Reverse Engineering. IEEE Comput. Soc., Los Alamitos (2012)
73. Feng, Y., Dreef, K., Jones, J.A., van Deursen, A.: Hierarchical Abstraction of Execution Traces for Program Comprehension. 26th Conference on Program Comprehension. Association for Computing Machinery (2018)
74. Montani, S., Leonardi, G., Striani, M., Quaglini, S., Cavallini, A.: Multi-level abstraction for trace comparison and process discovery. Expert Syst. Appl. **81**(C) (2017)
75. Bose, J.C.R.P., van der Aalst, W.M.P.: Abstractions in process mining: a taxonomy of patterns. In: Business Process Management. Springer, Berlin (2009)
76. Gad, R.: Improving packet capture trace import in trace compass with a data transformation DSL. In: IEEE 41st Annual Computer Software and Applications Conference, vol. 2 (2017)
77. Walrand, J.: Probability in Electrical Engineering and Computer Science. Springer, Berlin (2021)
78. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Form. Methods Syst. Des. **19**(3) (2001)
79. Deshmukh, J.V., Donzé, A., Ghosh, S., Jin, X., Juniwal, G., Seshia, S.A.: Robust online monitoring of signal temporal logic. Form. Methods Syst. Des. **51**(1) (2017)
80. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining counterexamples using causality. In: Computer Aided Verification, vol. 5643. Springer, Berlin (2009)
81. Behave. https://behave.readthedocs.io/
82. van der Sanden, B., Blankenstein, Y., Schiffelers, R., Voeten, J.: LSAT: Specification and analysis of product logistics in flexible manufacturing systems. In: 17th IEEE International Conference on Automation Science and Engineering. IEEE Press, New York (2021)
83. Eclipse Foundation. Eclipse LSAT. https://www.eclipse.org/lsat/
84. Verriet, J., van der Sanden, B., van der Veen, G., van Splunter, A., Lousberg, S., Hendriks, M., Basten, T.: Experiences and lessons from introducing model-based analysis in brown-field product family development. In: Proceedings of the 11th International Conference on Model-Based Software and Systems Engineering—MODELSWARD. INSTICC, SciTePress (2023)
85. Walkinshaw, N., Bogdanov, K.: Automated comparison of state-based software models in terms of their language and structure. ACM Trans. Softw. Eng. Methodol. **22**(2) (2013)
86. Lee, E.A.: Plato and the Nerd: The Creative Partnership of Humans and Technology. MIT Press, Cambridge (2018)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.