Technische Universiteit
Eindhoven
University of Technology

Inferring Timed Message Sequence Charts from Execution Traces of Large-scale Component-based Software Systems

Ruben Jonk, Jeroen Voeten, Marc Geilen, Rolf Theunissen, Yuri Blankenstein, Twan Basten, and Ramon Schiffelers

ES Reports
ISSN 1574-9517

ESR-2019-01
12 August 2019

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

Inferring Timed Message Sequence Charts from Execution Traces of Large-scale Component-based Software Systems

Ruben Jonk[1], Jeroen Voeten[2,1], Marc Geilen[1], Rolf Theunissen[2], Yuri Blankenstein[2], Twan Basten[1,2], and Ramon Schiffelers[3,1]

[1]Eindhoven University of Technology, Eindhoven, The Netherlands
[2]ESI (TNO), Eindhoven, The Netherlands
[3]ASML, Veldhoven, The Netherlands

***Abstract*—This paper introduces a new measurement-based approach to get insights in timing bottlenecks of existing large-scale component-based software systems. As a foundation we formalize a subset of the Message Sequence Charts standards (Z120 and UML2.0) called Timed Message Sequence Charts (TMSC). TMSCs capture the execution of component-based software systems in an intuitive way and are amenable to formal timing analysis. We introduce a scalable heuristics-based technique to automatically infer TMSCs from execution traces. We demonstrate the effectiveness of our approach by automatically computing critical paths in the software of an industrial lithography scanner to identify timing bottlenecks.**

I. Introduction

The foundations of the component-based software paradigm were established in the mid-nineties [4], [5] as a successor of the object-oriented paradigm. Nowadays it is in widespread use to address the development, quality and maintenance challenges of large-scale software applications [27]. Example application domains include, but are not limited to, enterprise applications [20], consumer electronics [26], avionics applications [25] and cyber-physical systems such as interventional X-ray machines [15] and lithography equipment [18]. The component-based software paradigm advocates system decomposition in independent and reusable components encapsulating functionality and offering well-defined services on their external interfaces. Components are independent in the sense that they can be developed and maintained by different development teams, allow heterogeneity in terms of their programming languages, and can be independently deployed on heterogeneous and distributed execution platforms. Reusability is an important characteristic of software components, allowing their services to be reused by different applications.

The benefits of reduced lead time and enhanced quality and maintenance of component-based software engineering [27], do not come for free. Major challenges arise when it comes to performance engineering. Getting insight in the performance (response time, throughput, resource utilization) of an instantiated component is challenging, because it depends not only on the component implementation, but also on the context the component is deployed into [13]. Context factors include the functions (services) a component requires from other components, resource contention on the execution platform on which it is deployed, and the parameters used to invoke the services of the component [13]. Although many performance prediction and measurement approaches have been proposed in the past, none of the approaches has gained widespread industrial use due to still immature component performance models, limited tool support, and missing large-scale case study reports. Many companies still rely on personal experience and hands-on approaches to deal with performance problems instead of using engineering methods [13], [23].

This paper introduces a new measurement-based approach to get insights in timing bottlenecks of existing large-scale component-based software systems. We choose Message Sequence Charts (MSCs) to form the foundation of our approach. The reason is that Message Sequence Charts i) are standardized by the International Telecommunication Union [24] and by the UML2.0 community [22], ii) represent the communication behaviour between components in an intuitive manner, and iii) are accepted and widely used in industry to specify and analyze the communication behaviour of components [16].

The contributions of this paper are as follows:

1) We introduce and formalize Timed Message Sequence Charts (TMSCs) including a subset of the features in the spirit of the standards (Z120 [24] and UML2.0 [22]), in such a way that they are amenable to formal timing analysis. We extend the work of [1] with explicit absolute timing of events and hierarchical nesting of functions. We further incorporate a notion of timing constraints as in the work on timed High-level Message Sequence Charts [19].
2) We introduce a scalable heuristics-based technique to automatically infer TMSCs from measured execution traces.
3) We demonstrate the effectiveness of our approach by automatically computing and visualizing critical paths in the software of an industrial lithography scanner to identify timing bottlenecks.

The paper is organized as follows. Related work is discussed in Section II. In Section III Timed Message Sequence Charts are introduced and formalized. We further define the notion of critical path in TMSC to facilitate the case study. In Section IV we introduce a heuristics-based approach to automatically infer TMSCs from execution traces. The effectiveness of the approach is demonstrated in Section V, where it is applied to identify timing bottlenecks in industrial lithography scanners of ASML. In Section VI, conclusions are drawn and challenges for future work are identified.

II. Related work

We target models that capture timed executions of component-based software systems in an intuitive way and are amenable to formal timing analysis and optimization. This implies that, next to capturing absolute timing, models should be able to express nested function executions in components and remote function calls across components.

Recent advances in model inference and formal analysis for timed systems are based on task precedence graphs [9], [10]. Task precedence graphs naturally capture system executions and their visualization in Gantt charts and support a plethora of formal techniques including critical path analysis, timed model checking and scheduling. Nested function executions and remote function calls can, in principle, be encoded in task precedence graphs, but this leads to awkward models that are hard to understand and analyze.

Message Sequence Charts (MSCs) are a better fit to capture timed executions of component-based software systems. The UTI Z.120 [24] and UML2.0 [22] standards support the specification of timing, nesting function executions and remote functions calls, but these standards are too broad for our purpose and do not support the timing bottleneck analysis (such as critical path analysis) that we target. We therefore aim at a formalization of a concise subset of the standards that, next to bottleneck analysis, allows for automated model inference.

Most existing formalizations of MSCs are untimed, such as the formalization by Alur and Yannakakis [1], the formalization of Hierarchical Message Sequence Charts (HMSCs) by Mauw [21], and Live Sequence Charts (LSCs, caputuring liveness properties of events) by Damm [6]. These formalizations come with corresponding inference techniques. For instance in the work of Kumar [14] HMSCs are mined from a set of execution traces, and in the work of Lo [17] LSCs are inferred from traces.

A timed formalization of HMSCs is introduced by Lucas [19]. In this formalism, timing constraints between events are included to specify timing intervals in which events should take place with respect to their predecessor(s). The formalism does not support the specification of absolute timing of events and does not allow nested function executions to be expressed.

Various visualization methods exist to express the execution of activities (in our context, functions). A widely used method is the use of Gantt charts [3]. Gantt charts show activities executed on resource in time. Although modern Gantt charts also shown dependencies, they do not represent call stacks or nested execution of functions. An intuitive visualization of call stacks is introduced by Gregg in his work on visualizing CPU profiling data, known as flame graphs [8]. In such graphs, the nesting of function calls is explicitly shown in an intuitive way. Flame graphs however abstract away from time and the ordering of function calls that occur during the execution.

In the development of TMSCs that we introduce in this paper, we were strongly inspired by the work of Alur and Yannakakis [1] and have used this work as a basis for our formalization. In addition, we incorporate the concept of timing constraints as introduced in the work of Lucas [19]. As a basis for visualizing the nesting of functions, we are inspired by flame graphs [8]

III. Timed Message Sequence Charts

A. Informal introduction

We consider the run-time execution of a component-based software application and represent it as a TMSC. We start with an example to give a taste of the involved concepts and notations, after which we give a formal definition in the next subsection. The example is presented in Fig. 1, showing a simplification of the case study, discussed further in Section V. The picture shows on the vertical axis components C_1, C_2 and C_3, each of which executes functions and exchanges messages at different moments in time. The horizontal axis on the bottom of the figure displays the time during which functions are executed and at which times messages are sent and received. With every function execution we associate a start event, shown as an open circle and a finish event, shown as a closed circle. Events occurring on the same component are totally ordered, whereas events across components are partially ordered according to the message exchanges between components. Start and finish events have unique labels, but for reasons of brevity these labels are omitted from the picture. Function executions are shown as rectangles labeled with the names of the functions that are executed. For instance, on component C_1 function l is executed, starting at time 0 and finishing at time 19. Functions can be executed within the context of another function representing nested local function calls. In the figure, nesting is represented by positioning the function execution being invoked on top of the function that invokes it. In the figure, the execution of function l involves repeated executions of functions f_1, f_2, f_3 and f_4. A function that is nested in another function must always start no sooner and finish no later than the function it is nested in. This ensures that the call stack is well-formed. The 'bottom' of the call stack represents the function that is executed on behalf of another component. Such an execution must be finished before another call can be serviced. Hence, components can only serve one remote function call at a time.

Components can exchange messages. These are shown as directed arrows labeled with a message name. For instance, upon the start of function f_1, component C_1 sends message g_1^{call} to C_2, which signifies the start of the execution of function g_1. After completion of this execution, component

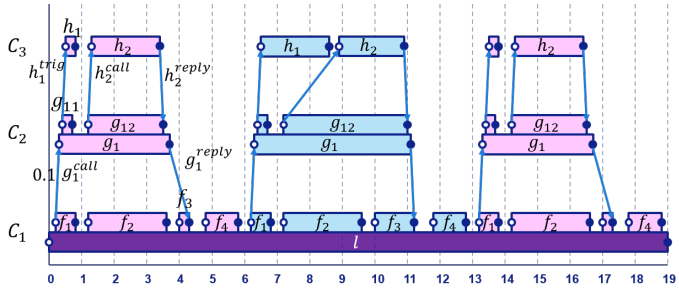


Fig. 1: A schematic view of a timed message sequence chart.

C_2 return reply message g_1^{reply} to C_1. This communication pattern encodes a remote function call. Arrows representing message exchanges always leave from (start or finish) events and terminate at (start or finish) events. Exchanging a message takes some amount of time. This time is indicated as a label of the message exchange. For instance, the leftmost message exchange between the start events of f_1 and g_1 is labelled 0.1. This indicates that the start of g_1 cannot occur earlier than the start time of f_1 plus 0.1 time units. In the figure, g_1 starts precisely 0.1 time units after the start of f_1, but in general this does not to be the case. The time label encodes a lower bound timing constraint.

Fig. 1 shows three different types of remote function calls that are supported by many component-based software frameworks. In this paper, we only discuss these patterns, albeit that in practise other patterns also exist. The first type is exemplified by message h_1^{trig} between components C_2 and C_3. This type of remote function call is called a *trigger*. The sender of such a trigger does not expect a reply and can thus continue its execution. The second type is a *blocking call*, an example of which is shown by messages h_2^{call} and h_2^{reply} between C_2 and C_3. A blocking call starts with a message between the start events of both functions and ends with a message between the end events running in the opposite direction. Here, the caller is waiting idly for a response from the callee. The third type is a *non-blocking call*, exemplified earlier as message g_1^{call} followed by message g_1^{reply}. Upon calling function g_1, C_1 continues executing functions f_2 and f_3, where f_3 is a call that idly waits until the reply g_1^{reply} from C_2 has been received. Notice that TMSCs do not support explicit concepts to express function call patterns. All these patterns can however be expressed in terms of the primitive formal concepts that are offered by TMSCs and will be explained in the next section.

B. Formal definition

We assume a set $\mathcal{E} = \mathcal{C} \times \mathcal{F} \times \mathbb{N}^+ \times \mathcal{S} \times \mathbb{R}_0^+$ of events, where, $\mathcal{C}$ is a set of component labels, $\mathcal{F}$ is a set of function labels, $\mathbb{N}^+$ is the set of positive natural numbers, $\mathcal{S} = \{\uparrow, \downarrow\}$, and $\mathbb{R}_0^+$ is the set of non-negative real numbers. If $e = (c, f, i, s, t) \in \mathcal{E}$ and $s = \uparrow$, then e denotes the start event of the i^{th} execution of function f on component c occurring at time t. Similarly, if $s = \downarrow$, then e denotes the finish event of the i^{th} execution of function f on component c occurring at time t. We further use $c(e)$, $f(e)$, $i(e)$, $s(e)$ and $t(e)$ to refer to c, f, i, s and t respectively. We let $\mathcal{M}$ denote the set of message labels that appear on the messages exchanged between components. We are now able to define the concept of TMSC.

Definition 1 (TMSC). *A TMSC is a three-tuple $(E, \rightarrow, m)$, where*

- *$E \subseteq \mathcal{E}$ is a set of events;*
- *$\rightarrow \subseteq E \times \mathbb{R}_0^+ \times E$ defines a timing constraint relation between events, and*
- *$m\colon \rightarrow \nmapsto \mathcal{M}$ is a partial function that maps the timing constraint relation to messages.*

For convenience we write $e_1 \xrightarrow{d} e_2$ to denote that $(e_1, d, e_2) \in \rightarrow$. The interpretation of timing constraint $e_1 \xrightarrow{d} e_2$ is that event e_2 will not occur earlier than d time units after the occurrence of e_1, i.e. $t(e_1) + d \leq t(e_2)$. TMSCs have to satisfy a number of conditions. Before these conditions are given, we define the concept of path.

Definition 2 (Path). *A sequence of events $e_0\ e_1\ \cdots\ e_n$ $(n \geq 1)$ of events is called a path if and only if $e_0 \xrightarrow{d_1} e_1 \xrightarrow{d_2} \ldots \xrightarrow{d_n} e_n$ for some $d_1, d_2, \cdots, d_n \in \mathbb{R}_0^+$. We write $e_0 \rightarrow^+ e_n$ if and only if there is a path $e_0\ e_1\ \cdots\ e_n$. If there is a path $e_0\ e_1\ \cdots\ e_n$ such that $c(e_0) = c(e_1) = \cdots = c(e_n) = c$, we write $e_0 \rightarrow_c^+ e_n$.*

Definition 3 (TMSC conditions). *A TMSC $(E, \rightarrow, m)$ satisfies each of the following conditions:*

1) *Relation $\rightarrow^+$ is a strict partial order on E and relation $\rightarrow_c^+$ is a strict total order on $E_c = \{e \in E \mid c(e) = c\}$. This implies that no cyclic paths exist and that between any two events occurring in a component a path exists that runs completely through that component.*
2) *If $(c, f, i, s, t) \in E$ and $(c, f, i, s, t') \in E$, then $t = t'$. This implies that an event is uniquely identified by its component name, function name, index (i) and event type ($\uparrow$ or $\downarrow$). For this reason event (c, f, i, s, t) will be conveniently referred to by expression $c.f(i)^s$.*
3) *If $c.f(i)^s \in E$, then for all $0 < j < i$ an event $c.f(j)^s \in E$ exists such that $c.f(j)^s \rightarrow_c^+ c.f(i)^s$. This implies that $c.f(i)^s$ refers to the start/finish event of the i^{th} execution of function f on component c.*
4) *$c.f(i)^{\uparrow} \in E$ iff $c.f(i)^{\downarrow} \in E$, and for all $c.f(i)^{\uparrow}, c.f(i)^{\downarrow} \in E$, $c.f(i)^{\uparrow} \rightarrow^+ c.f(i)^{\downarrow}$. This implies that a start event and consecutive end event always come in a pair, together representing a function execution.*
5) *For all $c.f(n)^{\uparrow}, c.f(n)^{\downarrow}, c.g(m)^{\uparrow}, c.g(m)^{\downarrow} \in E$, if $c.f(n)^{\uparrow} \rightarrow_c^+ c.g(m)^{\uparrow}$ and $c.g(m)^{\uparrow} \rightarrow_c^+ c.f(n)^{\downarrow}$, then $c.g(m)^{\downarrow} \rightarrow_c^+ c.f(n)^{\downarrow}$. This implies that functions are nested in a proper way, yielding well-formed call stacks.*
6) *For all $e_1, e_2 \in E$ and $d \in \mathbb{R}_0^+$, if $e_1 \xrightarrow{d} e_2$ then $t(e_1) + d \leq t(e_2)$. This implies that all timing constraints are met.*

The example shown in Fig. 1 is consistent with the definitions 1 and 3, albeit not all concepts are visualized in the

picture. Event tuples are not explicitly shown, but these can be conveniently referred to by the corresponding expressions. For instance the leftmost event on component C_1 is referred to by expression $C_1.l(1)^{\uparrow}$. To prevent clutter, the timing constraint relations $\rightarrow^{+}_{C_1}$, $\rightarrow^{+}_{C_2}$ and $\rightarrow^{+}_{C_3}$ are omitted from the picture. For instance, the time constraint $C_1.l(1)^{\uparrow} \xrightarrow{0.2} C_1.f_1(1)^{\uparrow}$ exists, which models the part of the execution of function l starting from event $C_1.l(1)^{\uparrow}$ and ending at event $C_1.f_1(1)^{\uparrow}$ and having execution time 0.2. As another example, time constraint $C_2.g_1(1)^{\downarrow} \xrightarrow{0} C_2.g_1(2)^{\uparrow}$ exists. This constraint is required because of condition 1 in Definition 3. The time value of this constraint is 0, indicating that event $C_2.g_1(2)^{\uparrow}$ could start immediately after $C_2.g_1(1)^{\downarrow}$. According to condition 6 in Definition 3, any other value between 0 and $t(C_2.g_1(2)^{\uparrow}) - t(C_2.g_1(1)^{\downarrow})$ would also be possible. The fact that event $C_2.g_1(2)^{\uparrow}$ does not start immediately is due to time constraint $C_1.f_1(2)^{\uparrow} \xrightarrow{0.1} C_2.g_1(2)^{\uparrow}$. Notice that both messages between components and the ordering of events on components are captured by the same time constraint concept. Timing constraints that capture message exchanges between components can be given a message name.

C. Critical-path analysis

In Section V we illustrate the effective use of TMSCs by performing critical path analysis in the software execution of an industrial lithography scanner. In this section, we will therefore define the notion of critical path on TMSCs.

Any two events in a TMSC can be related by a timing constraint. For instance, constraint $e_1 \xrightarrow{d} e_2$ implies that event e_2 cannot happen earlier than d units of time after the occurrence of e_1. In case event e_2 occurs precisely at time $t(e_1) + d$, the constraint is critical in the sense that a slight increase in $t(e_1)$ or d would render the constraint to be unsatisfied. Only if e_2 would be shifted in time the constraint could be satisfied again. Such a shift would not be necessary in case $t(e_2) > t(e_1) + d$. Conversely, there where $t(e_1) + d$ is smaller than $t(e_2)$, a similar increase in $t(e_1)$ or d would not lead to a shift in time for $t(e_2)$. Therefore, in such a case $e_1 \xrightarrow{d} e_2$ is not critical.

Definition 4 (Critical timing constraint)**.** *$e_1 \xrightarrow{d} e_2$ is critical iff $t(e_1) + d = t(e_2)$.*

Since a slight increase in $t(e_1)$ or d implies an increase in $t(e_2)$ for a critical timing constraint, this increase in $t(e_2)$ propagates across any critical timing constraint starting in e_2. Based on this intuition, we define the notions of critical path and maximal critical path.

Definition 5 ((Maximal) Critical path)**.** *A path $e_0 e_1 \cdots e_n$ is a critical path iff for each i such that $0 \leq i < n$, there exists a $d_i \in \mathbb{R}_0^+$ such that $e_i \xrightarrow{d_i} e_{i+1}$ is a critical timing constraint. A critical path $e_0 e_1 \cdots e_n$ is called a maximal critical path iff no $e \in E$ and $d \in \mathbb{R}_0^+$ exist such that $e \xrightarrow{d} e_0$ is a critical timing constraint.*

Algorithm 1 critical_path($(E, \rightarrow, m)$, e)

Require: TMSC $(E, \rightarrow, m)$, $e \in E$
Ensure: Set $\Pi(e)$ containing all critical timing constraints that appear in any maximal critical path towards e

```
1:  initialization: Π(e) ← ∅; S ← ⟨⟩; E' ← ∅
2:  S.push(e)
3:  while S not empty do
4:     e' = Q.pop()
5:     for all e'' ∈ E such that e'' -d-> e' do
6:        if e'' ∉ E' and t(e'') + d = t(e') then
7:           E' ← {e''} ∪ E'
8:           S.push(e'')
9:           Π(e) ← {e'' -d-> e'} ∪ Π(e)
10:       end if
11:    end for
12: end while
13: return Π(e)
```

We are interested in all maximal critical path(s) towards (that finish in) a specific event. In general this set can grow exponentially in the size of the TMSC. Marking the events and timing constraints that are part of any of the maximal critical paths can however be carried out in linear time by traversing all incoming critical timing constraints starting from that specific event. Algorithm 1 illustrates how to compute the set of critical timing constraints that are part of any maximal critical path towards a given event e. A depth first search is used to traverse through all incoming critical timing constraints. We start by initializing a stack and push event e onto the stack (lines 1 and 2). While there are events on the stack (line 3), we take the element on the top of the stack and check all its incoming timing constraints (lines 4 and 5). For all such timing constraints which are critical, a source event is pushed on the stack and the timing constraint is added to the output set. To prevent visiting the same event multiple times, we only add events to the stack we haven't seen before. (lines 6 through 9). The complexity of the algorithm is linear in the total number of events in the maximal critical path(s) times plus the timing constraints towards any such event.

A TMSC contains information about time stamps of events, while this information is not available for classical critical path algorithms based on task dependency graphs [12]. These algorithms compute the earliest start time (EST) and latest finish time for each task (LFT), as well as the slack for each task in the graph. Therefore, the time complexity of classical critical path algorithms is linear in the total number of edges and vertices of the task dependency graph. The critical path algorithm on TMSCs is also linear, but in the total number of edges (timing constraints) and vertices (events) that actually appear in the critical path(s), and with a worst-case complexity equal to the complexity of classical critical path algorithms in the case that all timing constraints are critical.

We illustrate critical paths on TMSCs using the example from Fig. 1. Fig. 2 illustrates the critical path towards event

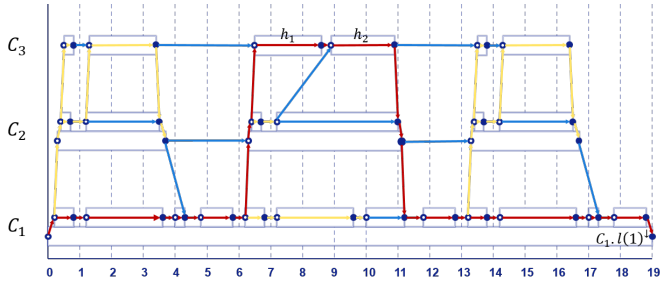


Fig. 2: The timed message sequence chart including additional timing information, marked with the critical path.

$C_1.l(1)^{\downarrow}$ on component C_1. The figure shows all the timing constraints that were omitted from the original Fig. 1. Each yellow and red coloured constraint $e_1 \xrightarrow{d} e_2$ is assumed to be critical (i.e. $t(e_1) + d = t(e_2)$). Each blue coloured timing constraint is assumed to be not critical (i.e. $t(e_1) + d < t(e_2)$). The way the timing constraints are obtained from execution traces is discussed in Section IV.

The rightmost event $C_1.l(1)^{\downarrow}$ on component C_1, marks the end of the function l. The time constraints shown in red correspond to the timing constraints in the (single) maximal critical path towards $C_1.l(1)^{\downarrow}$. In the first and third repetition of the functions f_1, f_2, f_3 and f_4, the critical path runs completely through the function executions executed by component C_1. During the second repetition, however, the critical path runs through function h_1 on component C_3 which lasts longer in this repetition (as is also shown in Fig. 1). Notice that the critical timing constraint between the executions of functions h_1 and h_2 (i.e. $C_3.h_1(2)^{\downarrow} \xrightarrow{0.3} C_3.h_2(2)^{\uparrow}$) is not a timing constraint imposed by the application, but rather a timing constraint resulting from the fact that each component can only execute one function at a time. In this case both the executions of h_1 and h_2 contend for resource (component) C_3. This type of contention is very common in component-based software systems due to the nature of reusability of components. We will see this type of contention in our case study on a lithography machine in Section V.

IV. Timed Message Sequence Chart Inference

In the previous section we have established the concept of TMSC. We are now ready to discuss the steps necessary to automatically infer TMSCs from execution traces. We first describe in Section IV-A how to obtain execution traces and interpret them to infer a preliminary TMSC. Here we assume the execution traces to be such that the resulting TMSC satisfies Definitions 1 and 3. In Section IV-B we explain how to heuristically deal with missing events in the trace due to windowing, and in Section IV-C heuristics are given to deal with untraced components.

Traces do not contain explicit information about the timing values of timing constraints. To deal with this missing information, all timing constraints are initially assumed to be critical in the preliminary TMSC. To obtain more realistic timing constraints, we define heuristics in Section IV-D which are based on the remote function call patterns described in Section III-A to refine the inferred TMSC.

A. Execution traces

In this section, we first identify the required information to be traced to infer a preliminary TMSC. Events are generated by instrumenting the software code at the start and end of every function call. This additional code is used to generate a trace of records. Each record contains the following information:

- a **time stamp** corresponding to the time that the event occurred;
- a **component label** corresponding to the component that generated the event;
- a **function label** corresponding to the function associated with the event;
- a **start/finish label** corresponding to the start or end of the function associated with the event;
- optionally, a **message label**, an **identifier** (unique to a message exchange) and a **type** (send or receive).

We assume that the ordering of records in the trace respects the ordering in which the records were generated. This means that the ordering of starting and finishing function executions per component is preserved in the trace. Further, a record that encodes the sending of message always precedes the record that encodes the corresponding message reception. In addition, we assume the time stamps of records to be non-decreasing.

Following Definition 1, the preliminary TMSC is of the form $(E, \rightarrow, m)$. For each record in the trace we construct a tuple (c, f, i, s, t) and add it to the set E of events. The values for c, f, s and t are derived directly from the record. The index i is derived from the ordering in which the records occur in the trace.

Relation $\rightarrow$ is derived in two steps. First the timing constraints between events occurring on the same component are related. For each pair of subsequent records related to the same component, a timing constraint between the corresponding

#	Time	Comp.	Func.	Start/Finish	Message	ID	Type
1	0.0	C_1	l	start			
2	0.2	C_1	f_1	start	g_1^{call}	$id1$	send
3	0.3	C_2	g_1	start	g_1^{call}	$id1$	receive
4	0.4	C_2	g_{11}	start	h_1^{trig}	$id2$	send
5	0.5	C_3	h_1	start	h_1^{trig}	$id2$	receive
6	0.7	C_2	g_{11}	finish			
7	0.8	C_3	h_1	finish			
8	0.8	C_1	f_1	finish			
9	1.2	C_2	g_{12}	start	h_2^{call}	$id3$	send
⋮							
56	19.0	C_1	l	finish			

Fig. 3: An excerpt of the trace in tabular format corresponding to Fig. 1.

events, say e_1 and e_2, is added. This timing constraint is of the form $e_1 \xrightarrow{t(e_2)-t(e_1)} e_2$. Notice that this timing constraint satisfies condition 6 of Definition 3, and furthermore is critical (see Definition 4). Second the timing constraints that are imposed by message exchanges are derived. For each pair of records in the execution traces with the same identifier, we add a timing constraints from the send event, say e_3, to the receive event, say e_4, namely $e_3 \xrightarrow{t(e_4)-t(e_3)} e_4$. It is assumed that records with the same identifier also carry the same message label. This message label is added to the newly added constraint through m.

As an example, we show in Fig. 3 in a tabular form a trace of records corresponding to the TMSC of Fig. 1. The trace records correspond to the first nine and the last event shown in Fig. 1. For instance, the first record in the trace corresponds to the start of the execution of function l on component C_1 at time 0.

Fig. 4a shows the events inferred from the trace. For example, event (C_1, l, 1, $\uparrow$, 0) corresponds to the first record in Fig. 3. Fig. 4b shows the timing constraints inferred from the trace of records, where events are referred to by their expressions. The first timing constraint that is inferred from the trace is $C_1.l(1)^{\uparrow} \xrightarrow{0.2} C_1.f_1(1)^{\uparrow}$. As described before, the time label 0.2 is computed as the time difference between the occurrences of both events, i.e. $t(C_1.f_1(1)^{\uparrow}) - t(C_1.l(1)^{\uparrow})$. The second timing constraint in Fig. 4b is associated with the message exchange of message g_1^{call} between components C_1 and C_2. Therefore, in addition to the timing constraint $C_1.f_1(1)^{\uparrow} \xrightarrow{0.1} C_2.g_1(1)^{\uparrow}$, we also set the message label $m(C_1.f_1(1)^{\uparrow} \xrightarrow{0.1} C_2.g_1(1)^{\uparrow})$ as g_1^{call}.

B. Heuristics to deal with missing events

Trace data may be incomplete in the sense that the trace may only contain information corresponding to a particular execution window in time. This implies that it may contain start event records without the corresponding finish event records, or vice versa. A preliminary TMSC inferred from such a trace does not property 4 of Definition 3. This can be remedied in two ways. The first approach is to discard records for which the counterparts are missing. Discarding such records leads to a valid preliminary TMSC. Another approach is to add missing records. For start event records that miss corresponding finish event records, these finish event records are included at the end of the trace in an order that reverses the order of the corresponding start event records. The time stamp of the additional records is set to the time stamp of the final record in the trace. Likewise, start event records are included at the start of the trace for all finish event records (that miss their counterpart), again in reversed order. The time stamp of the additional records is set to the time stamp of the first record in the trace. Adding records in this manner results in a valid preliminary TMSC.

(C_1, l, 1, $\uparrow$, 0);
(C_1, f_1, 1, $\uparrow$, 0.2);
(C_2, g_1, 1, $\uparrow$, 0.3);
(C_2, g_{11}, 1, $\uparrow$, 0.4);
(C_3, h_1, 1, $\uparrow$, 0.5);
(C_2, g_{11}, 1, $\downarrow$, 0.7);
(C_3, h_1, 1, $\downarrow$, 0.8);
(C_1, f_1, l, 1, $\downarrow$, 0.8);
(C_2, g_{12}, 1, $\uparrow$, 1.2);
$\vdots$
(C_1, l, 1, $\downarrow$, 19)

(a)

$C_1.l(1)^{\uparrow} \xrightarrow{0.2} C_1.f_1(1)^{\uparrow}$;
$C_1.f_1(1)^{\uparrow} \xrightarrow{0.1} C_2.g_1(1)^{\uparrow}$;
$C_1.f_1(1)^{\uparrow} \xrightarrow{0.6} C_1.f(1)^{\downarrow}$;
$C_2.g_1(1)^{\uparrow} \xrightarrow{0.1} C_2.g_{11}(1)^{\uparrow}$;
$C_2.g_{11}(1)^{\uparrow} \xrightarrow{0.1} C_3.h_1(1)^{\uparrow}$;
$C_2.g_{11}(1)^{\uparrow} \xrightarrow{0.3} C_2.g_{11}(1)^{\downarrow}$;
$C_3.h_1(1)^{\uparrow} \xrightarrow{0.3} C_3.h_1(1)^{\downarrow}$;
$\vdots$
$C_1.f_1(1)^{\downarrow} \xrightarrow{0.2} C_1.l(1)^{\downarrow}$

(b)

Fig. 4: The events (a) and timing constraints (b) inferred from the trace of Fig. 3.

C. Heuristics to deal with untraced components

It may not be possible to trace every component in a software application in practise, e.g. because available platforms do not offer appropriate tracing facilities (such as LTTng [7]). This may result in traces in which messages are not paired, i.e. in which only the sender or only the receiver of a message is recorded. There are two methods to deal with untraced components. The first approach is to ignore records with missing counterparts, and discard the message exchange. Doing so will ensure the validity of the conditions in Definition 3. The second approach is to insert a record corresponding to the missing record, giving it the same time stamp as its counterpart, and assign it a placeholder component label and function name based on the message label. A schematic view of this case in the form of a TMSC is given in Fig. 5. Here C_{g_1} represents an untraced component inferred from the g_1^{call} and g_1^{reply} messages. C_{g_1} executes the function g_1 corresponding to the message exchanges which are depicted by the arrows. Such a TMSC satisfies all conditions of Definition 3.

The decision to use one approach over the other depends on the assumptions one wants to make about the untraced component. The difference between the approaches become apparent in case we would like to perform timing variability analysis. In the first approach the untraced components are assumed to be non-critical for the timing behaviour of the traced components, while in the second approach they are assumed to be critical. Assume for instance that the first execution of function g_1 in Fig. 5 would take more time. In the

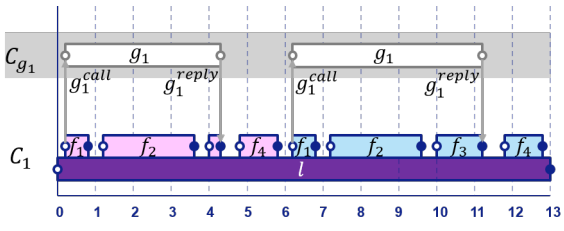
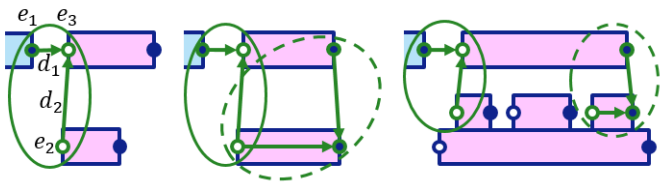


Fig. 5: The first two iterations from the example in Fig. 1, assuming C_2 and C_3 are not traced.



(a) (b) (c)

Fig. 6: Trigger (a), blocking call (b) and non-blocking call (c) patterns.

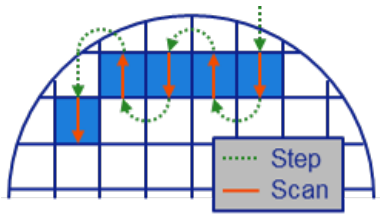


Fig. 7: A schematic overview of a wafer being exposed. Redrawn from Figure 15 in [2].

first approach, the executions on C_1 would remain the same, while in the second approach the first execution of f_4 together with all its successors would shift in time.

D. Heuristics for message exchanges

In Section III-A we discuss three distinct remote function call patterns that are common in component-based software systems. These patterns concern the trigger, blocking call, and non-blocking call patterns, and are depicted in Fig. 6. Events with two incoming timing constraints together with their source events are emphasized in (solid and dashed) ellipses. Recall that the values on the timing constraints are not measured in our approach. The reason is that these are very difficult to obtain in practise. In the preliminary TMSC the incoming timing constraints are determined to be critical. This is not realistic, since we would expect only one of these constraints to be critical. Therefore, we use a heuristic approach to determine more realistic constraints. This heuristic approach has turned out to work well in our industrial case study, see Section V.

To explain how to determine the critically in a more realistic way, we first consider the case of the trigger pattern in Fig. 6a. In this pattern we have two timing constraints $e_1 \xrightarrow{d_1} e_3$ and $e_2 \xrightarrow{d_2} e_3$ ending in e_3. Value d_1 represents the setup time, i.e. the time between finishing the function execution (denoted by event e_1) and the earliest start of the subsequent execution. Value d_2 represents the communication time, i.e. the time between sending a message (at the occurrence of event e_2) and the reception thereof. In our approach d_1 and d_2 are initially set to $t(e_3) - t(e_1)$ and $t(e_3) - t(e_2)$ respectively. Assume that in reality, these timing values are given by d_s and d_c respectively. To determine the criticality of $e_1 \xrightarrow{d_1} e_3$ and $e_2 \xrightarrow{d_2} e_3$, we assume a bound $\epsilon \geq 0$ such that $d_s, d_c \leq \epsilon$. We then adjust the values d_1 and d_2 according to the following assignments:

$$d_1 := min(d_1, max(d_2, \epsilon)), \tag{1}$$

$$d_2 := min(d_2, max(d_1, \epsilon)). \tag{2}$$

From the new value for d_1 it follows that the timing constraint $e_1 \xrightarrow{d_1} e_3$ is critical if and only if $t(e_3) - t(e_1) \leq t(e_3) - t(e_2)$ or $t(e_3) - t(e_1) \leq \epsilon$. Similarly, from the new value for d_2, the timing constraint $e_2 \xrightarrow{d_2} e_3$ is critical if and only if $t(e_3) - t(e_2) \leq t(e_3) - t(e_1)$ or $t(e_3) - t(e_2) \leq \epsilon$. In other words, the timing constraint with the smaller (or equal) difference in time between source event and target event is always critical, and both are critical if both values are smaller than or equal to the bound ϵ on d_s and d_c. Notice that for the case where $\epsilon = 0$, d_s and d_c are assumed to be negligible compared to the function execution times and criticality depends only on comparing the values d_1 and d_2.

Now that we have considered the case of the trigger pattern, we will consider the other patterns in Fig. 6. With respect to the blocking (b) and non-blocking (c) patterns we notice that the timing constraints depicted in the solid ellipses are identical to those of the trigger pattern, and can therefore be treated in the same way. In both the blocking and non-blocking patterns another event exists with two incoming timing constraints. These are depicted in the dashed ellipses. In these cases one of the timing constraints still involves a communication time, while the other timing constraint concerns the waiting time for a message to arrive. Determining the criticality for these cases follow the exact same line of reasoning as that of the trigger pattern. Computing the timing values of the constraints is therefore performed in the same way as described above for this pattern. Notice that for the blocking call (case (b)), this leads to the message exchange containing the reply message to always be critical.

V. Case-study

We perform a case study in the application domain of lithography. Lithography scanners are highly complex cyber-physical systems used to manufacture integrated circuits [2], [11]. In the next section we first give an overview of lithography scanners and identify the challenges to further improve these scanners, followed by applying the techniques introduced in this paper on an execution trace from an ASML test bench.

A. Lithography scanners

Lithography scanners use an optical system to project an image of a pattern on a quartz plate, called the reticle, onto a photosensitive layer on a substrate, called the wafer. Since one wafer can contain many ICs, typically 100 or more, the wafer needs to be repositioned from exposure to exposure. Exposures take place during scanning motions of the wafer, as illustrated in Fig. 7 (taken from [2]). A wafer is divided in image fields, which are scanned one by one according to a Step and Scan pattern. Each physical Step and Scan action involves a motion

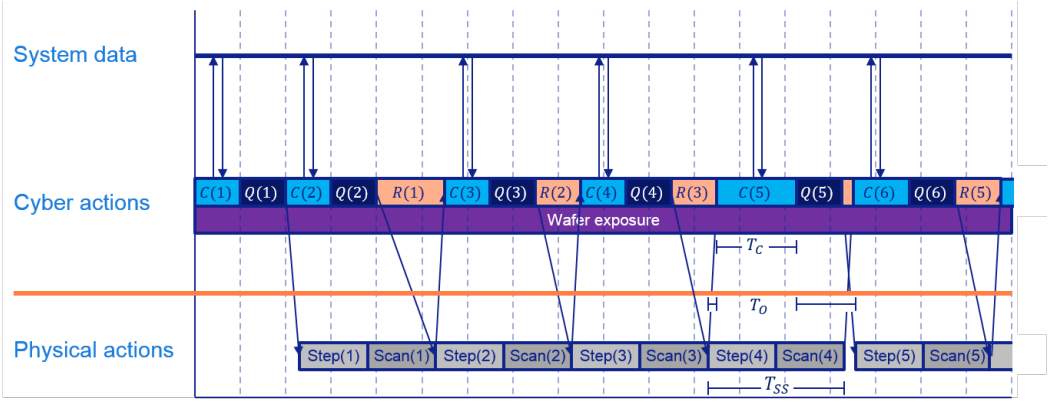


Fig. 8: A conceptual view of the wafer exposure loop with a queue depth of 2.

path of the wafer defined in terms of a setpoint profile which has to be tracked with nanometer positional accuracy. Due to physical disturbances, e.g. temperature perturbations, feedback setpoint adjustments are computed based on system-wide sensor and data input.

Fig. 8 shows an conceptual view of the actions required to expose a wafer. The physical actions are the Step and Scan actions described earlier. The cyber actions are executed as part of a control loop called *Wafer exposure*. The loop starts by executing action $C(1)$ to compute the setpoint adjustment for first Step and Scan actions. These results are then queued by $Q(1)$ to request the execution of actions $Step(1)$ respectively $Scan(1)$. Action $C(1)$ involves data retrieval from various sources, such as sensors and (shared) data stores storing system state. Then $C(2)$ and $Q(2)$ are executed to compute and queue the adjusted setpoints for $Step(2)$ and $Scan(2)$. Before $C(3)$ and $Q(3)$ are executed, the result $R(1)$ of $Scan(1)$ is first retrieved and, in general, before $C(i)$ and $Q(i)$ is executed, the result $R(i-2)$ of $Scan(i-2)$ is first retrieved ($3 \leq i \leq N$, where N is the total number of Scan actions to be performed). Hence, in this example, the control loop has a queue depth of 2, implying that at most two requests to perform a Step/Scan action are outstanding at the same time.

To obtain optimal system throughput, all Step and Scan actions must be executed in a concatenated fashion. It is not hard to see that this can only be achieved if the queue depth is at least two. In addition, a timing constraint on the execution of the cyber actions must hold. To formulate this timing constraint we let T_{SS} denote the best-case execution time of a Step/Scan action, see Fig. 8. We further let T_O denote the worst-case communication and queuing overhead, i.e. the total time to retrieve the results, to perform a queue action and to push the setpoints adjustments to the physics. Finally, we let $T_C(i)$ denote the execution time of $C(i)$. It is not hard to see that the Step and Scan actions are all concatenated if for all i ($2 \leq i \leq N$):

$$T_C(i) \leq T_{SS} - T_O. \tag{3}$$

A violation of this timing requirement, may result in a Scan action that is not immediately followed by a next Step action. Fig. 8 shows an example of this in which actions $Scan(4)$ and $Step(5)$ are not concatenated.

Due to resource contention $T_C(i)$ can vary for different values of i, even to such extent that the timing requirement is occasionally violated. To make the system robust against such sporadic deadline violations, the queue depth can be increased. An increase of this queue depth, however, enlarges the time difference between the moment of taking a snapshot of the system state and actuating the system, which decreases accuracy. Hence a delicate trade-off exists between system throughput and system accuracy. Determining the bottlenecks that cause sporadic deadline violations thus allows the system accuracy to be improved without sacrificing throughput. In the next section, we show how critical path analysis on automatically inferred TMSCs supports pinpointing these bottlenecks in a systematic way.

B. Inference and analysis results

We implemented the code instrumentation and inference techniques introduced in this paper in a proprietary tool. We limit ourselves to the system controller which runs on a Linux-based multi-core system and use LTTng [7] to efficiently retrieve execution traces from ASML test benches. The test involves the exposure of 70 wafers and involves 396 software components. The TMSC contains 48.7 million events, 24.3 million function execution, and 60.4 million timing constraints of which 11.7 million involves message exchanges. In the inference of the TMSC we chose $\epsilon = 0$ (see Section IV-D), since the communication, setup and waiting times are negligible (and thus almost the same) compared to the execution times of functions. Obviously, the inferred TMSC is too large to be visualized completely. Therefore we focus on a few setpoint adjustment computations and some of the involved software components.

An anonymized screenshot of the tool is shown in Fig. 9. At the bottom of the TMSC, the compute, queue and retrieve actions are annotated using the same colouring as in Fig. 8. C_1 denotes the component executing the expose control loop function. The figure shows six setpoint adjustment computations. The duration of the fifth computation exceeds the deadline,

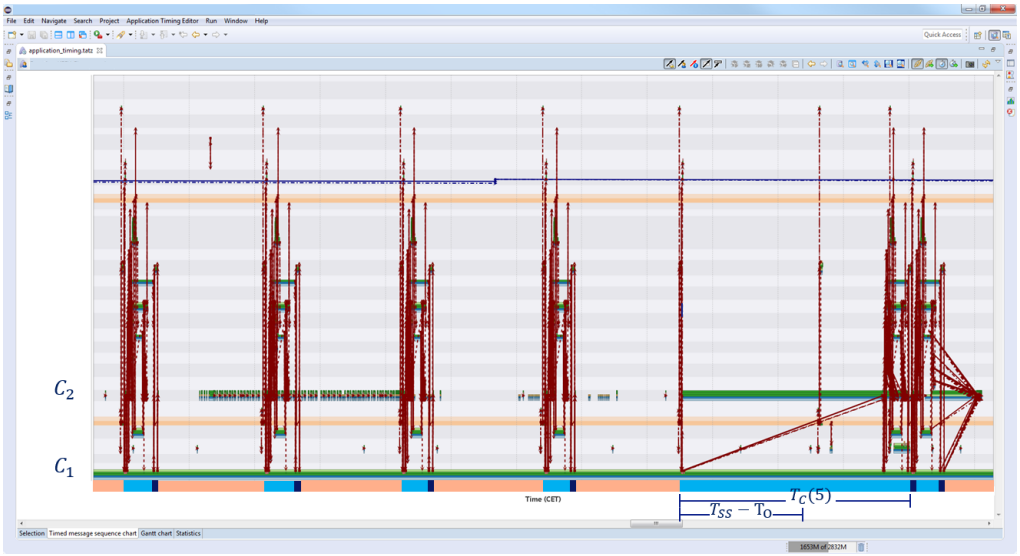


Fig. 9: A TMSC inferred from an execution trace of a testbench, showing a window of six setpoint adjustment computations.

i.e. $T_C(5) > T_{SS} - T_O$. Here $T_C(5)$ denotes the execution time of the fifth compute action ($C(5)$) relative to the start of the screenshot. Notice that this deadline violation results in the next setpoint computation to be executed immediately after queue action $Q(5)$ was performed.

To understand why the deadline of $C(5)$ was missed, we compute the critical path towards the finish event of $C(5)$ (on component C_1). The resulting critical path is shown as a red line in Fig. 10, starting from the start event of $C(5)$ (also on component C_1). Just after the start of $C(5)$, C_1 sends a trigger to C_2, followed by a blocking call to C_2. The is also is illustrated schematically in the TMSC embedded in the picture, which is similar as the running example of Fig. 1. Notice that the tail of the execution of $C(5)$ is not shown in this embedded TMSC. It turns out that the execution of the triggered function by C_2 takes an exceptionally large amount of time compared to the executions in the other setpoint adjustment computations. As a result, the execution of the blocking call is pushed forward in time delaying the finish event of the blocking call in C_1. This blocking call is therefore delayed due to component contention.

Notice that for reasons of confidentiality and clarity we have only been able to show the anonimized TMSC and critical path of a tiny part of the execution of the wafer exposure application. The approach, however, allows us to efficiently compute the critical path for any part of the measured application run and visualize the results.

VI. Conclusions

We presented a measurement-based approach based on Timed Message Sequence Charts (TMSCs), to analyze and visualize the timed behaviour of large-scale component-based software systems. A heuristic approach to automatically infer TMSCs from execution traces is presented together with a technique to compute critical paths in TMSCs.

We validated the technique on an industrial-sized wafer scanner application of ASML. The inferred TMSC involved 396 software components executing 24 million functions and exchanging 12 million messages. Discovering potential timing bottlenecks in a wafer scanner application has turned out to be very complex in practice. The TMSC approach presented in this paper is a step forward towards finding bottlenecks more efficiently. As such, the approach is very well-received by the developers community of ASML, not least because of its scalability, level of automation, and intuitive nature. The approach allows them to significantly reduce the time to narrow down potential timing bottlenecks.

The presented work establishes a sound foundation for more advanced analysis techniques. An example of such a technique is scalable robust model checking, i.e. timed model checking in the presence of timing variations in the application. In addition, we are interested in automated bottleneck analysis in case of timed property violations. Another topic of interest concerns automated mapping and scheduling techniques that guarantee the satisfaction of robust timing requirements.

References

[1] Rajeev Alur and Mihalis Yannakakis. Model checking of message sequence charts. In *International Conference on Concurrency Theory*, pages 114–129. Springer, 1999.

[2] Hans Butler. Position control in lithographic equipment [applications of control]. *IEEE control systems*, 31(5):28–47, 2011.

[3] Wallace Clark. *The Gantt chart: A working tool of management*. Ronald Press, 1922.

[4] Szyperski Clemens, Gruntz Dominik, and Murer Stephan. Component software: Beyond object-oriented programming. *Addison-Westley: Boston, MA, USA*, 1998.

[5] WT Council and GT Heineman. Component-based software engineering putting the pieces together. *Addison Weysley*, 2001.

[6] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Formal methods in system design*, 19(1):45–80, 2001.

[7] Mathieu Desnoyers and Michel R Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *Proceedings of the OLS (Ottawa Linux Symposium)*, volume 2006, pages 209–224. Linux Symposium, 2006.

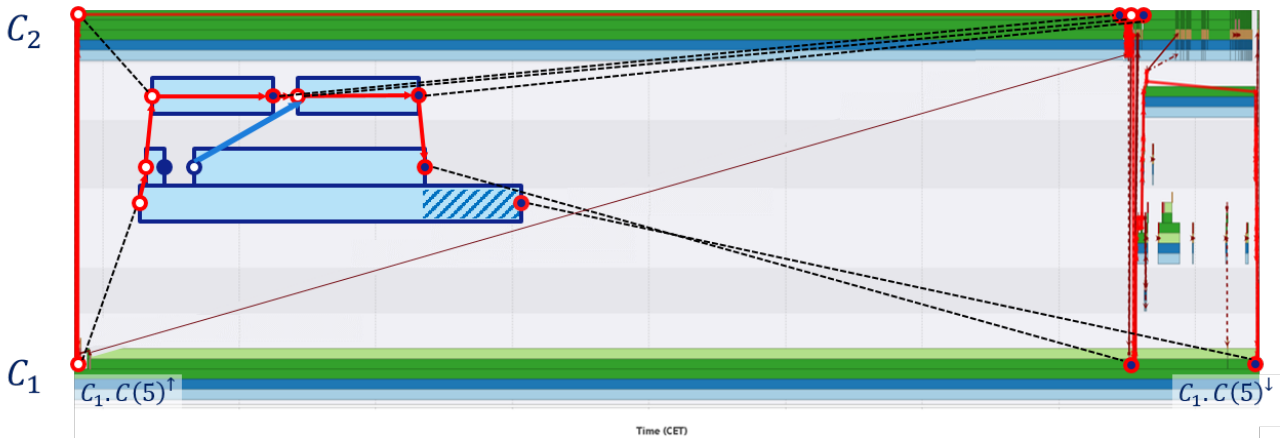


Fig. 10: A more detailed view of Fig. 9, showing the critical path for the fifth setpoint adjustment computation.

[8] Brendan Gregg. Flame graphs, 2015.

[9] Martijn Hendriks, Jacques Verriet, Twan Basten, Bart Theelen, Marco Brassé, and Lou Somers. Analyzing execution traces: critical-path analysis and distance analysis. *International Journal on Software Tools for Technology Transfer*, 19(4):487–510, 2017.

[10] Oleg Iegorov and Sebastian Fischmeister. Mining task precedence graphs from real-time embedded system traces. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 251–260. IEEE, 2018.

[11] Ruben Jonk, Jeroen Voeten, Marc Geilen, Twan Basten, and Ramon Schiffelers. Timing prediction for service-based applications mapped on linux-based multi-core platforms. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 130–139. IEEE, 2018.

[12] James E Kelley Jr and Morgan R Walker. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 160–173. ACM, 1959.

[13] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, 2010.

[14] Sandeep Kumar, Siau-Cheng Khoo, Abhik Roychoudhury, and David Lo. Mining message sequence graphs. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 91–100. IEEE, 2011.

[15] Ivan Kurtev, Mathijs Schuts, Jozef Hooman, and Dirk-Jan Swagerman. Integrating interface modeling and analysis in an industrial setting. In *MODELSWARD*, pages 345–352, 2017.

[16] Alexander A Letichevsky, Julia V Kapitonova, VP Kotlyarov, Vladislav A Volkov, and T Weigert. Semantics of message sequence charts. In *International SDL Forum*, pages 117–132. Springer, 2005.

[17] David Lo, Shahar Maoz, and Siau-Cheng Khoo. Mining modal scenario-based specifications from execution traces of reactive systems. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 465–468. ACM, 2007.

[18] Robin Loose, Bram van der Sanden, Michel Reniers, and Ramon Schiffelers. Component-wise supervisory controller synthesis in a client/server architecture. *IFAC-PapersOnLine*, 51(7):381–387, 2018.

[19] Philipp Lucas. Timed semantics of message sequence charts based on timed automata. *Electronic Notes in Theoretical Computer Science*, 65(6):160–179, 2002.

[20] Vlada Matena, Beth Stearns, and Linda DeMichiel. *Applying enterprise JavaBeans: component-based development for the J2EE platform*. Pearson Education, 2003.

[21] Sjouke Mauw and Michel A Reniers. High-level message sequence charts. In *SDL'97: Time for Testing*, pages 291–306. Elsevier, 1997.

[22] Zoltán Micskei and Hélène Waeselynck. Uml 2.0 sequence diagrams' semantics. *LAAS technical report no. 08389*, 37, 2008.

[23] Stephen Olatunde Olabiyisi, Elijah Olusayo Omidiora, Faith-Michael E Uzoka, Mbarika Victor, and Boluwaji A Akinnuwesi. A survey of performance evaluation models for distributed software system architecture. In *World Congress on Engineering 2012. July 4-6, 2012. London, UK.*, volume 2186, pages 35–43. International Association of Engineers, 2010.

[24] ITUT Recommendation. Z120. *Message Sequence Charts*, 2004.

[25] David C Sharp. Reducing avionics software cost through component based product line development. In *Digital Avionics Systems Conference, 1998. Proceedings., 17th DASC. The AIAA/IEEE/SAE*, volume 2, pages G32–1. IEEE, 1998.

[26] Rob Van Ommering, Frank Van Der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[27] Padmal Vitharana. Risks and challenges of component-based software development. *Communications of the ACM*, 46(8):67–72, 2003.