Automated Derivation of Application Workload Models for Design Space Exploration of Industrial Distributed Cyber-Physical Systems

Faezeh Sadat Saadatmand*, Todor Stefanov*, Ignacio González Alonso†,
Andy D. Pimentel‡, Benny Akesson§‡, Marius Herget‡, and Martin Bor‡

*Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands †ASML B.V., The Netherlands
‡Informatics Institute, University of Amsterdam, The Netherlands §TNO-ESI, The Netherlands

***Abstract*—Manufacturing companies of complex distributed cyber-physical systems (dCPS) are encountering challenges with respect to designing their next-generation machines. They need efficient Design Space Exploration (DSE) techniques to evaluate possible design decisions and their consequences on non-functional aspects of the systems. To enable scalable and efficient DSE of complex dCPS, it is essential to have abstract and coarse-grained models that are both accurate and capable of capturing dynamic application workloads.**

This paper addresses the scientific challenge of defining and automatically deriving an application workload model for DSE of complex dCPS. Our approach leverages trace analysis to derive a dynamic workload model that accurately represents computation and communication actions within an application in a timing agnostic manner. We demonstrate the effectiveness of our approach by applying it to the ASML Twinscan lithography machine, which is a real-world complex dCPS. The results of our study demonstrate an accuracy of over 90% in capturing the real application workloads.

***Index Terms*—Distributed Cyber-Physical systems, Application Workload Model, Design Space Exploration**

I. Introduction

Cyber-Physical Systems (CPS) comprise one of the largest information-technology sectors worldwide, which is a crucial driver for innovation in various industries, including health, industrial automation, robotics, avionics, and space. Nowadays, the embedded compute infrastructure of industrial CPS is based on heterogeneous multi-core or many-core subsystems that are distributed and connected via complex networks. The distributed subsystems contain hardware and software components that perform different tasks, including data processing, control, monitoring, and logging/reporting. The interconnection between the subsystems allows for the integration of different functionalities and features, making such distributed CPS (dCPS) very complex machines.

Manufacturing companies of complex dCPS are encountering challenges in evaluating the impact of design decisions on non-functional aspects at the early stages of designing their next-generation machines. As a result, designers of industrial complex dCPS need quick answers to so-called "what-if" questions with respect to possible design decisions and their consequences on system performance, cost, energy consumption, etc. This necessitates efficient and scalable system-level design space exploration (DSE) methods for dCPS [1]. These methods should integrate appropriate models of dCPS, simulation and optimization techniques, as well as supporting tools to facilitate the exploration of a wide range of design decisions.

Although many DSE methods have been developed for embedded Systems-on-Chip (SoCs), they are not (directly) applicable to complex dCPS due to several obstacles [2]. One of these obstacles is that these methods do not have an appropriate global system model that allows the behavior of complex dCPS to be captured. It is also very hard, or rather infeasible, to construct such a global system model manually for industrial-scale dCPS.

To overcome this obstacle, a global system model should be derived automatically at a high level of abstraction [2]. To create a separation of concerns, this abstract model should be based on the Y-chart approach [3], i.e., it should incorporate a hardware platform architecture model, an application workload model, including the software processes running on the platform, and an application-to-architecture mapping model. However, significant scientific challenges remain in the development of these models, such as defining highly abstract models that are also sufficiently accurate to enable efficient DSE of complex dCPS, as well as devising methods and tools for automated derivation of such models.

In this paper, we address the scientific challenge concerning the definition and automated derivation of an application workload model needed for DSE in the next generation of industrial complex dCPS. The development of such a model is challenging because it should possess several essential characteristics. Specifically, the model should be:

- **Abstract and coarse-grained:** This potentially allows the whole software infrastructure behavior of a complex dCPS to be effectively captured at a high-level of abstraction into a single model;
- **Timing and architecture agnostic:** Avoiding timing and architecture artifacts in the model, attributed to a specific hardware platform, ensures high flexibility to efficiently explore alternative mappings of the application workload onto a wide variety of hardware platforms;
- **Dependency-aware:** Explicitly capturing dependencies between software processes in the model allows efficient

exploration and exploitation of different degrees of parallelism when the application workload is mapped onto a hardware platform;
- **Mode-aware:** Capturing different modes that arise due to different dCPS software/hardware configurations, allows different application workload scenarios to be modeled.

Our main novel contributions can be summarized as follows:

1) We propose a highly abstract application model that is timing agnostic, as well as dependency- and mode-aware. It *mimics* the computational and communication behavior of the complex software infrastructure of dCPS and, more specifically, it represents the workload that this software infrastructure imposes on the underlying hardware platform architecture;
2) We propose a method to automatically derive our model from data collected during runtime of the software infrastructure of a complex dCPS. Such an approach is particularly valuable in cases where the analysis of the intricate source code to identify process dependencies across different application workload scenarios is exceptionally challenging, especially when multiple programming languages are involved. Moreover, our method is adaptable to various complex industrial dCPS, which typically incorporate runtime monitoring and data collection facilities for testing and diagnostics.
3) We implement our model and method in a software tool that generates an executable model for a discrete event simulator. We apply our approach to automatically derive an application workload model of a real-world industrial dCPS that is part of many lithography scanner machines manufactured by ASML.

The remainder of the paper is organized as follows: In Section II, we discuss related work. Section III presents our model and the data that need to be collected for automated model derivation. In Section IV, we present our method for deriving the model from the collected data. In Section V, we present the model evaluation setup and validation results. Finally, we conclude the paper in Section VI.

II. Related Work

Several research works have been conducted in the field of software application modeling to explore design possibilities in CPS. These works can be classified into the following two groups based on the complexity of the CPS applications.

Simple CPS Applications: The works in this group focus on relatively simple CPS use cases, such as a line follower robot [4], or a specific subsystem within a larger system. These works often use well-known models of computation (MoCs) to model the software application behaviour. For instance, some works utilize directed acyclic graphs (DAGs) to capture software task dependencies [5]–[9], while others use actor-oriented [10] or discrete-event models to describe interactions among tasks. Functional models are also employed to represent CPS application behavior, usually in a time-continuous manner, with differential equations [11]–[15].

Additionally, various tools have been developed based on these MoCs, such as Ptolemy [16], which is an actor-based simulator, or PTIDES [17], which allows modelling applications based on different MoCs or combination of them. Furthermore, frameworks like Octopus [18] and Daedalus [19] facilitate modeling techniques and co-simulation of models specifically for DSE. However, the aforementioned works often include a lot of specific details when modeling the behavioral aspects of applications. Therefore, they lack the necessary abstraction and coarse-grained representation required to effectively capture and model complex CPS applications for DSE and also cannot model dynamic workloads. In addition, they usually have scalability restrictions, limiting their ability to create a comprehensive application model of distributed systems for industry-relevant cases [2].

Complex CPS Applications: In the case of complex CPS applications, there are frameworks, such as PtolemyII [16], OpenMETA [20], BPMN4CPS [21], SysML2 [22], Maestro [23], and OpenModelica [24] that enable designers to model the behavior of industry-relevant systems for DSE purposes. However, the modeling process when utilizing these frameworks is mostly done manually due to a lack of tools that could help designers derive models in an automated way. Such manual modeling is a very time consuming and difficult process. In contrast, we propose a model derivation method implemented in a software tool that allows designers to avoid manual application modeling for an industry-relevant complex dCPS in case the designers need to perform DSE to design the next generation of dCPS. This is possible because initial models can be automatically derived using our method and tool from data collected during runtime of the software infrastructure of the current generation of the dCPS.

Finally, automated model inference is a well-established approach where inference is done by analyzing the source codes of an application or utilizing logs and traces of a system running the source code [25], [26]. These inference techniques are used for various purposes. Some studies utilize them for software testing and verification [27], [28], assisting engineers in identifying underlying issues, such as timing bottlenecks [29] or performance anomalies [30]. However, the aforementioned techniques primarily concentrate on the inference of *behavioral* models. Such models are not sufficiently abstract and coarse-grained for efficient DSE of complex dCPS. Moreover, many of these inference techniques are not applicable to complex dCPS because analyzing complex application source code is challenging.

Other studies leverage automated model inference to estimate the performance of Multiprocessor SoCs [31], [32], and parallel distributed systems [33] during their early design stages or in scenarios where real-time systems are assembled from black-box components [34]. However, the inferred models include timing and hardware architecture dependent information that does not allow the exploration of different application mappings on different hardware platforms.

In contrast, our method shifts the focus to automated derivation of application *workload* models that are sufficiently

abstract and coarse-grained, thereby enabling efficient DSE of complex dCPS. In addition, our method eliminates timing information from the derived models by considering only the order of computation and communication workloads and representing these workloads as abstract events.

III. Application Workload Model and Traces

In this section, we begin with presenting our application workload model in a semi-formal manner. Then, we provide a detailed explanation of the data that needs to be collected at runtime for automated derivation of the model. This data is represented as traces and we describe the process and required tools for collecting these traces.

A. Application Workload Model

Based on our observations of real-world dCPS, such as lithography scanner machines, industrial printers, and interventional X-ray machines, a complex dCPS software infrastructure can be seen, at a high level, as hundreds of software processes that are triggered by events and exchange messages with each other. When a process is triggered, it performs computation and data communication actions, referred to as "firing", and upon completion of the firing, it enters an "exit" state, waiting to be triggered and fire again. This fire-and-exit behavior repeats throughout the entire life cycle of a process.

Therefore, we model the workload behavior of the dCPS software infrastructure as a graph W represented as the tuple $W = (P, C)$, where $P = \{p_1, ..., p_{|P|}\}$ denotes a set of processes that model the corresponding processes in the dCPS software infrastructure. Additionally, $C = \{ch_1, ..., ch_{|C|}\}$ represents a set of communication channels that model the exchange of control or data messages between the processes. We denote each channel ch_j as a tuple $ch_j = (p_s, p_d)$, where ch_j is exclusively dedicated to communicating messages from source process p_s to destination process p_d. Figure 1 illustrates an example of a workload model W with a set of five processes $P = \{A, B, C, D, E\}$, depicted as squares, and a set of nine communication channels $C = \{ch_{AB}, ..., ch_{DE}\}$ represented by arrows. The direction of an arrow indicates the flow of messages communicated within the channel.

In our workload model, each process p_i is defined as a set of modes, i.e., $p_i = \{M_1, ..., M_{|p_i|}\}$ where every mode $M_q = \langle e_1, e_2, ..., e_{|M_q|}\rangle$ is a finite sequence of coarse-grained abstract events that models the types of actions performed during a single process firing, arranged in chronological order. Every event $e \in M_q$ falls into one of the following three categories.

Computation events: A computation event is denoted as $e^C = (sig)$ and models computation actions with an abstract workload described by signature sig. The specific content of the signature depends on the use case. For instance, sig could indicate the number of multiply-accumulate or floating-point operations performed by the computational action, or it could be a histogram showing the distribution of the number of executions of assembly instructions from an abstract instruction set. In our model, we defined sig as the number of cycles a process has been actively running on a CPU.

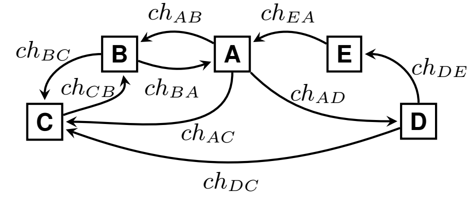


Fig. 1: Example of an application workload model W

Communication events: These events are crucial for modeling message exchange and synchronization between processes through channels. A communication event is classified as either a *write* event, denoted as e^W, or a *read* event denoted as e^R. A *write* event occurs when a process sends a message to another process, and a *read* event occurs when a process receives a message. Each e^W or e^R event is defined as a tuple $(ch, size, sig)$, where ch specifies the channel used for the communication, $size$ denotes the size of the communicated message in bytes, reflecting the communication workload of the event on the communication channel, and sig represents an abstract computation workload associated with the event's execution in the source/destination process, measured in cycles similar to a computation event and it contributes to computing the actual time required for sending or receiving a message.

Timer events: These events are internal triggers that originate within a process and initiate a computation or communication event after a specific amount of time has elapsed. These events allow processes to schedule actions that need to be performed after a certain period. A timer event is classified as either *timer setter*, denoted as e^{TS}, or *timer handler* denoted as e^{TH}. Event e^{TS} sets a timer with an absolute time value, and once that time has elapsed, a corresponding e^{TH} event is triggered to initiate other events. Each e^{TS} or e^{TH} event is defined as a tuple (τ, t), where τ is the timer identifier and t denotes the duration time, in seconds, set for timer τ.

Our application workload model, described above, possesses the essential characteristics needed for system-level DSE of complex dCPS, discussed in Section I. First, the model is rather abstract and coarse-grained because it captures the computation and communication actions within software processes as sequences of course-grained events with abstract workloads. Second, the model is timing agnostic because the sequences of events and the abstract workloads associated with the events do not include any timing. Next, the model is dependency-aware because it explicitly captures inter-process dependencies through the set of communication channels. Finally, the model is mode-aware because a process is defined as a set of modes where every mode captures different application workload scenarios.

B. Traces

As mentioned earlier, in order to derive our workload model, we need specific system data collected at runtime using tracing. We categorize and explain the traces into two groups based on their level of generality. The first group (Execution traces) is useful for all models, and the second group (System status

traces) varies based on the particular sig parameter chosen in different models.

1) Execution trace: We define the needed data $D = \{R_1, ..., R_{|D|}\}$ as a set of execution traces. Every execution trace $R_i \in D$ is collected individually for every system software/hardware configuration, thus R_i represents the workload scenario associated with the configuration. We define every execution trace $R_i = \langle r_1, ..., r_{|R_i|} \rangle$ as a sequence of records r_k that are collected over time at specific locations in the software code where trace points have been inserted. To capture a comprehensive view of the software execution, we strategically insert trace points at the *start* and *end* of every function call. Each individual record r_k is represented as a tuple $r_k = (ts, pn, fn, l, A)$, where ts is the timestamp of the record, pn is the name of the process running at the time, fn is the name of the function call executed in the process at the time, $l \in \{start, end\}$ denotes the location within the function code where the record is collected, and $A = (c, id, \lambda, \theta)$ is a tuple of attributes related to function call fn.

The function attribute $c \in \{send, receive, trigger, handler\}$ classifies function calls into four distinct categories. Function calls classified as $send$ or $receive$ are primarily used for data communication between software processes and allow tracking of message exchanges, thereby providing valuable insights into dependencies between processes. Function calls classified as $trigger$ or $handler$ provide insights into the utilization of internal timers within the processes. Attribute id is the message identifier of $send$ and $receive$ function calls, or the timer identifier of $trigger$ and $handler$ function calls. Furthermore, λ denotes the size of the exchanged message (in bytes), and θ indicates the duration (time in seconds) set for the timer.

As an example, an excerpt of an execution trace R_i is shown in Figure 2 and represented in a tabular format. Rows represent the sequence of collected records r_k and columns display the various data items of each record described above.

In addition, the example excerpt of the execution trace in Figure 2 is visualized in Figure 3, clearly showing the nested function calls within and the interactions between software processes A and B over time. Functions categorized under attribute c are visually highlighted with colors.

This particular scenario starts with process A which executes function f_0, and subsequently calls function f_{01} within f_0 to send a message i_0 with a size of 40 bytes to process B. The destination process of the message is distinguished by matching its id. Process B receives the message with identifier i_0 by executing function call g_0. After that, process B invokes function g_{11}, within g_1, to set a timer for a duration of 4 seconds. As depicted by an arrow, the timer is handled by function call g_3 later. Following the timer event, process B proceeds to send a message with a size of 100 bytes to process A, and process A calls function f_{03} within f_0 to receive the message. Additionally, there is another timer with identifier j_1 that is set and subsequently handled in process B.

To collect the execution traces $R_i \in D$ and monitor software processes in practice, various tools are available. One such tool is LTTng [35], an open-source tracing framework that

ts	pn	fn	l	c	id	λ	θ
0	A	f_0	start				
0.5	A	f_{01}	start	send	i_0	40	
1.4	B	g_0	start	receive	i_0	40	
2.0	A	f_{01}	end	send	i_0	40	
2.15	A	f_{02}	start				
2.85	B	g_0	end	receive	i_0	40	
3.0	B	g_1	start				
3.1	B	g_{11}	start	trigger	j_0		4
4.3	B	g_{11}	end	trigger	j_0		4
4.9	B	g_1	end				
⋮	⋮	⋮	⋮				
16.25	B	g_3	end	handler	j_1		4

Fig. 2: Excerpt of an example Execution Trace

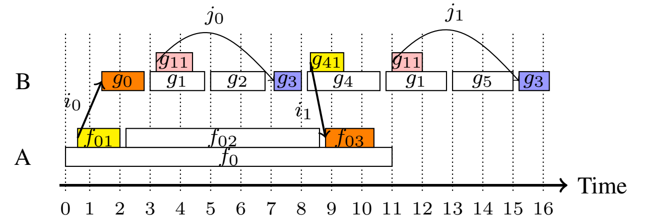


Fig. 3: Timing chart visualizing the traces in Figure 2

is compatible with most Linux distributions. LTTng provides predefined trace points at the kernel space of the operating system (OS), offering details about the running processes. Additionally, it allows for code instrumentation, enabling the collection of customized data from the OS user space.

2) System status trace: In addition to execution traces, to calculate the customized sig in our model, we also require traces that show the status of the system. These traces provide valuable information that is accessible to the operating system, including the status of processes (idle, interrupt, etc.) and the system frequency over time. To obtain and analyze the system status trace, many existing tools can be used. In our case, we utilize Trace Compass (TC) [36], an open-source tracing and analysis framework. TC allows users to obtain, visualize, and analyze traces generated by different tracing systems, including LTTng mentioned earlier.

IV. Model Derivation Method

In this section, we first present our method for deriving an application workload model from traces as a formal procedure and then briefly describe how the model is transformed into an executable format.

Formal procedure: The formal procedure of deriving an application workload model from trace is shown in Algorithm 1. It takes a set of execution traces D and system status trace TC as inputs, and generates a workload model W, as a graph of processes communicating via channels.

Algorithm 1 begins with deriving the topology of the graph, as shown in Lines 2-10. All trace records r_k are examined and based on the process name $r_k.pn$, stored in a record, a corresponding process is created if the process has not been created yet (Lines 4-5). Furthermore, if the record r_k has a

Algorithm 1: Workload Model Derivation from Traces

```
   Input  : D = {R_1, ..., R_|D|}, TC
   Output: W = (P, C)
1  P, C ← ∅;
2  foreach R_i ∈ D do
3      foreach r_k ∈ R_i do
4          if ∄ p_{r_k.pn} ∈ P then
5              p_{r_k.pn} ← ∅;    P ← P + p_{r_k.pn};
6          if r_k.A.c = receive then
7              r_j = findREC(R_i, send, start, r_k.A.id);
8              if ∄ ch_{(r_j.pn, r_k.pn)} ∈ C then
9                  ch_{(r_j.pn, r_k.pn)} = (p_{r_j.pn}, p_{r_k.pn});
10                 C ← C + ch_{(r_j.pn, r_k.pn)};
11     foreach p_x ∈ P do
12         H_x^i ← ∅;   /* set of tuples (ev, T_s, T_e)   */
13         H_x^i ← H_x^i + (∅, 0, 0);
14     for k = 1 to |R_i| do
15         if r_k.A.c = send ∧ r_k.l = start then
16             r_k' = findREC(R_i, send, end, r_k.A.id);
17             sig = getSIG(TC^{R_i}_{r_k.pn}, r_k.ts, r_k'.ts);
18             r_j = findREC(R_i, receive, start, r_k.A.id);
19             e_k^W = (ch_{(r_k.pn, r_j.pn)}, r_k.A.λ, sig);
20             H^i_{r_k.pn} ← H^i_{r_k.pn} + (e_k^W, r_k.ts, r_k'.ts);
21         if r_k.A.c = receive ∧ r_k.l = start then
22             r_k' = findREC(R_i, receive, end, r_k.A.id);
23             sig = getSIG(TC^{R_i}_{r_k.pn}, r_k.ts, r_k'.ts);
24             r_j = findREC(R_i, send, start, r_k.A.id);
25             e_k^R = (ch_{(r_j.pn, r_k.pn,)}, r_k.A.λ, sig);
26             H^i_{r_k.pn} ← H^i_{r_k.pn} + (e_k^R, r_k.ts, r_k'.ts);
27         if r_k.A.c = trigger ∧ r_k.l = start then
28             e_k^{TS} = (r_k.A.id, r_k.A.θ);
29             H^i_{r_k.pn} ← H^i_{r_k.pn} + (e_k^{TS}, r_k.ts, r_k.ts);
30         if r_k.A.c = handler ∧ r_k.l = start then
31             e_k^{TH} = (r_k.A.id, r_k.A.θ);
32             H^i_{r_k.pn} ← H^i_{r_k.pn} + (e_k^{TH}, r_k.ts, r_k.ts);
33 foreach p_x ∈ P do
34     for i = 1 to |D| do
35         M_i ← ∅;
36         if ∃ H_x^i then
37             t_s = r_k.ts : r_k ∈ R_i ∧ k = |R_i|;
38             H_x^i = H_x^i + (∅, t_s, t_s);
39             for y = 1 to |H_x^i| − 1 do
40                 if H_x^i[y].ev ≠ ∅ then
41                     M_i ← M_i + H_x^i[y].ev;
42                 t_s = H_x^i[y].T_e;    t_e = H_x^i[y+1].T_s;
43                 sig = getSIG(TC_x^{R_i}, t_s, t_e);
44                 e_y^C = (sig);    M_i ← M_i + e_y^C;
45         p_x ← p_x + M_i;
46 return (P, C)
```

function call of class $receive$ then a communication channel is created if the channel has not been created yet (Lines 6-10). To create the channel, Algorithm 1 finds the corresponding record r_j with a function call of class $send$, where this $send$ function call and the aforementioned $receive$ function call

Algorithm 2: findREC

```
   Input  : R, c, l, id
   Output: r
1  foreach r_i ∈ R do
2      if r_i.A.c = c ∧ r_i.l = l ∧ r_i.A.id = id then
3          return (r ← r_i)
```

have exchanged a message with the same identifier. Record r_j is found in Line 7 by using the procedure `findREC`. It is shown in Algorithm 2 and used at multiple places in Algorithm 1 to find a specific record in a given trace. It takes as inputs trace R, function call class c, trace point location l, and message identifier id. By examining all records in input trace R, `findREC` returns the record with function call class, location, and message identifier that match the inputs c, l, and id, respectively.

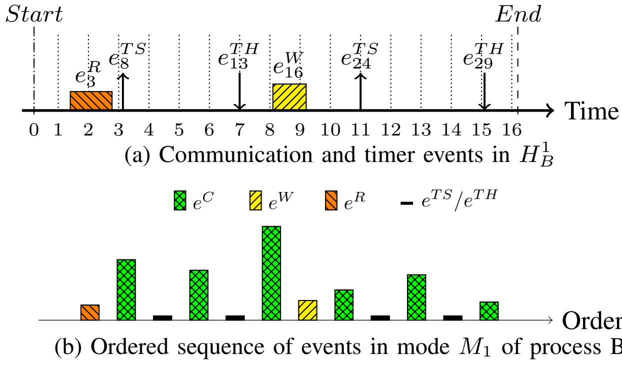
After deriving the topology of the graph containing all processes and communication channels, Algorithm 1 proceeds with the derivation of the modes of every process p_x by creating the sequence of communication, computation, and timer events associated with every mode.

First, in Lines 14-32, Algorithm 1 examines the records in every trace R_i in-order. For every record $r_k \in R_i$ with location $start$ and function call class $send$, $receive$, $trigger$, or $handler$, the algorithm creates a corresponding event and appends the event together with its start and end time to an ordered list of events H_x^i (all ordered lists are initialized in Lines 11-13).

In Lines 15-20, a write event $e_k^W = (ch, size, sig)$ is created and appended to the corresponding ordered list if the currently examined trace record r_k has location $start$ and a function call of class $send$. The attribute λ in record r_k provides the $size$ of the message. To determine the communication channel ch, Algorithm 1 needs to identify the source and destination processes. The process name pn in r_k identifies the source process. The identification of the destination process is done in Line 18 by using procedure `findREC` to find the receiving record r_j and taking the process name pn in r_j. The abstract workload sig is calculated in Line 17 by using the procedure `getSIG`, which will be explained later in this section.

Similarly, in Lines 21-26 of Algorithm 1, a Read event $e_k^R = (ch, size, sig)$ is created using procedure `findREC` and `getSIG`, and appended to a corresponding ordered list. In Lines 27-32, a timer setter event $e_k^{TS} = (\tau, t)$ or a timer handler event $e_k^{TH} = (\tau, t)$ is created and appended to a corresponding ordered list. The attributes id and θ in the examined record r_k provide the timer identifier τ and the duration time t set for the timer, respectively.

For example, let us apply Algorithm 1 on the execution trace in Figure 2 and consider process B in this trace. In Lines 12-13, the ordered list H_B^1 associated with mode 1 of process B is initialized. In Lines 14-32, communication and timer events are created and appended to H_B^1. The content of H_B^1 is visualized in Figure 4(a) where the left/right side



(a) Communication and timer events in H_B^1

(b) Ordered sequence of events in mode M_1 of process B

Fig. 4: Derivation of the first mode of process B

Algorithm 3: getSIG

Input : TC_{pn}^R, T_{start}, T_{end}
Output: sig

1. $cycles = 0$;
2. **foreach** $q \in TC_{pn}^R$ **do**
3.   **if** $T_{start} < q.OnCPU \leq T_{end}$ **then**
4. $cycles = cycles + q.OnCPU * q.freq$
5. **return** $(sig = cycles)$

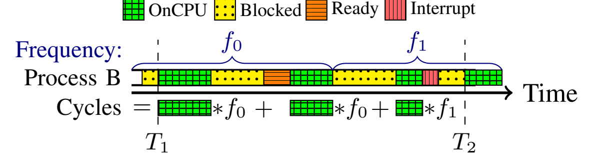


Fig. 5: Calculation of the signature (cycles) of process B

of a shaded box indicates the absolute start/end time of a communication event. The height of the box represents the abstract computation workload (signature) of the event, while the vertical arrows pointing upwards and downwards represent timer setter and timer handlers, respectively.

Second, in Lines 33-45 for every process p_x and trace R_i, Algorithm 1 derives mode $M_i \in p_x$. It is important to note that not every process is necessarily part of every trace, which means that the ordered list H_x^i may not exist for every process-trace combination. If H_x^i exists, all communication and timer events are moved in-order from the corresponding ordered list H_x^i to M_i (Lines 36-41). During this in-order movement of events, since the workload model is timing agnostic, the absolute start/end times of events are dropped (Line 41).

In addition, computation events are created by using the procedure `getSIG` and inserted in M_i after each communication and timer event (Lines 42-44). For example, consider process B, the trace in Figure 2, and the ordered list H_B^1 visualized in Figure 4(a). In Lines 37-44, communication, timer, and computation events are placed in M_1 in the correct order as explained above. These events and their order in the derived mode M_1 of process B are visualized in Figure 4(b). The height of a shaded box represents the abstract computation workload (signature) of a computation or communication event. It is worth noting that the width of all shaded boxes is the same, indicating that the notion of absolute start/end time of events is not present in M_1. Thus, M_1 captures only the order of events and their abstract workload.

Finally, Algorithm 1 returns the derived workload model $W = (P, C)$ in Line 46. As explained above, the algorithm uses procedure `getSIG` to calculate the signature of communication and computation events (Lines 17, 23, and 43). The procedure is shown in Algorithm 3. It takes as inputs system status trace TC_{pn}^R concerning process name pn and mode R, start time T_{start}, and end time T_{end}. It returns an abstract computation workload sig imposed by process $p_{pn} \in R$ within time frame $[T_{start}, T_{end}]$.

As explained in Section III-A, in our workload model the signature sig is defined as the number of cycles a process has been actively running on a CPU within a given time frame. To calculate this number, Algorithm 3 utilizes the system status trace TC_{pn}^R in the time frame $[T_{start}$,$T_{end}]$ (Lines 2-3), and considers the periods of time q when process p_{pn} has status `OnCPU`, indicating that the process is actively executed on a CPU. These periods $q.OnCPU$ are multiplied by the current CPU frequency and accumulated to obtain the total number of cycles (Line 4). As an example, Figure 5 illustrates the behavior of Algorithm 3 on process B, where the system status trace TC_B of process B within the time frame $[T_1$,$T_2]$ is visualized. During time interval $[T_1$,$T_2]$, process B has various statuses (`OnCPU`, `Blocked`, `Ready`, `Interrupted`), but Algorithm 3 uses only the `OnCPU` periods (depicted as green boxes) and the corresponding frequencies (f_0 and f_1) to calculate the total number of cycles.

Executable Model: To transform the derived application workload model into an executable format and integrate it with other system models (e.g., a hardware platform architecture model), we could utilize any system modeling and discrete-event simulation framework. In our case, we have chosen OMNeT++ [37], a powerful framework known for its extensive features and libraries, making it particularly well-suited for modeling and simulation of complex systems, such as large networks and distributed systems. To convert the models into OMNeT++ executable code for co-simulation, we have developed a custom code generator that produces multiple files needed for the co-simulation including individual event files for each mode of a process. Every event file contains the abstract events (and their attributes) of a process in the partial order that needs to be imposed during the simulation. Due to space limitations, more details about our OMNeT++ executable models implementation are omitted.

V. Experimental Evaluation and Results

This section presents the experimental approach, we have employed to evaluate the accuracy of our proposed workload model applied on a real-world dCPS, namely the ASML Twinscan lithography machine. First, we describe the setup used for our experiments, and then we present the results from the evaluation.

A. Experimental Setup

The ASML Twinscan machine utilizes advanced optics and precise positioning of reticles (also known as masks) to transfer circuit patterns onto silicon wafers. Various application workloads, called operation modes, are imposed by recipes that specify parameters and steps for efficient batch processing, i.e., performing multiple operations on a set of wafers simultaneously.

This machine consists of multiple hardware/software subsystems that run on different operating system (OS) platforms. Gathering traces from these diverse OS platforms requires the utilization of various specialized tools and facilities, a process that demands significant time, resources, and effort. Therefore, in our analysis and accuracy evaluation, we primarily focus on the main subsystem `Host`, which operates on the Linux OS platform. This approach allows us to dedicate our time and effort to evaluating the accuracy of our workload model within the scope of this subsystem, thereby obtaining valuable initial feedback in a timely manner before expanding our effort and investigation to other subsystems on other OS platforms.

However, to ensure a comprehensive and accurate evaluation of the `Host` subsystem and corresponding workload model, we also take into account its interaction with other subsystems. We abstract all components and functionalities of those subsystems into one model called Environmental module (`ENV`). This module is responsible for sending messages exactly at the same time `Host` receives these messages from other subsystems within the real Twinscan machine. All messages sent to `Host` and their timestamps are collected using the monitoring and tracing facilities of `Host` accessible to us.

The `ENV` module implements a simple message generator that "replays" the sending of the collected messages at their designated timestamps in the timeline. Such a message replay approach allows us to primarily focus on modeling and capturing the timing of events within `Host`, while maintaining accurate interaction with other subsystems. It is worth noting that the above approach can be used in other different real-world scenarios where complete tracing facilities may not be feasible/available for some subsystems. However, we can still capture and accurately model the application workload for the rest of the subsystems.

To evaluate the accuracy of the workload model, we capture the time of every message sent from `Host` to `ENV` during the simulation in OMNeT++. We call this captured time the $simulated$ timestamp of an outgoing message. Then, we compare the simulated timestamp of every outgoing message with a reference timestamp called $real$ timestamp of the same message that is collected using tracing facilities on the Twinscan machine.

By comparing the simulated and real timestamps of outgoing messages, we check how close our model is to the real timing behavior of `Host` in every time interval between receiving a message from `ENV` and sending an outgoing message to `ENV`. The smaller the difference Δ between the simulated and real timestamps, the more accurate our model is in terms of timing.

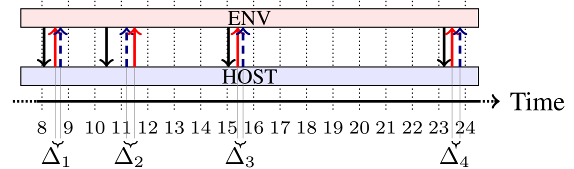


Fig. 6: Model evaluation: real and simulated timestamps

Figure 6 visualizes the aforementioned model evaluation approach. The black arrows from `ENV` to `Host` indicate the real times at which messages are received by `Host`. The red arrows from `Host` to `ENV` indicate the real timestamps of outgoing messages, and the dashed blue arrows show the simulated timestamps of the same outgoing messages. The differences between the simulated and real timestamps are denoted as Δ_1, Δ_2, etc. We consider our model accurate if these differences fall within an acceptable range, which depends on the specific use case of the dCPS under modeling and DSE.

B. Evaluation Results

We have automatically derived the workload model of the software application that processes wafers on the ASML Twinscan machines. The derived workload model includes **407** processes and **1408** channels in five different operation modes, demonstrating the large scale and complexity of real industrial application workloads. Table I reports the extensive total number of exchanged messages for each mode and the specific number of Δ values associated with messages sent from the `HOST` to `ENV`. These values have been obtained by comparing simulated and real timestamps, as elaborated in Section V-A.

According to ASML engineers, a significant portion of Δ values should fall within the range of one millisecond for the application use case of wafer processing on Twinscan in order to consider the accuracy of our model sufficient to be useful in practice.

TABLE I: Number of messages and Δ samples for the five operation modes of the wafer processing workload model

Modes	M_1	M_2	M_3	M_4	M_5
Num. of messages	152319	227482	231598	303497	254273
Num. of Δs	18975	31839	32070	42745	37214

Figure 7 depicts the distribution of the Δ values across various time ranges in a stacked bar chart. The x-axis shows the different modes (M_1 to M_5) and the y-axis shows the distribution in percentage. Every bar is segmented and each segment, depicted in a different color, corresponds to a time range. For example, the blue segment of a bar shows the percentage of Δ values that fall in the range of [0..1) milliseconds. Analysing Figure 7, we observe that the majority of Δ values for every mode, i.e., more than 90% of the values, are smaller than 1 millisecond.

In addition, Table II provides the average distribution of Δ values across different time ranges for the five operation

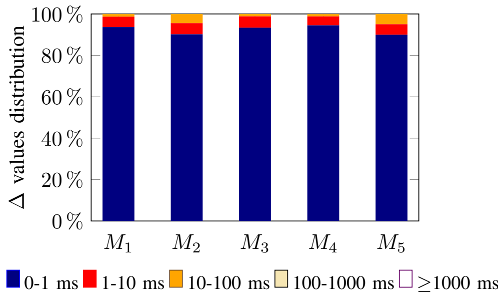


Fig. 7: Distribution of Differences Δ

modes. Notably, our analysis reveals that over 92% of the Δ values, on average, are concentrated within the microsecond range. Such exceptional level of precision, achieved by our model on the ASML machine, demonstrates the model effectiveness in accurately capturing the application workload, thereby facilitating effective DSE of complex dCPS.

TABLE II: Average distribution of the differences Δ between real and simulated timestamp values

Δ (ms)	**[0..1)**	**[1..10)**	**[10..100)**	**[100-1000)**	**≥ 1000**
Average%	92.34	5.05	2.55	0.016	0.001

VI. Conclusion

In this paper, we have presented an innovative approach for automated derivation of an application workload model for system-level DSE of complex dCPS. Our approach leverages runtime traces and transforms them into a timing-independent workload model that is aware of application processes' dependencies and operational modes. The workload model is abstract and coarse-grained as it does not contain the exact behavior of the application processes, function calls within the processes, and process communication protocols. Thus, it enables scalable and efficient DSE while accurately capturing dynamic application workloads. We have applied our approach to the main subsystem of the ASML Twinscan lithography machines, evaluating it across five distinct workload scenarios. The experimental results demonstrate high accuracy of our derived workload model, making it practically useful for DSE of real-world dCPS.

References

[1] B. van der Sanden *et al.*, "Model-driven system-performance engineering for cyber-physical systems : Industry session paper," in *EMSOFT*, 2021, pp. 11–22.
[2] M. Herget *et al.*, "Design space exploration for distributed cyber-physical systems: State-of-the-art, challenges, and directions," in *DSD*, 2022, pp. 632–640.
[3] B. Kienhuis *et al.*, "A methodology to design programmable embedded systems: the y-chart approach," *SAMOS*, pp. 18–37, 2002.
[4] J. Fitzgerald *et al.*, "Exploring the cyber-physical design space," *INCOSE*, vol. 27, pp. 371–385, 07 2017.
[5] Z. Shu *et al.*, "Cloud-integrated cyber-physical systems for complex industrial applications," *Mobile Networks and Applications*, vol. 21, pp. 865–878, 2016.
[6] K. Neubauer *et al.*, "Exact multi-objective design space exploration using aspmt," in *DATE*, 2018, pp. 257–260.
[7] G. Tanganelli *et al.*, "A methodology for the design and deployment of distributed cyber-physical systems for smart environments," *FGCS*, vol. 109, pp. 420–430, 2020.
[8] V. Richthammer *et al.*, "Search-space decomposition for system-level design space exploration of embedded systems," *TODAES*, vol. 25, no. 2, pp. 1–32, 2020.
[9] L. Gressl *et al.*, "Design space exploration for secure iot devices and cyber-physical systems," *TECS*, vol. 20, no. 4, pp. 1–24, 2021.
[10] I. Akkaya *et al.*, "Systems engineering for industrial cyber–physical systems using aspects," *PIEEE*, vol. 104, no. 5, pp. 997–1012, 2016.
[11] P. Bogdan *et al.*, "Towards a science of cyber-physical systems design," in *ICCPS*, 2011, pp. 99–108.
[12] A. Canedo *et al.*, "Architectural design space exploration of cyber-physical systems using the functional modeling compiler," *Procedia CIRP*, vol. 21, pp. 46–51, 2014.
[13] S. Narain *et al.*, "Design space exploration for cyber physical systems," Perspecta Labs Basking Ridge, United States, Tech. Rep., 2019.
[14] M. Amir *et al.*, "Pareto optimal design space exploration of cyber-physical systems," *Internet of things*, vol. 12, p. 100308, 2020.
[15] J. Wan *et al.*, "Functional model-based design methodology for automotive cyber-physical systems," *IEEE Systems Journal*, vol. 11, no. 4, pp. 2028–2039, 2015.
[16] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. [Online]. Available: http://ptolemy.org/books/Systems
[17] P. Derler *et al.*, "Ptides: A programming model for distributed real-time embedded systems," *University of California, Berkeley, EECS Technical Report*, 2008.
[18] T. Basten *et al.*, "Model-Driven Design-Space Exploration for Embedded Systems: The Octopus Toolset," in *ISoLA*, 2010, pp. 90–105.
[19] T. Stefanov *et al.*, *DAEDALUS: System-Level Design Methodology for Streaming Multiprocessor Embedded Systems on Chips*. Springer Netherlands, 2017, pp. 983–1018.
[20] J. Sztipanovits *et al.*, "OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber-Physical Systems," in *From Programs to Systems*. Springer Berlin Heidelberg, 2014, pp. 235–248.
[21] I. Graja *et al.*, "Demonstrating bpmn4cps: Modeling anc verification of cyber-physical systems," in *CCNC*, 2017, pp. 593–593.
[22] J. Gray *et al.*, "Reflections on the standardization of sysml 2," pp. 287–289, 2021.
[23] C. Thule *et al.*, "Maestro: the into-cps co-simulation framework," *Simulation Modelling Practice and Theory*, vol. 92, pp. 45–61, 2019.
[24] P. Fritzson *et al.*, "The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development," *MIC*, vol. 41, no. 4, pp. 241–295, 2020.
[25] W. Sun *et al.*, "An automatic software behavior model generation method for industrial cyber-physical system," in *INDIN*, vol. 1, 2020, pp. 897–902.
[26] D. Shin *et al.*, "Prins: scalable model inference for component-based system logs," *EMSE*, vol. 27, no. 4, p. 87, 2022.
[27] D. Lorenzoli *et al.*, "Automatic generation of software behavioral models," in *ICSE*, 2008, pp. 501–510.
[28] J. Geismann *et al.*, "A systematic literature review of model-driven security engineering for cyber–physical systems," *JSS*, vol. 169, p. 110697, 2020.
[29] R. Jonk *et al.*, "Inferring timed message sequence charts from execution traces of large-scale component-based software systems," 2019.
[30] H. Meyer *et al.*, "On the effectiveness of communication-centric modelling of complex embedded systems," in *ISPA*, 2018, pp. 979–986.
[31] T. Isshiki *et al.*, "Trace-driven workload simulation method for multiprocessor system-on-chips," in *DAC*, 2009, pp. 232–237.
[32] T. Kogel *et al.*, "Method for generating workload models from execution traces," Patent, Dec 08, 2016, uS20160357516A1.
[33] A. Mizan *et al.*, "An automatic trace based performance evaluation model building for parallel distributed systems (abstracts only)," *SIGMETRICS*, vol. 39, no. 3, p. 12, dec 2011.
[34] T. H. Feng *et al.*, "Automatic model generation for black box real-time systems," in *DATE*, 2007, pp. 1–6.
[35] M. Desnoyers *et al.*, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS*, vol. 2006, 2006, pp. 209–224.
[36] Trace Compass Developers, "Trace Compass," 2021, [Software]. [Online]. Available: https://tracecompass.org/
[37] A. Varga *et al.*, "An overview of the omnet++ simulation environment," in *SIMUTOOLS*, 2010.