

# CompDSE: A Methodology for Design Space Exploration of Computing Subsystems within Complex Cyber-Physical Systems

Faezeh Sadat Saadatmand, Todor Stefanov, Ignacio González Alonso,  
Andy D. Pimentel, and Benny Akesson

**Abstract**—Designing the next-generation complex distributed cyber-physical systems (dCPS) poses significant challenges for manufacturing companies, necessitating efficient Design Space Exploration (DSE) techniques to evaluate potential design decisions and their impact on non-functional aspects of the systems, such as performance, reliability, and energy consumption. This paper introduces CompDSE, a methodology designed to facilitate the DSE of complex dCPS, specifically focusing on the cyber components, i.e., the computing subsystems within dCPS. CompDSE defines and utilizes abstract models of the application workload, computing hardware platform, and workload-to-platform mapping of dCPS, automatically derived from runtime trace data, and integrates them into a discrete event simulation environment to explore various design points. We demonstrate the effectiveness of our methodology through a case study on the ASML Twinscan lithography machine, a complex industrial dCPS. The results showcase potential performance enhancements achieved by optimizing computing subsystems while considering physical constraints. Evaluating each design point takes under a minute, highlighting the CompDSE efficiency and scalability in tackling complex dCPS with large design spaces.

**Index Terms**—System Modeling, Simulation, Design Space Exploration, Distributed Cyber-Physical Systems

## I. INTRODUCTION

**D**ISTRIBUTED cyber-physical systems (dCPS) are increasingly vital in today’s high-tech landscape, with applications such as semiconductor lithography, industrial printing, and medical X-ray imaging. These systems integrate various computing subsystems, called cyber components, with physical processes to achieve greater efficiency, reliability, and functionality. As the complexity of dCPS grows, managing their design, particularly the optimization of the cyber components, becomes more challenging [1]. The cyber components,

consisting of distributed software processes running on multi-core or many-core processors connected through intricate networks, have a direct impact on the overall performance and functionality of dCPS. The design challenges concerning the cyber components are further exacerbated by the trend toward software-intensive dCPS, where more functionality is implemented in software due to its flexibility and adaptability [2].

Given the impact and growing complexity of the cyber components, the design of these components calls for efficient Design Space Exploration (DSE) methods [3]. Early design decisions significantly affect non-functional aspects, such as performance, cost, and energy consumption. This is particularly important for existing high-tech machines that are complex and require system-level DSE to optimize their next-generation designs, ensuring enhanced performance and adaptability in the face of evolving technological requirements [4]. Therefore, DSE methods that can capture the behavior of complex cyber components with sufficient accuracy, while maintaining abstraction for efficient exploration, are essential [5].

To enable system-level DSE of cyber components, one effective approach is to follow the well-established Y-chart approach [6]. The Y-chart approach is known for enhancing flexibility and model reuse through its separation-of-concerns principle, where the abstract system model consists of a model of the computing hardware platform architecture, an application workload model (including the software processes running on the platform), and a mapping model connecting the (processes in the) application model to the (computing resources in the) platform model. However, manually constructing these models for complex, large-scale industrial dCPS is highly impractical. Therefore, DSE methods must include tools that automate model derivation and facilitate fast system performance analysis, enabling the exploration of diverse design decisions.

While some initial research on automated workload modeling for industrial dCPS exists [7], a comprehensive methodology and tool support that address the challenges of DSE, in terms of abstract, yet sufficiently accurate modeling, and automated model derivation, for complex cyber components within large-scale dCPS is still lacking.

Therefore, in this paper, we propose **CompDSE**, a methodology designed to address these challenges by focusing on the computing components (cyber part) within complex dCPS.

Faezeh Sadat Saadatmand and Todor Stefanov are with Leiden Institute of Advanced Computer Science (LIACS), Leiden University, Leiden, 2333 CA, The Netherlands (e-mail: f.saadatmand@liacs.leidenuniv.nl; t.p.stefanov@liacs.leidenuniv.nl).

Ignacio González Alonso is with ASML B.V., Veldhoven, 5504 DR, The Netherlands (e-mail: ignacio.alonso@asm.com).

Andy D. Pimentel is with the Informatics Institute (IvI), University of Amsterdam, Amsterdam, 1098 XH, The Netherlands (e-mail: a.d.pimentel@uva.nl).

Benny Akesson is with TNO Embedded Systems Innovation (TNO-ESI), Eindhoven, 5656 AE, The Netherlands, and also with the Informatics Institute (IvI), University of Amsterdam, Amsterdam, 1098 XH, The Netherlands (e-mail: benny.akesson@tno.nl and k.b.akesson@uva.nl).

Our methodology defines, automatically derives, and integrates highly abstract models that are sufficiently accurate to enable efficient DSE of complex cyber components. Additionally, by generating abstract executable instances of the models for discrete event simulation, CompDSE enables fast system performance analysis, thereby facilitating efficient exploration of various design possibilities of the computing components.

While the primary focus of our work is on creating abstract models that balance accuracy with exploration efficiency and can be derived automatically, the CompDSE methodology is designed to be flexible and well-suited for integration with automated search algorithms, allowing for a highly automated DSE process when needed.

The key novel contributions of this paper can be summarized as follows:

- 1) We introduce the CompDSE methodology, which involves the definition of highly abstract models for the application workload, the computing hardware platform architecture, workload-to-architecture mapping, and the environmental influence. These models are automatically derived from data collected during the runtime of an existing complex dCPS, whose next-generation version is under exploration and design;
- 2) We present a toolbox that includes three distinct types of algorithms that we have devised and implemented to automate our CompDSE methodology: (1) algorithms for automated derivation and integration of the models used in CompDSE, (2) an algorithm that transforms the derived models into abstract executable instances with execution semantics for any discrete event simulation environment, and (3) an algorithm that generates specific code from the model instances for the OMNET++ discrete event simulator, enabling fast system performance analysis and efficient exploration of various system configurations;
- 3) We evaluate our CompDSE methodology and toolbox on a real-world industrial dCPS that is a major part of many lithography scanner machines manufactured by ASML. More specifically, we explore alternative design solutions in some of the computing resources to enhance machine performance while considering the limitations imposed by the physical components of the dCPS. For this purpose, we introduce the so-called  $\Delta$ -parameters approach to capture the impact of delays in the physical environment.

The remainder of the paper is organized as follows. In Section II, we discuss related work concerning the DSE of dCPS. Section III provides details on the data that needs to be collected at runtime for deriving the models used in CompDSE. In Section IV, we present our CompDSE methodology. Sections V and VI explain the models and the toolbox utilized in CompDSE, respectively. In Section VII, we demonstrate the merits of CompDSE by exploring alternative design points for lithography machines. Finally, we conclude the paper in Section VIII.

## II. RELATED WORK

Design Space Exploration (DSE) for Cyber-Physical Systems (CPS) has been extensively studied, leading to various

methodologies and tools. While the entire process can be broken down into multiple steps [5], it can be abstractly distinguished into two primary components: 1) system modeling and analysis methods, which evaluate different design objectives of a single design point (system configuration), and 2) search algorithms, which traverse the design space consisting of many design points to find optimal designs.

The first component involves preparing the infrastructure for system modeling and analysis, using various tools to create models representing the system. These models are used to evaluate specific DSE objectives, such as cost, energy consumption, reliability, or performance. Importantly, the models can vary in their level of abstraction depending on the objective. The accuracy of these models and the efficiency of the methods used to evaluate a single design point are crucial, as they are typically conflicting, i.e., more accurate models result in slower evaluations.

The second component concerns the search mechanisms used to explore the design space. This includes various DSE algorithms for multi-objective optimization [8], [9], automating the DSE process [10], [11], and limiting the search space for more efficient exploration [12]–[14].

This paper focuses on the first component, specifically system modeling and analysis for DSE, and remains agnostic to the search mechanisms used, meaning that any search mechanism can be integrated into the CompDSE methodology.

Several related studies concentrate on system modeling and analysis methods for DSE. Some of these are based on static methods like analytical models. For instance, the CNMA method [15] uses constraint programming for estimating power consumption and ensuring the timing correctness of a CPS. Similarly, the ArchEx 2.0 framework utilizes a high-level pattern-based specification language for CPS architecture exploration [16]. However, these static methods often lack the flexibility to handle dynamic interactions between processes effectively, especially when these interactions are complex and evolve over time.

Other studies have adopted simulation-based approaches that model and co-simulate the cyber and physical parts of CPS using tools based on differential equations or well-known models of computation (MoCs). For example, in [17], Matlab Simulink is used to model both the physical and controller parts of the CPS, along with a SoC as the cyber part. This approach explores design objectives such as energy consumption and control stability in an inverted-pendulum application. Another study [18] co-simulates a high-level physical environment model in Matlab Simulink and a control application model in SystemC. Similarly, in [19], a high-level partitioning approach extends the TTool (partitioning tool) with new SysML models that abstractly represent SystemC AMS components, applied to a medical device to reduce cost and increase portability. Despite their detailed modeling capabilities, these simulation-based methods are often time-intensive and do not scale well for large and complex industrial CPS.

Functional models have also been used to represent CPS behavior in co-simulation. For instance, [20] introduces a Functional Modeling Compiler (FMC) that synthesizes technology-

dependent solutions for architectural design space exploration using multidisciplinary simulations like AMESim and Modelica. While FMC facilitates multidisciplinary simulations, it still requires structured manual definition of system interactions, making it challenging to scale for highly complex CPS.

Most of the aforementioned tools and methods are limited to modeling small CPS or only parts of large dCPS, typically involving a limited number of modules such as sensors, actuators, signal transformations, CPUs, and memory. Tools like xCPS [21] and iCyPhy [22] incorporate techniques to improve CPS modeling, but xCPS still requires significant manual effort despite some automation, and iCyPhy, while supporting modular system decomposition, struggles with large-scale systems due to state explosion. Additionally, both tools rely on detailed behavioral modeling of cyber and physical components, which poses significant challenges for large-scale industrial CPS like the ASML Twinscan machine, where the physical part comprises numerous sensors and actuators, and the cyber part consists of multiple distributed subsystems running hundreds of software processes.

Furthermore, the detailed behavior modeling required by the aforementioned methods often makes the evaluation of a single design point time-consuming. Given the large design space of complex dCPS, this makes DSE using these methods excessively time-intensive. Consequently, there is a need for necessary abstraction and coarse-grained representation to model the behavior of complex dCPS for DSE without relying on manual efforts. Moreover, most applications modeled by the aforementioned tools exhibit static behavior, with a fixed order of tasks for both the physical part (sensors, actuators) and the cyber part (data processing). In contrast, complex industrial dCPS often feature dynamic behavior, such as adaptive processing loads, or various software/hardware configurations. These dynamic behavior necessitate a tool that supports flexible modeling and exploration with sufficient accuracy and speed.

A recent approach, introduces an efficient method for CPS architecture exploration using contract-based design and sub-graph isomorphism [23]. This method improves on ArchEx 2.0 by providing better compositionality and scalability in system design space exploration. However, it focuses on structural validation, meaning it verifies the correctness of system architecture but does not analyze how different configurations impact execution time and system performance, making it unsuitable for performance-driven exploration in complex industrial CPS.

The work presented in [7] addresses only the lack of application workload models for dCPS in DSE by proposing an approach for automated workload model derivation. Their 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. However, it does not provide a comprehensive and complete modeling methodology for the DSE of computing subsystems within complex industrial-scale dCPS as well as the necessary algorithms and tools to support such methodology. In contrast, our paper proposes such comprehensive DSE methodology that uses the aforementioned workload model derivation but also introduces additional models, algorithms,

and tools necessary for the modeling of the cyber part of dCPS and the DSE analysis of the cyber part. More specifically, our methodology involves automated derivation of abstract models of the dCPS cyber part (computing components) for each system configuration, and automated model-to-model transformation and code generation to facilitate effective simulation-based performance analysis for DSE. Although we do not explicitly model the physical part of the dCPS, we consider its effects on the cyber part by using the  $\Delta$ -parameters approach we have devised. This approach enables us to effectively handle the complexity and dynamic nature of complex dCPS.

To highlight the differences between our approach and existing DSE tools/methodologies, we provide a comparative analysis in Table I, which distinguishes them based on model abstraction level, scalability for large industrial CPS, degree of automation, performance analysis type, and handling of dynamic behavior. Unlike many existing tools, CompDSE offers higher abstraction, full automation, and explicit support for dynamic behavior, making it well-suited for large-scale industrial dCPS.

### III. BACKGROUND: TRACES

To derive the CompDSE models, we need to collect specific system data from the operating system (OS) during runtime. This data must be gathered only once in the form of event traces from all subsystems (hosts) of a dCPS, and for each *operation mode*, where an operation mode corresponds to a distinct system software/hardware configuration that defines a specific execution pattern of the system. It is essential to ensure that the system is functioning correctly during the trace collection process.

For every host  $h_i$ , there are two sets of traces that are collected, namely set  $US_{h_i}$  containing *execution traces* collected from trace points placed in the user-space of the OS and set  $KS_{h_i}$  containing *system status traces* accessible in the kernel space of the OS. These sets of traces from all  $n$  hosts in a dCPS are gathered into larger collections  $US = \{US_{h_1}, \dots, US_{h_n}\}$  and  $KS = \{KS_{h_1}, \dots, KS_{h_n}\}$  that are given as inputs of the CompDSE model derivation algorithms.

#### A. Execution Traces

Every set  $US_{h_i} = \{E_1, \dots, E_m\}$  is a set of execution traces for a dCPS with  $m$  operation modes. Each execution trace  $E_j \in US_{h_i}$  is a sequence of records  $r_k$  collected over time at specific locations in the software code where trace points have been strategically inserted. These trace points can capture timestamps  $ts$ , process names  $pn$ , function calls  $fn$ , the location of the trace point within the function code  $l$ , and the attributes  $A$  of a function based on the function type. Therefore, each record  $r_k$  is represented as a tuple  $r_k = (ts, pn, fn, l, A)$ . In our data collection approach, we collect traces from the *start* and *end* of four types of functions: *send*, *receive*, *trigger*, and *handler*. Functions *send* and *receive* track message exchanges between processes, capturing message identifiers and message sizes (in bytes). Functions *trigger* and *handler* provide insights into internal timer usage

TABLE I: Comparison of DSE Tools for Industrial CPS Modeling

Approach/Tool	Abstraction Level	Scalability for Large Industrial CPS	Degree of Automation	Performance Analysis Type	Handling of Dynamic Behavior
CNMA [15]	Low	No	Semi-automated	Analytical	No
ArchEx 2.0 [16]	Medium	Limited	Semi-automated	Analytical	No
Extended ArchEx 2.0 [23]	Medium	Yes	Semi-automated	Unknown*	No
Matlab Simulink [17] [18]	Low	No	Manual	Co-simulation	No
Extended TTool [19]	Medium	Limited	Semi-automated	Simulation-based	No
FMC [20]	Medium	Limited	Semi-automated	Co-simulation	No
xCPS [21]	Low	Limited	Semi-automated	Constraint-based	Limited
iCyPhy [22]	Low	No	Semi-automated	Formal verification	No
CompDSE	High	Yes	Fully automated	Simulation-based	Yes

\* Performance analysis in [23] is unclear as the focus is on structural validation, not execution-time evaluation.

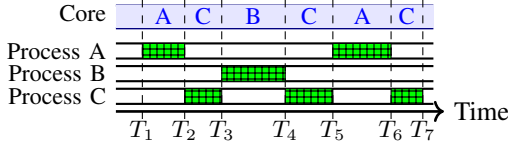


Fig. 1: Deriving OnCPU timing by using context switch events

in a process, capturing timer identifiers and durations (in seconds).

Any message or timer identifier  $id$  includes Lamport timestamp  $lmp$ . Lamport time is a logical clock algorithm used in distributed systems to order events and ensure a consistent sequence of operations across different nodes without relying on synchronized physical clocks [24]. Besides  $lmp$ , for message exchanges, to derive the inter-host communications, we also need to collect the IP address  $ip$  and port number  $prt$  for both source and destination processes in the  $send$  and  $receive$  functions. Therefore, the message identifier  $id$  is represented as a tuple  $id = (lmp, ip_{src}, ip_{dst}, prt_{src}, prt_{dst})$ .

### B. System Status Traces

Every set  $KS_{h_i}$  provides valuable information including the resource utilization (e.g., CPU, memory), process states (idle, interrupted, etc.), and system frequency over time. To derive our workload model, these traces are vital for pinpointing periods when processes are actively executed on a CPU (OnCPU status). Identifying these periods relies on collecting specific kernel events, with context switches being key among them. These context switch events indicate when a process is scheduled onto the CPU core and when it switches to another process, highlighting the active CPU running periods.

Fig. 1 visualizes the OnCPU status derived from context switch events on a CPU core. The transitions between processes on the core are indicated by changes in the process labels (A, B, C), and the green bars represent the periods when each process is actively running on the CPU. The timing of OnCPU status for a process begins when it is scheduled onto the core and ends when the core switches to another process.

In Linux OS, tools like Trace Compass [25] simplify the analysis of kernel traces by using predefined analysis models to extract OnCPU timings. However, on platforms using VxWorks OS, analysts usually have to manually develop these models by examining the collected traces. Given the heterogeneous nature of a dCPS, we need different standard tools to

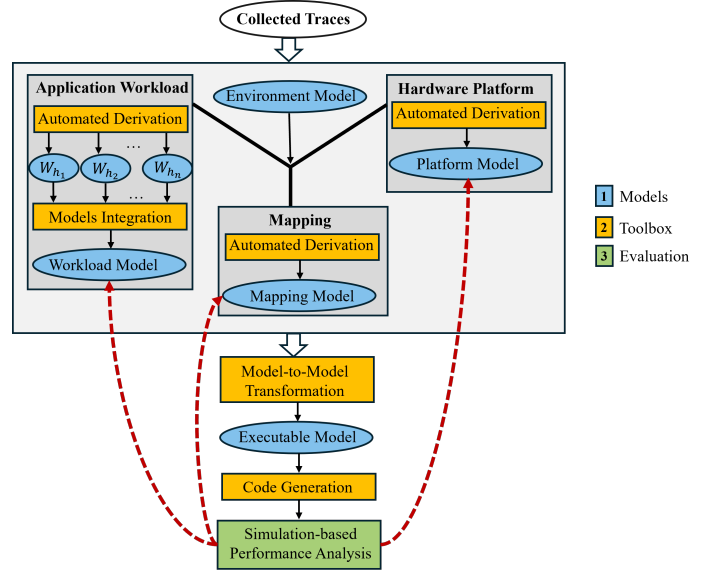


Fig. 2: The CompDSE Methodology

collect execution and system status traces. For example, we utilize LTTng [26] for Linux server-based systems and Wind River System Viewer [27] for VxWorks embedded real-time systems in the ASML Twinscan machine. These tools record events such as system calls, interrupts, context switches, and message exchanges, thereby allowing monitoring and analysis of system resources and application behaviors.

## IV. OVERVIEW OF THE COMPDSE METHODOLOGY

Our CompDSE methodology, illustrated in Fig. 2, leverages the Y-chart approach [6] to systematically define and automatically derive abstract models for the application workload, computing hardware platform architecture, and workload-to-platform mapping. Additionally, it incorporates a model to account for the effects of the physical environment on computing processes. These models are derived from collected traces, described in Section III, and are transformed into abstract executable instances for discrete event simulation. This transformation enables simulation-based performance analysis, allowing us to explore the design space by independently modifying each model.

The CompDSE methodology consists of three main components highlighted in different colors in Fig. 2:

- *Models*: Representations of the application workload, computing hardware platform, mapping, and the environmental influence, which collectively define the behavior of a dCPS. The Application Workload Model captures computation patterns, inter-process communications, and timing dependencies, representing software behavior. The Computing Hardware Platform Model abstracts system resources, including CPU specifications, scheduling policies, and network interconnects, enabling performance evaluation under different configurations. The Mapping Model defines the allocation of software processes to computing hosts, determining the execution impact of deployment choices on overall system performance. The Environmental Influence Model accounts for delays introduced by the physical environment’s interaction with computing processes, ensuring that external influences on execution behavior are properly represented in the simulation;
- *Toolbox*: A set of algorithms for automated model derivation and integration, model-to-model transformation, and code generation. The automated model derivation and integration algorithms extract the application workload model, hardware platform model, and mapping model from execution traces. The model-to-model transformation algorithms transform these formal models into abstract executable instances, while the code generation algorithms further translate the executable model into simulator-specific code, establishing the simulation environment for DSE;
- *Evaluation*: Performance evaluation of a single design point (system configuration) through simulation-based analysis within the prepared simulation environment.

The red feedback arrows in Fig. 2 represent the DSE process, which involves searching the design space by assessing different design points, i.e., different application realizations, computing hardware platforms, and mappings.

In the remainder of this paper, we explain the aforementioned components in detail. First, we provide detailed descriptions of the models (Section V). Then, we examine the methods and algorithms used in the toolbox (Section VI). Finally, we evaluate alternative design points for an ASML lithography machine (Section VII).

## V. MODELS USED IN COMPDSE

In this section, we define and present the abstract models, essential for the CompDSE methodology based on the Y-chart approach, in a semi-formal manner.

### A. Application Workload Model

As shown in Fig. 2, our workload model integrates several workload models  $W_{h_i}$ , where each  $W_{h_i}$  is derived from traces corresponding to host  $h_i$  in a dCPS. For a detailed definition of  $W_{h_i}$ , please refer to [7]. Since this work provides a complete explanation of the automated workload model derivation, we do not repeat those details here but instead describe how our integrated workload model  $WK$  extends it. The super graph  $WK$  combines the workload graphs of all hosts into a single

overarching structure. The super graph  $WK$  is represented by the tuple  $(P, C)$ , where similar to [7],  $P$  is a set of processes modeling the corresponding software processes in the dCPS software infrastructure distributed across multiple hosts, while  $C$  represents a set of communication channels that model the exchange of control or data messages between specific source ( $src$ ) and destination ( $dst$ ) processes.

Each process  $p_i \in P$  is defined as a set of modes, where every mode  $M_q$  is a finite sequence of coarse-grained abstract events that models the types of actions performed during a single operation of the process, arranged in chronological order. These events are categorized into the following event types:

- **Computation Event**: Models computation actions with an abstract workload described by a signature ( $sig$ ). In our model, the signature is the number of cycles a process has been actively running on a CPU within a given time frame. It is computed as:

$$sig = T_{active} \cdot f_{CPU} \quad (1)$$

where  $T_{active}$  is the time the process was actively running on the CPU, and  $f_{CPU}$  is the CPU clock frequency.

- **Communication Event**: Models message exchanges, categorized as *Write* (sending) or *Read* (receiving) events. Each event includes the corresponding channel  $ch \in C$ , message size denoted as  $size$  in bytes, and a signature  $sig$ . The signature represents the abstract computation workload in the source/destination process needed for writing/reading. This signature is similar to the signature used for computation events.
- **Timer Event**: Internal triggers that originate within a process and initiate a computation or communication event after a specific amount of time has elapsed. A timer event is classified as either *timer setter* or *timer handler*. Timer setter event sets a timer with an absolute time value, and once that time has elapsed, a corresponding timer handler is triggered to initiate other events. Each timer event has a timer identifier  $id$  and also duration time  $t$  in seconds.

### B. Computing Hardware Platform Model

We model the computing hardware platform of a dCPS as the tuple  $PL = (H, N, L)$ . Set  $H = \{h_1, \dots, h_{|H|}\}$  is the set of hosts that represent the computing subsystems in the dCPS hardware platform, and set  $N = \{swch_1, \dots, swch_{|N|}\}$  is the set of switches that facilitate network communication among the hosts. Set  $L$  consists of tuples  $(h_i, swch_j)$ , where each tuple specifies that host  $h_i \in H$  is connected to a network via switch  $swch_j \in N$ . Each host  $h_i$  is defined as a tuple  $(ip, sch, crs)$ , where  $ip$  denotes a set of IP addresses used to identify  $h_i$  in the network,  $sch$  specifies the policy used to schedule the processes running on  $h_i$ , and  $crs$  is the set of cores available in  $h_i$ , each with its frequency  $f$ . Each network  $swch_j$  is defined as a tuple  $(sbn, bw)$ , where  $sbn$  indicates the subnet mask of the switch and  $bw$  is the switch bandwidth.

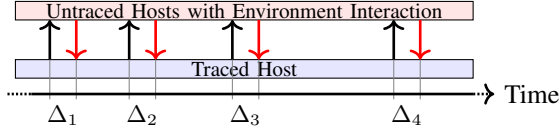


Fig. 3: Modeling the impact of physical environment

### C. Mapping Model

We model the mapping associated with a dCPS configuration as the set  $MP = \{MP_1, \dots, MP_m\}$ , where  $MP_j \in MP$  specifies the mapping of the application workload in operation mode  $j$  to the computing hardware platform of the dCPS. Every  $MP_j = \{mp_1, \dots, mp_{|P|}\}$  is a set of tuples  $mp = (p, h)$  specifying that process  $p \in P$  is mapped onto host  $h \in H$ , where  $P$  is the set of processes in the integrated workload model  $WK$  and  $H$  is set of hosts in the platform model  $PL$ .

### D. Environmental Influence Model

In dCPS, some embedded hosts interact directly with the physical environment through actuators and sensors. These hosts process real-world inputs and influence system behavior but may not always be fully traced due to system constraints. However, their interactions with traced computing processes introduce timing variations that impact overall system performance. To account for these effects, we introduce the  $\Delta$  parameters approach, which abstractly represents the timing impact of the physical environment on computing processes.

The  $\Delta$  parameters are derived from execution traces by analyzing timestamps of message exchanges between traced computing processes and untraced embedded hosts interacting with the physical environment. As illustrated in Fig. 3, the up arrows represent Write events where a traced process sends messages to an untraced embedded host, while the down arrows represent Read events where the traced process receives responses. Since the exact process on the untraced host is unknown, we identify the corresponding interaction using the combination of IP address and port number, ensuring that each  $\Delta$  parameter is correctly attributed to the same communication flow.

For each Read event, a corresponding  $\Delta$  parameter is computed as the difference between the real timestamp of the Read event and the preceding Write event, specifically for the same communication flow:

$$\Delta_i = T_{\text{read},i}^{pt} - T_{\text{write},i}^{pt} \quad (2)$$

where  $T_{\text{write},i}^{pt}$  is the timestamp of the Write event sent from a specific traced process  $p_t$  on a traced host, and  $T_{\text{read},i}^{pt}$  is the timestamp of the corresponding Read event received by the same traced process  $p_t$ .

This approach enables us to model the influence of untraced embedded hosts and their interactions with the physical environment in system simulations, ensuring a more accurate representation of real-world delays.

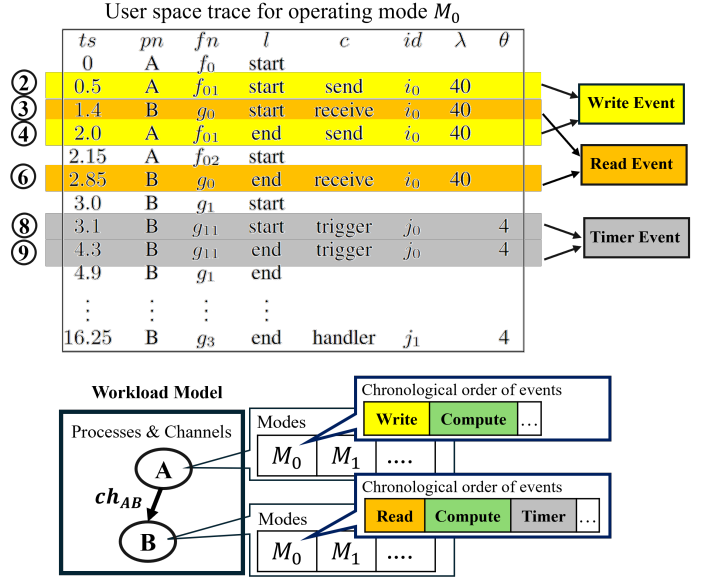


Fig. 4: Workload Model Derivation for each Host

## VI. THE COMPDSE TOOLBOX

In this section, we present the CompDSE toolbox that contains several algorithms devised and implemented to support the modeling and exploration process.

### A. Automated Derivation and Integration

As shown in Fig. 2, the Automated Derivation algorithms are used to automatically derive the required workload models  $W_{h_i}$ , platform model  $PL$ , and mapping model  $MP$  from traces. Additionally, the Model Integration algorithm combines the workload models  $W_{h_i}$  from each host  $h_i$  into a comprehensive application workload model  $WK$ . Here, we outline the operational details of these algorithms, thereby describing their main functionality.

1) **Application Workload:** The formal procedure for the derivation of workload models  $W_{h_i}$  from traces and their integration into application workload model  $WK$  is outlined in Algorithm 1. This algorithm takes sets of execution traces  $US$  and system status traces  $KS$  as inputs, and generates a comprehensive workload model  $WK$  represented as a set of processes  $P$  communicating via set of channels  $C$ . The algorithm begins with deriving the workload model  $W_{h_i}$  for each host (Lines 2-3) using the Workload Model Derivation (WLMD) approach described in detail in [7]. This is followed by the integration of the individual models  $W_{h_i}$  into a unified application workload model encompassing all traced hosts (Lines 4-12). Below, we provide a brief overview of the WLMD approach and an explanation of the integration steps in Algorithm 1.

**Workload Model Derivation (WLMD):** The WLMD approach and its implementation, presented in [7], is used to automatically derive the workload model  $W_{h_i}$  for each host  $h_i$  from the set of execution traces  $US_{h_i}$  across all operation modes. Fig. 4 illustrates the key concepts of this approach, showing an excerpt from execution trace  $E_0$  corresponding to

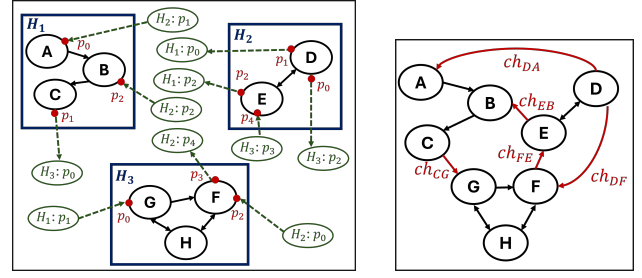
operation mode  $M_0$ . Some important records in trace  $E_0$  are highlighted and numbered (see circles 2 to 9 in the figure).

Initially, WLMD analyzes the execution traces in set  $US_{h_i}$  to create set of processes  $W_{h_i}.P$  and identify message exchanges, facilitating the creation of the set of communication channels  $W_{h_i}.C$ . Considering the example in Fig. 4, since record 2 is a *send* function from process  $A$ , and record 3 is a *receive* function to process  $B$  with the same message identifier  $i_0$ , WLMD creates processes  $A$  and  $B$  as well as communication channel  $Ch_{AB}$  between them. Subsequently, the *send* and *receive* functions are translated into *Write* and *Read* communication events within workload model  $W_{h_i}$ , as shown for processes  $A$  (records 2,4) and  $B$  (records 3,6). These events are then incorporated into the chronological order of events for processes  $A$  and  $B$  in mode  $M_0$  as visualized by the yellow and orange boxes in Fig. 4. Timer events, as exemplified for process  $B$  (records 8-9), are also integrated into the workload model. In addition, after creating the communication and timer events (yellow, orange, and gray boxes in Fig. 4), WLMD places the computation events (green boxes) in the intervals between them. Finally, to calculate the signature (*sig*) of both computation and communication events, WLMD uses the set of system status traces  $KS_{h_i}$ . It considers periods when the process is actively executed on a CPU (OnCPU status), multiplying these periods by the relevant CPU frequency and accumulating them to obtain the total number of CPU clock cycles for each event.

**Workload Model Integration:** As explained above, after deriving the workload model for each host  $W_{h_i}$ , the intra-host communication among the processes running on host  $h_i$  is well-defined within  $W_{h_i}$  by its set of communication channels  $W_{h_i}.C$ . However, inter-host communication remains incomplete because the source or destination processes on other hosts have not been identified, as illustrated in Fig. 5a. This figure shows a simplified computing system consisting of processes  $A$  to  $H$  distributed across three hosts  $H_1$ ,  $H_2$ , and  $H_3$ . The dashed green arrows represent inter-host communication, where the source or destination process needs to be identified. The red dots on the processes indicate the ports involved in inter-host communications, which must be considered when identifying the communication channels.

To address this, Algorithm 1 first identifies inter-host communications based on workload model  $W_{h_i}$  of each host  $h_i$ . It examines each communication channel  $ch_j \in W_{h_i}.C$  to determine if the source or destination host differs from  $h_i$  (Lines 5-10), indicating connections between processes running on different hosts. The procedure *findPeerProc* (Lines 8 and 10) is then employed to resolve the identities of these unknown source or destination processes. After completing the information for inter-host communications, the sets of processes  $W_{h_i}.P$  and channels  $W_{h_i}.C$  for all traced hosts  $h_i$  are integrated into the sets  $P$  and  $C$ , respectively (Lines 11-12), thereby creating the comprehensive application workload model  $WK = (P, C)$ , as shown in Fig. 5b, where the previously unknown sources or destinations of inter-host communication in Fig. 5a are identified (indicated by red arrows).

The aforementioned *findPeerProc* procedure is detailed in Algorithm 2. It takes as inputs workload model  $W_h$  of host



(a) Distributed Workload Models (b) Integrated Workload Model

Fig. 5: Workload Model Integration

---

**Algorithm 1: Derive and Integrate Workload Models**

---

**Input :**  $US = \{US_{h_1}, \dots, US_{h_n}\}, KS = \{KS_{h_1}, \dots, KS_{h_n}\}$   
**Output:**  $WK = (P, C)$

- 1  $P, C \leftarrow \emptyset; W_{tmp} \leftarrow \emptyset;$
- 2 **foreach**  $US_{h_i} \in US$  **do**
- 3    $W_{h_i} = WLM D(US_{h_i}, KS_{h_i}); W_{tmp} \leftarrow W_{tmp} + W_{h_i};$
- 4 **foreach**  $W_{h_i} \in W_{tmp}$  **do**
- 5   **foreach**  $ch_j \in W_{h_i}.C$  **do**
- 6      $h_{src} = ch_j.src.h; h_{des} = ch_j.des.h;$
- 7     **if**  $h_{src} \neq h_i$  **then**
- 8        $ch_j.src.p = \text{findPeerProc}(W_{h_{src}}, ch_j, S);$
- 9     **if**  $h_{des} \neq h_i$  **then**
- 10       $ch_j.des.p = \text{findPeerProc}(W_{h_{dst}}, ch_j, D);$
- 11 **foreach**  $W_{h_i} \in W_{tmp}$  **do**
- 12    $P \leftarrow P + W_{h_i}.P; C \leftarrow C + W_{h_i}.C;$
- 13 **return**  $(P, C)$

---



---

**Algorithm 2: findPeerProc**

---

**Input :**  $W_h, ch, flagT$   
**Output:** *peerProc*

- 1 **foreach**  $ch_i \in W_h.C$  **do**
- 2   **if**  $(ch_i.src.h = ch.src.h \wedge ch_i.dst.h = ch.dst.h) \wedge$   
 $(ch_i.src.prt = ch.src.prt \wedge ch_i.dst.prt = ch.dst.prt)$   
**then**
- 3     **if**  $flagT = S$  **then**
- 4        $\text{return } peerProc \leftarrow ch_i.src.p$
- 5     **else**
- 6        $\text{return } peerProc \leftarrow ch_i.des.p$

---

$h$ , channel  $ch$ , and target flag  $flagT \in \{S, D\}$  that indicates whether the source ( $S$ ) or destination ( $D$ ) process of channel  $ch$  is unknown. The procedure then searches for the unknown process in workload model  $W_h$  by matching the source and destination hosts and ports of the given channel  $ch$  (Lines 1-2). Once a match is found, it returns either the source or destination process as *peerProc* based on the target flag ( $flagT$ ) (Lines 3-6).

2) **Hardware Platform:** The formal procedure for deriving hardware platform model  $PL = \{H, N, L\}$  from traces is outlined in Algorithm 3. This algorithm takes sets of execution traces  $US$  and system status traces  $KS$  as inputs, and generates a platform model  $PL$ , represented by the set of hosts  $H$ , the set of switches  $N$ , and the set of links  $L$ .

**Algorithm 3: Platform Model Derivation from Traces**


---

**Input :**  $US = \{US_{h_1}, \dots, US_{h_n}\}$ ,  $KS = \{KS_{h_1}, \dots, KS_{h_n}\}$   
**Output:**  $PL = (H, N, L)$

```

1  $H, N, L \leftarrow \emptyset;$ 
2 foreach  $US_{h_i} \in US$  do
3    $ip_i = \text{getIPs}(US_{h_i});$ 
4    $nCr = \text{number of cores from } KS_{h_i}; crs_i \leftarrow \emptyset;$ 
5   for  $j = 1$  to  $nCr$  do
6      $f_j = \text{related frequency from } KS_{h_i};$ 
7      $cr_j = (f_j); crs_i \leftarrow crs_i + cr_j;$ 
8    $sch_i \leftarrow \emptyset; h_i = (ip_i, sch_i, crs_i); H \leftarrow H + h_i;$ 
9 foreach  $h_i \in H$  do
10  foreach  $ip_j \in h_i.ip$  do
11     $sbn_j = \text{subnetMask}(ip_j);$ 
12    if  $\nexists swch_j \in N$  such that  $swch_j.sbn = sbn_j$  then
13       $bw_j = 0; swch_j = (sbn_j, bw_j);$ 
14       $N \leftarrow N + swch_j;$ 
15     $l_j = (h_i, swch_j); L \leftarrow L + l_j;$ 
16 return  $(H, N, L)$ 

```

---

**Algorithm 4: getIPs**


---

**Input :**  $US_h$   
**Output:**  $ipSet$

```

1  $ipSet \leftarrow \emptyset;$ 
2 foreach  $r_k \in US_h.E_1$  do
3   if  $r_k.A.f_n = \text{send}$  then
4      $ip_k = r_k.A.id.ip_{src};$ 
5     if  $ip_k \notin ipSet$  then
6        $ipSet \leftarrow ipSet + ip_k;$ 
7 return  $ipSet$ 

```

---

The algorithm begins by creating the set of traced hosts  $H$  (Lines 2-8). For each set of execution traces  $US_{h_i}$  of host  $h_i$ , the algorithm starts by deriving the set of IP addresses  $ip_i$ , associated with host  $h_i$ , using the *getIP* procedure (Line 3). Next, the algorithm proceeds by utilizing the set of system status traces  $KS_{h_i}$  of host  $h_i$  to determine the number of cores  $nCr$  and their frequencies  $f_j$  (Lines 4-7). These frequencies serve as initial values in the model and can be adjusted during the exploration phase. Similarly, the policy  $sch_i$  for scheduling the tasks running on host  $h_i$  is initially set to unknown (Line 8) and it is tuned based on the designer's preferences during the exploration phase.

Once the set of hosts  $H$  is finalized (Line 8), Algorithm 3 proceeds by iterating over each host  $h_i \in H$ . For every IP address  $ip_j$  associated with  $h_i$ , the algorithm calculates the subnet mask  $sbn_j$  (Line 11). Then, it checks if there exists a switch  $swch_j$  in the set of switches  $N$  that shares the same subnet mask  $sbn_j$  (Line 12). If such a switch does not exist, the algorithm creates a new switch  $swch_j$  with  $sbn_j$ , sets its initial bandwidth  $bw_j$  to zero, and adds the  $swch_j$  to  $N$  (Lines 13-14). Finally, a link  $l_j$  is created and added to  $L$  specifying that host  $h_i$  is connected to switch  $swch_j$  (Line 15). Note, host  $h_i$  may be connected to multiple switches if it has multiple IP addresses  $ip_j \in h_i.ip$  (Line 10).

The aforementioned *getIP* procedure is detailed in Algorithm 4. It takes execution trace  $US_h$  of host  $h$  as an input and

**Algorithm 5: Mapping Model Derivation from Traces**


---

**Input :**  $US = \{US_{h_1}, \dots, US_{h_n}\}$   
**Output:**  $MP = \{MP_1, \dots, MP_m\}$

```

1  $MP \leftarrow \emptyset;$ 
2 for  $j = 1$  to  $m$  do
3    $MP_j \leftarrow \emptyset; MP \leftarrow MP + MP_j;$ 
4 foreach  $US_{h_i} \in US$  do
5   foreach  $E_j \in US_{h_i}$  do
6     foreach  $r_k \in E_j$  do
7        $mp_k = (p_{r_k.pn}, h_i); MP_j \leftarrow MP_j + mp_k;$ 
8 return  $MP$ 

```

---

returns set  $ipSet$  of unique IP addresses associated with host  $h$ . First, the algorithm initializes an empty set  $ipSet$  (Line 1). Then, for each record in the execution trace  $US_h$ , it checks whether the function name in the attributes is *send* (Lines 2-3). If so, it extracts the source IP address and adds it to  $ipSet$  (Lines 4-6). The algorithm considers records from operation mode  $E_1$  because the hardware configuration remains static across all modes; therefore, selecting  $E_1$  simplifies our analysis without sacrificing accuracy. Finally, it returns the complete set of unique IP addresses of the host (Line 7).

3) **Mapping:** The formal procedure for deriving the mapping model from traces is outlined in Algorithm 5. This algorithm takes set  $US$  of execution traces as an input, and generates a mapping model  $MP$  which specifies the mapping of the application workload, in every operation mode, onto the computing hardware platform of a dCPS. During DSE, this initial mapping model can be modified in order to evaluate alternative design choices, such as moving application processes from one host to another in order to better balance the application workload over the hosts.

Algorithm 5 begins by initializing an empty mapping set  $MP_j$  for each operation mode  $M_j$  (Lines 2-3). It then examines the records  $r_k$  in every execution trace  $E_j \in US_{h_i}$  associated with  $M_j$  and host  $h_i$  (Lines 4-6) to identify every process  $p_{r_k.pn}$  running on  $h_i$ . This examination results in creating tuples  $(p_{r_k.pn}, h_i)$  specifying that process  $p_{r_k.pn}$  is mapped onto host  $h_i$  in operation mode  $M_j$ . The created tuples are subsequently added to the mapping set  $MP_j$  associated with operation mode  $M_j$  (Line 7).

**B. Model-to-Model Transformation and Code Generation**

This section describes how the automatically derived formal Workload, Platform, and Mapping models are transformed into abstract executable instances, called model *entities*, designed for use with any discrete event simulator. This transformation is facilitated by specialized algorithms that generate a separate model entity for every process  $p_i \in P$  in the workload model  $WK$ , for every host  $h_i \in H$  and switch  $swch_j \in N$  in the platform model  $PL$ , and for the mapping  $MP$ . Within this framework, all these workload, mapping, and platform entities interact and synchronize through event-driven messaging, thereby operating concurrently. Each model entity maintains a queue, denoted as  $Q$ , to buffer incoming event messages from



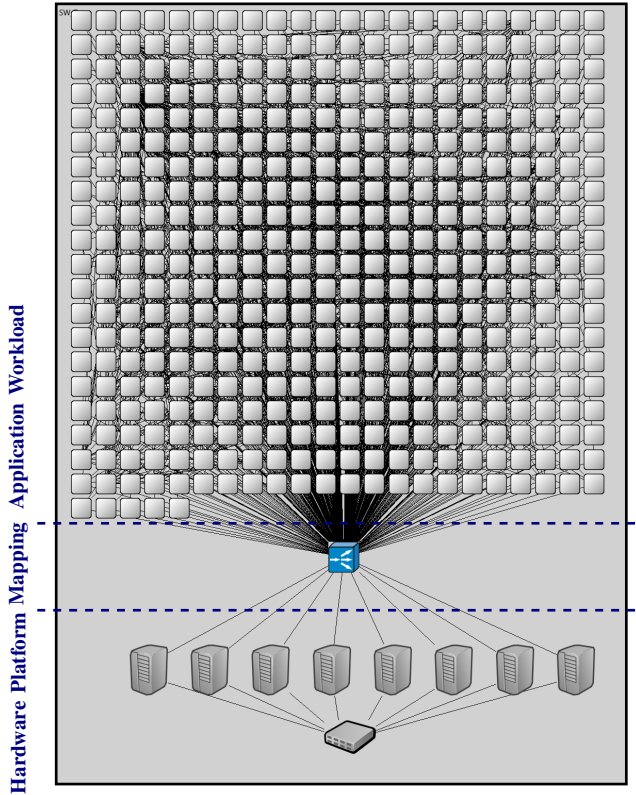


Fig. 6: Application Workload, Computing Hardware Platform, and Mapping layers in OMNET++

other model entities. All interacting entities form the abstract executable model of a dCPS.

After obtaining the executable model which is independent of any discrete event simulation environment, a code generation step is necessary to transform the executable model into specific executable code for a selected simulator, thereby establishing our final simulation environment for DSE. For our case, we have chosen the OMNET++ simulation environment [28] due to its robustness, scalability, and suitability for simulating complex distributed systems and large-scale networks. Fig. 6 shows the environment of our simulation, visualized as three layers of entities: the application workload, the mapping, and the computing hardware platform. Below, we further elaborate on the behavior of each abstract executable model entity.

1) **Application Workload:** In the executable application workload model (the top part of Fig. 6), each process is represented as an entity with individual event files for each operation mode, containing abstract events arranged chronologically along with their attributes. All process entities are connected to the mapper entity (the blue box in Fig. 6). Listing 1 sketches the behavior of a process entity as a pseudo code. The process entity retrieves events from its event file (Line 16). If the retrieved event is a Write, Compute, or Read event then it requires hardware platform resources for execution. Thus, the process entity sends a request message to the mapper entity (Lines 23-26) and waits until receiving a "done" message, allowing to proceed to the next event

```

1 runNext = True; waitForTimer = False
2 while (True) {
3   if (Q is not empty) { // Check for received events
4     event = take the first element of Q;
5     if (sender of the event is mapper) //done message
6       runNext = True;
7     else { //timeoff event
8       add the ID of event to the timeout list;
9       if (waitForTimer) {
10        runNext = True;
11        waitForTimer = False;}
12    }}
13 // send request to the mapper
14 if (runNext and not waitForTimer) {
15   runNext = False;
16   nextEvent = get the next event of sequence;
17   if (type of nextEvent is setTimer)
18     Schedule a timeoff message that will trigger
19     after nextEvent time;
20   else if (type of nextEvent is timerHandler) {
21     if (timeout contains the ID of nextEvent)
22       runNext = True;
23   }
24   else { //Read, Write and Compute events
25     send nextEvent to the mapper;
26     waitForTimer = True;
27     runNext = False;}
28 }

```

Listing 1: Execution behavior of a Process entity

```

1 while (True) {
2   if (Q is not empty) { // Check for received events
3     event = take the first element of Q;
4     if (sender of event is a process) {
5       s_Host = determine the host mapped to the
6       sender process of event;
7       send event to s_Host;}
8     else // The event is from a host
9       send a "done" message to the source process
10      of event;
11 }

```

Listing 2: Execution behavior of the Mapper entity

(Lines 5-6). Timer events, however, do not require a request sent to the mapper entity. Instead, for timer events, the process entity schedules a "timeoff" message to be sent to itself after a specified timer duration (Lines 17-18). Once the scheduled "timeoff" message is received, the timer is considered off (Lines 7-11). If the event is a timer handler and the process entity has already received the "timeoff" message, it proceeds to retrieve the next event (Lines 20-21). If the "timeoff" message has not yet been received, the process waits until it arrives.

2) **Mapping:** The abstract executable model contains one mapper entity (the middle part of Fig. 6) which is connected to all process entities (the top part of Fig. 6) and host entities (the bottom part of Fig. 6). As explained above, process entities send Write, Compute, and Read request messages to the mapper entity, which subsequently forwards these requests to the corresponding host entities based on resource allocations defined in a mapper file derived from the tuples  $(p_{rk}, pn, h_i)$  in the mapping model  $MP$ . In addition to this forwarding of request messages, the mapper also manages the transmission of "done" messages from host entities back to the requesting process entities indicating that the requests have been completed by the hosts. Listing 2 illustrates the behavior of the mapper entity as a pseudo code.

3) **Hardware Platform:** In the executable hardware platform model (the bottom part of Fig. 6), each host and switch is represented as an entity operating concurrently with other entities. All host and switch entities are interconnected according to the set of links  $L$  specified in the platform model  $PL$ . The behavior of a host entity and a switch entity are sketched as a pseudo code in Listing 3 and Listing 4, respectively. The host entity receives Write, Compute, and Read request messages (events) from the mapper (Lines 4-5 in Listing 3). If the request message is of type Compute or Write event, the host entity proceeds with scheduling the event for execution (Lines 13-14). For a request message of type Read event, the host entity first checks (Lines 6-10 in Listing 3) whether a data message, sent from the corresponding source process via the network of switch entities, has been received. If the data message has not been received yet, the host entity waits until it arrives before scheduling the Read event for execution (Line 24). Once the execution of any requested event is completed, the host entity notifies the requesting process entity by sending a 'done' message through the mapper entity (Lines 17, 22).

During the execution of an event, just a time delay is performed, based on the event's signature ( $sig$  described in Section V-A), in order to abstractly simulate the compute workload of the event. For Write events, after simulating its compute workload by the corresponding delay, the event is forwarded to a switch entity in the network (Lines 15-16 in Listing 3). The switch entity applies a time delay and transmits the event to the destination host entity (Lines 9-10 in Listing 4). This time delay abstractly simulates the network congestion delay and transmission delay of a data message communicated between hosts via the network (Line 8 in Listing 4). In our abstract model, the network congestion delay is the trimmed mean interval between sent and received messages taken from execution traces  $US$ , excluding the transmission delay. The trimmed mean is computed as:

$$T_\alpha(X) = \frac{1}{(1-2\alpha)N} \sum_{i=\alpha N+1}^{(1-\alpha)N} x_{(i)} \quad (3)$$

where  $X = \{x_1, x_2, \dots, x_N\}$  is the ordered dataset sorted in ascending order,  $\alpha$  is the proportion of data to be trimmed from both ends,  $N$  is the total number of values in the dataset, and  $x_{(i)}$  represents the  $i$ -th smallest value in the sorted dataset.

In our model, we set  $\alpha = 0.1$ , meaning that the top 10% and bottom 10% of values are removed to mitigate the influence of outliers [29]. The transmission delay is calculated by dividing the message size ( $size$  in Section V-A) by the switch bandwidth ( $bw_j$  in Section VI-A2) (Line 7 in Listing 4).

## VII. EVALUATION OF THE COMPDSE METHODOLOGY

In this section, we evaluate the accuracy and efficiency of our CompDSE methodology and demonstrate its applicability on a real industrial case by conducting a small DSE experiment utilizing the OMNET++ simulator for performance analysis.

```

1 while (True) {
2   if (Q is not empty) { // Check for received events
3     event = take the first element of Q;
4     if (sender of event is the mapper)
5       add event to readyQ;
6     else if (sender of event is network) {
7       add event to the receiveList;
8       if (exists an e in waitList and source of e
9         is the same as event source)
10        take e from waitList and add it to
11        readyQ;}
12   }
13   while (readyQ is not empty) {
14     exeEv = take next event from readyQ;
15     if (type of exeEv is not Read) {
16       allocate resource to execute exeEv;
17       if (type of exeEv is Write)
18         send exeEv to network;
19       send "done" message to mapper for exeEv;
20     }
21     else { // Read event
22       if (exists an e in receiveList and source of
23         e is the same as exeEv source) {
24         take e from the receiveList and allocate
25         resource to execute it;
26         send "done" message to mapper for exeEv;}
27     else // Message has not yet arrived
28       add exeEv to waitList;
29   }
30 }

```

Listing 3: Execution behavior of a Host entity

```

1 while (True) {
2   if (Q is not empty) { // Check for received events
3     event= take the first element of Q;
4     add event to the readyQ;
5     while (readyQ is not empty) {
6       event = take the next event from readyQ;
7       transitDelay = divide the message size of
8         event by switch bandwidth;
9       delay = transitDelay + congestDelay;
10      destHost = determine the host mapped to
11        destination process of event;
12      send event to destHost at (simTime + delay);
13    }
14  }
15 }

```

Listing 4: Execution behavior of a Switch entity

### A. Case Study: ASML Twinscan Lithography Machine

ASML lithography machines stand at the forefront of semiconductor manufacturing, playing a pivotal role in the cutting-edge fabrication of silicon wafers. These machines utilize advanced optics and precise positioning of reticles (also known as masks) to transfer circuit patterns onto silicon wafers with remarkable accuracy. As its name suggests, the ASML Twinscan machine utilizes a dual-stage scanning mechanism. In the first stage, various sensors measure the precise location of the wafer. In the second stage, using the information from the first stage and utilizing actuators, the exposure process is executed. Running these two stages simultaneously on a set of wafers enhances production efficiency.

To initiate the wafer processing operation, precise recipes must be defined. These recipes specify the number of batches, the number and source of wafers for each batch, and, most importantly, various exposure and measurement parameters. The diversity in recipes results in different application workloads, which correspond to predefined operation modes.

The cyber part of ASML Twinscan machines consists of a complex software infrastructure distributed across heterogeneous multi-core subsystems connected via networks. These

subsystems, running on various platform operating systems (OS), are classified based on their functionality as either embedded real-time or server-based systems. Within this infrastructure, hundreds of processes operate concurrently, exchanging thousands of messages per second to ensure precise coordination between the subsystems. This high volume of communication and processing adds to the complexity of managing and optimizing system performance, particularly when accounting for the real-time constraints of the embedded systems and the scalability of the server-based components.

### B. Experimental Setup

For our experiment, we have traced a subset of the hosts in the ASML Twinscan machine testbench, including one Linux-based server and several real-time embedded systems running Wind River VxWorks OS. Due to technical limitations such as some hosts being "bare-metal", i.e., not running an OS, it is not possible to trace all hosts. However, we consider the effects of such untraced hosts on the overall dCPS performance during our system simulations by using  $\Delta$  parameters as explained in Section V-D.

During simulation, the simulated timestamp of every Write event, corresponding to a message sent to an untraced host, is recorded. When the simulator encounters a Read event corresponding to a message received from an untraced host, it first computes the difference between the current simulation time and the recorded timestamp of the preceding Write event. This calculated difference represents the elapsed time in the simulation since the preceding Write event occurred. The simulator then subtracts this elapsed time from the  $\Delta$  parameter value corresponding to the Read event in order to determine the remaining time that should be waited before scheduling the Read event for execution. This approach ensures that the impact of the physical environment (untraced hosts) on the dCPS timing performance is accurately represented in the simulation.

### C. Evaluation Results

1) *Efficiency*: The Efficiency of CompDSE is one of its core strengths, facilitating system designers to effectively explore the vast design space of computing subsystems within a complex distributed cyber-physical system (dCPS). Table II provides an overview of the complexity of the automatically derived model, which captures the workload imposed on the traced hosts within the ASML Twinscan machine. It also demonstrates how CompDSE efficiently handles this model complexity.

Table II is divided into two sections. The first section outlines the complexity metrics, including the number of traced hosts, processes, and communication channels. Specifically, the CompDSE methodology is applied to model the ASML Twinscan machine for a wafer batch operation involving five wafers, where **8** hosts are traced, **445** application processes are modeled, and **2057** communication channels between these processes are identified.

The huge number of possibilities (in the range of  $8^{445}$ ) to map the 445 processes onto the 8 hosts creates a design

TABLE II: Complexity of dCPS and efficiency of CompDSE

Complexity Metric	Value
Number of traced hosts	8
Number of Processes	445
Number of Channels	2057
Steps	Time Required
Adding trace points manually	about 5 minutes
Collecting traces automatically	4.33 minutes
Automated models derivation and integration	29.5 hours
Code generation for OMNET++	1.82 minutes
Single point evaluation	54.5 seconds

space of extraordinary size. Furthermore, when considering additional factors - such as the number of cores per host, different operating frequencies, and various scheduling policies - the number of possible configurations expands into an astronomical number of combinations. These numbers highlight the high complexity of the derived model and the corresponding design space, which is typical for industrial-scale dCPS.

The second section in Table II provides the execution time of the various steps within CompDSE, visualized in Fig. 2. The time required to manually add trace points in the software infrastructure depends on the number of trace points needed. For our case, we added trace points at the start and end of four functions within a shared library used by all processes. Adding these 8 trace points takes approximately 5 minutes and needs to be done only once. To ensure that tracing does not affect the system's real-time performance, we monitored the wafer processing time before and after enabling tracing and observed no measurable difference.

Afterward, the time required for trace collection depends on the duration of the application's execution. In our case, processing five wafers takes 4.33 minutes. This trace collection process must be repeated for each operation mode. Automated Model Derivation and Integration take 29.5 hours but needs to be performed only once. The Model-to-Model Transformation and Code Generation for the OMNET++ simulator is completed in about 2 minutes. The simulation to evaluate the performance of one system configuration requires **54.5 seconds**.

Given the high complexity of the aforementioned model, this less-than-a-minute simulation-based performance analysis demonstrates the remarkable efficiency of CompDSE in evaluating a single design configuration. Such rapid evaluation is especially beneficial for industrial dCPS, where quick insights into the performance of complex machines are critical. Moreover, such rapid performance evaluation of a single design configuration is crucial to enable fast exploration of large design spaces. For example, by integrating an automated search algorithm within CompDSE, such as a genetic algorithm, and utilizing the rapid simulation-based performance analysis as a fitness function, we could accelerate significantly the exploration of large design spaces, making the CompDSE methodology even more scalable for future designs of complex dCPS. Integrating an automated search algorithm in CompDSE is out of the scope of this paper.

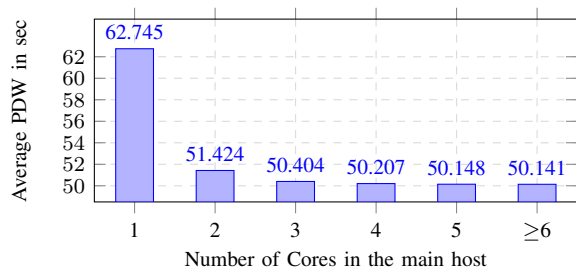


Fig. 7: The impact of the number of cores on PDW

2) *Accuracy*: In our experiment, the average processing duration per wafer (PDW) measured on the real ASML Twinscan machine is **50.548 seconds**, whereas the simulation of our dCPS model of the same machine indicates an average PDW of **50.141 seconds**, resulting in a negligible error of approximately 0.8%. This fact together with the aforementioned less-than-a-minute performance analysis demonstrates the potential of CompDSE for sufficiently accurate/trustworthy and fast exploration of complex dCPS.

3) *Exploration*: In our small DSE experiment, we first explore the effect of varying the number of cores in the main host (the Linux-based server) on the dCPS performance in terms of average PDW. The default configuration of the main host has sixteen cores. However, as shown in Fig. 7, increasing the number of cores beyond six does not lead to any significant changes in the average PDW. When we decrease the number of cores from six to three, the average PDW does not increase significantly, i.e., the increase is in the range of milliseconds. Further decreasing the number of cores leads to a more substantial increase in the average PDW. The above findings suggest that the original design of the main host with sixteen cores in the ASML Twinscan machine might be an overestimation to avoid computing hardware resource bottlenecks. Thus, having a host with eight cores (half of the original design) could lower overall costs without compromising the machine performance, which could be considered in future system designs.

Next, we explore the effect of increasing the number of cores in the traced embedded hosts (the VxWorks-based real-time embedded systems) while keeping the main host with its default configuration. The results show that increasing the number of cores from one to four in these embedded hosts decreases the average PDW by 2.148 seconds. This improvement indicates that while additional cores enable the system to handle more software processes in parallel, the overall impact on the average PDW is limited. This limitation is most likely due to the original design of the software infrastructure running in the ASML Twinscan machine where potential parallel processing in the software is not heavily exploited.

Finally, we examine the impact of varying the clock frequencies of the cores in the traced main and embedded hosts on the average PDW by conducting a series of simulations with different clock frequency settings, supported by the cores, across all hosts. The results, plotted in Fig. 8, show the relationship between the average PDW and clock frequency

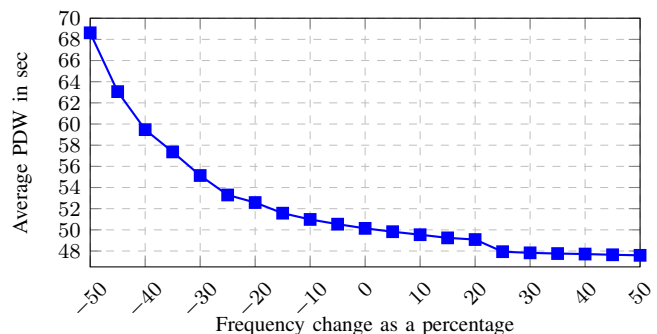


Fig. 8: The impact of the clock frequency of cores on PDW

changes relative to the base clock frequency setting of each host's multi-core CPU. The base frequency setting is denoted as 0 on the horizontal axis, while the average PDW is depicted on the vertical axis. The horizontal axis depicts the clock frequency changes as a percentage of the base clock frequency.

It can be seen in Fig. 8 that increasing the clock frequency of all traced hosts' multi-core CPUs with 25% results in noticeable decrease of the average PDW from 50.141 to 48 seconds. However, increasing further the clock frequency yields only a marginal decrease of the average PDW, often within the millisecond range, as visualized by the curve's plateau in Fig. 8. This plateau suggests that the performance gain in terms of PDW does not scale with the clock frequency increase when it is above 25%. The plateau indicates that the PDW is limited to around 48 seconds. This limitation arises from the performance of some untraced hosts reflected by the  $\Delta$  parameter values discussed in Section VII-B.

## VIII. CONCLUSIONS

In this paper, we introduce CompDSE, a novel methodology for efficient Design Space Exploration (DSE) of computing subsystems within complex distributed cyber-physical systems. CompDSE leverages automated derivation of abstract models for an application workload, a computing hardware platform, and a workload-to-platform mapping from runtime trace data, integrating them into a discrete event simulation environment for performance evaluation. We demonstrate the benefits of CompDSE by applying it on the ASML Twinscan lithography machine. Our exploration of various design configurations to optimize computing resources reveals potential performance improvements. A significant advantage of CompDSE is its ability to evaluate each design scenario in under a minute, demonstrating its suitability/scalability for/to industrial-scale applications. While our small DSE experiment has been done manually, CompDSE is well-suited for integration with search algorithms, such as genetic algorithms, to further enhance its automation.

## REFERENCES

- [1] R. Alur, *Principles of Cyber-Physical Systems*. The MIT Press, 2015.
- [2] S. Acur *et al.*, "Vision and outlook for systems architecting and systems engineering in the high-tech equipment industry," TNO, Tech. Rep. R10542, 2024.
- [3] B. Meier *et al.*, "Htsm systems engineering roadmap," *Tech. rep.*, 2020.

- [4] B. van der Sanden *et al.*, “Model-driven system-performance engineering for cyber-physical systems : Industry session paper,” in *EMSOFT*, pp. 11–22, 2021.
- [5] M. Herget *et al.*, “Design space exploration for distributed cyber-physical systems: State-of-the-art, challenges, and directions,” in *DSD*, pp. 632–640, 2022.
- [6] B. Kienhuis *et al.*, “A methodology to design programmable embedded systems: the y-chart approach,” *SAMOS*, pp. 18–37, 2002.
- [7] F. S. Saadatmand *et al.*, “Automated derivation of application workload models for design space exploration of industrial distributed cyber-physical systems,” *ICPS*, May. 2024.
- [8] P. Terway *et al.*, “Dispatch: design space exploration of cyber-physical systems,” *arXiv preprint arXiv:2009.10214*, 2020.
- [9] R. Weber *et al.*, “Multi-objective design space exploration for cyber-physical systems satisfying hard real-time and reliability constraints,” in *Proceedings of IDEAL*, vol. 14.
- [10] J. Bakakeu *et al.*, “Multi-objective design space exploration for the integration of advanced analytics in cyber-physical production systems,” in *IEEM*, pp. 1866–1873, 2018.
- [11] Y. Vanommeslaeghe *et al.*, “Automatic generation of workflows for efficient design space exploration for cyber-physical systems,” in *iThings, GreenCom, CPSCoM, SmartData, and Cybermatics*, pp. 346–351, 2021.
- [12] M. Thompson *et al.*, “Exploiting domain knowledge in system-level mpsoC design space exploration,” *Journal of Systems Architecture*, vol. 59, no. 7, pp. 351–360, 2013.
- [13] Y. Vanommeslaeghe *et al.*, “Leveraging domain knowledge for the efficient design-space exploration of advanced cyber-physical systems,” in *DSD*, pp. 351–358, 2019.
- [14] Y. Vanommeslaeghe *et al.*, “Ontological reasoning in the design space exploration of advanced cyber-physical systems,” *Microprocessors and Microsystems*, vol. 85, p. 104151, 2021.
- [15] S. Narain *et al.*, “Design space exploration for cyber physical systems,” Perspecta Labs, Tech. Rep., Nov. 2019.
- [16] D. Kirov *et al.*, “Archex: An extensible framework for the exploration of cyber-physical system architectures,” in *DAC*, pp. 1–6, 2017.
- [17] M. Amir *et al.*, “Pareto optimal design space exploration of cyber-physical systems,” *Internet of things*, vol. 12, p. 100308, 2020.
- [18] N. Mühleis *et al.*, “A co-simulation approach for control performance analysis during design space exploration of cyber-physical systems,” *ACM SIGBED Review*, vol. 8, no. 2, pp. 23–26, 2011.
- [19] D. Genius *et al.*, “High-level partitioning and design space exploration for cyber physical systems.” in *MODELSWARD*, pp. 84–91, 2020.
- [20] 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.
- [21] S. Adyanthaya *et al.*, “xcps: A tool to explore cyber physical systems,” *ACM sigbed review*, vol. 14, no. 1, pp. 81–95, 2017.
- [22] P. Nuzzo *et al.*, “Methodology and tools for next generation cyber-physical systems: The icyphy approach,” in *INCOSE*, vol. 25, no. 1, pp. 235–249, 2015.
- [23] Y. Xiao *et al.*, “Efficient exploration of cyber-physical system architectures using contracts and subgraph isomorphism,” in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6. IEEE, 2024.
- [24] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” in *Concurrency: the Works of Leslie Lamport*, pp. 179–196, 2019.
- [25] Trace Compass Developers, “Trace Compass [Software],” <https://tracecompass.org/>, 2024.
- [26] M. Desnoyers *et al.*, “The lttng tracer: A low impact performance and behavior monitor for gnu/linux,” in *OLS*, vol. 2006, pp. 209–224, 2006.
- [27] D. Wilner, “Windview: a tool for understanding real-time embedded software through system vizualization,” *ACM Sigplan Notices*, vol. 30, no. 11, pp. 117–123, 1995.
- [28] A. Varga *et al.*, “An overview of the omnet++ simulation environment,” in *SIMUTools*, 2010.
- [29] M. L. Jones, “Trimmed means and the robustness of estimation,” *Journal of Statistical Research*, vol. 8, no. 2, pp. 200–210, 2014.