IET Cyber-Physical Systems: Theory & Applications

WILEY

The Institution of Engineering and Technology

ORIGINAL RESEARCH OPEN ACCESS

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

Faezeh Sadat Saadatmand[1] | Todor Stefanov[1] | Ignacio González Alonso[2] | Andy D. Pimentel[3] | Benny Akesson[3,4]

[1]Leiden Institute of Advanced Computer Science (LIACS), Leiden University, Leiden, the Netherlands | [2]ASML B.V., Veldhoven, the Netherlands | [3]Informatics Institute (IvI), University of Amsterdam, Amsterdam, the Netherlands | [4]TNO Embedded Systems Innovation (TNO-ESI), Eindhoven, the Netherlands

Correspondence: Faezeh Sadat Saadatmand (f.saadatmand@liacs.leidenuniv.nl)

Received: 30 October 2024 | **Revised:** 18 March 2025 | **Accepted:** 26 April 2025

Handling Editor: Junlong Zhou

Funding: This research was supported by the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO) under project number 17930.

Keywords: cyber-physical systems | design space exploration | embedded systems | simulation | system modelling

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 nonfunctional 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, that is, the computing subsystems within dCPS. CompDSE defines and utilises 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 optimising 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.

1 | Introduction

Distributed 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 optimisation of the cyber components, becomes more challenging [1]. The cyber components, consisting of distributed software processes running on multicore or manycore 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 nonfunctional 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 optimise their next-generation designs, ensuring enhanced performance and adaptability in the face of evolving technological requirements [4]. Therefore, DSE methods that can capture the behaviour 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.

Although some initial research on automated workload modelling for industrial dCPS exists [7], a comprehensive methodology and tool support that address the challenges of DSE, in terms of abstract, yet sufficiently accurate modelling, 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. 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.

Although 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 summarised 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 Δ-parameters approach to capture the impact of delays in the physical environment.

The remainder of the paper is organised as follows: In Section 2, we discuss related work concerning the DSE of dCPS. Section 3 provides details on the data that needs to be collected at runtime for deriving the models used in CompDSE. In Section 4, we present our CompDSE methodology. Sections 5 and 6 explain the models and the toolbox utilised in CompDSE, respectively. In Section 7, we demonstrate the merits of CompDSE by exploring alternative design points for lithography machines. Finally, we conclude the paper in Section 8.

2 | Related Work

Design space exploration (DSE) for cyber-physical systems (CPS) has been extensively studied, leading to various methodologies and tools. Although the entire process can be broken down into multiple steps [5], it can be abstractly distinguished into two primary components: (1) system modelling 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 modelling 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; that is, 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 multiobjective optimisation [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 modelling 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 modelling and analysis methods for DSE. Some of these are based on static methods such as 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 utilises 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 the study by Amir and Givargis [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 the study by Genius et al. [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 modelling 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 behaviour in co-simulation. For instance, A. Canedo and J. H. Richter [20] introduce a functional modelling compiler (FMC) that synthesises technology-dependent solutions for architectural design space exploration using multidisciplinary simulations such as AMESim and Modelica. Although 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 modelling 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 such as xCPS [21] and iCyPhy [22] incorporate techniques to improve CPS modelling, 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 behavioural modelling of cyber and physical components, which poses significant challenges for large-scale industrial CPS such as 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 behavioural modelling 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 behaviour of complex dCPS for DSE without relying on manual efforts. Moreover, most applications modelled by the aforementioned tools exhibit static behaviour, 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 behaviour, such as adaptive processing loads or various software/hardware configurations. These dynamic behaviours necessitate a tool that supports flexible modelling and exploration with sufficient accuracy and speed.

A recent approach introduces an efficient method for CPS architecture exploration using contract-based design and subgraph 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 analyse how different configurations impact execution time and system performance, making it unsuitable for performance-driven exploration in complex industrial CPS.

The work presented by Saadatmand et al. [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 modelling 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 modelling 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 Δ-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 1, which distinguishes them based on model abstraction level, scalability for large industrial CPS, degree of automation, performance analysis type and handling of dynamic behaviour. Unlike many existing tools, CompDSE offers higher abstraction, full automation and explicit support for dynamic behaviour, making it well suited for large-scale industrial dCPS.

TABLE 1 | Comparison of DSE tools for industrial CPS modelling.

Approach/tool	Abstraction level	Scalability for large industrial CPS	Degree of automation	Performance analysis type	Handling of dynamic behaviour
CNMA [15]	Low	No	Semiautomated	Analytical	No
ArchEx 2.0 [16]	Medium	Limited	Semiautomated	Analytical	No
Extended ArchEx 2.0 [23]	Medium	Yes	Semiautomated	Unknown[a]	No
Matlab Simulink [17, 18]	Low	No	Manual	Co-simulation	No
Extended TTool [19]	Medium	Limited	Semiautomated	Simulation based	No
FMC [20]	Medium	Limited	Semiautomated	Co-simulation	No
xCPS [21]	Low	Limited	Semiautomated	Constraint based	Limited
iCyPhy [22]	Low	No	Semiautomated	Formal verification	No
CompDSE	High	Yes	Fully automated	Simulation based	Yes

[a]Performance analysis in the study by Xiao et al. [23] is unclear as the focus is on structural validation, not execution time evaluation.

3 | 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 $US = \{US_{h_1}, ..., US_{h_n}\}$ and $KS = \{KS_{h_1}, ..., KS_{h_n}\}$ that are given as inputs of the CompDSE model derivation algorithms.

3.1 | Execution Traces

Every set $US_{h_i} = \{E_1, ..., E_m\}$ 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 in a process, capturing timer identifiers and durations (in seconds).

Any message or timer identifier *id* includes the 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 synchronised physical clocks [24]. Besides *lmp*, for message exchanges, to derive the interhost 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 $id = (lmp, ip_{src}, ip_{dst}, prt_{src}, prt_{dst})$.

3.2 | System Status Traces

Every set KS_{h_i} provides valuable information including the resource utilisation (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.

Figure 1 visualises 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 such as 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 collect execution and system status traces. For example, we utilise 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 behaviours.

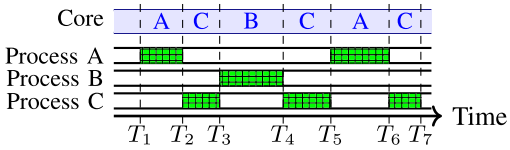


FIGURE 1 | Deriving `onCPU` timing by using context switch events.

4 | Overview of the CompDSE Methodology

Our CompDSE methodology, illustrated in Figure 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 3, 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 colours in Figure 2:

- *Models*: Representations of the application workload, computing hardware platform, mapping and the environmental influence, which collectively define the behaviour of a dCPS. The application workload model captures computation patterns, interprocess communications and timing dependencies, representing software behaviour. 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 behaviour 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, whereas 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.

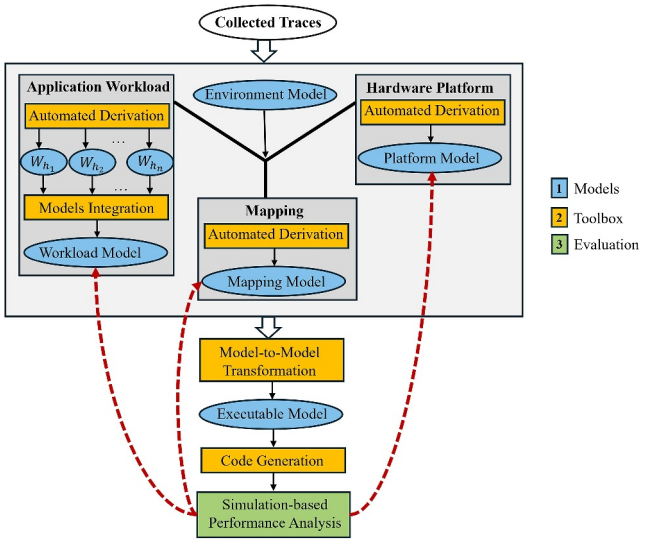


FIGURE 2 | The CompDSE methodology.

The red feedback arrows in Figure 2 represent the DSE process, which involves searching the design space by assessing different design points, that is, different application realisations, 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 5). Then, we examine the methods and algorithms used in the toolbox (Section 6). Finally, we evaluate alternative design points for an ASML lithography machine (Section 7).

5 | 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 semiformal manner.

5.1 | Application Workload Model

As shown in Figure 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 the study by Saadatmand et al. [7]. Because 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 supergraph WK combines the workload graphs of all hosts into a single overarching structure. The supergraph WK is represented by the tuple (P, C), where, similar to the study by Saadatmand et al. [7], P is a set of processes modelling the corresponding software processes in the dCPS software infrastructure distributed across multiple hosts, whereas 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 model the types of actions performed during a single operation of the process, arranged in chronological order. These events are categorised 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 follows:

$$sig = T_{\text{active}} \cdot f_{\text{CPU}} \tag{1}$$

  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, categorised as *Write* (sending) or *Read* (receiving) events. Each event includes the corresponding channel $ch \in C$, the 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 a *timer setter* or a *timer handler*. A 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 a duration time t in seconds.

5.2 | Computing Hardware Platform Model

We model the computing hardware platform of a dCPS as the tuple $PL = (H, N, L)$. Set $H = \{h_1, ..., h_{|H|}\}$ is the set of hosts that represent the computing subsystems in the dCPS hardware platform, and set $N = \{swch_1, ..., swch_{|N|}\}$ 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.

5.3 | Mapping Model

We model the mapping associated with a dCPS configuration as the set $MP = \{MP_1, ..., MP_m\}$, 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 $MP_j = \{mp_1, ..., mp_{|P|}\}$ 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 the set of hosts in the platform model PL.

5.4 | 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 behaviour but may not always be fully traced because of 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 Δ parameters approach, which abstractly represents the timing impact of the physical environment on computing processes.

The Δ parameters are derived from execution traces by analysing timestamps of message exchanges between traced computing processes and untraced embedded hosts interacting with the physical environment. As illustrated in Figure 3, the up arrows represent Write events where a traced process sends messages to an untraced embedded host, whereas the down arrows represent Read events where the traced process receives responses. Because 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 Δ parameter is correctly attributed to the same communication flow.

For each Read event, a corresponding Δ 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:

$$\Delta_i = T^{p_t}_{\text{read},i} - T^{p_t}_{\text{write},i} \tag{2}$$

where $T^{p_t}_{\text{write},i}$ is the timestamp of the Write event sent from a specific traced process p_t on a traced host and $T^{p_t}_{\text{read},i}$ 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.

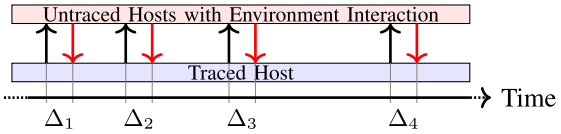


FIGURE 3 | Modelling the impact of the physical environment.

6 | The CompDSE Toolbox

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

6.1 | Automated Derivation and Integration

As shown in Figure 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.

6.1.1 | Application Workload

The formal procedure for the derivation of workload models W_{h_i} from traces and their integration into the 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 a 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 the study by Saadatmand et al. [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.

6.1.1.1 | Workload Model Derivation (WLMD). The WLMD approach and its implementation, presented in the study by Saadatmand et al. [7], are 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. Figure 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 2–9 in the figure).

ALGORITHM 1 | Derive and integrate workload models.

Input : $US = \{US_{h_1}, .., US_{h_n}\}, KS = \{KS_{h_1}, .., KS_{h_n}\}$
Output: $WK = (P, C)$
1 $P, C \leftarrow \emptyset$; $W_{tmp} \leftarrow \emptyset$;
2 **foreach** $US_{h_i} \in US$ **do**
3 $\quad W_{h_i} = WLMD(US_{h_i}, KS_{h_i})$; $W_{tmp} \leftarrow W_{tmp} + W_{h_i}$;
4 **foreach** $W_{h_i} \in W_{tmp}$ **do**
5 $\quad$ **foreach** $ch_j \in W_{h_i}.C$ **do**
6 $\quad\quad h_{src} = ch_j.src.h$; $h_{des} = ch_j.des.h$;
7 $\quad\quad$ **if** $h_{src} \neq h_i$ **then**
8 $\quad\quad\quad ch_j.src.p = \text{findPeerProc}(W_{h_{src}}, ch_j, S)$;
9 $\quad\quad$ **if** $h_{des} \neq h_i$ **then**
10 $\quad\quad\quad ch_j.des.p = \text{findPeerProc}(W_{h_{dst}}, ch_j, D)$;
11 **foreach** $W_{h_i} \in W_{tmp}$ **do**
12 $\quad P \leftarrow P + W_{h_i}.P$; $C \leftarrow C + W_{h_i}.C$;
13 **return** (P, C)

Initially, WLMD analyses the execution traces in set US_{h_i} to create the 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 Figure 4, because 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 visualised by the yellow and orange boxes in Figure 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 grey boxes in Figure 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.

6.1.1.2 | Workload Model Integration. As explained above, after deriving the workload model for each host W_{h_i}, the intrahost 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, interhost communication remains incomplete because the source or destination processes on other hosts have not been identified, as illustrated in Figure 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.

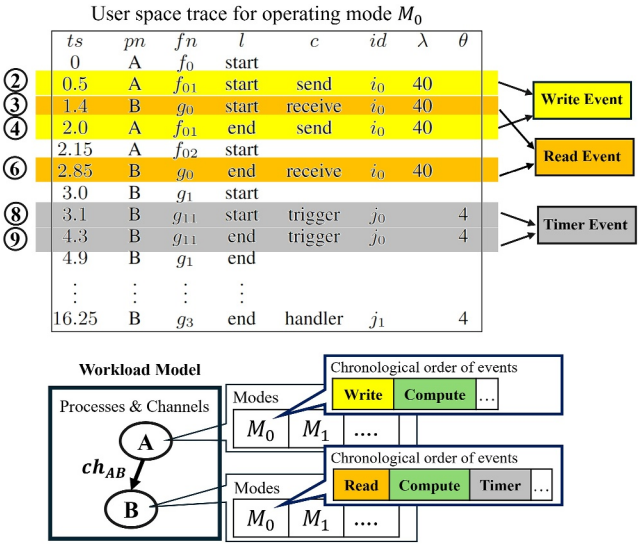
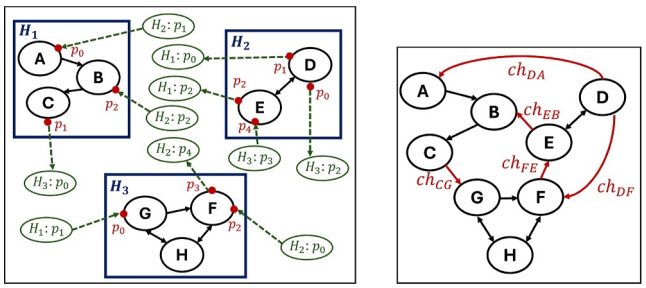


FIGURE 4 | Workload model derivation for each host.

The dashed green arrows represent interhost communication, where the source or destination process needs to be identified. The red dots on the processes indicate the ports involved in interhost communications, which must be considered when identifying the communication channels.

To address this, Algorithm 1 first identifies interhost 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 interhost 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 Figure 5b, where the previously unknown sources or destinations of interhost communication in Figure 5a are identified (indicated by red arrows).

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



(a) (b)

FIGURE 5 | Workload model integration. (a) Distributed Workload Models. (b) Integrated Workload Model.

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 **return** $peerProc \leftarrow ch_i.src.p$
5 **else**
6 **return** $peerProc \leftarrow ch_i.des.p$

6.1.2 | 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.

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 utilising 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 finalised (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 $swch_j$ 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 returns set *ipSet* of unique IP addresses associated with host h. First, the algorithm initialises an empty set *ipSet* (Line 1). Then, for each record in the execution trace US_h, it checks whether the

ALGORITHM 3 | Platform model derivation from traces.

Input : $US = \{US_{h_1}, .., US_{h_n}\}, KS = \{KS_{h_1}, .., KS_{h_n}\}$
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 $nCr =$ number of cores from KS_{h_i}; $crs_i \leftarrow \emptyset$;
5 **for** $j = 1$ ***to*** nCr **do**
6 $f_j =$ 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)

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).

6.1.3 | 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 initialising 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).

ALGORITHM 4 | getIPs.

Input : US_h
Output: $ipSet$
1 $ipSet \leftarrow \emptyset$;
2 **foreach** $r_k \in US_h.E_1$ **do**
3 **if** $r_k.A.f_n = send$ **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** $ipSet$

ALGORITHM 5 | Mapping model derivation from traces.

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

6.2 | 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 specialised 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 synchronise through event-driven messaging, thereby operating concurrently. Each model entity maintains a queue, denoted as *Q*, to buffer incoming event messages from 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. Figure 6 shows the environment of our simulation, visualised as three layers of entities: the application workload, the mapping and

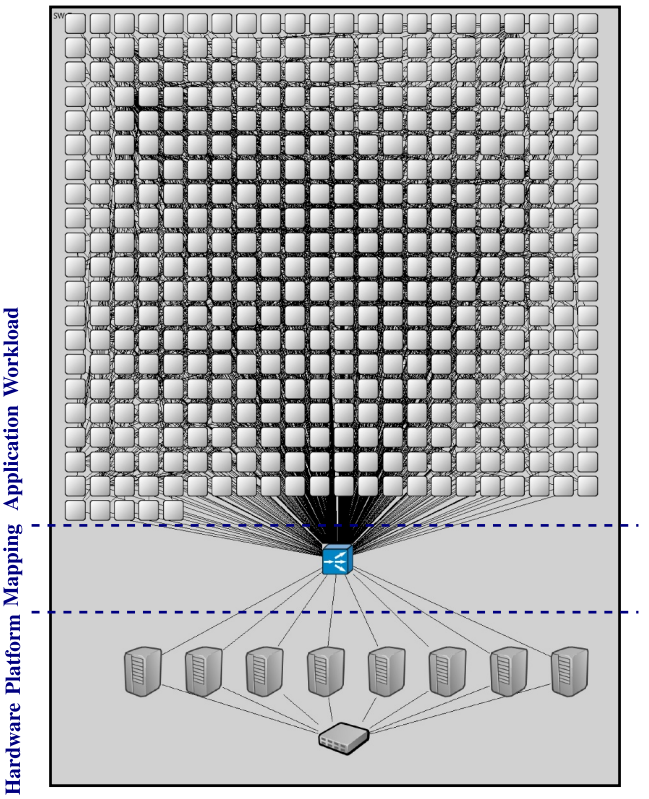


FIGURE 6 | Application workload, computing hardware platform and mapping layers in OMNET++.

the computing hardware platform. Below, we further elaborate on the behaviour of each abstract executable model entity.

6.2.1 | Application Workload

In the executable application workload model (the top part of Figure 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 Figure 6). Listing 1 sketches the behaviour of a process entity as pseudocode. 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 it to proceed to the next event (Lines 5–6). Timer events, however, do not require a request to be 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.

6.2.2 | Mapping

The abstract executable model contains one mapper entity (the middle part of Figure 6) which is connected to all process entities (the top part of Figure 6) and host entities (the bottom part of Figure 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 $(p_{r_k.pn}, h_i)$ 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 behaviour of the mapper entity as pseudocode.

6.2.3 | Hardware Platform

In the executable hardware platform model (the bottom part of Figure 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 behaviour of a host entity and a switch entity are sketched as pseudocode in Listings 3 and 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 5.1), 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

LISTING 1 | Execution behaviour of a process entity.

```
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
                   after nextEvent time;
19         else if (type of nextEvent is timerHandler) {
20             if (timeout contains the ID of nextEvent)
21                 runNext = True;
22         }
23         else {  //Read, Write and Compute events
24             send nextEvent to the mapper;
25             waitForTimer = True;
26             runNext = False;}
27     }
28 }
```

LISTING 2 | Execution behaviour of the mapper 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
                   sender process of event;
6              send event to s_Host;}
7          else   // The event is from a host
8              send a "done" message to the source process
                   of event;
9      }}
```

LISTING 3 | Execution behaviour 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          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
                   is the same as event source)
9                  take e from waitList and add it to
                       readyQ;}
10     }
11     while (readyQ is not empty) {
12         exeEv = take next event from readyQ;
13         if (type of exeEv is not Read) {
14             allocate resource to execute exeEv;
15             if (type of exeEv is Write)
16                 send exeEv to network;
17             send "done" message to mapper for exeEv;
18         }
19         else {  // Read event
20             if (exists an e in receiveList and source of
                     e is the same as exeEv source) {
21                 take e from the receiveList and allocate
                       resource to execute it;
22                 send "done" message to mapper for exeEv;}
23             else // Message has not yet arrived
24                 add exeEv to waitList;
25         }}
26 }
```

LISTING 4 | Execution behaviour of a switch 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
                   event by switch bandwidth;
8              delay = transitDelay + congestDelay;
9              destHost = determine the host mapped to
                   destination process of event;
10             send event to destHost at (simTime + delay);
11         }}
12 }
```

received messages taken from execution traces *US*, excluding the transmission delay. The trimmed mean is computed as follows:

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

where $X = \{x_1, x_2, ..., x_N\}$ is the ordered dataset sorted in ascending order, α 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 5.1) by the switch bandwidth (bw_j in Section 6.1.2) (Line 7 in Listing 4).

7 | Evaluation of the CompDSE Methodology

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

7.1 | 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 utilise 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 utilises 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 utilising 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 multicore 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 optimising system performance, particularly when accounting for the real-time constraints of the embedded systems and the scalability of the server-based components.

7.2 | Experimental Setup

For our experiment, we 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. Because of technical limitations such as some hosts being 'bare metal', that is, 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 Δ parameters as explained in Section 5.4.

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 Δ 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.

7.3 | Evaluation Results

7.3.1 | 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 2 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 2 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 modelled and 2057 communication channels between these processes are identified.

The huge number of possibilities (in the range of $\mathbf{8^{445}}$) to map the 445 processes onto the 8 hosts creates a design 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 2 provides the execution time of the various steps within CompDSE, visualised in Figure 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 min and needs to be done only once. To ensure that tracing did not affect the system's real-time performance, we monitored the wafer processing time before and after enabling tracing and observed no measurable difference.

TABLE 2 | 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 min
Collecting traces automatically	4.33 min
Automated models derivation and integration	29.5 h
Code generation for OMNET++	1.82 min
Single point evaluation	54.5 s

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 min. This trace collection process must be repeated for each operation mode. Automated model derivation and integration take 29.5 h but need to be performed only once. The model-to-model transformation and code generation for the OMNET++ simulator are completed in about 2 min. The simulation to evaluate the performance of one system configuration requires 54.5 s.

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 utilising 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.

7.3.2 | Accuracy

In our experiment, the average processing duration per wafer (PDW) measured on the real ASML TWINSCAN machine is 50.548 s, whereas the simulation of our dCPS model of the same machine indicates an average PDW of 50.141 s, 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.

7.3.3 | 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 16 cores. However, as shown in Figure 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; that is, 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 16 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 s. This improvement indicates that although 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 Figure 8, show the relationship between the average PDW and clock frequency changes relative to the base clock frequency setting of each host's multicore CPU. The base frequency setting is denoted as 0 on the horizontal axis, whereas 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 Figure 8 that increasing the clock frequency of all traced hosts' multicore CPUs by 25% results in a noticeable decrease in the average PDW from 50.141 to 48 s. However, further increasing the clock frequency yields only a marginal decrease in the average PDW, often within the millisecond range, as visualised by the curve's plateau in Figure 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 s. This limitation arises from the performance of some untraced hosts, as reflected by the Δ parameter values discussed in Section 7.2.

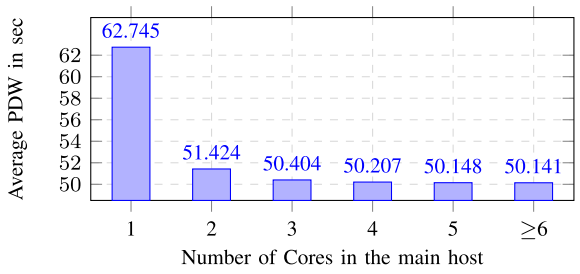


FIGURE 7 | The impact of the number of cores on PDW.

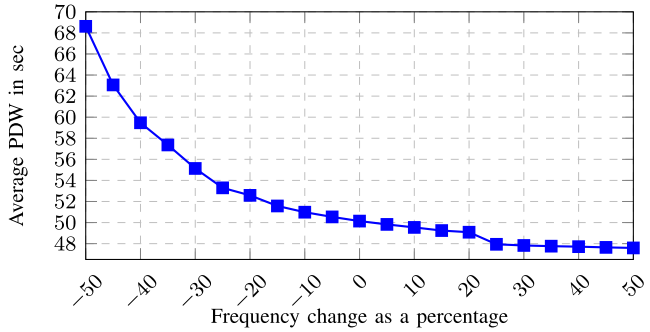


FIGURE 8 | The impact of the clock frequency of cores on PDW.

8 | 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 optimise 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. Although 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.

Author Contributions

Faezeh Sadat Saadatmand: conceptualization, formal analysis, investigation, methodology, software, validation, visualization, writing – original draft. **Todor Stefanov:** conceptualization, funding acquisition, investigation, methodology, resources, supervision, writing – review and editing. **Ignacio González Alonso:** data curation, funding acquisition, project administration, resources, supervision, writing – review and editing. **Andy D. Pimentel:** funding acquisition, project administration, resources, writing – review and editing. **Benny Akesson:** data curation, resources, writing – review and editing.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The data supporting the findings of this study is proprietary and confidential, as it belongs to ASML. Because of restrictions on data sharing, the experimental data cannot be made publicly available. However, the methodology and analysis used in this paper are fully described within the manuscript.

References

1. R. Alur, *Principles of Cyber-Physical Systems* (MIT Press, 2015).

2. S. Acur and T. Hendriks, "Vision and Outlook for Systems Architecting and Systems Engineering in the High-Tech Equipment Industry," TNO, Technical Report R10542 (2024).

3. B. Meier, M. Skelin, F. Beenker, and W. Leibbrandt, "HTSM Systems Engineering Roadmap," Technical Report (2020).

4. B. van der Sanden, Y. Li, J. van den Aker, et al., "Model-Driven System-Performance Engineering for Cyber-Physical Systems: Industry Session Paper," in *EMSOFT* (2021), 11–22.

5. M. Herget, F. S. Saadatmand, M. Bor, et al., "Design Space Exploration for Distributed Cyber-Physical Systems: State-of-the-Art, Challenges, and Directions," in *DSD* (2022), 632–640.

6. B. Kienhuis, E. F. Deprettere, P. Van der Wolf, and K. Vissers, *A Methodology to Design Programmable Embedded Systems: The Y-Chart Approach* (SAMOS, 2002), 18–37.

7. F. S. Saadatmand, T. Stefanov, I. G. Alonso, et al., *Automated Derivation of Application Workload Models for Design Space Exploration of Industrial Distributed Cyber-Physical Systems* (ICPS, 2024).

8. P. Terway, K. Hamidouche, and N. K. Jha, "Dispatch: Design Space Exploration of Cyber-Physical Systems," arXiv preprint, arXiv:2009.10214 (2020).

9. R. Weber, S. Henkler, and A. Rettberg, "Multi-Objective Design Space Exploration for Cyber-Physical Systems Satisfying Hard Real-Time and Reliability Constraints," in *Proceedings of the IDEAL Workshop at CPSWeek, CEUR Workshop Proceedings* (2014), 1337, 57–66.

10. J. Bakakeu, J. Fuchs, T. Javied, et al., "Multi-Objective Design Space Exploration for the Integration of Advanced Analytics in Cyber-Physical Production Systems," in *IEEM* (2018), 1866–1873.

11. Y. Vanommeslaeghe, J. Denil, B. Van Acker, and P. De Meulenaere, "Automatic Generation of Workflows for Efficient Design Space Exploration for Cyber-Physical Systems," in *iThings, GreenCom, CPSCom, SmartData, and Cybermatics* (2021), 346–351.

12. M. Thompson and A. D. Pimentel, "Exploiting Domain Knowledge in System-Level MPSoC Design Space Exploration," *Journal of Systems Architecture* 59, no. 7 (2013): 351–360, https://doi.org/10.1016/j.sysarc.2013.05.023.

13. Y. Vanommeslaeghe, J. Denil, J. De Viaene, D. Ceulemans, S. Derammelaere, and P. De Meulenaere, "Leveraging Domain Knowledge for the Efficient Design-Space Exploration of Advanced Cyber-Physical Systems," in *DSD* (2019), 351–358.

14. Y. Vanommeslaeghe, J. Denil, J. De Viaene, D. Ceulemans, S. Derammelaere, and P. De Meulenaere, "Ontological Reasoning in the Design Space Exploration of Advanced Cyber–Physical Systems," *Microprocessors and Microsystems* 85 (2021): 104151, https://doi.org/10.1016/j.micpro.2021.104151.

15. S. Narain, E. Mak, T. Huster, et al., "'Design Space Exploration for Cyber Physical Systems,' Perspecta Labs," Technical Report (2019).

16. D. Kirov, P. Nuzzo, R. Passerone, and A. Sangiovanni-Vincentelli, "ArchEx: An Extensible Framework for the Exploration of Cyber-Physical System Architectures," in *DAC* (2017), 1–6.

17. M. Amir and T. Givargis, "Pareto Optimal Design Space Exploration of Cyber-Physical Systems," *Internet of Things* 12 (2020): 100308, https://doi.org/10.1016/j.iot.2020.100308.

18. N. Mühleis, M. Glaß, L. Zhang, and J. Teich, "A Co-Simulation Approach for Control Performance Analysis During Design Space Exploration of Cyber-Physical Systems," *ACM SIGBED Review* 8, no. 2 (2011): 23–26, https://doi.org/10.1145/2000367.2000372.

19. D. Genius, I. Bournias, L. Apvrille, and R. Chotin, "High-Level Partitioning and Design Space Exploration for Cyber Physical Systems," in *MODELSWARD* (2020), 84–91, https://doi.org/10.5220/0009171600840091.

20. A. Canedo and J. H. Richter, "Architectural Design Space Exploration of Cyber-Physical Systems Using the Functional Modeling Compiler," *Procedia CIRP* 21 (2014): 46–51, https://doi.org/10.1016/j.procir.2014.03.183.

21. S. Adyanthaya, H. A. Ara, J. Bastos, et al., "xCPS: A Tool to Explore Cyber Physical Systems," *ACM SIGBED Review* 14, no. 1 (2017): 81–95, https://doi.org/10.1145/3036686.3036696.

22. P. Nuzzo, A. L. Sangiovanni-Vincentelli, and R. M. Murray, "Methodology and Tools for Next Generation Cyber-Physical Systems: The iCyPhy Approach," in *INCOSE*, Vol. 25, no. 1 (2015), 235–249.

23. Y. Xiao, C. Oh, M. Lora, and P. Nuzzo, "Efficient Exploration of Cyber-Physical System Architectures Using Contracts and Subgraph Isomorphism," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (IEEE, 2024), 1–6.

24. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," in *Concurrency: The Works of Leslie Lamport* (2019), 179–196.

25. Trace Compass Developers. "Trace Compass [Software]," (2024), https://tracecompass.org/.

26. M. Desnoyers and M. R. Dagenais, "The LTTng Tracer: A Low Impact Performance and Behavior Monitor for GNU/Linux," in *OLS*, Vol. 2006 (2006), 209–224.

27. D. Wilner, "WindView: A Tool for Understanding Real-Time Embedded Software Through System Vizualization," *ACM Sigplan Notices* 30, no. 11 (1995): 117–123, https://doi.org/10.1145/216633.216674.

28. A. Varga and R. Hornig, "An Overview of the OMNeT++ Simulation Environment," in *SIMUTools* (2008), 1–10.

29. M. L. Jones, "Trimmed Means and the Robustness of Estimation," *Journal of Statistical Research* 8, no. 2 (2014): 200–210.