



Universiteit
Leiden
The Netherlands

Towards design space exploration of industrial distributed cyber-physical systems

Saadatmand, F.S.

Citation

Saadatmand, F. S. (2026, September 30). *Towards design space exploration of industrial distributed cyber-physical systems*. Retrieved from <https://hdl.handle.net/1887/4310251>

Version: Publisher's Version

License: [Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden](#)

Downloaded from: <https://hdl.handle.net/1887/4310251>

Note: To cite this publication please use the final published version (if applicable).

Chapter 3

CompDSE Methodology and Models

Publications related to this chapter

- Faezeh Sadat Saadatmand, Todor Stefanov, Ignacio González Alonso, Andy D. Pimentel, and Benny Akesson. CompDSE: A Methodology for Design Space Exploration of Computing Subsystems Within Complex Cyber-Physical Systems. *IET Cyber-Physical Systems: Theory & Applications*, 10(1), 2025.
- Faezeh Sadat Saadatmand, Todor Stefanov, Ignacio González Alonso, Andy D. Pimentel, Benny Akesson, Marius Herget, and Martin Bor. Automated Derivation of Application Workload Models for Design Space Exploration of Industrial Distributed Cyber-Physical Systems. In *Industrial Cyber-Physical Systems (ICPS)*, pages 1–8. IEEE, 2024 (**Best Paper Award**).

In this chapter, we introduce the CompDSE methodology, a structured framework for conducting system-level DSE on complex industrial dCPS, addressing challenge **CHL1**. We then describe the abstraction models used within CompDSE, that are intended to address the modeling challenge **CHL2**. Additionally, we explain how these models are represented as executable entities in a discrete-event simulation environment, contributing to the resolution of challenge **CHL4**.

The chapter is organized as follows. Section 3.1 presents the problem statement and motivates the need for a new methodology for DSE in large-scale industrial dCPS. Section 3.2 outlines the key contributions of the CompDSE methodology and its supporting models. Section 3.3 reviews related work on DSE in the CPS domain and associated modeling approaches. Section 3.4 introduces the structure of the methodology and its core components. Section 3.5 details the abstract models – application workload, hardware platform, mapping, and environmental influence – and explains how the corresponding executable entities are used in simulation environments to enable system-level performance evaluation. Section 3.6 discusses the design space supported by the proposed framework. Finally, Section 3.7 concludes the chapter.

3.1 Problem Statement

DSE is a promising approach for evaluating system-level performance in the early stages of the design cycle. However, existing DSE tools – mostly developed for embedded systems or small-scale CPS – are not suited for the complexity of industrial dCPS. These systems feature large-scale, heterogeneous software and hardware infrastructures.

A key limitation of these tools is the lack of an integrated methodology that combines system-level modeling, automated model derivation, and scalable system simulation. As a result, existing tools fail to support efficient exploration of the large design spaces typical in industrial settings. More specifically, no consistent DSE framework exists to support the construction of highly abstract yet sufficiently accurate models that reflect the behavior of complex industrial dCPS. These systems often consist of hundreds of independently developed processes running across heterogeneous hardware platforms and require tight synchronization through event-driven execution. Additionally, they operate in multiple modes, each associated with distinct software behavior that must be captured.

In the absence of such DSE methodology and model infrastructure, designers of industrial dCPS lack effective methods and tools to evaluate “what-if” scenarios and explore architectural alternatives. This gap represents a significant barrier to the scalable and efficient design of next-generation dCPS.

3.2 Contributions

This chapter makes the following key contributions towards enabling scalable and efficient DSE for industrial dCPS:

1. **The CompDSE methodology:** We introduce CompDSE, a structured methodology for conducting system-level DSE tailored to the cyber components of industrial dCPS. The methodology integrates efficient system-level modeling with supporting tools for automated model derivation, transformation, and performance evaluation. CompDSE is particularly suited to existing complex industrial dCPS that are being explored for next-generation upgrades.
2. **Proper modeling:** Building on the CompDSE methodology, we define a set of highly abstract, yet sufficiently accurate models: the application workload model, the computing hardware platform model, and the workload-to-platform mapping model. Additionally, we introduce a highly abstract environmental influence

model to account for timing effects caused by subsystems that interact with the physical environment but are not explicitly traced and modeled.

3. **Executable simulation entities:** We define how the aforementioned abstract models are instantiated as executable entities that interact during discrete-event simulation to model the system behavior.

3.3 Related Work

In reviewing related work on DSE, two key aspects deserve particular attention: 1) the methods and algorithms used for DSE in CPS, and 2) how the core models are defined within the DSE methodologies. Among these models, software application modeling presents unique challenges due to the complexity, variability, and scale of software infrastructures in industrial systems. Notably, the way application behavior is modeled influences the required level of abstraction of the underlying hardware platform, as well as the modeling granularity of mapping models between application and platform. Accordingly, this related work section is organized into three parts. The first part reviews system-level DSE methodologies relevant to CPS. The second part focuses on prior work in software application modeling, highlighting the limitations of existing approaches. The third part reviews existing approaches to hardware platform modeling and application-to-platform mapping, and explains the rationale for their selection.

3.3.1 System-Level DSE Methodologies for dCPS

DSE for CPS has been extensively studied, leading to various methodologies and tools. While the entire process can be broken down into multiple steps (as explained in Section 2.1), 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, and 2) search algorithms, which traverse the design space consisting of many design points to find (Pareto-)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, and/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 [98, 117], automating the DSE process [11, 112], and limiting the search space for more efficient exploration [102, 110, 111].

This section 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 [74] 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 [58]. 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 MoCs. For example, in [7], 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 [73] co-simulates a high-level physical environment model in Matlab Simulink and a control application model in SystemC.

Similarly, in [39], 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, [23] 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 [2] and iCyPhy [77] 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 subgraph isomorphism [120]. 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 the 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.

To highlight the differences between our approach and existing DSE tools/methodologies, we provide a comparative analysis in Table 3.1, which distinguishes them based on the abstraction level of models, 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.

Table 3.1: Comparison of the DSE tools for industrial CPS modeling

Approach/tool	Abstraction level	Scalability for industrial CPS	Degree of automation	Performance analysis type	Support dynamics
CNMA [74]	Low	No	Semiautomated	Analytical	No
ArchEx 2.0 [58]	Medium	Limited	Semiautomated	Analytical	No
Extended ArchEx 2.0 [120]	Medium	Yes	Semiautomated	Unknown*	No
Matlab Simulink [7] [73]	Low	No	Manual	Co-simulation	No
Extended TTool [39]	Medium	Limited	Semiautomated	Simulation-based	No
FMC [23]	Medium	Limited	Semiautomated	Co-simulation	No
xCPS [2]	Low	Limited	Semiautomated	Constraint-based	Limited
iCyPhy [77]	Low	No	Semiautomated	Formal verification	No
CompDSE	High	Yes	Fully automated	Simulation-based	Yes

* Performance analysis in [120] is unclear as the focus is on structural validation, not execution-time evaluation.

3.3.2 Application Workload Modeling for DSE

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

Simple CPS Applications: The works in this group focus on relatively simple CPS use cases, such as a line follower robot [36], or a specific subsystem within a larger system. These works often use well-known MoCs to model the software application behaviour. For instance, some works utilize directed acyclic graphs (DAGs) to capture software task dependencies [45, 75, 85, 91, 97], while others use actor-oriented [3] or discrete-event models to describe interactions among tasks. Functional models are also employed to represent CPS application behavior, usually in a time-continuous manner, with differential equations [7, 20, 23, 74, 114].

Additionally, various tools have been developed based on these MoCs, such as Ptolemy [82], which is an actor-based simulator, or PTIDES [29], which allows modelling applications based on different MoCs or combination of them. Furthermore, frameworks like Octopus [15] and Daedalus [92] facilitate modeling techniques and co-simulation of models specifically for DSE. However, the aforementioned works often include a lot of specific details when modeling the behavioral aspects of applications. Therefore, they lack the necessary abstraction and coarse-grained representation required to effectively capture and model complex CPS applications for DSE and also cannot model dynamic workloads. In addition, they usually have scalability restrictions, limiting their ability to create a comprehensive application model of distributed systems for industry-relevant cases.

Complex CPS Applications: In the case of complex CPS applications, there are frameworks, such as PtolemyII [82], OpenMETA [95], BPMN4CPS [43], SysML2 [44],

Maestro [103], and OpenModelica [37] that enable designers to model the behavior of industry-relevant systems for DSE purposes. However, the modeling process when utilizing these frameworks is mostly done manually due to a lack of tools that could help designers derive models in an automated way. Such manual modeling is a very time consuming and difficult process. In contrast, we propose a model derivation method implemented in a software tool that allows designers to avoid manual application modeling for an industry-relevant complex dCPS in case the designers need to perform DSE to design the next generation of the dCPS. This is possible because initial models can be automatically derived using our method and tool from data collected during runtime of the software infrastructure of the current generation of the dCPS.

Finally, automated model inference is a well-established approach where inference is done by analyzing the source codes of an application or utilizing logs and traces of a system running the source code [90, 94]. These inference techniques are used for various purposes. Some studies utilize them for software testing and verification [38, 64], assisting engineers in identifying underlying issues, such as timing bottlenecks [54] or performance anomalies [69]. However, the aforementioned techniques primarily concentrate on the inference of *behavioral* models. Such models are not sufficiently abstract and coarse-grained for efficient DSE of complex dCPS. Moreover, many of these inference techniques are not applicable to complex dCPS because analyzing complex application source code is challenging.

Other studies leverage automated model inference to estimate the performance of Multiprocessor SoCs [53, 59], and parallel distributed systems [70] during their early design stages or in scenarios where real-time systems are assembled from black-box components [34]. However, the inferred models include timing and hardware architecture dependent information that does not allow the exploration of different application mappings on different hardware platforms.

In contrast, our method shifts the focus to automated derivation of application workload models that are sufficiently abstract and coarse-grained, thereby enabling efficient DSE of complex dCPS. In addition, our method eliminates timing information from the derived models by considering only the order of computation and communication workloads and representing these workloads as abstract events.

3.3.3 Hardware Platform and Application-to-Platform Mapping Modeling for DSE

In DSE, while application, platform, and mapping models are conceptually distinct in the Y-chart modeling approach, the semantics assigned to platform resources and mapping decisions are influenced by the abstraction level at which the application behavior is modeled. In the literature, this relationship is reflected in the wide range of platform modeling approaches, from detailed execution-level models to highly abstract analytical models [40, 89].

At the detailed execution-oriented end of this spectrum are high-fidelity virtual platform approaches that aim to approximate the behavior of the real system as closely as possible during performance evaluation. Representative examples include Simics [66], gem5 [17], and QEMU-based virtual platforms [16], which often rely on instruction-set emulation to execute unmodified software applications. In these approaches, the execution platform is modeled at a very fine granularity, and software execution is performed on or alongside the platform model under near-realistic conditions. Application-to-platform mapping is typically fixed or explicitly defined for a specific platform instance, and performance metrics are obtained by directly observing execution behavior. While such high-fidelity virtual platforms can provide highly accurate performance estimates when carefully calibrated, they require detailed knowledge of platform components and their interactions, as well as substantial effort to reconstruct and integrate a faithful system representation. Consequently, emulation-supported virtual platforms are predominantly used for late-stage validation of specific configurations, but are less suitable for early-stage or large-scale design space exploration, where platform parameters and mappings must be varied extensively.

Moving toward higher levels of abstraction, other virtual platform approaches model the application and the execution platform using executable but abstract representations, enabling efficient performance evaluation and DSE. In these frameworks, the application behavior is typically modeled as concurrent tasks or processes, which are executed on abstract representations of processing elements, interconnects, and other platform resources [22, 41, 84]. Representative system-level design and DSE frameworks following this approach include Daedalus [92, 101], ESPAM [76], Koski [55], PeaCE [48], SCE [31], SystemCoDesigner [49], MAMPS [60], and the DOL framework [99]. These frameworks have been primarily developed in the context of embedded and MPSoC systems. Application-to-platform mapping specifies the binding of tasks or processes to abstract resources, and the performance impact of mapping decisions emerges

during simulation through scheduling, synchronization, and resource contention effects. Compared to emulation-supported virtual platforms, these abstract virtual platforms significantly reduce the modeling effort and simulation cost, while still preserving executable semantics sufficient for DSE.

At a higher level of abstraction than virtual platforms, service-based and event-driven simulation approaches represent the execution platform in terms of abstract services rather than executable hardware components. In these approaches, platform resources such as processing elements, communication links, and memories are modeled as service providers with abstract performance characteristics. Application-to-platform mapping determines which platform services are used by each application component, and performance effects such as resource contention and scheduling delays emerge through discrete-event simulation rather than through execution of application code. Representative examples of this class of approaches include the Sesame system-level modeling and simulation framework for MPSoC design space exploration [33], as well as frameworks such as MAPS [25], which have been developed to support efficient system-level performance analysis and DSE for embedded and distributed platforms. Similar to virtual-platform approaches, these frameworks have been predominantly applied to embedded and MPSoC systems.

At the opposite end of the spectrum are purely analytical approaches, where platform resources are described through mathematical models such as algebraic expressions or queuing formulations [13], or curve-based abstractions as used in Real-Time Calculus (RTC) [26, 100] and Modular Performance Analysis (MPA) [115]. In these approaches, application-to-platform mapping is typically represented as an explicit assignment of tasks or workloads to abstract resources, and performance metrics, including execution time or worst-case delay bounds, are derived by combining workload parameters, resource characteristics, and mapping decisions within the analytical model. This makes such methods computationally efficient and particularly useful in timing-critical systems, but also reliant on simplifying assumptions regarding execution order, interference, and resource contention, which can limit their applicability to complex and dynamic systems.

The platform modeling approach proposed in this thesis falls within the class of service-based and event-driven platform modeling approaches. Similar to existing service-based frameworks, the execution platform is described through an abstract architectural specification capturing available computational and communication resources. This deliberate abstraction captures the platform behavior in terms of system-level timing and resource-contention effects, without modeling detailed micro-architectural

or physical properties. In this context, application-to-platform mapping represents the assignment of software processes to execution hosts, while the allocation of processes to individual cores within a host is resolved dynamically at runtime by the execution platform. Although such host-level mappings can be specified manually, doing so becomes time-consuming when the number of software processes is large and mappings vary across different design points. For this reason, execution traces are used in this work to automatically derive an initial application-to-platform mapping of an existing system, which serves as a starting point for exploring alternative mappings during DSE.

The platform model can be derived from different information sources, including hardware documentation describing the structural composition of dCPS. However, in the targeted industrial context, extracting and maintaining abstract platform models from detailed hardware documentation is a time-consuming and error-prone process, particularly when multiple system configurations and testbench setups coexist and evolve over time. Moreover, much of the structural detail contained in such documentation is not required at the chosen abstraction level. For this reason, execution traces are used in this work to derive an initial abstract platform model corresponding to an existing system configuration.

3.4 Overview of the CompDSE Methodology

The CompDSE methodology, illustrated in Figure 3.1, leverages the Y-chart approach to systematically define and automatically derive abstract models for the application workload, computing hardware platform architecture, and workload-to-platform mapping. Additionally, it incorporates environmental influence model to account for the effects of the physical environment on computing processes.

These models are derived from collected user-space and kernel-space traces, explained in detail in Chapter 4, and are transformed into abstract executable entities for discrete event simulation. This transformation enables performance evaluation through simulation, allowing system designers to explore the design space by independently modifying each model. All models are implemented using EMF, as introduced in Section 2.6.2.

The CompDSE methodology consists of three main components highlighted in different colors in Figure 3.1:

Models: Representations of the application workload, computing hardware platform, mapping, and the environmental influence, which collectively define the behavior

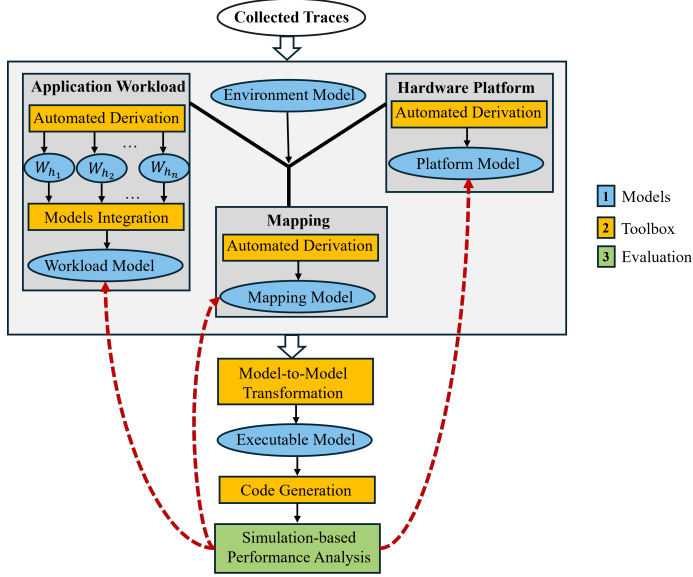


Figure 3.1: The CompDSE methodology

of a dCPS. The Application Workload Model captures computation patterns, and inter-process communications, 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 subsystems (hosts), determining the execution impact of deployment choices on the 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 entities, 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 through simulation-based analysis within the prepared simulation environment.

The red feedback arrows in Figure 3.1 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.

The remainder of this chapter focuses on the detailed specification of the Models defined within the CompDSE methodology, while the Toolbox and Evaluation are presented in Chapter 4 and Chapter 5, respectively.

3.5 Models used in CompDSE

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

3.5.1 Application Workload Model

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

Therefore, we model the workload behavior of the dCPS software infrastructure as a super graph WK represented as the tuple $WK = (P, C)$, where $P = \{p_1, \dots, p_{|P|}\}$ denotes a set of processes that model the corresponding processes in the dCPS software infrastructure distributed across multiple hosts. Additionally, $C = \{ch_1, \dots, ch_{|C|}\}$ represents a set of communication channels that model the exchange of control or data messages between the processes. We denote each channel ch_j as a tuple $ch_j = (p_s, p_d)$, where ch_j is exclusively dedicated to communicating messages from source process p_s to destination process p_d . Figure 3.2 illustrates an example of a workload model WK with a set of eight processes $P = \{A, B, C, D, E, F, G, H\}$, depicted as squares, and a set of sixteen communication channels $C = \{ch_{AB}, \dots, ch_{HG}\}$ represented by arrows. The direction of an arrow indicates the flow of messages communicated within the channel.

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

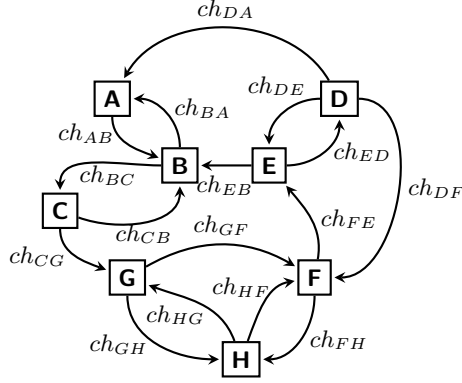


Figure 3.2: Example of an application workload model WK

three categories:

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

$$sig = T_{\text{onCPU}} \cdot f_{\text{CPU}} \quad (3.1)$$

where T_{onCPU} is the time the process has been actively running on the CPU, and f_{CPU} is the CPU clock frequency.

- **Communication events:** These events are crucial for modeling message exchange and synchronization between processes through channels. A communication event is classified as either a *write* event, denoted as e^W , or a *read* event, denoted as e^R . A *write* event occurs when a process sends a message to another process, and a *read* event occurs when a process receives a message. Each e^W or e^R event is defined as a tuple $(ch, size, sig)$, where ch specifies the channel used for the communication, $size$ denotes the size of the communicated message in bytes, reflecting the communication workload of the event on the communication channel, and sig represents an abstract computation workload associated with the

event's execution in the source/destination process, measured in cycles similar to a computation event and it contributes to computing the actual time required for sending or receiving a message.

- **Timer events:** These events are internal triggers that originate within a process and initiate a computation or communication event after a specific amount of time has elapsed. These events allow processes to schedule actions that need to be performed after a certain period. A timer event is classified as either *timer setter*, denoted as e^{TS} , or *timer handler* denoted as e^{TH} . Event e^{TS} sets a timer with an absolute time value, and once that time has elapsed, a corresponding e^{TH} event is triggered to initiate other events. Each e^{TS} or e^{TH} event is defined as a tuple (τ, t) , where τ is the timer identifier and t denotes the duration time, in seconds, set for timer τ . For a timer handler event, the duration attribute t corresponds to the timeout value defined by the associated timer setter. Although not strictly required for simulation, it is included to maintain a uniform representation of timer events in the workload model.

As shown in Figure 3.1, the global workload model WK is built by integrating multiple per-host models W_{h_i} , each derived from execution traces collected from host h_i . Each W_{h_i} is structurally similar to WK , containing its own processes and channels, and the union of these models forms the global supergraph that represents the comprehensive distributed application workload.

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

3.5.2 Computing Hardware Platform Model

We model the computing hardware platform of a dCPS as the tuple $PL = (H, N, L)$. Set $H = \{h_1, \dots, h_{|H|}\}$ 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 network identifiers used to identify h_i in the communication infrastructure (e.g., static IP addresses in the ASML case study), 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.

3.5.3 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 .

3.5.4 Environmental Influence Model

In a dCPS, some embedded hosts interact directly with the physical environment through actuators and sensors. These hosts process real-world inputs and influence the 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 the 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 obtained during the derivation of the workload model by analyzing timestamps of message exchanges between traced computing processes and untraced embedded hosts interacting with the physical environment. As illustrated in Figure 3.3, the up arrows represent *write* events where a traced process sends messages to an untraced embedded process, 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 Δ parameter is correctly attributed to the same communication flow.

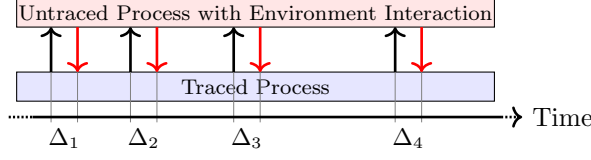


Figure 3.3: Modeling the impact of the physical environment

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:

$$\Delta_i = T_{\text{read},i}^{p_t} - T_{\text{write},i}^{p_t} \quad (3.2)$$

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

3.5.5 Executable Models

Once the abstract models of the application workload, computing platform, and mapping are derived, they must be transformed into intermediate executable entities, which can be used with any discrete event simulator. This transformation generates a separate model entity for each process $p_i \in P$ in the workload model WK , each host $h_i \in H$ and switch $swch_j \in N$ in the platform model PL , and for the mapping MP .

It is important to note that the environment model does not correspond to an executable simulation entity; instead, its effect is incorporated during simulation through the Δ parameters applied by the process entities, as explained in Chapter 5.

Figure 3.4 illustrates an example of the application workload model, the mapping model, and the computing hardware platform model transformed into executable entities. The top layer represents the workload model, consisting of process entities and the communication channels between them. In this example, there are process entities A to H, connected by 16 channels. Arrows with two heads indicate bidirectional communication, meaning two separate channels exist between those processes. The middle layer shows the mapping model entity (Mapper), which defines the allocation of

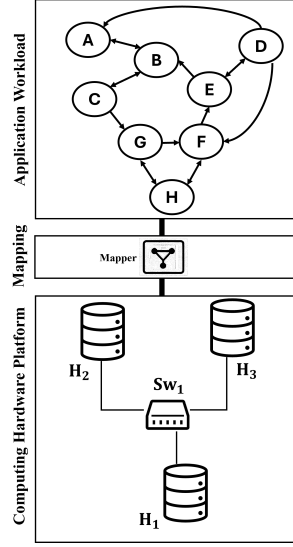


Figure 3.4: The application workload, computing hardware platform, and mapping model entities

process entities to hardware resource entities. The bottom layer presents the hardware platform model, which includes host and switch entities. In this example, there are three host entities, H_1 , H_2 , and H_3 , that are connected through a single switch entity Sw_1 .

Together, all these entities form the abstract executable model of a dCPS. All entities interact and synchronize through event-driven messaging, operating concurrently within the simulation framework. The overall execution principle is that each application workload process entity sends an execution request to the mapper entity. The mapper forwards this request to the mapped hardware platform entity, which executes the request and returns a completion notification. This completion is propagated back to the originating process and allows execution of subsequent events to continue. Each model entity maintains a queue, denoted as Q , to buffer incoming event messages from other entities. Below, we further elaborate on the behavior of each abstract executable model entity.

Application Workload Process Entities

In the executable application workload model (the top part of Figure 3.4), each process is represented as an entity with individual event files for each operation mode, containing

```

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 3.1: Execution behavior of a Process entity

abstract events arranged chronologically along with their attributes. All process entities are connected to the mapper entity (the middle part in Figure 3.4).

Listing 3.1 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 (Lines 5-6). Timer events, however, do not require a request sent to the mapper entity. Instead, when encountering a *timer setter* event, the process entity schedules a “timeoff” message to be delivered to itself after the specified timer duration (Lines 17–18). Setting a timer does not block the execution of the process entity; the event-processing loop continues handling other incoming messages from the queue Q , including completion notifications from the mapper or previously scheduled self-messages. Once the scheduled “timeoff” message is received, the timer is considered off (Lines 7–11). If the event is a *timer handler* and the corresponding “timeoff” message has already been received, the process proceeds to retrieve the next event (Lines 20–21); otherwise, it waits until the required timeout has elapsed. If the “timeoff” message has not yet been received, the process waits until it arrives.

```

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 3.2: Execution behavior of the Mapper entity

Mapper Entity

The abstract executable model contains one mapper entity (the middle part of Figure 3.4) which is connected to all process entities (the top part of Figure 3.4) and host entities (the bottom part of Figure 3.4). 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 3.2 illustrates the behavior of the mapper entity as a pseudo code.

Hardware Platform Entities

In the executable hardware platform model (the bottom part of Figure 3.4), 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 3.3 and Listing 3.4, respectively. The host entity receives *write*, *compute*, and *read* request messages (events) from the mapper (Lines 4-5 in Listing 3.3). 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 3.3) 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).

```

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
9         source)
10        take e from waitList and add it to readyQ;}
11   }
12   while (readyQ is not empty) {
13     exeEv = take next event from readyQ;
14     if (type of exeEv is not read) {
15       allocate resource to execute exeEv;
16       if (type of exeEv is write)
17         send exeEv to network;
18       send "done" message to mapper for exeEv;
19     }
20     else { // read event
21       if (exists an e in receiveList and source of e is the same as exeEv
22         source) {
23         take e from the receiveList and allocate resource to execute it;
24         send "done" message to mapper for exeEv;}
25       else // Message has not yet arrived
26         add exeEv to waitList;
27     }
28   }
29 }

```

Listing 3.3: 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 event by switch bandwidth;
8       delay = transitDelay + congestDelay;
9       destHost = determine the host mapped to destination process of event
10      ;
11      send event to destHost at (simTime + delay);
12    }
13  }
14 }

```

Listing 3.4: Execution behavior of a Switch entity

During the execution of an event, just a time delay is performed, based on the event's signature (*sig* described in Section 3.5.1), 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 3.3). The switch entity applies a time delay and transmits the event to the destination host entity (Lines 9-10 in Listing 3.4). 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 3.4). In our abstract model, the network congestion delay is the trimmed mean interval between

sent and received messages taken from traces, excluding the transmission delay. The trimmed mean is computed as:

$$T_\alpha(X) = \frac{1}{(1 - 2\alpha)N} \sum_{i=\alpha N+1}^{(1-\alpha)N} x_{(i)} \quad (3.3)$$

where $X = \{x_1, x_2, \dots, x_N\}$ 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 [27]. The transmission delay is calculated by dividing the message size by the switch bandwidth (Line 7 in Listing 3.4).

3.6 Design Space

Based on the defined application, platform, and mapping models, this section specifies the design space that can be explored using the proposed methodology. The supported design points for exploration span both the hardware platform and the application-to-platform mapping layers.

The initial hardware platform configuration from which exploration starts is derived from execution traces of an existing system instance. This trace-derived model provides a practical and consistent starting point for DSE. The absence of certain components or connections in the initial trace-derived model does not imply that such elements cannot be explored. During subsequent DSE iterations, additional computational and communication resources can be added manually or automatically by the exploration process to construct and evaluate alternative platform configurations, including extensions and next-generation variants that were not present in the original/initial execution trace.

At the platform layer, the design space includes variations in the number of hosts participating in execution and the number of processing cores available per host. In addition, performance characteristics of computational and communication resources can be varied, such as processing core frequency and network bandwidth. The network structure itself can also be modified by changing the number of switches and the topology describing how hosts are interconnected. Furthermore, different scheduling policies for the processes mapped on a host (host-level scheduling) can be selected, enabling the analysis of their impact on the system execution behavior and performance.

At the mapping layer, the design space comprises alternative assignments of appli-

cation processes to hosts. By modifying the mapping, it is possible, for example, to reassign computationally intensive or frequently activated processes to less loaded hosts and observe the resulting performance effects. Similarly, new hosts can be introduced into the platform model, and selected processes can be migrated to these hosts to study load redistribution and scalability.

It is important to note that core-level process allocation and scheduling within a host are not treated as explicit design points. Instead, these aspects are resolved dynamically during simulation based on resource availability and contention. Nevertheless, changes in the selected host-level scheduling policy directly influence this dynamic behavior and therefore indirectly affect the execution timing and overall system performance.

The design space defined above is subject to a number of structural and practical constraints that ensure the feasibility of DSE simulations. For executing the DSE simulation, at least one host with at least one execution core must be defined, and all application processes in the workload model must be allocated to a host. As a result, configurations without an explicit network model are possible, for example when all processes are mapped to a single host. The ranges of platform parameters, such as processor frequency and network bandwidth, are constrained by the characteristics of the target system. New scheduling policies can also be defined and evaluated. In cases where a scheduling policy allows task preemption, additional implementation effort is required to support task interruption, but such policies are compatible with the proposed methodology.

3.7 Conclusion

This chapter introduced CompDSE, a structured methodology for DSE of computing subsystems in complex industrial dCPS, composed of required models, a toolbox, and simulation-based evaluation support. The methodology is centered around three core abstract models: the application workload model, the computing hardware platform model, and the workload-to-platform mapping model. These models are defined independently, yet are designed to work in combination, offering a high level of abstraction while maintaining the accuracy required for meaningful system-level performance analysis. This aspect will be validated through the experimental results presented in Chapter 5.

In addition to these core models, the methodology includes an environmental influence model that captures the timing impact of untraced embedded hosts interacting with the physical environment (e.g., through sensors and actuators) that cannot be

explicitly modeled. This model plays a supporting role by approximating the indirect effects of such physical interactions on system timing, thereby enhancing simulation accuracy without introducing modeling complexity.

Furthermore, this chapter described how each abstract model is represented as executable simulation entities, and explained the execution behavior of these entities within a discrete-event simulation environment. During a system simulation, the interaction among these entities emulates the behavior of a dCPS and enables system-level performance evaluation and analysis.

