Towards Design Space Exploration of Industrial Distributed Cyber-Physical Systems

Faezeh Sadat Saadatmand

Towards Design Space Exploration of Industrial Distributed Cyber-Physical Systems

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van rector magnificus prof.dr. S. de Rijcke,
volgens besluit van het college voor promoties
te verdedigen op woensdag 30 september 2026
klokke 10:00 uur

door

Faezeh Sadat Saadatmand

geboren te Tehran, Iran
in 1992

Promotores:
Dr. T.P. Stefanov
Prof. dr. N. Mentens

Promotiecommissie:

Prof. dr. M.M. Bonsangue	
Prof. dr. H.C.M. Kleijn	
Prof. dr. R.V. van Nieuwpoort	
Prof. dr. ir. A.A. Basten	Technische Universiteit Eindhoven
Prof. dr. O. Bringmann	Universität Tübingen
Prof. dr. ir. A.L. Varbanescu	Universiteit Twente

Copyright © 2026 Faezeh Sadat Saadatmand.

This research was carried out within the DSE2.0 project, "Design Space Exploration 2.0: Towards Optimal Design of Complex, Distributed Cyber-Physical Systems", funded by the Dutch Research Council (NWO) under project number 17930.

Cover: designed by the author.

To my dear family

Contents

List of Figures **xi**

List of Tables **xiii**

List of Abbreviations **xv**

1 Introduction **1**

1.1 Context and Motivation 1
1.2 Scope and Focus of This Thesis 2
1.3 Scientific Challenges 3
1.4 Research Contributions 7
1.5 Thesis Structure 10

2 Background **13**

2.1 Design Space Exploration 14
2.1.1 General Workflow of DSE 15
2.1.2 Y-Chart Modeling Approach 16
2.2 Limitations of Embedded DSE Tools for Industrial dCPS 17
2.3 ASML TWINSCAN Machine 20
2.4 Event-Driven Execution in Distributed Systems 20
2.5 Execution Traces 21
2.6 Simulation and Modeling Environment 22
2.6.1 Discrete-Event Simulation using OMNet++ 23
2.6.2 Eclipse Modeling Framework 23
2.6.3 Model Transformation with Xtend 23

3 CompDSE Methodology and Models **25**
3.1 Problem Statement . . . 26
3.2 Contributions . . . 26
3.3 Related Work . . . 27
3.3.1 System-Level DSE Methodologies for dCPS . . . 27
3.3.2 Application Workload Modeling for DSE . . . 30
3.3.3 Hardware Platform and Application-to-Platform Mapping Modeling for DSE . . . 32
3.4 Overview of the CompDSE Methodology . . . 34
3.5 Models used in CompDSE . . . 36
3.5.1 Application Workload Model . . . 36
3.5.2 Computing Hardware Platform Model . . . 38
3.5.3 Mapping Model . . . 39
3.5.4 Environmental Influence Model . . . 39
3.5.5 Executable Models . . . 40
3.6 Design Space . . . 45
3.7 Conclusion . . . 46

4 CompDSE Tools **49**
4.1 Problem Statement . . . 50
4.2 Contributions . . . 50
4.3 Related Work . . . 51
4.4 Execution Traces for Model Derivation . . . 52
4.4.1 Application-level Execution Traces . . . 53
4.4.2 System-level Execution Traces . . . 55
4.5 The CompDSE Toolbox . . . 56
4.5.1 Automated Derivation and Integration . . . 56
4.5.2 Model-to-Model Transformation . . . 65
4.5.3 Code Generation . . . 67
4.6 Conclusion . . . 71

5 CompDSE Evaluation **73**
5.1 Problem Statement . . . 74
5.2 Contributions . . . 74
5.3 Experimental Setup . . . 74
5.4 Evaluation Results . . . 77
5.4.1 Efficiency . . . 77

5.4.2 Accuracy . . . 78
5.4.3 Exploration . . . 78
5.5 Conclusion . . . 80

6 Workload Model Refinement 83
6.1 Problem Statement . . . 84
6.2 Contributions . . . 84
6.3 Related Work . . . 85
6.4 Motivation . . . 87
6.5 Application Workload Model Refinement . . . 89
6.6 Collected Execution Traces . . . 91
6.7 Automated Derivation of Refined Workload Model . . . 92
6.8 Feasibility for Parallel Execution . . . 93
6.8.1 Determining Parallel Execution Eligibility of Blocks . . . 93
6.8.2 Simulation Strategy for Parallel Execution of Blocks . . . 96
6.9 Experimental Evaluation . . . 99
6.9.1 Experimental Setup . . . 99
6.9.2 Results . . . 101
6.10 Conclusion . . . 104

7 Summary, Discussion, and Future Directions 105
7.1 Summary of Challenges and Contributions . . . 105
7.2 Discussion and Future Directions . . . 106
7.3 Concluding Remark . . . 111

Bibliography 113

English Summary 125

Nederlandse Samenvatting 127

List of Publications 129

Curriculum Vitae 131

Acknowledgments 133

List of Figures

2.1 Illustration of the Y-chart modeling approach 17
2.2 Heatmap of the DSE runtime (in hours) by varying the number of application processes and processing components 19

3.1 The CompDSE methodology 35
3.2 Example of an application workload model WK 37
3.3 Modeling the impact of the physical environment 40
3.4 The application workload, computing hardware platform, and mapping model entities 41

4.1 An excerpt of an example application-level execution trace 54
4.2 Visualization of the complete function executions, highlighting traced functions from Figure 4.1 54
4.3 Process status over time 55
4.4 Deriving `OnCPU` timing by using context switch events 55
4.5 Derivation of the workload model of the first mode in process B 60
4.6 Calculation of the signature (cycles) of process B 61
4.7 The workload model integration 62
4.8 Model-to-Model transformation from formal models (left) to executable simulation entities (right) 66
4.9 The application workload, computing hardware platform, and mapping layers in OMNet++ 68
4.10 The C++ class skeleton of the `Mapper` entity for the OMNet++ simulation 69
4.11 Example of a generated event file for Process B in operation mode M_1 70
4.12 Example of a generated platform file 70
4.13 Example of a generated mapping file 71

5.1 The simulation setup of the ASML TWINSCAN machine in OMNet++ 76
5.2 The impact of the number of cores on PDW 79
5.3 The impact of the clock frequency of cores on PDW 80

6.1 The impact of request handling order on execution timing 88
6.2 The refined application workload model class diagram 89
6.3 Refinement of the application workload model 90
6.4 Automated derivation of the refined application workload model . . . 92
6.5 The dependency check of execution blocks 95
6.6 An example of parallel execution of blocks 98
6.7 Three ENV delay modeling scenarios 101
6.8 The effect of ENV delay reduction and number of cores on PDW. Top: inter-process only. Bottom: inter- and intra-process 102
6.9 The effect of frequency scaling on PDW for 8-core platform with 100% ENV delay reduction 104

List of Tables

1.1 Relation between the thesis chapters and research challenges 10

3.1 Comparison of the DSE tools for industrial CPS modeling 30

5.1 Scale of the traced system and model derivation 75
5.2 Time required for each step in the CompDSE workflow 78

6.1 Additional trace points for the refined application workload model . . 91
6.2 Comparison of PDW between real system execution and simulation considering inter-process parallelism only 101

List of Abbreviations

CPS	Cyber-Physical Systems
CTF	Common Trace Format
dCPS	Distributed Cyber-Physical Systems
DSE	Design Space Exploration
EMF	Eclipse Modeling Framework
FCFS	First Come, First Served
FMC	Functional Modeling Compiler
GA	Genetic Algorithm
MoC	Models of Computation
MPSoC	Multiprocessor System-on-Chip
OS	Operating System
PDW	Processing Duration per Wafer
SoC	System-on-Chip

Chapter 1

Introduction

1.1 Context and Motivation

Cyber-Physical Systems (CPS) are systems that tightly integrate computational (cyber) components with physical processes. The cyber part consists of embedded software and computing hardware – often based on heterogeneous multi-core or many-core processors – responsible for control, data processing, monitoring, and decision-making. The physical part includes mechanical, electrical, or chemical components that interact with the environment through sensors and actuators. Together, these tightly coupled parts enable CPS to perform complex, adaptive tasks across application domains such as healthcare, industrial automation, robotics, avionics, and space.

In modern industrial applications, these systems often take the form of distributed CPS (dCPS), where multiple cyber-physical subsystems, each with its own computing and sensing capabilities, are networked together. Such distributed architecture enhances functionality, scalability, and system-wide coordination, but also introduces substantial architectural and behavioral complexity.

As the size and complexity of dCPS continue to grow, manufacturers face significant challenges in designing their next-generation machines based on dCPS. A key challenge is to evaluate how design decisions influence non-functional properties of dCPS such as performance, energy consumption, and cost. These evaluations must take place early in the design and development process, when critical architectural choices are still open. To support this, system designers need methods that can efficiently explore design alternatives and provide rapid answers to "what-if" questions about the system behavior under different configurations. This necessitates scalable and efficient system-level Design Space Exploration (DSE) methods tailored to the unique characteristics of industrial dCPS [107]. Such methods must integrate appropriate models of hardware and software behavior, efficient simulation and optimization techniques, and supporting tools for exploring a wide range of configurations.

Most of the existing and well established DSE methods have been developed for embedded Systems-on-Chip (SoC) and are not directly applicable to industrial dCPS due to several obstacles. First, these methods typically lack a global system model that can efficiently capture the behavior of distributed and heterogeneous components interconnected with a complex network infrastructure. Second, manually constructing such model is extremely difficult and often infeasible due to the scale and dynamic behavior of industrial dCPS. Third, even when suitable models can be constructed, often there is no scalable simulation and optimization engine that can efficiently explore large system-level design spaces.

The aforementioned limitations call for new methodologies specifically designed for industrial dCPS. These methodologies must combine efficient high-level system modeling, automated model derivation techniques, and scalable simulation tools to support meaningful and practical system-level evaluation and design space exploration.

1.2 Scope and Focus of This Thesis

To address the limitations discussed above, this thesis presents and develops a methodology for scalable and efficient DSE tailored to industrial dCPS. The focus is on enabling informed design-time decisions during the early development phase of the cyber part of dCPS, which plays a central role in shaping the system behavior and performance. As dCPS expand in size and variability, the design and optimization of their cyber components becomes increasingly complex, especially in the face of heterogeneous architectures and dynamic runtime conditions [6]. This complexity is further amplified by the trend toward software-intensive architectures, where more functionality is implemented in software to facilitate adaptability and faster system evolution [1]. As a result, the cyber part not only introduces many of the key design challenges but also offers the most leverage for system-level optimization, making it a strategic focus for DSE.

The proposed methodology in this thesis defines appropriate system-level models at a high level of abstraction, while preserving sufficient accuracy to support meaningful performance analysis and architectural trade-off evaluation. In addition to the model definitions, it incorporates techniques and tools for automatically deriving these models from run time data collected from existing dCPS and transforming them into executable representations. These representations form the foundation for scalable simulation and performance analysis.

To validate the methodology, it is applied to the TWINSCAN lithography system,

one of the advanced machines developed by ASML [108]. This system represents a complex industrial dCPS and provides a concrete basis for assessing the effectiveness of the proposed DSE methodology.

1.3 Scientific Challenges

To realize a practical and scalable DSE methodology for industrial dCPS, several scientific and technical challenges must be addressed. These challenges cover the entire methodology, from defining a conceptual modeling framework for system-level DSE to developing executable models, automation tools, and efficient simulation environments. In particular, DSE must be adapted to the nature of industrial dCPS, that are typically distributed, heterogeneous, and exhibit dynamic execution behavior. These characteristics introduce obstacles that cannot be adequately addressed by traditional embedded system design techniques.

A fundamental starting point is the construction of the overall DSE methodology. As indicated in Section 1.1, the methodology must combine system modeling, automated model derivation and transformation techniques, and system performance evaluation tools. This requirement leads to the first challenge:

CHL1: Devise a DSE methodology for the cyber part of complex industrial dCPS that integrates efficient system modeling, automated model derivation and transformation, and system performance evaluation into a coherent and scalable workflow.

A mature and effective DSE methodology for the cyber part of dCPS requires system modeling that encompasses three core models [50, 57]:

1. A **hardware platform model** describing the available computing infrastructure, such as processing components and interconnection networks;

2. An **application software model** describing the behavior of distributed software processes and their interactions;

3. A **mapping model** specifying the assignment of the software processes to the hardware processing components.

Taking into account the high complexity of industrial dCPS, the aforementioned

models must represent the system at a high level of abstraction in order to manage the complexity and enable efficient and scalable DSE.

Additionally, industrial dCPS typically operate in multiple operation modes. An operation mode represents a specific operational condition resulting from a particular combination of hardware and/or software configuration parameters that are externally imposed by the system setup or usage scenario. Different operation modes may lead to different execution behaviors, for example in terms of resource utilization, execution order, and interaction patterns between system components. Therefore, the system modeling must be able to capture multiple operation modes in an abstract way.

Moreover, to support model reuse during exploration of the application software mapped onto different dCPS hardware platform architectures, the application software model must remain independent from platform-specific artifacts. All these requirements lead to the second challenge:

CHL2: Define an application software model and a hardware platform model that are independent of each other and highly abstract, yet sufficiently accurate to support meaningful and practical performance analysis of dCPS in different operation modes.

Even with such models defined, manually constructing them for an industrial-scale dCPS is infeasible. Such dCPS often consists of hundreds of interacting software processes distributed across multiple computing nodes, with behavior that varies significantly across operation modes and configurations. Capturing this behavior accurately, while ensuring the modeling process remains scalable, is extremely challenging. In addition, utilizing techniques and tools for automated source code analysis to infer system behavior is often infeasible. Many industrial dCPS are composed of huge codebases developed in multiple programming languages, making it extremely difficult, or even impossible, to extract software process dependencies and mode-specific behaviors through static source code analysis. Moreover, the source code may not always be fully accessible due to intellectual property constraints.

Nevertheless, most industrial dCPS include runtime monitoring and logging mechanisms to support debugging, diagnostics, and testing. This existing infrastructure can be used to collect execution traces containing detailed runtime data collected from real system executions. Traces record events such as process activations, message exchanges, and synchronization points, providing a practical and scalable source of

system behavior insights across different system operation modes. Moreover, such system execution traces could be leveraged to derive abstract system models in an automated way. This enables scalable model derivation across heterogeneous subsystems requiring insignificant manual engineering effort and very limited (or no) access to the application software source code. Such trace-based model derivation approach is particularly valuable for complex industrial dCPS, where static analysis of the source code is infeasible.

However, the usefulness of system execution traces for automated system model derivation depends on the type of collected data, its level of detail, and whether it reflects the system behavior across relevant scenarios (system operation modes). The right level of detail and the type of runtime data to be collected are crucial factors in making the trace-based model derivation both accurate and scalable. This leads to the third challenge:

CHL3: Identify the runtime system execution data to collect, and specify how to use it for automated derivation of the highly abstract and sufficiently accurate application software and hardware platform models needed for DSE of complex industrial dCPS.

After obtaining the abstract models, we need to represent them in an appropriate form which allows performance analysis. There are generally two approaches to achieving this: translating the models into analytical representations for analytical-based performance analysis or transforming them into executable representations for simulation-based performance analysis.

Analytical approaches, such as using queueing models or formal verification techniques, operate on mathematical abstractions to derive performance metrics. These techniques are effective and sufficiently accurate when the system structure and behavior are static and predictable to a large extent, thereby allowing strong simplifying assumptions. However, such conditions rarely hold in large-scale industrial dCPS, where software components are developed independently, run on heterogeneous platforms, and exhibit mode-dependent behavior that varies across operational configurations. These systems often rely on event-driven communication and synchronization, and involve tight coordination among hundreds of processes. Analytical approaches struggle to accommodate this level of behavioral variability and fine-grained timing complexity, making them inadequate for evaluating realistic industrial scenarios. Attempting to

translate such complex systems into overly simplified analytical models risks losing essential details, leading to inaccurate performance insights.

In contrast, simulation approaches do not rely on static and strict structural assumptions. Instead, they evaluate the system through the replay or generation of representative execution scenarios, including varying workloads in different operation modes. This flexibility allows simulation approaches to faithfully capture event-driven interactions, runtime variability, and concurrent execution behavior. Additionally, in many cases, simulations scale effectively with system complexity. They support the exploration of a wide range of design alternatives by reusing the same abstract models defined per operation mode, while varying some parameters in the mapping and/or hardware platform models. This is particularly valuable in DSE, where the objective is not to evaluate a single design instance, but to assess a large set of possible architectures and configurations. For these reasons, a simulation approach is adopted in this work as the foundation for performance analysis in DSE. To enable simulation-based performance analysis, the abstract models must be transformed into executable simulation entities, thereby introducing the next challenge:

CHL4: Generate executable entities from the abstract models that are suitable for rapid simulation which enables efficient performance analysis of complex industrial dCPS.

Finally, while existing DSE studies for dCPS primarily focus on analyzing and optimizing the hardware platform and the assignment of software processes to hardware processing components, they often overlook a crucial factor: the internal structure and execution order of the software workload itself. The execution order of software processes and tasks within every process significantly impacts dCPS performance, particularly in terms of exploiting potential parallelism. Specifically, two types of parallelism are essential:

1. **Inter-process parallelism** which involves parallel execution among different software processes;
2. **Intra-process parallelism** referring to parallel execution of computation and communication tasks within a single process.

To make these types of parallelism analyzable and amenable for exploration during DSE, the application software model must capture both inter- and intra-process dependencies at a suitable level of abstraction. Such a model not only enables DSE

to identify and explore opportunities for parallel execution, but also ensures that any increase in hardware parallelism, such as adding more hardware processing components in the cyber part of dCPS, can be effectively utilized by exposing sufficient degree of software-level parallelism. This is particularly valuable for large-scale software, running on industrial dCPS, that is often proprietary and expensive to significantly modify but may still benefit from performance improvements through better scheduling and hardware utilization. This leads to our final challenge:

CHL5: Automatically derive an abstract application software model that captures both inter- and intra-process dependencies in order to enable the analysis, exploration, and exploitation of available parallelism without significantly modifying the original software infrastructure in complex industrial dCPS.

1.4 Research Contributions

This thesis addresses the five scientific challenges presented in Section 1.3 by devising, developing, and validating a comprehensive methodology, called **CompDSE**, for scalable and efficient DSE tailored to the computing (cyber) parts of complex industrial dCPS. The methodology integrates system-level modeling, automated model derivation and transformations, and simulation-based performance analysis into a coherent and tool-supported workflow.

To address challenge **CHL1**, the thesis introduces CompDSE as a structured DSE methodology composed of three main components: abstract models, an integrated toolbox, and simulation-based environment. The abstract models are derived from runtime execution traces, collected from an existing industrial dCPS, where DSE is applied to explore and design the next-generation alternatives of the dCPS. The toolbox includes algorithms for automated model derivation from execution traces, model-to-model transformations to convert the derived model into executable entities, and simulation code generation. These components enable the performance evaluation of each design configuration through discrete-event simulation.

In response to challenge **CHL2**, we devise three core models: an *application workload model* representing the software infrastructure of dCPS, a *delay-based hardware platform model* representing the hardware infrastructure of dCPS, and a *workload-to-platform mapping model* specifying the distribution of the software workload on the hardware

platform. These models are defined at a very high level of abstraction while maintaining sufficient accuracy for system-level performance evaluation of complex industrial dCPS.

The abstract application workload model captures the software workload as a set of processes that exchange messages via communication channels. For each dCPS operation mode, a process is represented as a time-agnostic sequence of computation and communication events that reflects the software execution flow without relying on platform-specific information. Every event has an abstract signature that models the workload of the event imposed on the hardware platform. Besides the workload signature, communication events include the size of the message which is communicated. Since we can derive the model from several execution traces representing different dCPS operation modes, the model captures mode variability in dCPS, enabling different mode-related software workloads to be modeled under varying configurations.

The abstract delay-based hardware platform model captures the hardware platform topology and, when transformed to executable entities, emulates computation and communication delays associated with the corresponding events in the abstract workload model. The hardware platform model consists of *hosts* connected to one or more *switches*, where each host contains multiple computing cores characterized by specific operating frequencies. While hosts are responsible for emulating the delays of computation events, switches emulate the delays of communication events as network transfer delays associated with exchanged messages between processes over a communication network.

The workload-to-platform mapping model specifies the assignment of processes in the application workload model to the hosts in the hardware platform model, thereby relating these two core models to form an abstract dCPS system model. In addition to the three core models, we have developed and incorporated in CompDSE an *environmental influence model* to account for timing effects originating from subsystems that could not be traced due to system constraints. These subsystems, which typically interact with the physical environment (e.g., sensors, actuators, or mechanical components), are not explicitly modeled. However, their interaction with traced components (hosts) introduces timing variations that influence the overall system performance. Therefore, the environmental influence model complements the abstract dCPS system model by implicitly and abstractly modeling the untraceable subsystems.

To address challenge **CHL3**, we study thoroughly traceable components within complex industrial dCPS and observe that the runtime data needed to derive the aforementioned abstract models is spread across both the user space and kernel space of the operating system (OS) running on these components. Therefore, the OS is

selected and utilized as the source for obtaining execution traces that combine standard kernel-space events with custom user-space events generated via software source code instrumentation.

In the user space, traces are collected by adding trace points in key library functions such as send and receive, that are commonly used by software processes and located in shared libraries. This approach avoids the need to instrument the source code of individual processes. The collected user-space traces include event timestamps, process names, and details of the instrumented functions, such as the size of the messages transferred by processes. These traces are primarily used to derive the processes, communication channels, and computation and communication events in the abstract application workload model. They also contribute to the derivation of the mapping model and parts of the abstract hardware platform model.

In the kernel space, relevant standard kernel events are collected to derive the abstract workload signatures of the events in the workload model and to provide complementary information for deriving the abstract hardware platform model. Having execution traces with the collected data, described above, we devise and implement several algorithms to automatically derive the three core abstract models utilized within the CompDSE methodology. These algorithms are included in the CompDSE toolbox mentioned earlier.

In addressing challenge **CHL4**, we develop and utilize model-to-model transformation algorithms, included in the CompDSE toolbox, that transform the automatically derived abstract application workload, platform, and mapping models into executable simulation entities. These executable entities include process entities for the application workload model, a centralized mapper entity representing the mapping model, and host and switch entities for the hardware platform model. The process entities generate and send computation and communication events to the mapper entity, which dispatches them to the appropriate host entities according to the specified mapping. Communication events are routed from the source host through the switch entities to the destination host. All these entities can interact and synchronize with the help of any discrete-event simulation engine, where host and switch entities simply emulate computation and communication timing delays only in order to support efficient and scalable system performance evaluation. In the CompDSE toolbox, we provide code generation algorithms that convert these executable entities into a specific code format targeting the well-known OMNet++ discrete-event simulation environment [113].

Finally, in response to challenge **CHL5**, we propose an important refinement of the abstract application workload model in order to capture both inter- and intra-

Chapters vs. Challenges	CHL1	CHL2	CHL3	CHL4	CHL5
Chapter 3: CompDSE Method+Models	✓	✓		✓	
Chapter 4: CompDSE Tools			✓	✓	
Chapter 5: CompDSE Evaluation				✓	
Chapter 6: Workload Model Refinement					✓

Table 1.1: Relation between the thesis chapters and research challenges

process parallelism. The most essential feature of the refined model is the grouping of cohesive computation and communication events into semantically meaningful tasks within every process that potentially could run in parallel. Each task corresponds to a sequence of events that need to be performed in response to an internal or external trigger that fires a process. Moreover, we propose a dependency analysis approach to identify opportunities for safely reordering and executing these tasks in parallel, without requiring significant modifications to the source code of the software infrastructure in dCPS. This approach enables deeper insights into the concurrency characteristics of the software workload and reveals optimization opportunities for better hardware resource utilization.

1.5 Thesis Structure

The remainder of this thesis is structured as follows:

Chapter 2 provides the necessary background to understand the thesis contributions. It introduces the principles of DSE, the Y-chart modeling approach, and the limitations of current DSE tools when applied to complex industrial dCPS. It also presents the ASML TWINSCAN machine as a representative industrial dCPS and discusses the nature of event-driven execution in such systems. Additionally, it explains the distinction between user-space and kernel-space execution traces and outlines general strategies for collecting them, providing foundational context for the trace-based model derivation presented in later chapters. It also introduces some existing facilities used to support modeling, transformation, and simulation activities throughout the thesis.

Chapters 3 through 6 present, in great detail, the main research contributions that directly address the scientific challenges introduced in Section 1.3. Table 1.1 lists these chapters and shows the research challenges addressed in each chapter.

Chapter 3 describes the proposed CompDSE methodology, defines its abstract models, and explains how these models are represented as executable entities suitable

for discrete-event simulation, thereby addressing challenges **CHL1** and **CHL2**, and partly challenge **CHL4**.

Chapter 4 presents the toolbox of CompDSE. More specifically, it explains in detail the specific user-space and kernel-space execution traces and describes the algorithms for automatically deriving the abstract models from these execution traces, thereby addressing challenge **CHL3**. In addition, it introduces the algorithms to transform the derived abstract models into executable simulation entities and to generate simulation-ready code targeting OMNet++, thereby addressing challenge **CHL4**.

Chapter 5 evaluates the accuracy and efficiency of the CompDSE methodology by applying it to the ASML TWINSCAN machine. The efficiency is measured by recording the time required to evaluate individual design configurations. Additionally, through a set of manual DSE experiments, the chapter demonstrates the methodology's capability to evaluate the performance of alternative hardware platform designs. This confirms that the CompDSE methodology is sufficiently fast and practical for DSE in industrial dCPS, thereby addressing the efficiency aspect of challenge **CHL4**.

Chapter 6 presents the refinement of the abstract workload model to capture both inter- and intra-process dependencies. It also describes in detail the dependency analysis approach to identify opportunities for parallel execution of tasks within processes, thereby addressing challenge **CHL5**.

Finally, Chapter 7 concludes the thesis by summarizing the main contributions and outlining directions for future work.

Chapter 2

Background

Publications related to this chapter

- Marius Herget, Faezeh Sadat Saadatmand, Martin Bor, Ignacio González Alonso, Todor Stefanov, Benny Akesson, and Andy D. Pimentel. Design Space Exploration for Distributed Cyber-Physical Systems: State-of-the-art, Challenges, and Directions. In *Euromicro Conference on Digital System Design (DSD)*, pages 632–640. IEEE, 2022.

In this chapter, we provide background on key concepts, technologies, and facilities relevant to the thesis. Section 2.1 introduces DSE, outlining its general workflow and the Y-chart modeling approach as a foundation for system-level design analysis. Section 2.2 discusses the limitations of existing embedded DSE tools when applied to large-scale industrial dCPS. Section 2.3 describes the ASML TWINSCAN machine as a representative industrial case study, followed by Section 2.4, which explains event-driven execution as a common execution paradigm in such systems. Section 2.5 clarifies the distinction between user-space and kernel-space execution traces and outlines general strategies for collecting them. Finally, Section 2.6 reviews the simulation environments and modeling frameworks employed throughout this thesis.

2.1 Design Space Exploration

DSE refers to a systematic process for exploring alternative system designs by evaluating and comparing configurations based on one or more optimization objectives. It is widely used in early design phases of a system to support informed decisions based on analysis, helping system developers understand the implications of different design choices before actual implementation, thereby reducing costly iterations and late-stage changes of the system. This is particularly valuable when designing complex systems where multiple interacting components, resource constraints, and conflicting objectives must be carefully balanced [68].

A system configuration evaluated within the design space is referred to as a *design point*. Each design point represents a unique combination of system parameters and design decisions that can be assessed against one or more optimization objectives. DSE is typically formulated as either a *single-objective* or *multi-objective* optimization problem. In single-objective DSE, the focus is on optimizing a specific goal such as execution time, energy consumption, or safety. In contrast, multi-objective DSE evaluates trade-offs among multiple and often conflicting goals – such as latency, reliability, power consumption, and cost – to identify *Pareto-optimal* solutions that offer balanced trade-offs. To guide the search and evaluate each design point, a *fitness* function is defined that quantifies how well a particular configuration satisfies the objective(s). This function may be a single metric or a composite score combining several criteria, depending on the DSE formulation.

DSE is commonly characterized along two complementary dimensions: 1) the evaluation of individual design points using appropriate models and analysis methods, and 2) the systematic coverage of the design space, given that exhaustive exploration is typically infeasible [46]. In practice, these two dimensions are closely related. More accurate evaluation of a design point often requires more complex models and higher computational effort. This, in turn, limits how many design points can be explored and the degree to which the exploration can be fully automated. These aspects together define the core challenges addressed by DSE methodologies.

Although DSE is a general methodology with applications across a wide range of domains, including hardware design, software optimization, robotics, and CPS, it has received significant attention in the embedded systems domain. This is because embedded systems are particularly constrained by non-functional requirements such as timing, energy efficiency, and reliability, and typically operate under tight resource limitations. In this context, DSE is often used to explore architectural configura-

tions, task-to-resource mappings, and scheduling strategies to meet these constraints effectively [79]. Early DSE toolchains for embedded systems were often built around ESL (Electronic System Level) modeling approaches [40], which describe the system behavior and architecture at a high level of abstraction. These approaches were realized using different modeling infrastructures, including but not limited to SystemC-based frameworks. Well-known ESL frameworks supporting DSE for embedded and MPSoC systems include SystemCoDesigner [56], Metropolis [12], Ptolemy II [28], and Daedalus [92, 101]. While these frameworks differ in their underlying modeling formalisms and tool support, they share a common focus on early evaluation of architectural alternatives and application-to-platform mappings. For example, Metropolis and Ptolemy II emphasize on flexible modeling and analysis across multiple models of computation (MoCs), whereas Daedalus emphasizes on automated exploration and synthesis flows, and SystemCoDesigner focuses on tool-supported exploration and architectural synthesis for embedded MPSoCs.

The importance of DSE is increasingly recognized in industry due to rising system complexity, growing demands for product customization, and the need to ensure high performance under diverse and dynamic operating conditions. A recent study based on a survey of practitioners and researchers highlights that while performance challenges are well known in industrial contexts, there is still limited adoption of structured DSE methods in practice [107]. The findings show that many performance issues are detected too late in the development cycle, often during integration or system operation, leading to costly redesigns. Consequently, there is a clear industrial demand for earlier and more systematic evaluation of design alternatives to mitigate these late-stage design problems.

2.1.1 General Workflow of DSE

The DSE process is typically structured around a systematic workflow composed of five key steps [78]:

1. **Modeling:** Abstract models are developed to capture the system's structure and behavior at a high level. This system modeling typically follows the well-known and established Y-chart approach [57], as briefly explained in Section 2.1.2.

2. **Design Space Definition:** The design space is defined by identifying all relevant parameters, decisions, and constraints that characterize potential system configurations. This includes, for example, the number and type of hardware

processing components, communication infrastructure, memory hierarchy, application task scheduling policies, and mappings between application tasks and hardware components. The resulting design space may be extremely large and multi-dimensional.

3. **Exploration Strategy:** Various strategies can be used to explore the design space, depending on its size and complexity. These include exhaustive search in the design space, random sampling, manual selection, or heuristic and metaheuristic methods. Common metaheuristic techniques include Genetic Algorithms (GA), Simulated Annealing, Particle Swarm Optimization, and Ant Colony Optimization [96]. These methods aim to efficiently explore large design spaces while identifying design solutions that perform well with respect to the system's optimization objectives and constraints.

4. **Performance Estimation:** For each selected design point during the search in the design space, one or several performance metrics are estimated using analytical techniques, simulation techniques, or a combination of both. Typical metrics include system execution time, energy consumption, resource utilization, reliability, and cost.

5. **Analysis and Selection:** The evaluated design points are analyzed to identify Pareto-optimal design solutions and to understand trade-offs between conflicting optimization objectives. Based on this analysis, designers can select one or more system design solutions that best align with the system's goals and constraints.

2.1.2 Y-Chart Modeling Approach

The Y-chart modeling approach is a well-established methodology that structures the system-level modeling and exploration process to support DSE in embedded systems and CPS [57]. It provides a structured way to separate concerns during system-level design by modeling the system from three distinct perspectives. These perspectives are typically visualized in the form of a "Y" shape, as shown in Figure 2.1, where each branch represents one model contributing to the performance evaluation process. The three core model types in the Y-chart are as follows:

- **Hardware platform model** describing the available computing infrastructure, such as processing components and interconnection networks;

- **Application software model** describing the behavior of distributed software processes and their interactions;

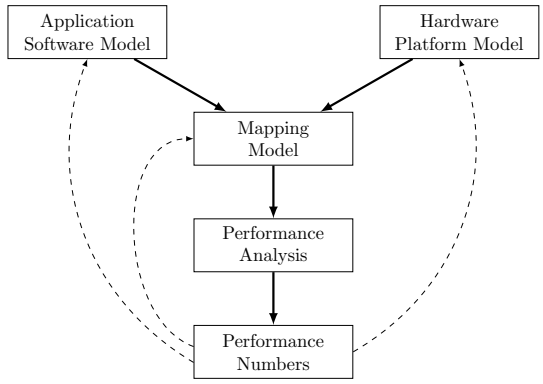


Figure 2.1: Illustration of the Y-chart modeling approach

- **Mapping model** specifying the assignment of the software processes to the hardware processing components.

The interaction between these three models defines the scope of design alternatives considered during exploration. Each unique combination of an application, a platform, and a mapping constitutes a design point that can be evaluated in terms of performance.

As depicted by the dashed arrows in Figure 2.1, the Y-chart supports an iterative design process. Insights gained from performance analysis can be used to refine the application model, adjust the platform architecture, or improve the mapping. This feedback loop enables designers to explore alternatives systematically, reason about trade-offs, and converge toward optimal or satisfactory design solutions. Depending on the design objective, one or more of these dimensions may be fixed while the others are explored.

A key advantage of the Y-chart approach is its separation of concerns, which allows the application and the platform model to be independently developed, analyzed, and reused. Such model separation supports scalability, improves design re-usability, and enables efficient exploration of large design spaces.

2.2 Limitations of Embedded DSE Tools for Industrial dCPS

While the foundational principles of DSE, such as separation of concerns and abstract modeling, apply to both embedded systems and dCPS, existing embedded DSE tools exhibit significant scalability limitations when applied to large-scale or industrial dCPS.

In this subsection, we demonstrate these limitations through an experiment.

In our experiment, we used SESAME (Simulation of Embedded System Architectures for Multilevel Exploration) [80], a well-established framework for system-level modeling, simulation, and design space exploration of embedded Multiprocessor System-on-Chips (MPSoCs). We selected SESAME because it supports high-level abstract modeling of both application workloads and hardware architectures, which enables efficient DSE for these systems. For the application model, we chose a publicly available multimedia streaming application. This choice is motivated by the observation that, at a high level of abstraction, the dataflow nature of many industrial CPS workloads aligns well with the streaming application model.

Because such MPSoC applications are typically small (the selected one includes only six processes), we replicate the application multiple times to simulate larger workloads. For the platform model, we defined an MPSoC architecture with four types of processors, each with different cost and performance characteristics. To simplify the communication between processing components, all components share a single bus and memory. To eliminate the impact of communication bottlenecks, we model an idealized communication system by setting all memory and communication costs to zero.

To approximate the scale of industrial dCPS, we scaled the platform to support up to several hundred processing components. In our experiment, we explore different mappings of the application workload (processes) onto the underlying platform architecture. We used a GA with a population size of 20, and each DSE run consists of 50 search iterations (generations), exploring a total of 1000 design points per run. Figure 2.2 shows the average DSE runtime as a heatmap, with the number of application processes on the X-axis and the number of processing components on the Y-axis.

Each experiment is conducted two to four times using the same input parameters, and the results are averaged. Due to resource constraints, some experiments are conducted only once or twice. These are marked with white shading in Figure 2.2.

The 458 experiments accumulate a total runtime of approximately 64 days and 2 hours. The results show that the DSE runtime increases significantly, growing faster than linearly with respect to the number of application processes and processing components. This effect is especially pronounced when scaling the number of application processes, primarily due to increased simulation overhead.

Even in this simplified setting with idealized communication and replicated independent applications, exploring a design space involving 100 application processes and tens of processing components requires substantial computational effort. While the number of possible mappings is large, the overall structure of the design space remains

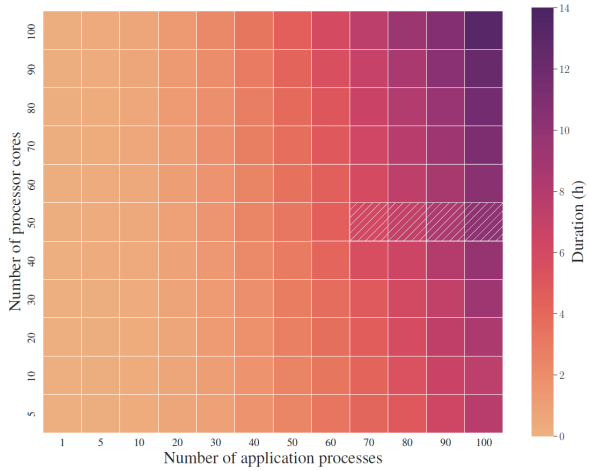


Figure 2.2: Heatmap of the DSE runtime (in hours) by varying the number of application processes and processing components

relatively simple and predictable due to the absence of inter-dependencies between application processes and the use of an idealized communication model in the platform.

Because of these simplifications, such as allowing every process to communicate with any other process regardless of the processing component to which it is assigned, our experiments do not account for the overhead of repairing invalid design points. In realistic scenarios involving heterogeneous architectures and interdependent processes, additional overhead from invalid mappings and complex interactions would likely result in longer runtimes. Therefore, our experimental results represent a best-case scenario with respect to the required DSE runtime.

The aforementioned findings show that even under simplified conditions, current embedded DSE tools are only partially effective when applied to (emulated) large-scale industrial dCPS. In more realistic cases, the optimization process is likely to converge more slowly toward design points that satisfy performance constraints, while also exhibiting reduced search efficiency and encountering a higher proportion of infeasible configurations. These limitations would further hinder the practical applicability of existing tools to complex industrial systems.

This example highlights the need for new modeling abstractions and scalable DSE tools specifically tailored to the characteristics of industrial dCPS, where evaluating realistic application workloads mapped onto large-scale dCPS architectures within acceptable time bounds remains a fundamental challenge.

2.3 ASML TWINSCAN Machine

As representative examples of large-scale industrial dCPS, the ASML lithography machines [108] play a central role in semiconductor manufacturing by enabling the production of high-density integrated circuits. These machines use precision optics to project circuit patterns, defined by reticles or masks, onto photoresist-coated silicon wafers. A wafer is a thin, circular slice of semiconductor material, typically processed in batches called *lots*, each consisting of multiple wafers handled together during manufacturing.

As the name suggests, the ASML *TWINSCAN* machine employs a dual-stage scanning mechanism. In the first stage, various sensors measure the precise location of the wafer. In the second stage, this measurement data is used, along with a set of actuators, to perform the actual exposure process. By executing these two stages simultaneously on separate wafers, the system achieves higher production throughput and improved efficiency.

To initiate wafer processing, detailed *recipes* must be defined. These recipes specify the number of lots, the number and source of wafers in each lot, and, most importantly, various exposure and measurement parameters. The diversity in recipes results in different application workloads, each corresponding to a specific *operation mode*.

The cyber part of the ASML TWINSCAN machines consists of a complex software infrastructure distributed across heterogeneous multi-core subsystems connected via high-speed networks. These subsystems, running on various OS platforms, are categorized based on their function as either embedded real-time or server-based systems. The server-based subsystems primarily run on *Linux* OS, whereas most of the real-time embedded subsystems operate on *Wind River VxWorks* OSs.

Within this infrastructure, hundreds of processes execute concurrently and exchange thousands of messages per second to maintain tight coordination across subsystems. This high volume of communication and processing adds to the complexity of managing and optimizing the system performance, particularly when accounting for the real-time constraints of the embedded subsystems and the scalability of the server-based subsystems.

2.4 Event-Driven Execution in Distributed Systems

High-performance distributed systems, including industrial dCPS, commonly use an event-driven execution model, where software processes respond to incoming events

rather than executing in a fixed sequence. At the core of such execution model is the event loop, which continuously listens for system events, such as messages, timer expirations, or scheduling triggers, and asynchronously invokes the appropriate tasks.

When an event occurs, the corresponding process executes the required computation or communication tasks and responds by either sending a message or triggering another event. This reactive approach enables efficient handling of multiple requests by ensuring that processes are activated only in response to relevant events, thereby avoiding unnecessary or periodic execution. Depending on the system's workload characteristics and available resources, the event-driven execution may follow a single-threaded strategy where events are handled sequentially, or a multi-threaded approach where events are processed in parallel by worker threads.

2.5 Execution Traces

Execution traces provide a foundational mechanism to observe, analyze, and monitor the behavior of a system. They record events during the system runtime, allowing detailed reconstruction of how software components and hardware resources interact over time. In dCPS, tracing is essential for understanding the coordination and interaction among concurrently executing components. An industrial dCPS, such as the ASML TWINSCAN machine, often includes built-in capabilities for collecting traces, logs, and runtime statistics to support system-level observability.

An execution trace is a temporally ordered sequence of runtime events, such as function calls, memory accesses, task activations, or message transfers. Each event typically includes metadata such as timestamps, thread or process identifiers, and action descriptors. Traces can capture both high-level application behavior and low-level hardware activities. In general, traces can be collected at different system levels via the operating system (OS):

- **User-space traces:** Collected from the user-space part of the system, typically by instrumenting application software. Developers can insert *tracepoints* to record custom events and parameters. These traces are useful for analyzing the logical flow of application-level processes and their interactions.
- **Kernel-space traces:** Collected from the kernel-space of the OS, providing insight into low-level system behavior such as context switches, system calls, interrupt handling, and CPU scheduling. These are essential for diagnosing timing issues and resource contention in multitasking or real-time environments.

Execution traces may be stored in various formats, from human-readable to compact binary encodings. Binary formats are generally preferred for efficiency and are processed offline using specialized tools after collection. In Linux-based systems, the *Common Trace Format* (CTF) is a binary format that is widely used. The `LTTng` (Linux Trace Toolkit: Next Generation) framework [30] is a well-known tool for collecting both user-space and kernel-space traces in CTF format. For post-analysis and visualization, the open-source tool `Trace Compass` [104] supports CTF and provides interactive capabilities such as filtering, correlation across traces, and timeline inspection.

Tracing is also supported in embedded real-time operating systems. In particular, the Wind River VxWorks OS provides built-in tracing capabilities through the `System Viewer` tool [119], which records events from both the kernel space and user space. Trace data is stored in a proprietary binary format (with the `.wvr` extension) and can be analyzed and visualized using the `Wind River Workbench environment`.

When setting up trace collection, several practical factors must be considered, such as which events to record, where to store them, and how to transfer the traced data for analysis. In some systems, traces are transferred after the entire collection process is complete, while in others, they must be transferred during the system runtime. For instance, in the TWINSCAN machines, embedded subsystems may lack sufficient memory to store traces locally. Instead, traces are streamed over the network during runtime. In such cases, it is important to manage the network bandwidth carefully to avoid trace streaming interference with the system's real-time behavior during trace collection.

2.6 Simulation and Modeling Environment

This section outlines the simulation and modeling environment used to support the system model transformation and performance evaluation in our CompDSE methodology. We employ a combination of tools to carry out these activities. The OMNet++ library and framework is used to simulate our models for performance evaluation. The model construction is carried out using the Eclipse Modeling Framework (EMF), while the model transformation is implemented using the Xtend programming language. Each of these tools and their roles in our methodology are described in the following subsections.

2.6.1 Discrete-Event Simulation using OMNet++

OMNet++ [113] is an open source, extensible, and component-based discrete event simulation environment designed for building and simulating large-scale communication networks and distributed systems. It supports a hierarchical module structure, event scheduling, and message passing between components, making it suitable for modeling systems with complex interactions.

In our work, OMNet++ is used as the simulation environment for system performance evaluation. Within this environment, we simulate the system models in three layers: application workload, hardware platform, and mapping. This setup allows us to observe the performance outcomes of different design points by analyzing how these layers are combined and interact with one another.

Although the system performance evaluation in the CompDSE methodology can be conducted using any discrete event simulator with the appropriate code generation and integration steps, we select OMNet++ because it supports relatively large components and provides good simulation performance.

2.6.2 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [93] is a modeling and code generation framework for building structured data models. It allows developers to define models using a metamodeling approach and automatically generates corresponding Java code for model manipulation, serialization, and data persistence.

In our work, EMF is used to construct models representing different layers of the system, including the application workload, hardware platform, and mapping. These models are described using *Ecore*, the EMF's metamodeling language, which provides a structured way to describe model elements, attributes, and relationships. By leveraging EMF, we ensure consistency across model representations and enable automation in model transformation and integration processes.

2.6.3 Model Transformation with Xtend

Xtend [32] is a high-level programming language built on top of Java, designed to support model transformation and template-based text generation. It offers concise syntax, type inference, and powerful templating features that make it well-suited for generating code and configuration files from the models. Xtend integrates seamlessly with the Java environment, allowing developers to reuse existing Java libraries and tools,

which is particularly beneficial in model-driven engineering workflows. In our approach, Xtend is used to implement transformation rules and code generation templates at multiple levels: from data in execution traces to system models described in EMF format, from EMF-format models to executable entities, and from executable entities to simulation-ready code compatible with OMNet++.

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

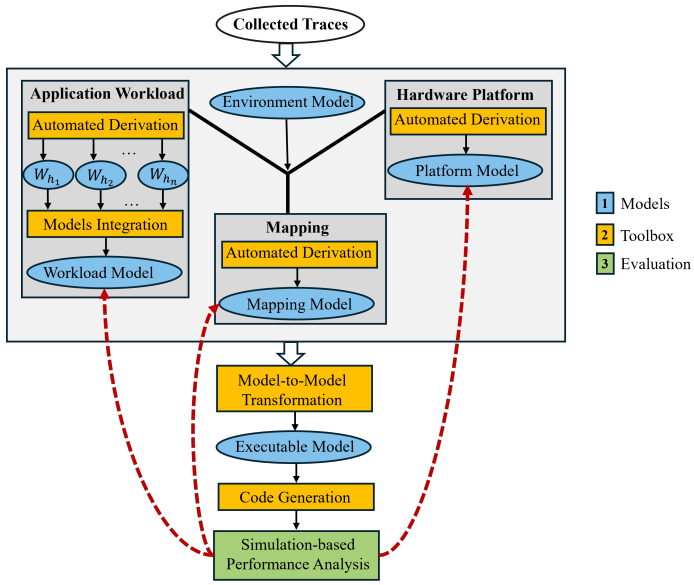


Figure 3.1: The CompDSE methodology

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 $P = \{p_1, ..., p_{|P|}\}$ denotes a set of processes that model the corresponding processes in the dCPS software infrastructure distributed across multiple hosts. Additionally, $C = \{ch_1, ..., ch_{|C|}\}$ 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 $C = \{ch_{AB}, ..., ch_{HG}\}$ 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., $p_i = \{M_1, ..., M_{|p_i|}\}$ where every mode $M_q = \langle e_1, e_2, ..., e_{|M_q|} \rangle$ is a finite sequence of coarse-grained abstract events that models the types of actions performed during a single process firing, arranged in chronological order. Every event $e \in M_q$ falls into one of the following

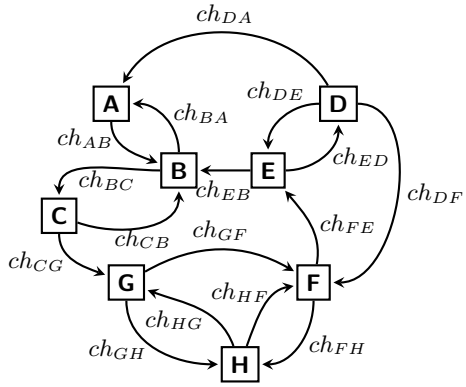


Figure 3.2: Example of an application workload model WK

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:

$$\text{sig} = T_{\texttt{OnCPU}} \cdot f_{\text{CPU}} \tag{3.1}$$

  where $T_{\texttt{OnCPU}}$ 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 $H = \{h_1, \ldots, h_{|H|}\}$ is the set of hosts that represent the computing subsystems in

the dCPS hardware platform, and set $N = \{swch_1, \ldots, swch_{|N|}\}$ 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 $MP = \{MP_1, \ldots, MP_m\}$, 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 $MP_j = \{mp_1, \ldots, mp_{|P|}\}$ 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.

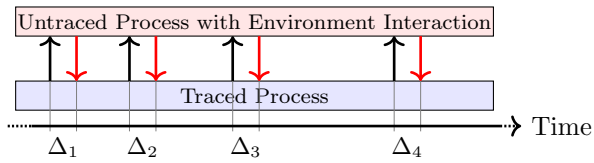


Figure 3.3: Modeling the impact of the physical environment

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:

$$\Delta_i = T^{p_t}_{\text{read},i} - T^{p_t}_{\text{write},i} \tag{3.2}$$

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.

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

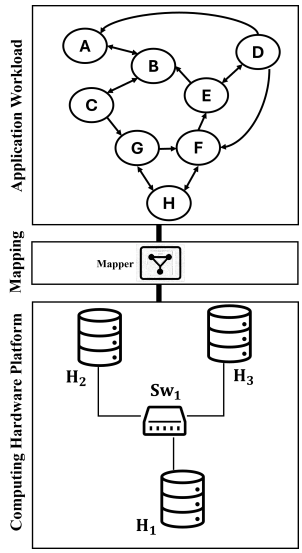


Figure 3.4: The application workload, computing hardware platform, and mapping model entities

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).

```
while (True) {
    if (Q is not empty) { // Check for received events
        event = take the first element of Q;
        if (sender of event is the mapper)
            add event to readyQ;
        else if (sender of event is network) {
            add event to the receiveList;
            if (exists an e in waitList and source of e is the same as event
                source)
                take e from waitList and add it to readyQ;}
    }
    while (readyQ is not empty) {
        exeEv = take next event from readyQ;
        if (type of exeEv is not read) {
            allocate resource to execute exeEv;
            if (type of exeEv is write)
                send exeEv to network;
            send "done" message to mapper for exeEv;
        }
        else {  // read event
            if (exists an e in receiveList and source of e is the same as exeEv
                source) {
               take e from the receiveList and allocate resource to execute it;
               send "done" message to mapper for exeEv;}
            else // Message has not yet arrived
                add exeEv to waitList;
        }}
}
```

Listing 3.3: Execution behavior of a Host entity

```
while (True) {
    if (Q is not empty) { // Check for received events
        event= take the first element of Q;
        add event to the readyQ;
        while (readyQ is not empty) {
            event = take the next event from readyQ;
            transitDelay = divide the message size of event by switch bandwidth;
            delay = transitDelay + congestDelay;
            destHost = determine the host mapped to destination process of event
                ;
            send event to destHost at (simTime + delay);
        }}
}
```

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:

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

where $X = \{x_1, x_2, \ldots, x_N\}$ 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.

Chapter 4

CompDSE Tools

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.

This chapter presents the CompDSE Toolbox, which supports the methodology introduced in Chapter 3, by describing the necessary algorithms, we have devised, to implement the tools. It addresses challenge **CHL3** by defining the required execution traces and enabling the automated derivation of models. It also contributes to challenge **CHL4** by enabling the automated generation of executable model entities suitable for simulation.

Section 4.1 discusses the challenges related to model derivation and simulation readiness. Section 4.2 outlines the key contributions of this chapter, which collectively form the CompDSE Toolbox. Section 4.3 discusses existing trace-based tools for system analysis. Section 4.4 provides a detailed description of the execution traces used for model derivation. Section 4.5 explains the algorithms for deriving and integrating the application workload, hardware platform, and mapping models. It also discusses how these models are transformed into executable entities and describes the code generation process for OMNet++ simulation. Finally, Section 4.6 concludes the chapter.

4.1 Problem Statement

As indicated earlier in this thesis, DSE at the system level for industrial dCPS relies on abstract models that represent the behavior of these complex systems. However, manually constructing these models is impractical for real-world industrial dCPS due to their large scale, software heterogeneity, and dynamic operational behavior.

To address this issue, model derivation must be automated using execution trace data, collected during runtime of a system. However, collecting such data during runtime of industrial large-scale systems presents its own challenges: the execution tracing must not interfere with the system correctness and performance, yet it must provide enough semantic detail to enable the automated derivation of accurate models.

Additionally, to support scalable and efficient DSE, there is a critical need for algorithms and tooling capable of automatically deriving the required models from such execution traces, coherently integrating them into a comprehensive representation, and transforming them into executable entities suitable for simulation. Such toolbox must ensure generality, scalability, and applicability across diverse system configurations.

4.2 Contributions

This chapter presents the CompDSE Toolbox, a set of tools and supporting algorithms that automate the methodology introduced in Chapter 3. The toolbox enables trace-based model derivation and prepares the derived models for performance evaluation by transforming them into executable entities for simulation. The key contributions are:

1. **Execution trace definition:** We define the execution traces required for model derivation and describe how they can be collected from user-space and kernel-space levels using non-intrusive methods that preserve system correctness;

2. **Automated model derivation and integration:** We present the core algorithms in the CompDSE Toolbox for deriving abstract models of the application workload, computing hardware platform, and workload-to-platform mapping from execution traces;

3. **Model-to-model transformation and code generation:** We demonstrate how the derived models can be transformed into executable entities and present concrete examples of simulation-ready code generation for the OMNet++ simulator, showcasing the full flow from trace acquisition to simulation.

4.3 Related Work

Tracing and trace-based analysis have been widely studied as a means to capture and analyze the execution behavior of systems. Depending on the intended purpose, traces can be collected using a wide range of mechanisms and at different abstraction levels, ranging from application-level instrumentation to hardware-level tracing of execution activities. In industrial settings, tracing infrastructures are typically designed to balance low overhead, limited intrusiveness, and practical feasibility under operational constraints.

A dominant use of tracing in both academic and industrial contexts is performance monitoring and diagnostics. Widely used tools and frameworks such as DTrace [24], Perf [116], and LTTng [30] rely on execution traces to analyze system behavior and identify performance bottlenecks in production environments. In industrial CPS, the Platform Performance Suite (PPS) [105] provides a set of tools for trace-based performance analysis based on the Timed Message Sequence Charts (TMSC) formalism [54]. It supports analyses such as critical-path identification and root-cause analysis. Similarly, in modern distributed software systems, observability frameworks such as OpenTelemetry [63] define tracing models to capture end-to-end request flows. In these frameworks, traces are composed of smaller execution segments, called spans, which together capture how a request flows across multiple services [21]. These approaches are primarily designed for operational observability and debugging and typically rely on explicit instrumentation at the application or middleware level.

Beyond performance monitoring, execution traces have also been applied in several other domains. In the area of security, system-call traces and execution sequences have been analyzed for anomaly and intrusion detection [51, 65]. In power and energy analysis, hardware performance events and execution traces have been used to estimate system power consumption and identify energy hotspots [18]. Tracing has also been applied in reliability and fault analysis to support automated diagnosis from runtime records [52].

Execution traces and measurement data have also been used in execution-time analysis, particularly in embedded and real-time systems. In measurement-based approaches, traces or timing measurements are collected to estimate or bound timing properties such as worst-case execution time (WCET) or worst-case response time (WCRT), often in combination with static analysis techniques [118]. In these approaches, traces are used to support the derivation of conservative timing guarantees for scheduling analysis, rather than to derive abstract workload models.

While the aforementioned works demonstrate the broad applicability of tracing as an analysis technique, the abstractions derived from traces are typically tailored to domain-specific objectives, such as diagnosis, verification, or fine-grained performance analysis of a given system configuration. Consequently, many existing approaches rely on detailed trace information. In contrast, in this thesis, we use traces for deriving abstract models suitable for DSE. For this purpose, trace data does not need to capture fine-grained execution details.

In the context of DSE, trace-based techniques have also been explored, primarily in the embedded and MPSoC domains. For example, the SESAME framework uses abstract event traces as application stimuli for trace-driven co-simulation and evaluation of alternative architectures and mappings [109]. Trace-driven workload simulation has also been applied to performance estimation of parameterized MPSoC architectures, such as estimating cycle counts under varying architectural parameters [53]. These approaches are typically applied to relatively simple embedded applications, where the execution context and application structure are known in advance.

In contrast, the approach proposed in this thesis targets large-scale, distributed, industrial systems, where execution behavior emerges from interactions among multiple subsystems. Rather than assuming pre-defined abstract traces, our method automatically derives abstract models from execution traces of individual subsystems and subsequently integrates these models into a system-level representation.

4.4 Execution Traces for Model Derivation

Deriving the CompDSE models requires collecting specific execution traces from the target dCPS. For each system operation mode, traces are collected once from all subsystems (hosts), capturing the execution behavior under a specific hardware/software configuration or usage scenario. In this context, although operation modes are conceptually related to scenarios because both describe distinct execution contexts, their roles differ. Scenario-based design typically defines scenarios as part of the design methodology to guide exploration or validation. In contrast, in this work, operation modes are observed and they are execution contexts determined by existing system configurations and usage conditions, and are not introduced as additional design abstractions.

For every host h_i, there are two sets of traces that are collected, namely set US_{h_i} containing *application-level execution traces* collected from user-space of the OS and set KS_{h_i} containing *system-level execution 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

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

4.4.1 Application-level Execution Traces

Each set $US_{h_i} = \{E_1, \ldots, E_m\}$ contains the application-level execution traces for host h_i across m operation modes. Each 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 inserted. To capture a comprehensive view of the software execution, we strategically insert trace points at the *start* and *end* of the required function calls.

Each individual record r_k is represented as a tuple $r_k = (ts, pn, fn, l, A)$, where ts is the timestamp of the record, pn is the name of the process running at the time, fn is the name of the function call executed in the process at the time, $l \in \{start, end\}$ denotes the location within the function code where the record is collected, and $A = (c, id, \lambda, \theta)$ is a tuple of attributes based on the function type fn.

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

Any message identifier *id* includes Lamport timestamp *lmp*. Lamport time is a logical clock algorithm used in distributed systems to order events and ensure a consistent sequence of operations across different nodes without relying on synchronized physical clocks [62]. Besides *lmp*, for message exchanges, to derive the inter-host communications, we also need to collect the IP address *ip* and port number *prt* for both source and destination processes in the *send* and *receive* functions. Therefore, the message identifier *id* is represented as a tuple $id = (lmp, ip_{src}, ip_{dst}, prt_{src}, prt_{dst})$.

As an example, an excerpt of an application-level execution trace E_j is shown in Figure 4.1, represented in a tabular format. Rows represent the sequence of collected records r_k and columns display the various data items of each record described above.

Figure 4.2 presents a timeline-based visualization of the same execution from Figure 4.1, illustrating the full sequence of function calls (including those not traced)

ts	*pn*	*fn*	*l*	*c*	*id*	λ	θ
0.5	A	f_{01}	start	send	i_0	40	
1.4	B	g_0	start	receive	i_0	40	
2.0	A	f_{01}	end	send	i_0	40	
2.85	B	g_0	end	receive	i_0	40	
3.1	B	g_{11}	start	trigger	j_0		4
4.3	B	g_{11}	end	trigger	j_0		4
7.1	B	g_3	start	handler	j_0		4
8.0	B	g_3	end	handler	j_0		4
8.15	B	g_{41}	start	send	i_1	100	
8.8	A	f_{03}	start	receive	i_1	100	
9.25	B	g_{41}	end	send	i_1	100	
10.43	A	f_{03}	end	receive	i_1	100	
11.0	B	g_{11}	start	trigger	j_1		4
12.0	B	g_{11}	end	trigger	j_1		4
15.15	B	g_3	start	handler	j_1		4
16.25	B	g_3	end	handler	j_1		4

Figure 4.1: An excerpt of an example application-level execution trace

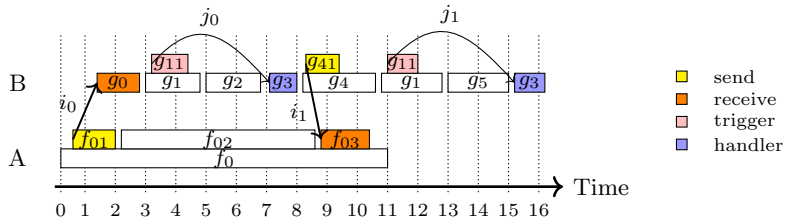


Figure 4.2: Visualization of the complete function executions, highlighting traced functions from Figure 4.1

along with communication and timer operations across two processes A and B. The specific function calls that have been traced are visually highlighted using distinct colors.

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

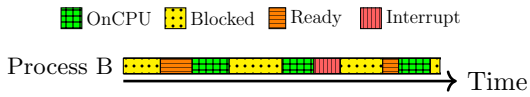


Figure 4.3: Process status over time

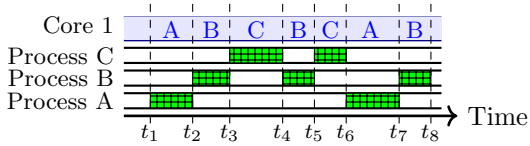


Figure 4.4: Deriving `OnCPU` timing by using context switch events

handled in process B.

4.4.2 System-level Execution Traces

Every set KS_{h_i} provides valuable information, including resource utilization, process states, and system frequency over time. Processes can be in various states, such as Ready, Interrupted, etc., as illustrated by the short timeline for Process B in Figure 4.3. To derive our workload model, system-level execution traces are essential for identifying periods when processes are actively executed on a CPU (denoted as the `OnCPU` state), which is shown in green in Figure 4.3. Identifying these periods relies on collecting specific kernel events, with context switches playing a key role. These events indicate when a process is scheduled onto a CPU core and when it is switched out, thus marking the intervals of active CPU usage.

Figure 4.4 visualizes the `OnCPU` status derived from context switch events on a CPU core. The transitions between processes on the core are indicated by changes in the process labels (A, B, C), and the green bars represent `OnCPU` timing of each process. The timing of `OnCPU` state for a process begins when it is scheduled onto the core and ends when the core switches to another process.

In Linux OS, the tool Trace Compass [104] simplifies 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.

4.5 The CompDSE Toolbox

In this section, we present the CompDSE Toolbox that contains several algorithms devised and implemented to support the modeling and exploration process. All implementations are carried out using Xtend, as introduced in Section 2.6.3.

4.5.1 Automated Derivation and Integration

As shown in Figure 3.1, the automated derivation algorithms are used to automatically derive the required workload models W_{h_i}, platform model PL, and mapping model MP from execution 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.

Algorithms for Application Workload Model Derivation and Integration

The formal procedure for the derivation of workload models W_{h_i} from traces and their integration into application workload model WK is outlined in Algorithm 1. This algorithm takes sets of application-level execution traces US and system-level execution traces KS as inputs, and generates a comprehensive workload model WK represented as a set of processes P communicating via set of channels C. The algorithm begins with deriving the workload model W_{h_i} for each host (Lines 2-4) using the Workload Model Derivation (`WLMD`) algorithm. This is followed by the integration of the individual models W_{h_i} into a unified application workload model encompassing all traced hosts (Lines 5-14). Below, we provide explanation of the `WLMD` algorithm and the final integration steps in Algorithm 1.

1. **Workload Model Derivation (`WLMD`)**

The formal procedure of the `WLMD` algorithm is presented in Algorithm 2. It takes as input a set of application-level execution traces US_{h_i} and a system-level execution trace KS_{h_i} for host h_i, and produces the corresponding workload model W_{h_i} for that host.

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

Algorithm 1: Workload Model Derivation and Integration

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

```
1  P, C ← ∅; W_tmp ← ∅;
2  foreach US_hi ∈ US do
3      W_hi = WLMD(US_hi, KS_hi);
4      W_tmp ← W_tmp + W_hi;
5  foreach W_hi ∈ W_tmp do
6      foreach ch_j ∈ W_hi.C do
7          h_src = ch_j.src.h; h_des = ch_j.des.h;
8          if h_src ≠ h_i then
9              ch_j.src.p = findPeerProc(W_h_src, ch_j, S);
10         if h_des ≠ h_i then
11             ch_j.des.p = findPeerProc(W_h_dst, ch_j, D);
12 foreach W_hi ∈ W_tmp do
13     P ← P + W_hi.P;
14     C ← C + W_hi.C;
15 return (P, C)
```

(Lines 4-5). Furthermore, if the record r_k has a function call of class *receive* then a communication channel is created if the channel has not been created yet (Lines 6-10).

To create the channel, Algorithm 2 finds the corresponding record r_j with a function call of class *send*, where this *send* function call and the aforementioned *receive* function call have exchanged a message with the same identifier. Record r_j is found in Line 7 by using the procedure `findREC`. It is shown in Algorithm 3 and used at multiple places in Algorithm 2 to find a specific record in a given trace. It takes as inputs trace E, function call class c, trace point location l, and message identifier id. By examining all records in input trace E, `findREC` returns the record with function call class, location, and message identifier that match the inputs c, l, and id, respectively.

After deriving the topology of the graph containing all processes and communication channels, Algorithm 2 proceeds with the derivation of the operation modes of every process p_x by creating the sequence of communication, computation, and timer events associated with every mode.

First, in Lines 14-32, Algorithm 2 examines the records in every trace E_i in-order. For every record $r_k \in E_i$ with location *start* and function call class *send*, *receive*, *trigger*, or *handler*, the algorithm creates a corresponding event and appends the event together with its start and end time to an ordered list of events H_x^i (all ordered

Algorithm 2: WLMD

Input : $US_h = \{E_1, ..., E_m\}$, KS_h
Output : $W_h = (P, C)$

$P, C \leftarrow \emptyset$;
foreach $E_i \in US_h$ **do**
  foreach $r_k \in E_i$ **do**
    if $\nexists\ p_{r_k.pn} \in P$ **then**
      $p_{r_k.pn} \leftarrow \emptyset$; $P \leftarrow P + p_{r_k.pn}$;
    if $r_k.A.c = receive$ **then**
      $r_j = \text{findREC}(E_i, send, start, r_k.A.id)$;
      if $\nexists\ ch_{(r_j.pn, r_k.pn)} \in C$ **then**
        $ch_{(r_j.pn, r_k.pn)} = (p_{r_j.pn}, p_{r_k.pn})$;
        $C \leftarrow C + ch_{(r_j.pn, r_k.pn)}$;
  foreach $p_x \in P$ **do**
    $H_x^i \leftarrow \emptyset$; `/* set of tuples` (ev, T_s, T_e) `*/`
    $H_x^i \leftarrow H_x^i + (\emptyset, 0, 0)$;
  for $k = 1$ ***to*** $|E_i|$ **do**
    if $r_k.A.c = send \wedge r_k.l = start$ **then**
      $r_{k'} = \text{findREC}(E_i, send, end, r_k.A.id)$;
      $sig = \text{getSIG}(KS_h^{E_i}, p_x.pn, r_k.ts, r_{k'}.ts)$;
      $r_j = \text{findREC}(E_i, receive, start, r_k.A.id)$;
      $e_k^W = (ch_{(r_k.pn, r_j.pn)}, r_k.A.\lambda, sig)$;
      $H_{r_k.pn}^i \leftarrow H_{r_k.pn}^i + (e_k^W, r_k.ts, r_{k'}.ts)$;
    if $r_k.A.c = receive \wedge r_k.l = start$ **then**
      $r_{k'} = \text{findREC}(E_i, receive, end, r_k.A.id)$;
      $sig = \text{getSIG}(KS_h^{E_i}, p_x.pn, r_k.ts, r_{k'}.ts)$;
      $r_j = \text{findREC}(E_i, send, start, r_k.A.id)$;
      $e_k^R = (ch_{(r_j.pn, r_k.pn,)}, r_k.A.\lambda, sig)$;
      $H_{r_k.pn}^i \leftarrow H_{r_k.pn}^i + (e_k^R, r_k.ts, r_{k'}.ts)$;
    if $r_k.A.c = trigger \wedge r_k.l = start$ **then**
      $e_k^{TS} = (r_k.A.id, r_k.A.\theta)$;
      $H_{r_k.pn}^i \leftarrow H_{r_k.pn}^i + (e_k^{TS}, r_k.ts, r_k.ts)$;
    if $r_k.A.c = handler \wedge r_k.l = start$ **then**
      $e_k^{TH} = (r_k.A.id, r_k.A.\theta)$;
      $H_{r_k.pn}^i \leftarrow H_{r_k.pn}^i + (e_k^{TH}, r_k.ts, r_k.ts)$;
foreach $p_x \in P$ **do**
  for $i = 1$ ***to*** m **do**
    $M_i \leftarrow \emptyset$;
    if $\exists\ H_x^i$ **then**
      $t_s = r_k.ts : r_k \in E_i \wedge k = |E_i|$;
      $H_x^i = H_x^i + (\emptyset, t_s, t_s)$;
      for $y = 1$ ***to*** $|H_x^i| - 1$ **do**
        if $H_x^i[y].ev \neq \emptyset$ **then**
          $M_i \leftarrow M_i + H_x^i[y].ev$;
        $t_s = H_x^i[y].T_e$; $t_e = H_x^i[y+1].T_s$;
        $sig = \text{getSIG}(KS_h^{E_i}, p_x.pn, t_s, t_e)$;
        $e_y^C = (sig)$; $M_i \leftarrow M_i + e_y^C$;
    $p_x \leftarrow p_x + M_i$;
return (P, C)

Algorithm 3: findREC

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

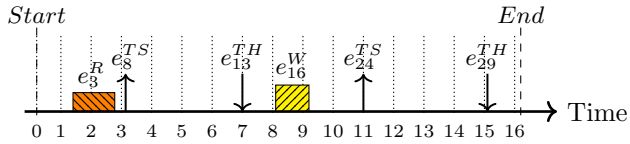
lists are initialized in Lines 11-13).

In Lines 15-20, a write event $e_k^W = (ch, size, sig)$ is created and appended to the corresponding ordered list if the currently examined trace record r_k has location *start* and a function call of class *send*. The attribute λ in record r_k provides the *size* of the message. To determine the communication channel *ch*, Algorithm 2 needs to identify the source and destination processes. The process name *pn* in r_k identifies the source process. The identification of the destination process is done in Line 18 by using procedure `findREC` to find the receiving record r_j and taking the process name *pn* in r_j. The abstract workload *sig* is calculated in Line 17 by using the procedure `getSIG`, which will be explained later in this section.

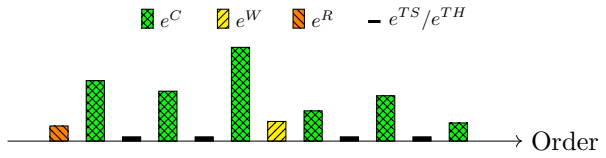
Similarly, in Lines 21-26 of Algorithm 2, a *read* event $e_k^R = (ch, size, sig)$ is created using procedure `findREC` and `getSIG`, and appended to a corresponding ordered list. In Lines 27-32, a timer setter event $e_k^{KS_h} = (\tau, t)$ or a timer handler event $e_k^{TH} = (\tau, t)$ is created and appended to a corresponding ordered list. The attributes *id* and θ in the examined record r_k provide the timer identifier τ and the duration time t set for the timer, respectively.

For example, let us apply Algorithm 2 on the application-level execution trace in Figure 4.1 and consider process B in this trace. In Lines 12-13, the ordered list H_B^1 associated with operation mode 1 of process B is initialized. In Lines 14-32, communication and timer events are created and appended to H_B^1. The content of H_B^1 is visualized in Figure 4.5(a) where the left/right side of a shaded box indicates the absolute start/end time of a communication event. The height of the box represents the abstract computation workload (signature) of the event, while the vertical arrows pointing upwards and downwards represent timer setter and timer handlers, respectively.

Second, in Lines 33-45 for every process p_x and trace E_i, Algorithm 2 derives mode $M_i \in p_x$. It is important to note that not every process is necessarily part of every trace, which means that the ordered list H_x^i may not exist for every process-trace combination. If H_x^i exists, all communication and timer events are moved in-order from



(a) Communication and timer events in H_B^1



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

Figure 4.5: Derivation of the workload model of the first mode in process B

the corresponding ordered list H_x^i to M_i (Lines 36-41). During this in-order movement of events, since the workload model is timing agnostic, the absolute start/end times of events are dropped (Line 41).

In addition, computation events are created by using the procedure `getSIG` and inserted in M_i after each communication and timer event (Lines 42-44). For example, consider process B, the trace in Figure 4.1, and the ordered list H_B^1 visualized in Figure 4.5(a). In Lines 37-44, communication, timer, and computation events are placed in M_1 in the correct order as explained above. These events and their order in the derived mode M_1 of process B are visualized in Figure 4.5(b). The height of a shaded box represents the abstract computation workload (signature) of a computation or communication event. It is worth noting that the width of all shaded boxes is the same, indicating that the notion of absolute start/end time of events is not present in M_1. Thus, M_1 captures only the order of events and their abstract workload.

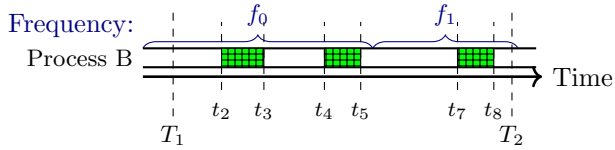
Finally, Algorithm 2 returns the derived workload model $W_h = (P, C)$ in Line 46. As explained above, the algorithm uses procedure `getSIG` to calculate the signature of communication and computation events (Lines 17, 23, and 43). The procedure, shown in Algorithm 4, takes as input the kernel-space trace KS_h^E for host h_i in operation mode E, the process name pn, start time T_{start}, and end time T_{end}. It returns an abstract computation workload sig imposed by process $p_{pn} \in E$ within time frame $[T_{start}, T_{end}]$.

As explained in Section 3.5.1, in our workload model the signature sig is defined as the number of cycles a process has been running on a CPU within a given time frame.

Algorithm 4: getSIG

Input : KS_h^E, pn, T_{start}, T_{end}
Output : sig

1 $cycles = 0$;
2 **foreach** $q \in KS_h^E$ **do**
3 **if** $q.process = pn$ ***and*** $T_{start} < q.\texttt{OnCPU} \leq T_{end}$ **then**
4 $cycles \leftarrow cycles + q.\texttt{OnCPU} \times q.freq$;
5 **return** *(sig = cycles)*



$$Cycles_{[T_1,T_2]} = (t_3 - t_2) \times f_0 + (t_5 - t_4) \times f_0 + (t_8 - t_7) \times f_1$$

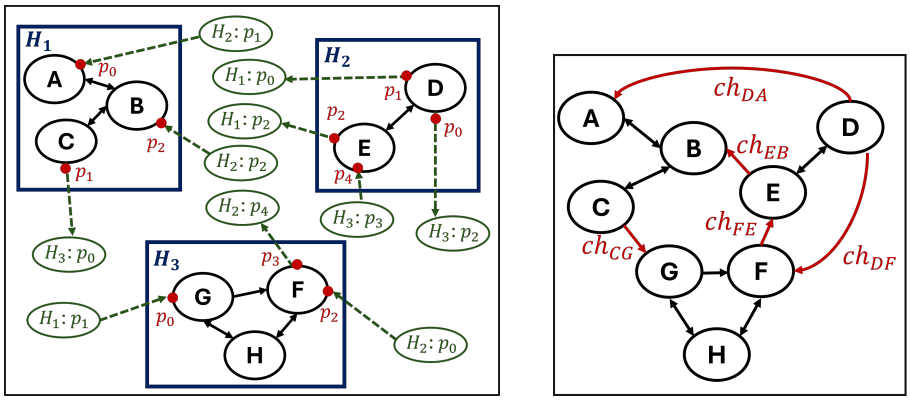
Figure 4.6: Calculation of the signature (cycles) of process B

To calculate this number, Algorithm 4 utilizes the system-level execution trace KS_h^E in the time frame [T_{start},T_{end}] (Lines 2-3), and considers the periods of time q when process p_{pn} has `OnCPU` state. These periods q.`OnCPU` are multiplied by the current CPU frequency and accumulated to obtain the total number of cycles (Line 4).

As an example, Figure 4.6 visualizes the `OnCPU` intervals for process B in the time frame [T_1, T_2], demonstrating the behavior of Algorithm 4. The algorithm uses these intervals and the corresponding frequencies (f_0 and f_1) to compute the total number of cycles during this time frame.

2. Workload Model Integration

As explained above, after deriving the workload model for each host W_{h_i}, the intra-host communication among the processes running on host h_i is well-defined within W_{h_i} by its set of communication channels $W_{h_i}.C$. However, inter-host communication remains incomplete because the source or destination processes on other hosts have not been identified, as illustrated in Figure 4.7a. This figure shows a simplified computing system consisting of processes A to H distributed across three hosts H_1, H_2, and H_3. The dashed green arrows represent inter-host communication, where the source or destination process needs to be identified. The red dots on the processes indicate the ports involved in inter-host communications, which must be considered when identifying the communication channels.



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

Figure 4.7: The workload model integration

To address this, Algorithm 1 first identifies inter-host communications based on workload model W_{h_i} of each host h_i. It examines each communication channel $ch_j \in W_{h_i}.C$ to determine if the source or destination host differs from h_i (Lines 6-11), indicating connections between processes running on different hosts. The procedure `findPeerProc` (Lines 9 and 11) is then employed to resolve the identities of these unknown source or destination processes. After completing the information for inter-host communications, the sets of processes $W_{h_i}.P$ and channels $W_{h_i}.C$ for all traced hosts h_i are integrated into the sets P and C, respectively (Lines 12-14). This creates the comprehensive application workload model $WK = (P, C)$, as shown in Figure 4.7b, where the previously unknown sources or destinations of inter-host communication in Figure 4.7a are identified (indicated by red arrows).

The aforementioned `findPeerProc` procedure is detailed in Algorithm 5. It takes as inputs 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).

Algorithm for Hardware Platform Model Derivation

The formal procedure for deriving hardware platform model $PL = \{H, N, L\}$ from traces is outlined in Algorithm 6. This algorithm takes sets of application-level execution

Algorithm 5: findPeerProc

```
   Input  : W_h, ch, flagT
   Output : peerProc
1  foreach ch_i ∈ W_h.C do
2      if (ch_i.src.h = ch.src.h ∧ ch_i.dst.h = ch.dst.h) ∧ (ch_i.src.prt =
         ch.src.prt ∧ ch_i.dst.prt = ch.dst.prt) then
3          if flagT = S then
4              return peerProc ← ch_i.src.p
5          else
6              return peerProc ← ch_i.des.p
```

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

traces US and system-level execution 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 application-level 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 `getIPs` procedure (Line 3). Next, the algorithm proceeds by utilizing the set of system-level execution traces KS_{h_i} of host h_i to determine the number of cores nCr and their frequencies f_j (Lines 4-7).

Algorithm 7: getIPs

```
   Input  : US_h
   Output : ipSet
1  ipSet ← ∅;
2  foreach r_k ∈ 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 ∉ ipSet then
6              ipSet ← ipSet + ip_k;
7  return ipSet
```

These frequencies serve as initial values in the model and can be adjusted during the exploration phase. Similarly, the policy sch_i for scheduling the tasks running on host h_i is initially set to unknown (Line 8) and it is tuned based on the designer's preferences during the exploration phase.

Once the set of hosts H is finalized (Line 9), Algorithm 6 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 12). 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 13). If such a switch does not exist, the algorithm creates a new switch $swch_j$ with sbn_j, sets its initial bandwidth bw_j to zero, and adds the $swch_j$ to N (Lines 14-15). Finally, a link l_j is created and added to L specifying that host h_i is connected to switch $swch_j$ (Line 16). Note, host h_i may be connected to multiple switches if it has multiple IP addresses $ip_j \in h_i.ip$ (Line 11).

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

Algorithm 8: Mapping Model Derivation from Traces

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

```
1 MP ← ∅;
2 for j = 1 to m do
3     MP_j ← ∅; MP ← MP + MP_j;
4 foreach US_{h_i} ∈ US do
5     foreach E_j ∈ US_{h_i} do
6         foreach r_k ∈ E_j do
7             mp_k = (p_{r_k.pn}, h_i); MP_j ← MP_j + mp_k;
8 return MP
```

Algorithm for Mapping Model Derivation

The formal procedure for deriving the mapping model from traces is outlined in Algorithm 8. This algorithm takes set US of application-level 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 8 begins by initializing an empty mapping set MP_j for each operation mode M_j (Lines 2-3). It then examines the records r_k in every execution trace $E_j \in US_{h_i}$ associated with M_j and host h_i (Lines 4-6) to identify every process $p_{r_k.pn}$ running on h_i. This examination results in creating tuples $(p_{r_k.pn}, h_i)$ specifying that process $p_{r_k.pn}$ is mapped onto host h_i in operation mode M_j. The created tuples are subsequently added to the mapping set MP_j associated with operation mode M_j (Line 7).

4.5.2 Model-to-Model Transformation

After deriving the formal models of the application workload, hardware platform, and mapping, the next step, as illustrated in Figure 3.1, is to prepare the models for simulation by transforming them into abstract executable entities. This transformation is performed by a set of model-to-model (M2M) transformation algorithms that operate on the structure and semantics of the formal models.

Figure 4.8 summarizes the three formal input models (shown as class diagrams on the left), the M2M transformation paths, and the resulting executable entities (on

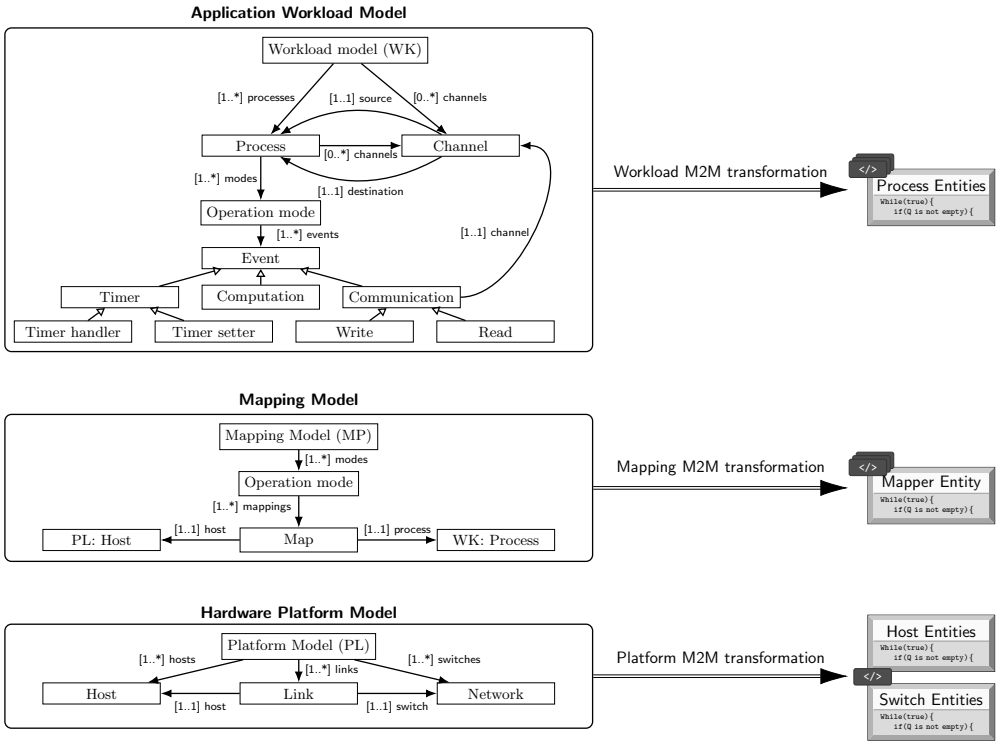


Figure 4.8: Model-to-Model transformation from formal models (left) to executable simulation entities (right)

the right). Each entity consists of a *data structure* capturing its attributes and a general *code template* that defines its behavior. The code template for each entity is described in Section 3.5.5 and drives the entity's execution and its interactions with other entities in the simulation. Entities interact by exchanging messages to synchronize and trigger simulation events.

Workload M2M transformation

Taking the formal application workload model, represented in our methodology implementation as the class diagram structure shown in the top part of Figure 4.8, the M2M transformation algorithm parses this structure and creates one Process Entity for each Process class. The general code template of the Process Entity is shown in Listing 3.1. For each Operation mode of a Process class, the transformation algorithm also generates a separate data structure that collects all associated Event classes and their attributes. For Computation events, attributes record the workload signatures;

for Communication events, attributes specify the source or destination, the workload signature, and the message size; for Timer events, attributes include the timer duration and corresponding timer identifier.

At the start of a simulation run, each Process Entity instantiates events from its data structure and also reads the environment (ENV) data structure, which represents the environmental influence model and provides delay estimates (Δ parameters) for incoming messages originating from untraced embedded hosts.

Mapping M2M transformation

Taking the formal mapping model, represented as the class diagram structure shown in the middle part of Figure 4.8, the M2M transformation algorithm parses this structure and creates one Mapper Entity. The general code template of the Mapper Entity is shown in Listing 3.2. For each Operation mode of the formal mapping model, the transformation also generates a separate data structure that, for that operation mode, defines the allocation of every process from the workload model (WK:Process) to its corresponding host in the platform model (PL:Host).

Platform M2M transformation

Taking the formal hardware platform model, represented as the class diagram shown in the bottom part of Figure 4.8, the M2M transformation algorithm parses this structure and creates one Host Entity for each Host class and one Switch Entity for each Network class. The general code template of the Host Entity and Switch Entity is shown in Listing 3.3 and Listing 3.4, respectively. The transformation algorithm also generates a data structure for entity attributes and the platform topology. For each Host Entity, attributes include the number of cores, core frequencies, IP addresses, and the scheduling policy. For each Switch Entity, attributes include the network bandwidth. The platform topology, derived from the Link class, specifies how Host Entities and Switch Entities are connected.

4.5.3 Code Generation

After obtaining the executable models, which are independent of any discrete event simulation environment, a code generation step is necessary to transform them into concrete executable code for a specific simulation environment which is selected. In our case, we select the OMNet++ simulation environment due to its scalability, as discussed in Section 2.6.

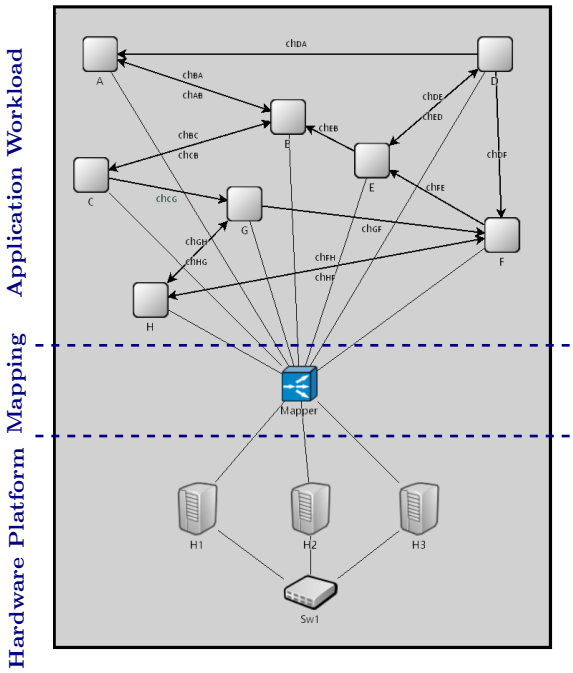


Figure 4.9: The application workload, computing hardware platform, and mapping layers in OMNet++

To enable co-simulation of the models with OMNet++, we develop a custom code generator that produces the files required for execution. These generated files include OMNet++ specific simulation files, C++ code files for the entities generated from their general code templates, and entity input files generated from their corresponding data structures. The primary OMNet++ specific simulation file is the Network Description (NED) file, which specifies the simulation modules and their topology. In our case, this file integrates the workload, mapping, and platform entities as simulation modules into a single OMNet++ network. A visualization of this network, based on the example in Figure 4.7, is shown in Figure 4.9. As can be seen, the NED file is structured into three layers, namely Application Workload layer (including the Process entities), Hardware Platform layer (including the Host and Switch entities), and Mapping layer (including the Mapper entity). During simulation, each entity in the network reads its corresponding generated entity input file.

As an example of the generated C++ code for the entities, Figure 4.10 shows the C++ class skeleton of the Mapper entity used in our simulation. The figure presents only the class interface, including its member variables and functions. The body of

*Mapper.cc

```
#include <fstream>
#include "HWMsg_m.h"
#include "AppMsg_m.h"
#include "generalLib.h"

using namespace omnetpp;
using namespace std;


class Mapper : public cSimpleModule
{
  private:
    vector<processor> processorList;
    ofstream map_log_File ;
  protected:
    virtual void initialize() override;
    virtual void handleMessage(cMessage *msg) override;
    virtual void finish() override;
    string find_allocated_processor (string processName);
    void sendToHW(AppMsg *msg);
    void writeToLogFile(string log);
};
```

Figure 4.10: The C++ class skeleton of the `Mapper` entity for the OMNet++ simulation

function `handleMessage` is invoked on message arrival from another simulation module and `handleMessage` processes the message as specified by the Mapper entity's general code template (Listing 3.2).

In addition to the generated C++ code for the entities, the simulation execution relies on the entity input files. A primary input file for each process entity is its own event file. This file is generated from the process entity's data structure and contains a partially ordered sequence of abstract events together with their attributes. At the simulation runtime, each process entity reads events from its event file and handles them according to their event type, thereby driving the simulation behavior. An example of an event file for process B in Figure 4.5 is shown in Figure 4.11. It is generated from the data structure of the entity corresponding to process B with the sequence of abstract events illustrated in Figure 4.5(b). Each row in Figure 4.11 specifies the event type and its relevant attributes.

Besides the process event files, the generator creates a platform file, derived from the data structure corresponding to the host and switch entities, that specifies the system hardware platform configuration. This file lists the host and switch entities and their attributes. An example platform file, generated from the platform model entities shown in Figure 3.4, is presented in Figure 4.12. Hosts and switches are interconnected using link declarations.

The generator also produces a mapping file derived from the mapper entity's data

```
   EventType          dstP/srcP/ID      #cycles/Time      #bytes
1  read               |    <- A         |    10           |    40
2  compute            |                 |    40           |
3  timer setter       |    id:0         |    4s           |
4  compute            |                 |    33           |
5  timer handler      |    id:0         |    4s           |
6  compute            |                 |    62           |
7  write              |    -> A         |    13           |    73
8  compute            |                 |    20           |
9  timer setter       |    id:1         |    4s           |
10 compute            |                 |    30           |
11 timer handler      |    id: 1        |    4s           |
12 ...                |    ...          | ...             | ...
```

Figure 4.11: Example of a generated event file for Process B in operation mode M_1

```
1  <!-- Platform File -->
2  <Platform>
3  <host name="H1" schedule="FCFS">
4    <core id="H1_1" frequency="500"/> <!-- MHz -->
5  </host>
6  <host name="H2" schedule="FCFS">
7    <core id="H2_1" frequency="250"/>
8    <core id="H2_2" frequency="350"/>
9  </host>
10 <host name="H3" schedule="FCFS">
11   <core id="H3_1" frequency="300"/>
12   <core id="H2_2" frequency="300"/>
13 </host>
14
15 <switch name="Sw1" bandWidth="1000"/> <!-- Mbps -->
16
17 <links>
18   <link from="H1" to="Sw1"/>
19   <link from="H2" to="Sw1"/>
20   <link from="H3" to="Sw1"/>
21 </links>
22 </platform>
```

Figure 4.12: Example of a generated platform file

structure. A sample mapping file, corresponding to the same model used in the platform file, is shown in Figure 4.13.

Finally, the generator creates an ENV file that contains all the Δ delay parameters. Each entry in this file corresponds to a communication flow between a traced process and an untraced embedded host. Specifically, it records the timing difference between a message sent (*write*) by the traced process and the corresponding response (*read*) it receives.

```
<!-- Mapping File -->
<mapping>
  <bind process="A" host="H1"/>
  <bind process="B" host="H1"/>
  <bind process="C" host="H1"/>
  <bind process="D" host="H2"/>
  <bind process="E" host="H2"/>
  <bind process="F" host="H3"/>
  <bind process="G" host="H3"/>
  <bind process="H" host="H3"/>
</mapping>
```

Figure 4.13: Example of a generated mapping file

4.6 Conclusion

This chapter introduced the CompDSE Toolbox, a set of algorithms that enable automated trace-based model derivation and simulation support for DSE in large-scale dCPS. We first defined the execution traces required for model derivation, ensuring that their collection does not disturb system behavioral correctness. These traces are collected from user space as application-level execution traces, and from kernel space as system-level execution traces. We then detailed the algorithms for automatically deriving and integrating application workload, hardware platform, and workload-to-platform mapping models. In addition, we demonstrated how the derived models can be transformed into executable entities suitable for use in any discrete-event simulation environment, and showed how simulation-ready code can be generated for the OMNet++ simulator. Together, these steps complete the tooling support needed to transition from trace collection to simulation input, laying the foundation for scalable DSE in distributed cyber-physical systems as will be demonstrated in Chapter 5.

Chapter 5

CompDSE Evaluation

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.

In this chapter, we evaluate the CompDSE methodology using an industrial case study, namely the ASML TWINSCAN machine. The goal is to assess the efficiency and accuracy of the proposed methodology in supporting DSE of complex industrial dCPS, thereby addressing the efficiency aspect of challenge **CHL4**.

Section 5.1 highlights the need for evaluation, including the validation of both accuracy and efficiency. Section 5.2 outlines the contributions of this chapter, including the evaluation itself and a set of DSE experiments. Section 5.3 presents the experimental setup. This is followed by Section 5.4, which presents the experimental results, compares simulated timing with real-time measurements to assess accuracy, and records the simulation duration to assess the efficiency for evaluating the performance of a single design point. This section also explores alternative configurations of certain subsystems within the ASML TWINSCAN machine and compares their performance evaluation results. Finally, Section 5.5 concludes the chapter.

5.1 Problem Statement

The previous chapters introduced the CompDSE methodology, including the abstract models required for DSE of complex dCPS, the algorithms for their automated derivation from execution traces, and the generation of simulation-ready code. However, while the methodology, modeling approach, and supporting tools have been fully developed and described, their effectiveness has to be validated in a realistic industrial context. In particular, it is necessary to assess how accurately the derived models reflect the actual system behavior required for effective DSE, and how efficient the system simulation is for rapid performance evaluation of each design point.

5.2 Contributions

This chapter evaluates the CompDSE methodology using the ASML TWINSCAN lithography machine, introduced in Section 2.3, as a representative industrial dCPS case study. While the experiments are based on the TWINSCAN machine, the methodology and findings are applicable to other complex dCPS with similar architectural characteristics. The main contributions of this chapter are as follows:

1. **Accuracy validation:** We evaluate how closely the simulation results produced by the CompDSE models align with the actual TWINSCAN system behavior, thereby validating the fidelity of the modeling process. In particular, we compare the average time required for wafer processing in the simulated and real systems;

2. **Efficiency assessment:** We assess the time required to simulate a single design point, demonstrating the efficiency of the CompDSE methodology despite the complexity and scale of the TWINSCAN industrial dCPS;

3. **DSE experiments:** We conduct simulation-based DSE experiments to evaluate alternative architectural configurations of some TWINSCAN subsystems. These experiments aim to identify potential system performance improvements while respecting physical and operational constraints of the system.

5.3 Experimental Setup

To evaluate the accuracy and efficiency of the proposed methodology, we conducted an experiment based on real trace data collected from the ASML TWINSCAN machine

Table 5.1: Scale of the traced system and model derivation

Complexity Metric	Value
Number of traced hosts	8
Number of processes	445
Number of channels	2057

testbench. We traced a subset of the hosts in the testbench, including one Linux-based server and several real-time embedded systems running the Wind River VxWorks OS. Due to technical limitations such as some hosts being "bare-metal" (i.e., not running any operating system), it was not possible to trace all hosts. However, we consider the effects of these untraced hosts on the overall dCPS performance during our system simulations by using Δ parameters, as explained in Section 3.5.4.

To incorporate Δ parameters 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 *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 (including untraced hosts) on the dCPS timing performance is accurately represented in the simulation.

Table 5.1 summarizes the complexity metrics of the generated models, including the number of traced hosts, modeled processes, and identified communication channels. In this experiment, the CompDSE methodology is applied to a wafer lot operation involving five wafers. A total of **8** hosts are traced, **445** application processes are identified and modeled, and **2057** communication channels between these processes are identified. These channels are derived to support message exchange and synchronization, independent of how the hardware platform model is abstracted.

In our integrated simulation setup, synchronization is handled by the hardware platform layer. Messages exchanged between processes are transferred through the simulated hardware platform. When a process attempts to read a message, it can only proceed after receiving a confirmation from its allocated host, indicating that the message is now available for reading. As a result, the aforementioned communication

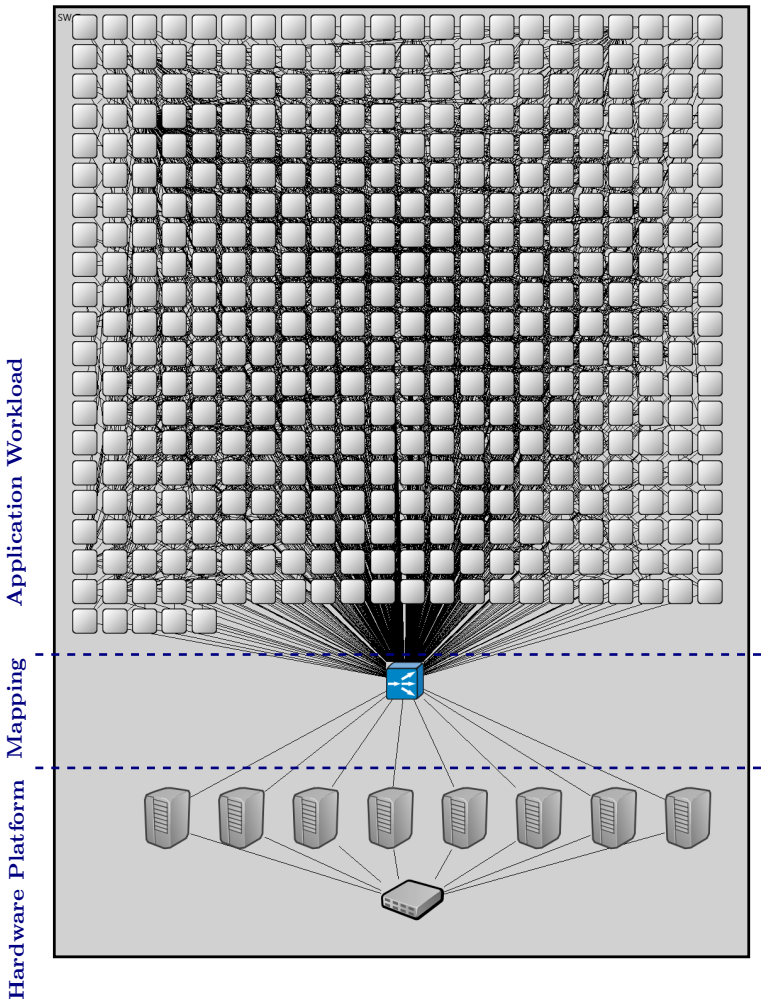


Figure 5.1: The simulation setup of the ASML TWINSCAN machine in OMNet++

channels serve as logical abstractions and are not explicitly represented in the NED file.

Figure 5.1 shows a visualization of the simulation environment used in our experiment for the TWINSCAN machine, implemented in OMNet++ and organized into three layers. The names of the entities (processes, hosts, etc.) are omitted due to confidentiality imposed by ASML. In the platform layer, the eight traced hosts used in our simulation are connected to a central switch.

5.4 Evaluation Results

After describing the experimental setup above, in this section we evaluate the model accuracy, simulation efficiency, and explore some alternative system configurations.

5.4.1 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 dCPS. By considering the system complexity, shown in Table 5.1, 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.

Table 5.2 provides the execution time of the various steps within CompDSE, visualized in Figure 3.1. The time required to manually add tracepoints in the software infrastructure depends on the number of tracepoints needed. For our case, we add tracepoints at the start and end of four functions within a shared library used by all processes. Adding these 8 tracepoints takes approximately 5 minutes and needs to be done only once. To ensure that tracing does not affect the system's real-time performance, we monitored the wafer processing time before and after enabling tracing and observed no measurable difference.

Afterward, the time required for trace collection depends on the duration of the application's execution. In our case, processing five wafers takes 4.33 minutes. This trace collection process must be repeated for each operation mode. After the trace collection, the automated model derivation and integration take 29.5 hours but needs to be performed only once. The model-to-model transformation and code generation for the OMNet++ simulator is completed in about 2 minutes. The simulation to evaluate the performance of one system configuration requires **54.5 seconds**.

Given the high complexity of the aforementioned model, this less-than-a-minute simulation-based performance analysis demonstrates the remarkable efficiency of CompDSE in evaluating a single design configuration. Such rapid evaluation is especially beneficial for industrial dCPS, where quick insights into the performance of complex machines are critical. Moreover, such rapid performance evaluation of a single design

Table 5.2: Time required for each step in the CompDSE workflow

Steps	Time Required
Adding tracepoints manually	about 5 minutes
Collecting execution traces automatically	4.33 minutes
Automated models derivation and integration	29.5 hours
Automated model transformation and code generation	1.82 minutes
Single point evaluation	54.5 seconds

configuration is crucial to enable fast exploration of large design spaces. For example, by integrating an automated search algorithm within CompDSE, such as a GA, and utilizing the rapid simulation-based performance analysis as a fitness function, we could significantly accelerate the exploration of large design spaces, making the CompDSE methodology even more scalable for future designs of complex dCPS.

5.4.2 Accuracy

In our experiment, the average processing duration per wafer (PDW) measured on the real ASML TWINSCAN machine is **50.548 seconds**, whereas the simulation of our derived system model of the same TWINSCAN machine indicates an average PDW of **49.183 seconds**, resulting in an error of 2.7%. This deviation reflects the reconstruction accuracy of the proposed modeling approach, indicating how closely the simulation reproduces the observed system behavior at the chosen level of abstraction. Together with the less-than-a-minute performance analysis time, this demonstrates that the model provides a faithful and computationally efficient abstraction of the system. Since the derived workload model preserves the structural dependencies and interaction patterns observed in the real execution, and the explored design alternatives modify only resource-related parameters within the same abstraction level, the validated baseline serves as a consistent reference for evaluating relative performance differences across configurations within the defined design space.

5.4.3 Exploration

Building on the validated baseline model, we conduct a small DSE experiment within the design space defined in Section 3.6. We first explore the effect of varying the number of cores in the main host (the Linux-based server) on the TWINSCAN system performance in terms of average PDW. The default configuration of the main host has sixteen cores. However, as shown in Figure 5.2, increasing the number of cores

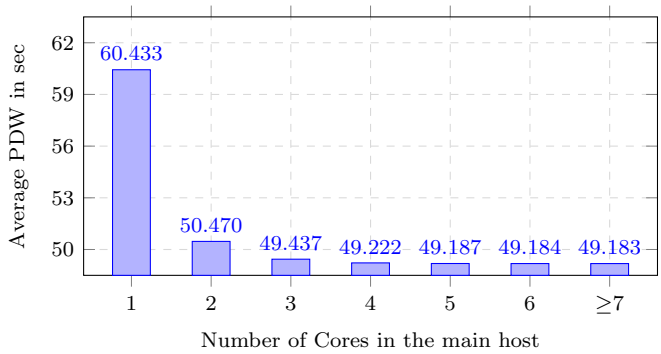


Figure 5.2: The impact of the number of cores on PDW

beyond seven does not lead to any significant changes in the average PDW. When we decrease the number of cores from seven to three, the average PDW does not increase significantly, i.e., the increase is in the range of milliseconds. Further decreasing the number of cores leads to a more substantial increase in the average PDW. The above findings suggest that the original design of the main host with sixteen cores in the ASML TWINSCAN machine might be an overestimation to avoid computing hardware resource bottlenecks. Thus, having a host with eight cores (half of the original design) could lower overall costs without compromising the machine performance, which could be considered in future system designs.

Next, we explore the effect of increasing the number of cores in the traced embedded hosts (the VxWorks-based real-time embedded systems) while keeping the main host with its default configuration. The results show that increasing the number of cores from one to four in these embedded hosts decreases the average PDW by 0.477 seconds. This small reduction indicates that while additional cores enable the system to handle more software processes in parallel, the overall impact on the average PDW is limited. This limitation is most likely due to the original design of the software infrastructure running in the ASML TWINSCAN machine where potential parallel processing in the software is not heavily exploited.

Finally, we examine the impact of varying the clock frequencies of the cores in the traced main and embedded hosts on the average PDW by conducting a series of simulations with different clock frequency settings, supported by the cores, across all hosts. The results, plotted in Figure 5.3, show the relationship between the average PDW and clock frequency changes relative to the base clock frequency setting of each host's multi-core CPU. The base frequency setting is denoted as 0 on the horizontal

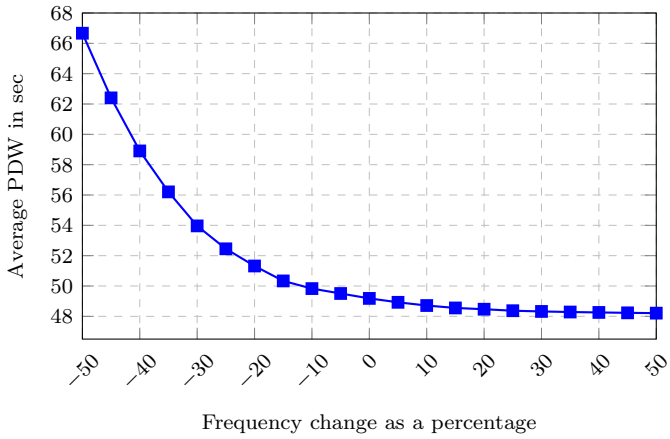


Figure 5.3: The impact of the clock frequency of cores on PDW

axis, while the average PDW is depicted on the vertical axis. The horizontal axis depicts the clock frequency changes as a percentage of the base clock frequency.

It can be seen in Figure 5.3 that increasing the clock frequency of all traced hosts' multi-core CPUs with 25% results in decrease of the average PDW from 49.183 to 48.371 seconds. However, increasing further the clock frequency yields only a marginal decrease of the average PDW, often within the millisecond range, as visualized by the curve's plateau in Figure 5.3. This plateau suggests that the performance gain in terms of PDW does not scale with the clock frequency increase when it is above 25%. The plateau indicates that the PDW is limited to around 48 seconds. This limitation arises from the performance of some untraced hosts reflected by the Δ parameter values.

The DSE experiments presented in this section focus on parameter variations of the traced platform configuration, specifically changes in computational resources such as the number of cores and clock frequencies. Although the design space defined in Section 3.6 also allows structural platform modifications, including the addition of hosts or changes in topology, such structural variations were not part of the present validation.

5.5 Conclusion

This chapter evaluates the CompDSE methodology using a realistic industrial case study, namely the ASML TWINSCAN lithography machine. The results show that the models, automatically derived by CompDSE, closely reproduce the observed behavior of

the real system under the same architectural configuration. In particular, the simulated average wafer processing time differs by 2.7% from the measured system processing time, which reflects the reconstruction accuracy of the proposed modeling approach at the adopted level of abstraction. Despite the complexity and scale of the model, each design point could be simulated in under one minute, demonstrating that the methodology remains computationally efficient. Building on the validated baseline model, we conduct a small DSE experiment within the defined design space to assess how architectural changes affect the system performance. The experiment demonstrates that reducing the number of cores can lower the system cost without significantly affecting the performance, while adjusting clock frequencies allows designers to observe how much performance gain can be achieved within the limitations of the physical environment. These findings highlight the applicability of CompDSE in supporting early-stage architectural design decisions.

Although the exploration in this chapter is conducted manually, the CompDSE methodology is well-suited for integration with search algorithms, such as GA, to further enhance automation and enable efficient large-scale optimization of complex dCPS.

Chapter 6

Workload Model Refinement

Publications related to this chapter

- Faezeh Sadat Saadatmand, Todor Stefanov, Andy D. Pimentel, Benny Akesson, and Ignacio González Alonso. Unraveling Parallelism in Automated Workload Modeling for Distributed Cyber-Physical Systems. In *Euromicro Conference on Digital System Design (DSD)*, pages 24–33. IEEE, September 2025.

This chapter introduces the refined application workload model and a method for parallelism analysis, addressing challenge **CHL5**. Section 6.1 identifies the technical limitations of the initial workload model, used in the CompDSE methodology, that prevent the representation of intra-process parallelism. Section 6.2 outlines the key contributions of this chapter. Section 6.3 provides an overview of the related work with respect to the chapter contributions. Section 6.4 explains the motivation for the refinement of the application workload model for large-scale industrial dCPSs. Section 6.5 describes the structure of the refined workload model and its support for parallel execution. Section 6.6 describes which execution traces need to be collected for model refinement. Section 6.7 explains the automated procedure to derive the refined workload model. Section 6.8 introduces the algorithm used to detect parallel execution opportunities based on the refined workload model. Section 6.9 presents experimental results on the ASML TWINSCAN case study. Finally, Section 6.10 concludes the chapter.

6.1 Problem Statement

The CompDSE methodology enables exploration of different platform configurations and process-to-host mappings, as it is common in system-level DSE for dCPSs. While the derived workload models capture accurately inter-process dependencies based on runtime traces and ensure a valid sequential ordering of communication and computation events within each process, the models assume a fixed, static execution order of events for each operation mode of a process. This abstraction limits the exploration space by excluding alternative execution orders that may exist within the same process.

More specifically, the initial application workload model, presented in Chapter 3, does not represent internal task-level dependencies within a process. As a result, potential intra-process parallelism, which can be crucial for improving performance on modern multi-core CPUs in the hosts, remains unexploited during DSE. To overcome this limitation, an enhanced modeling approach is needed: one that captures both inter-process and intra-process dependencies and enables the identification and exploration of different sequential and parallel execution orders of processes and tasks within processes.

6.2 Contributions

In this chapter, we propose significant refinements to the initial application workload model in CompDSE, enabling it to capture both inter-process and intra-process dependencies. Building on this model, we present a method to identify tasks within each process that can be safely reordered or executed in parallel, without violating the system behavior or requiring modifications to the existing software implementation. More specifically, the key novel contributions are as follows:

1. Our refined application workload model incorporates detailed execution traces, enabling automated analysis and identification of parallel execution opportunities between software processes and tasks within individual processes in large-scale dCPS.

2. We propose a method that identifies eligible tasks within each process that can be safely reordered or executed in parallel with other tasks. The method requires no application-specific knowledge and is compatible with the event-driven execution model (Section 2.4) commonly used in industrial dCPSs.

3. We validate our refined application workload model and the proposed method by automatically deriving models from execution traces collected from the ASML TWINSCAN lithography machine.

6.3 Related Work

Workload modeling approaches in CPS can be broadly grouped into two categories: 1) approaches that do not exploit parallelism and rely on abstract models for system-level optimization, and 2) approaches that capture parallelism by representing concurrent behavior or by improving execution order using scheduling, profiling, or simulation.

Workload Modeling Without Parallelism: Several approaches focus on abstract workload modeling for system-level optimizations without considering parallelism. One study proposes a formal modeling framework based on stochastic processes and fractal methods to probabilistically capture task execution and communication patterns [67]. While this study offers analytical insights into system resource utilization and scheduling, it requires manual mathematical definitions and relies on fixed statistical structures, limiting scalability and adaptability needed for complex industrial dCPSs.

Another study presents a power-aware workload model that integrates computational delays and energy constraints into scheduling decisions [8]. It captures complex workload behaviors, such as irregular patterns and long-term dependencies, using advanced mathematical formulations. However, the model is not automatically derived and depends on predefined analytical profiles, making it inflexible across different dCPS configurations.

Another approach combines stochastic Petri nets, fault trees, and Markov models to assess performance, reliability, and resource utilization [71]. Though effective in analyzing system-level trade-offs, it relies on manually constructed workload scenarios and predefined interaction patterns, which limits its adaptability.

While all these approaches support system-wide analysis, they share two key limitations: they depend on manually defined workloads, which are impractical for large-scale industrial dCPSs, and they assume fixed execution orders, preventing the exploration and exploitation of potential parallelism.

Workload Modeling Incorporating Parallelism: Several studies focus on parallelism detection and modeling within CPS workloads. Identifying safe parallel execution opportunities generally relies on analyzing dependencies between computational tasks or execution units. In related domains such as compiler technology and automatic code parallelization, dependency analysis has been extensively studied to determine

whether tasks/processes and operations can be reordered or executed concurrently without violating data or control dependencies [42, 83]. These analyses are typically performed statically at source code or intermediate-representation level and aim to exploit data-level [5, 14], task-level [19, 47], instruction-level [10, 72], or loop-level parallelism [4, 35, 61].

However, such static dependency analysis is not directly applicable to large-scale industrial dCPS. These systems do not consist of a single centralized program, but of multiple distributed and heterogeneous processes. As a result, the dependencies that govern their execution are typically not explicitly represented in the source code or intermediate representations, especially given the scale and distribution of industrial software systems. Instead, execution dependencies emerge from runtime interactions between processes, making dependency analysis based on runtime information a more appropriate approach in this context.

At the simulation and modeling level, one study explores parallelism in transaction-level modeling (TLM) by analyzing how communication and synchronization mechanisms impact simulation speed in Parallel Discrete Event Simulation (PDES) environments [9]. The authors propose techniques to enhance model-level parallelism in SystemC TLM and evaluate various buffering and communication schemes using a deep neural network case. While effective for simulation optimization, the approach does not model execution dependencies or generate application workload models.

Another approach addresses software inefficiencies by detecting performance antipatterns using queuing network models derived from UML specifications [81]. While the approach effectively quantifies the performance impact of antipatterns and helps identify software architectural bottlenecks, it relies on manually created design artifacts and performance scenarios, which limits its scalability for large-scale industrial dCPS. Additionally, the method focuses on identifying performance antipatterns at the architectural level through resource and deployment modeling, rather than capturing software execution dependencies.

In another study, an AI-driven scheduler, called MCDS, combines Monte Carlo Tree Search with deep surrogate models to optimize workflow execution in edge-cloud systems [106]. It models workflows as Directed Acyclic Graphs (DAGs) and predicts long-term Quality of Service (QoS). However, it assumes that the workflow structure is known in advance and remains fixed, and it also incurs significant computational overhead, making it unsuitable for large-scale dCPS with different configurations.

A more scalable direction is proposed in [87], where application workload models are automatically derived from runtime system traces. While this method supports

DSE of hardware platforms and software-to-hardware mapping in complex dCPS, it assumes fixed task execution orders and does not explore intra-process parallelism.

The aforementioned workload modeling approaches either lack necessary automation and adaptability or do not support parallelism exploration across both inter- and intra-process levels. Specifically, many existing methods rely on manual workload specifications or predefined task execution orders, limiting their use in evolving industrial dCPS configurations.

In contrast, our approach automatically derives refined workload models for each hardware/software configuration using execution traces. These refined models capture both inter- and intra-process dependencies and support the identification of tasks that can be safely reordered or executed in parallel, without modifying the existing software implementation.

6.4 Motivation

In large-scale industrial dCPS, such as the ASML TWINSCAN machine, performance optimization through parallel execution is essential, but achieving it is challenging. These systems typically rely on mature software that has been incrementally developed over many years and involves complex interactions among hundreds of processes. Even minor code changes can disrupt the execution flow and lead to unforeseen downtime or system failures. This raises an important question: How can opportunities for parallel execution be systematically and safely uncovered without modifying the software source code?

To make this challenge more concrete, consider a simplified but representative example of how the forced sequential execution model used in many industrial systems can lead to unnecessary delays. Figure 6.1 illustrates this situation in an event-driven system with single-threaded processes, where requests are queued and handled one at a time. The figure compares two scenarios, each demonstrating a different sequence of request handling across four processes (A, B, C, and D) over time (represented along the vertical axis).

In Scenario 1, process A sends a request `get_state` to process B. To respond, B sends a request `new_state` to process C and waits for its response before replying to A. Meanwhile, process D sends a request `getParam_x` to B to retrieve a parameter needed for executing task `t_x`. However, because B handles requests sequentially, thus this new request is queued until B becomes available, causing the delay shown by the hatched rectangle. Only after completing process A's request, process B can handle process D's

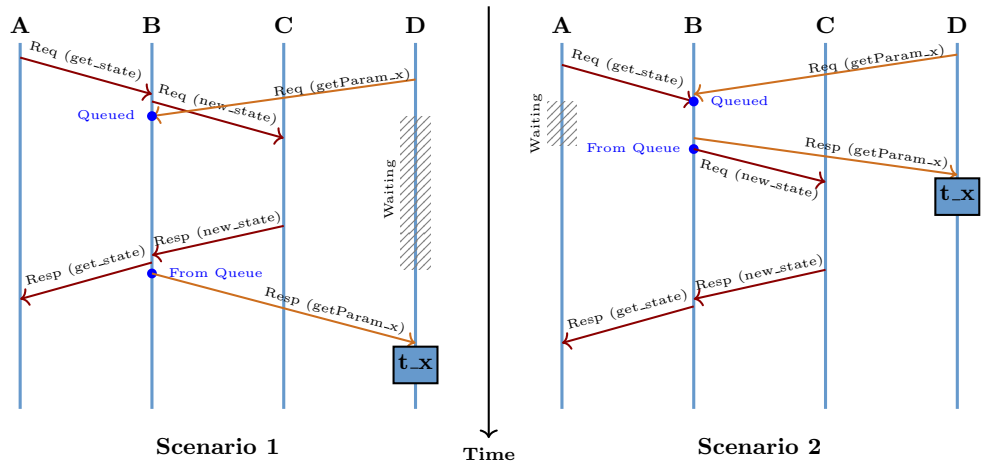


Figure 6.1: The impact of request handling order on execution timing

request, and then send the response.

In Scenario 2, the order of requests arrival is reversed: request `getParam_x` (from D) arrives before `get_state` (from A). By prioritizing process D's request, the waiting time for D is significantly reduced, enabling earlier execution of task `t_x`. Moreover, because `getParam_x` is a relatively lightweight operation, it completes quickly and allows process B to handle process A's request shortly afterward, resulting in less total delay for both. This simple change in execution order demonstrates how intelligent reordering of requests, even without concurrency, can lead to performance gains. The benefit could be even more significant in systems that support multi-threading to concurrently handle independent requests like the independent requests from process A and D to process B.

This example highlights the potential for substantial performance improvements through execution reordering or parallel task execution. However, to support such reasoning, a more detailed and analyzable model of software execution is required, beyond the static, inter-process-only abstractions used in the initial application workload model of CompDSE. To address this limitation, we introduce a refined application workload model in this chapter, one that captures both inter- and intra-process dependencies, enabling the identification of parallelism opportunities without requiring software source code modifications.

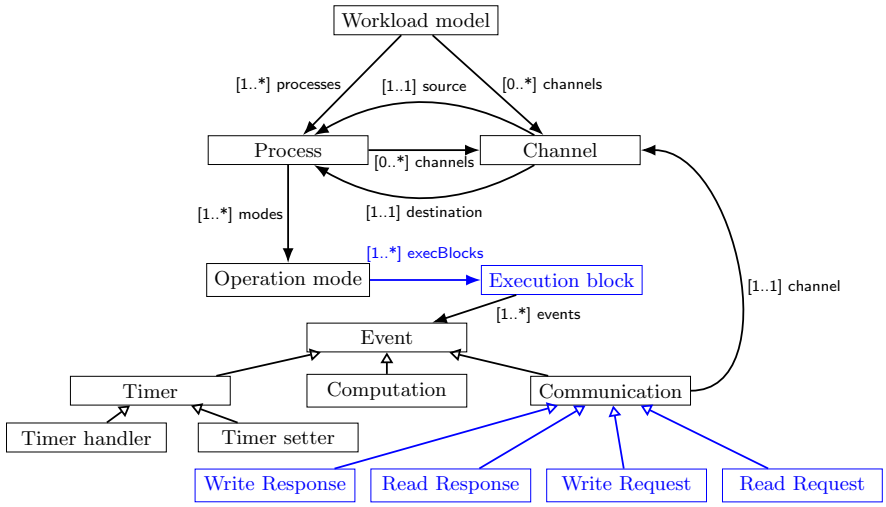


Figure 6.2: The refined application workload model class diagram

6.5 Application Workload Model Refinement

Our refined workload model, illustrated in Figure 6.2, enhances the representation of execution behavior in dCPS. The blue elements in the diagram highlight the key extensions that will be introduced in this section, compared to the initial model described in Section 3.5.1. The workload is structured as a graph of processes and communication channels. Each channel has a source and destination process, and each process may operate in multiple modes to reflect different software/hardware configurations. Our refinement focuses on the internal structure of each operation mode to support intra-process parallelism.

The most significant change in the refined model is the introduction of the *execution block*. Instead of representing each operation mode as a flat sequence of events, we group related events into semantically meaningful segments. Each *execution block* encapsulates a cohesive sequence of communication, computation, or timer events that occur in response to a specific incoming event, such as a received request or response from another process, or the expiration of an internal timer. For example, when a process receives a blocking request, all tasks required to generate the corresponding response – from receiving the request to producing the final reply, including any intermediate computation or further message exchanges – are grouped into a single *execution block*. This level of granularity is chosen not only because it reflects the structure of event-driven execution in dCPS, but also because it captures a contextually

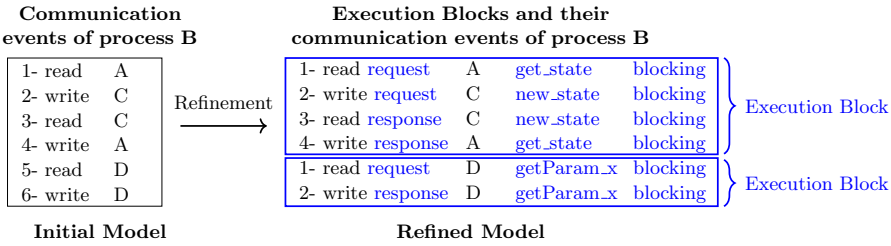


Figure 6.3: Refinement of the application workload model

consistent unit of execution, which is essential for safely identifying dependencies and analyzing parallelism.

Building on the definition of execution blocks, the refined workload model also introduces a more detailed classification of communication events. While the initial model has only two types, read and write, the refined model distinguishes between four types of communication events: read request, read response, write request, and write response. Each communication event also includes a service name, identifying the requested functionality or data, and also a flag indicating whether the interaction is blocking or non-blocking. A blocking request suspends the process until a corresponding response is received, while a non-blocking one allows the process to proceed with other tasks. The other event types, *computation* and *timer*, remain unchanged. Events are assigned to *execution blocks* based on their timestamps and are ordered chronologically within each block. This event order is always preserved.

Figure 6.3 compares the initial and refined workload models for process B, based on Scenario 1 from the example illustrated in Figure 6.1. In this case, process B receives two blocking requests, one from process A (requesting the service `get_state`) and another from process D (requesting `getParam_x`). To respond to the request from process A, process B must send a new request (with service name `new_state`) to process C and wait for a response. As shown, in the initial model (in the left part of Figure 6.3), events form a flat sequence which must be processed in strict order. This structure prevents early handling of new requests, even if they are independent, because they are blocked by preceding events. In contrast, the refined model (in the right part of Figure 6.3) segments the sequence into distinct execution blocks based on request/response boundaries and service names, thereby enabling potential reordering or parallel execution of the blocks under certain conditions that are discussed in Section 6.8.

Table 6.1: Additional trace points for the refined application workload model

Function	Collected Data
handle event loop	timestamp, process name, trace point location, function type (message handler or internal timer handler)
receive request receive response send request send response	timestamp, process name, function type, trace point location, command name

6.6 Collected Execution Traces

Like the initial application workload model, two types of traces, namely application-level execution traces and system-level execution traces are used to derive the refined workload model. While system-level execution traces remain unchanged as used in the initial derivation approach, application-level execution traces require additional information to be collected in order to support automated derivation of the refined application workload model. As discussed in Section 2.4, the processes rely on an event loop to handle both internal and external events. To observe the flow of event processing, we instrument the handle event loop function with trace points at both its start and end. These points record the timestamp, process name, trace point location (start or end), and function type, classified as either a *message handler* or an *internal timer handler*.

In addition to the event loop, we instrument the core message-processing functions. For incoming messages, the event loop receives the message from the appropriate interface (e.g., a file descriptor or socket) and dispatches it to either the *receive request* or *receive response* function, based on header analysis. Similarly, for outgoing messages, the *send request* and *send response* functions are used. All four functions are instrumented at both entry and exit points, capturing the timestamp, process name, function type, trace location, and command name (representing the requested or responded functionality). Since these functions reside in a shared library, the new trace points are automatically inherited by all processes. Table 6.1 summarizes the newly introduced trace points, including the function types, and the data collected.

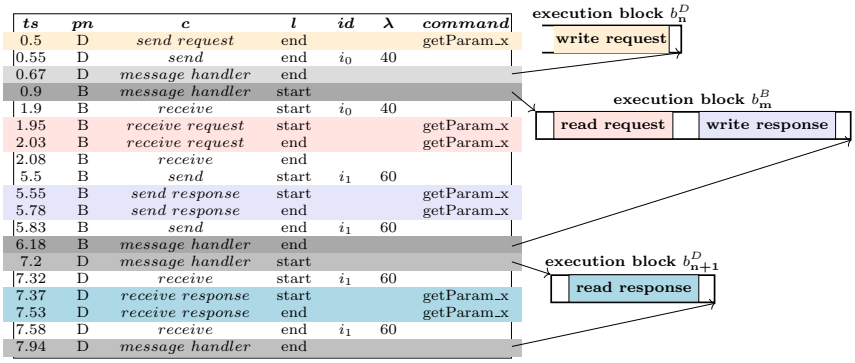


Figure 6.4: Automated derivation of the refined application workload model

6.7 Automated Derivation of Refined Workload Model

To derive the refined application workload model, the aforementioned enhanced execution traces are processed and transformed into the corresponding model components shown in Figure 6.2. This section focuses specifically on the refinement-related aspects, i.e., the blue components in Figure 6.2. The remaining model components are derived by following the initial derivation procedure described in Section 4.5.1. Figure 6.4 illustrates a snippet of collected trace data (left) and its corresponding model components in the refined workload model (right). The trace data includes the timestamp (ts), process name (pn), function type (c), trace point location (l), message identifier (id), message size (λ), and command name. This example is part of the message exchange, illustrated in Figure 6.1, where process D sends a request to process B using the command `getParam_x`. After receiving and processing the request, process B sends a response back, which is later received by process D. This trace segment highlights the key data required for constructing the refined workload model. The derivation process includes three steps: 1) identifying execution blocks, 2) mapping communication events, and 3) determining their blocking behavior. Each step is explained below:

Step 1 - Identify Execution Blocks: Each invocation of the message handler or internal timer handler function in the trace defines the boundaries of an execution block. The start and end trace points determine the temporal scope of each execution block. As shown in Figure 6.4, the gray-highlighted rows indicate the full boundaries of execution blocks b_m^B (for process B) and b_{n+1}^D (for process D). For the earlier block b_n^D, only its end point is visible in the trace snippet.

Step 2 - Map Communication Events: Traces from the four message-passing functions – *receive request*, *receive response*, *send request*, and *send response* – are mapped to the corresponding refined communication event types: read request, read response, write request, and write response. Each event is assigned to the execution block whose temporal boundaries contain its timestamp. Every communication event also includes a service name, extracted from the trace (as the command name), indicating the requested functionality. In Figure 6.4, process B execution block b_m^B includes two communication events: read request and write response. For process D, the earlier block b_n^D may contain multiple communication events, but based on the visible portion of the trace, the last identifiable one is a write request. The following block, b_{n+1}^D, contains only a read response communication event.

Step 3 - Determine Blocking Behavior: Each communication event is classified as blocking or non-blocking based on whether its corresponding counterpart appears within the same execution block. If the paired event (e.g., a write response for a read request) is present in the same block and has the same service name with reversed source and destination processes, the event is considered blocking; otherwise, it is non-blocking. For instance, in Figure 6.4, the read request in b_m^B is blocking because its paired write response occurs in the same block. In contrast, the read response in b_{n+1}^D is non-blocking, as its corresponding write request occurred earlier in b_n^D.

6.8 Feasibility for Parallel Execution

After deriving the refined application workload model, an analysis is conducted for each process to determine which execution blocks within the process can be safely reordered or executed in parallel. This section describes our analysis approach and is divided into two subsections, i.e., Section 6.8.1 focuses on determining eligibility for parallel execution of blocks, and Section 6.8.2 describes how this information is used during simulation of the modeled dCPS.

6.8.1 Determining Parallel Execution Eligibility of Blocks

To enable parallel execution during simulation, we first determine which execution blocks can safely run in parallel. Algorithm 9 outlines the procedure. It takes two execution blocks from the same process p in operation mode m, and returns a boolean indicating whether they are eligible for parallel execution. The algorithm checks two conditions: 1) whether the blocks are independent (Line 1), and 2) whether they do

Algorithm 9: Check Eligibility for Running in Parallel

Input: $block_1^{p,m}$, $block_2^{p,m}$
Output: areEligible

```
1 dep ← checkDependency(block_1^{p,m}, block_2^{p,m});
2 res ← checkSharedReadResponse(block_1^{p,m}, block_2^{p,m});
3 if dep = false and res = false then
4     return true;
5 return false;
```

Algorithm 10: checkDependency

Input: $block_1^{p,m}$, $block_2^{p,m}$
Output: dependent

```
1 h_1 ← getHistory(block_1^{p,m});   h_2 ← getHistory(block_2^{p,m});
2 for x_i ∈ h_1, y_j ∈ h_2 do
3     if x_i = END or y_j = END then
4         return true;
5     if x_i.block = y_j.block then
6         return isBlocking(x_i.event);
7 return false;/* no shared execution block                    */
```

not share a read response event (Line 2). The blocks are considered eligible for parallel execution only if both conditions are met (Lines 3-4). These checks are described in detail below.

Check 1: Independence of execution blocks: Algorithm 10 formalizes the procedure for determining whether two execution blocks are independent. It receives two blocks of process p in the same operation mode m, where $block_1^{p,m}$ occurs before $block_2^{p,m}$, and returns a boolean indicating whether they are dependent. The algorithm begins by retrieving the call history of each block (Line 1). The call history of a given block is defined as the sequence of execution blocks – along with their triggering events – that recursively led to the block's activation.

To illustrate this concept, Figure 6.5 shows four processes (A, B, C, and D) in mode m_0, each containing several execution blocks (e.g., A2, B7, B8, etc.). Each execution block includes a sequence of events, represented as vertical slices within the block. Events highlighted in orange indicate write operations (write request or write response) that trigger execution of blocks in other processes. Blue arrows represent inter-process communication. Suppose we want to determine whether execution blocks D5 and D6 in process D are dependent. The call history of D5 is obtained by tracing communication

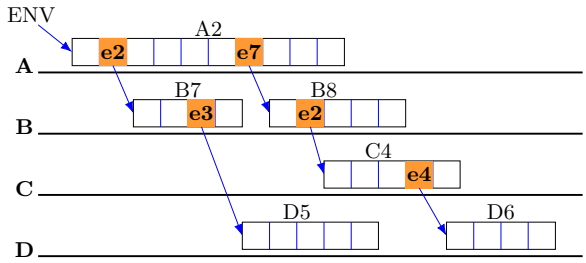


Figure 6.5: The dependency check of execution blocks

backward: from B7 (triggered by event e3), to A2 (triggered by e2), and finally to the environment (ENV), resulting in:

$$\text{B7:e3} \rightarrow \text{A2:e2} \rightarrow \text{END}$$

If an execution block is triggered by a timer event or ENV communication, we mark the end of its call history with `END` because there is no preceding triggering block to trace further. A similar procedure is followed for block D6, resulting in the following call history:

$$\text{C4:e4} \rightarrow \text{B8:e2} \rightarrow \text{A2:e7} \rightarrow \text{END}$$

Once the call histories are obtained, Algorithm 10 compares them to identify the first shared ancestor block (Lines 2-6). If either of the histories reaches `END` without finding a common ancestor, the blocks are conservatively marked as dependent (Lines 3-4), because the causal relationship cannot be fully established. This conservative assumption ensures safe reordering in the absence of traceable triggering dependencies. If both histories terminate at different root blocks without any shared ancestor, the blocks are considered independent (Line 7). However, if a shared ancestor is found (Line 5), the triggering events within that block are compared. If the earlier event is blocking, its execution may impose a constraint on the ordering of the resulting execution blocks. In this case, the algorithm conservatively concludes that the blocks are dependent to preserve the original execution order and prevent unintended behavior (Lines 5-6).

In the example in Figure 6.5, the first shared block is A2, which contains two triggering events e2 and e7. To assess the dependency between D5 and D6, we examine the blocking status of e2. If e2 is a blocking write request, the process must wait for a response before executing e7. Similarly, if e2 is a blocking write response, it means the associated request occurred earlier in the same block and must be completed first. In both cases, the strict ordering of events implies that not only the events

themselves, but also the execution blocks they trigger, must preserve the original sequence and cannot be reordered. In contrast, if e2 is a non-blocking write request, the response will be handled in a future execution block, so execution of e7 is not dependent on it. Likewise, if e2 is a non-blocking write response, the corresponding request was received in a previous block, and this response simply completes that interaction without constraining subsequent events. In such cases, the execution blocks are considered independent.

Check 2: Shared read response event detection: After determining the independence of two execution blocks, we further check whether they share a read response event. This step ensures that the blocks are not competing to receive the same response, which could otherwise lead to deadlocks or violations of causal message ordering. To perform this check, we inspect all events in both blocks to identify a shared read response originating from the same source process and associated with the same service name. If such an event is found, the blocks are not eligible for parallel execution.

It is important to note that it is not necessary to compare every pair of blocks within the same mode of a process. Instead, the analysis is limited to a bounded-size window of neighboring blocks that are likely candidates for reordering. In our implementation, we examine the 30 closest successors of each block. However, runtime profiling shows that, in practice, fewer than 10 neighboring blocks typically run in parallel. This window size is configurable and can be adapted to systems with different workload characteristics or higher potential for parallel execution.

6.8.2 Simulation Strategy for Parallel Execution of Blocks

Once the set of eligible blocks for parallel execution is identified using Algorithm 9, this information is utilized to enable safe concurrent execution during simulations of the modeled dCPS. The refined application workload model is transformed into a format compatible with discrete-event simulators. For our experiments, we use the OMNet++ simulator.

To support parallel execution, we extend the runtime behavior of each process compared to the initial model, and we introduce a shared parallel eligibility file. This file defines, for each process and operation mode, which execution blocks are eligible to run in parallel with one another.

The runtime behavior of each process is illustrated in Listing 6.1, which presents the pseudocode for how execution blocks are managed during simulation. At the start

```
1  void Initialization() {
2      blockList =load_ordered_executionBlocks();
3      readyList = empty; runningList = empty;
4      firstBlk = blockList.front();
5      readyList.add(firstBlk);
6      seqPtr = firstBlk.id;
7      if (allowExecution(firstBlk.initEv)) {
8      runningList.add(firstBlk);
9      readyList.remove(firstBlk);}}
10 void onMessageReceived(Message msg) {
11     msgBlk = findRelatedBlock(msg, blockList);
12     eligible = true;
13     foreach (Block bl: runningList) {
14         if (!checkEligibility(msgBlk, bl)) {
15             eligible = false; break; }}
16     if (eligible)
17         runningList.add(msgBlk);}
18 void onExecutionFinished(Block* doneBlock) {
19     runningList.remove(doneBlock);
20     if (doneBlock.id == seqPtr) {
21         nextBlock = blockList.front();
22         seqPtr = nextBlock.id;
23         eligible = true;
24         foreach (Block bl: runningList) {
25             if (!checkEligibility(nextBlock, bl)){
26                 eligible = false; break;}}
27         if (eligible) {
28             readyList.add(nextBlock);
29             if (allowExecution(nextBlock.initEv)){
30             runningList.add(nextBlock);
31             readyList.remove(nextBlock);}}}
```

Listing 6.1: Runtime behaviour of a process in the simulation

of the simulation (`Initialization` procedure, Line 1), each process loads its list of execution blocks (referred to as the `blockList`) while preserving their original order (Line 2). This ordering is particularly important for blocks triggered by the ENV or by internal timer expirations, as they are not eligible to run in parallel with any other blocks and must therefore be executed sequentially, following the original execution order. To track this order, a dedicated pointer called `seqPtr` is maintained. It always points to the head block of the `blockList`, even after that block has been moved to the `readyList` or `runningList`. The pointer is updated only after the block it references has completed execution.

In addition to the `blockList`, the simulation maintains two other structures: A `runningList`, which stores information about currently executing blocks, such as their event counters and status flags (e.g., whether they are waiting for a message or for platform-side execution), as well as a `readyList`, which holds blocks that are eligible to execute as soon as their triggering conditions are satisfied. Both lists are initialized as empty at the beginning of the simulation (Listing 6.1, Line 3).

The simulation begins by moving the first block from the `blockList` to the `readyList` and updating the `seqPtr` accordingly (Lines 4-6). Each execution block includes an

Block	Parallel
b_1	b_3, b_4
b_2	-
b_3	b_5, b_6
b_4	b_5, b_6
b_5	b_6, b_7
b_6	b_9
b_7	-

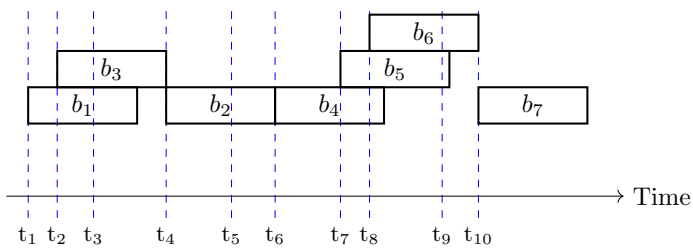


Figure 6.6: An example of parallel execution of blocks

initial communication or timer handler event, depending on what triggered the block, denoted as `initEv`. This event defines the condition under which the block becomes eligible for execution. If `initEv` is a timer handler, the block must wait until the corresponding timer goes off before it becomes eligible for execution. If it is a read request or read response, the block must wait until the corresponding message is received. If it is a write request, the block can be moved directly from the `readyList` to the `runningList` and begin execution (Lines 7-9).

During simulation, when a process receives a message (`onMessageReceived` procedure, Line 10), it searches for the first execution block in its `blockList` whose `initEv` matches the received message (Line 11). The simulator then checks whether this block can be added to the `runningList`, which is determined by verifying that it is eligible to run in parallel with all currently executing blocks (Lines 12-15). If so, it is added to the `runningList` and begins execution (Lines 16-17).

Once a block completes its execution (`onExecution Finished` procedure, Line 18), it is removed from the `runningList` (Line 19). If it is the block currently pointed to by `seqPtr`, the simulator advances `seqPtr` and retrieves the next head block in the original sequence (Lines 20-22). If the new block is eligible to run in parallel with all currently executing blocks, it is added to the `readyList` (Lines 23-28). If its `initEv` allows immediate execution – either because it is a write request, or a read request or read response for which the corresponding message has already been received, or a timer event whose time has already passed – it is transferred directly to the `runningList` (Lines 29-31). If the block is not yet eligible to be added to the `readyList`, it must wait until one or more active blocks complete before it can proceed. This mechanism ensures that execution blocks are scheduled as early as possible to maximize parallelism, while still preserving the original block order when blocks are not eligible to run in parallel.

Figure 6.6 illustrates a scenario involving parallel execution. The table on the left indicates, for each execution block, which subsequent blocks are eligible to run in

parallel. As noted earlier, only blocks that appear later in the `blockList` are considered for reordering, as earlier blocks are assumed to have already completed. The simulation begins at time t_1 with the execution of block b_1, assuming its `initEv` is a write request, and `seqPtr` is initialized to point to b_1. At t_2, the message associated with block b_3's `initEv` arrives. Since b_1 and b_3 are eligible to run in parallel based on the table, b_3 is immediately scheduled for execution. At t_3, the message corresponding to block b_2's `initEv` is also received, making it a candidate for execution. However, due to dependency constraints, b_2 cannot run concurrently with the currently executing blocks (b_1 and b_3), and is therefore not scheduled at this point. Once b_3 completes its execution, the simulator retrieves the next block at the head of the `blockList` and updates `seqPtr` to b_2. Because the message for b_2 has already been received, it is immediately moved to the `runningList` and begins execution.

At time t_5, the message corresponding to the `initEv` of block b_4 is received, making it ready for potential execution. However, b_4 cannot run concurrently with the currently executing block b_2 based on the table in the left part of Figure 6.6, so the message is queued. Once b_2 completes its execution, the next block at the head of the `blockList` is retrieved, and `seqPtr` is updated to point to b_4. At time t_7 and t_8, the messages for b_5 and b_6 are received. Since both blocks are eligible to run in parallel with the currently executing block b_4, they are scheduled and executed concurrently. At time t_9, the message related to `initEv` of block b_7 is received. However, b_7 cannot start yet because it is not eligible to run in parallel with either b_5 or b_6. Finally, at time t_{10}, after both b_5 and b_6 complete their execution, b_7 becomes the next block at the head of the `blockList` and is scheduled for execution.

6.9 Experimental Evaluation

To evaluate the effectiveness of our refined application workload model and the automated parallelism detection method, we conduct an experimental study using trace data collected from the ASML TWINSCAN lithography machine.

6.9.1 Experimental Setup

Our evaluation focuses on the server subsystem of the TWINSCAN machine, which runs on a Linux-based platform and communicates with multiple embedded real-time subsystems. We concentrate on the server subsystem because it executes the majority of software processes and is therefore the most critical location for uncovering

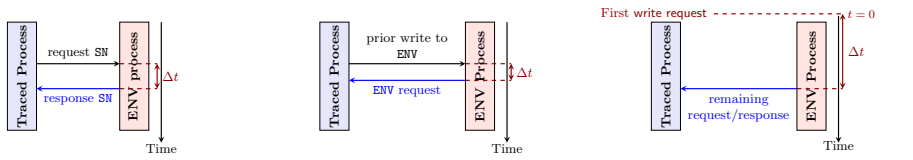
parallelism opportunities. The embedded subsystems, in contrast, operate under strict real-time constraints, where modifying execution behavior is undesirable, and collecting their traces would require additional engineering effort that is not necessary to demonstrate the capabilities of the proposed refined model. From the collected traces of a wafer lot operation involving five wafers, we derive an application workload model containing **309** software processes and **1521** inter-process communication channels. The workload model includes **489459** execution blocks and **597210** messages, representing communications in which the traced host acted as sender, receiver, or both.

To evaluate the wafer processing time, we identified service calls in the traces that correspond to wafer-handling events. These calls, observed in specific request and response messages, are used to mark the start and end points of each wafer's operation. The PDW is then computed as the time difference between the corresponding start and end events.

To model the effects of the environment on the system's timing behavior, we use the ENV model with associated delay parameters, denoted as Δ, as explained in Section 3.5.4. The refinement of the workload model, particularly through the introduction of more detailed communication event types and the inclusion of service names, enables a more fine-grained estimation of the values of these delay parameters. Figure 6.7 illustrates the three scenarios considered in our ENV delay modeling approach. In most cases, since each communication event includes a service name (`SN`), we directly determine the delay parameter value Δ by calculating the time difference between a traced process's write request to an ENV process and the corresponding read response from that ENV process using the same service name, as shown in Figure 6.7(a).

For situations where the ENV initiates a request toward the traced process (`ENV` request), we estimate the delay Δ based on the most recent write request or write response previously sent by the same traced process to that ENV process. We assume that the ENV request occurs shortly after this preceding write event, as illustrated in Figure 6.7(b).

Finally, in cases where no matching service name (`SN`) is available or no prior write event can be found, we conservatively assign the delay based on the time difference from the first root-level write request in the model. This event marks the simulation's starting point and represents the earliest available interaction, as shown in Figure 6.7(c).



(a) Matched Request-Response (b) Inferred from Prior Write (c) Considering the Simulation Start

Figure 6.7: Three ENV delay modeling scenarios

Table 6.2: Comparison of PDW between real system execution and simulation considering inter-process parallelism only

Wafer	Reality (s)	Simulation (s)
1	47.638	47.651
2	56.764	56.782
3	50.766	50.782
4	47.715	47.735
5	47.770	47.788
Average	**50.130**	**50.148**

6.9.2 Results

First, we validate the accuracy of our simulation model by comparing the simulated PDW with real execution data from the TWINSCAN machine. This comparison is performed under a baseline configuration that includes only inter-process communication and does not leverage intra-process parallelism. This setup ensures that the simulation reflects the actual system behavior and provides a reliable baseline for further performance evaluation.

Table 6.2 shows the PDW for five wafers from both real and simulated executions. The average PDW in the real system is 50.1309 s, compared to 50.1481 s in simulation, a negligible difference of 17.2 ms, corresponding to an error of only 0.034 %. This confirms that the simulation model provides sufficient accuracy for performance analysis.

Next, we examine the impact of enabling intra-process parallelism alongside inter-process parallelism. We vary two key parameters: the number of CPU cores in the platform model, and the value of Δ in the ENV delay model, which controls how quickly new execution blocks triggered by environment interactions become eligible for execution and thus increases the opportunity for parallelism. Five levels of ENV delay reduction are tested: 0 % (baseline), 25 %, 50 %, 75 %, and 100 % (representing immediate ENV response upon sending a request from the traced server). As a reminder, execution blocks triggered by ENV interactions must be executed sequentially in their original order and are not eligible for parallel execution.

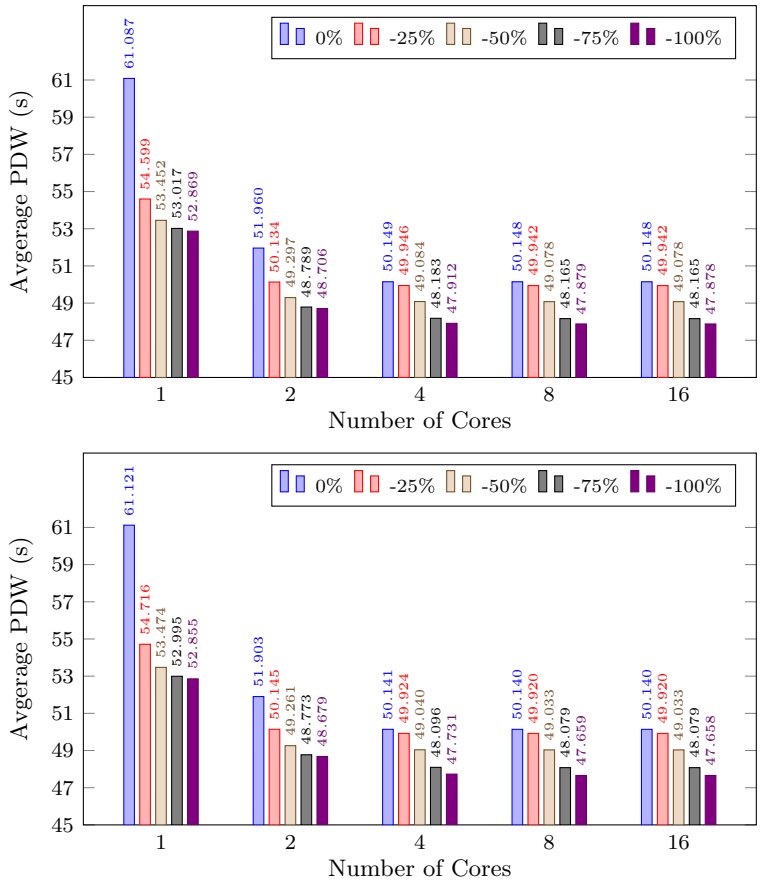


Figure 6.8: The effect of ENV delay reduction and number of cores on PDW. Top: inter-process only. Bottom: inter- and intra-process

Figure 6.8 shows the effect of the aforementioned parameters on the average PDW. The top chart reflects inter-process parallelism only, while the bottom chart includes both inter- and intra-process parallelism. Each bar corresponds to a specific ENV delay level. The horizontal axis indicates the number of CPU cores (the default server configuration has 16 cores), and the vertical axis shows the average PDW in seconds. As shown in both charts, decreasing ENV delays and increasing the number of cores consistently reduce the average PDW.

The results demonstrate that intra-process parallelism yields additional performance gains, particularly when ENV delays are reduced. At the baseline level (0 % delay reduction), the improvement on the default 16-core platform is marginal, at around 8 ms,

indicating that the environment latency remains the dominant bottleneck. Although intra-process execution is accelerated, the system must still wait for ENV interactions, especially those modeled with conservative assumptions anchored to the root-level write request, as previously shown in Figure 6.7(c), which limits the overall benefit. In contrast, when ENV delays are fully eliminated (100 %), the gain increases to approximately 220 ms, highlighting that the effectiveness of intra-process parallelism depends heavily on the environment responsiveness in this case study.

Despite this improvement, the overall gain remains modest, which can be attributed to two main factors. First, the software under study was not originally designed with parallel execution in mind, likely to avoid unintended side effects such as race conditions or inconsistent execution behavior. As a result, the structure of the system may inherently limit the amount of parallelism that can be safely exploited. Second, our method for identifying parallelizable blocks is deliberately conservative. As discussed in Check 1 (Section 6.8.1), conservative assumptions are necessary when trace information is insufficient to guarantee safe block independence. These assumptions prevent potential deadlocks in the system but may also restrict opportunities for concurrent execution.

Finally, we evaluate the impact of CPU frequency scaling using a fixed configuration of 8 cores under 100 % ENV delay reduction. This choice is made because, as shown in Figure 6.8, the difference between using 8 and 16 cores is not significant and falls within the simulation error margin.

Figure 6.9 shows the effect of frequency scaling on PDW under both inter-process-only and combined inter- and intra-process parallelism exploitation cases. The base frequency setting is denoted as 0 on the horizontal axis, while the average PDW is depicted on the vertical axis. The horizontal axis depicts the clock frequency changes as a percentage of the base clock frequency. As CPU frequency increases, performance improves in both cases due to shorter execution times. However, the performance difference between the two cases gradually decreases, because frequency scaling accelerates all execution blocks, including those that must run sequentially. As sequential execution becomes less time-consuming, the relative advantage of parallelism exploitation diminishes. Consequently, the two cases converge in performance at higher frequencies.

All our results above indicate that multi-threaded execution of code blocks within event-driven processes (i.e., exploiting the available intra-process parallelism) can yield measurable performance improvements, particularly when the responsiveness of other embedded hosts (ENV) increases. However, they also reveal a saturation point,

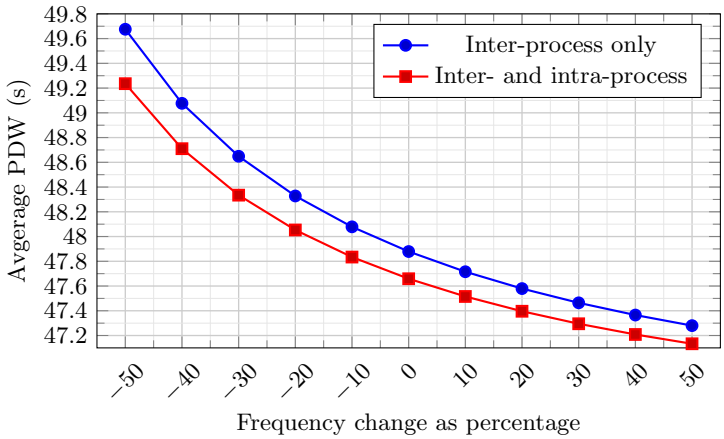


Figure 6.9: The effect of frequency scaling on PDW for 8-core platform with 100% ENV delay reduction

beyond which such multi-threaded execution no longer leads to performance gains unless changes are made to the software structure of dCPS in order to create (more) blocks and processes that can really execute in parallel. This underscores the value of our refined application workload model, not only in enabling realistic performance evaluation of an existing software structure of dCPS but also in supporting exploratory what-if analyses with respect to potential changes in the software structure.

6.10 Conclusion

This chapter presents a refined application workload model that captures both inter-process and intra-process execution dependencies in complex dCPS. The model is derived from execution traces and supports a method for identifying tasks (code blocks) within software processes that can be safely reordered or executed in parallel without modifying the existing software structure. We validate our approach on the ASML TWINSCAN lithography machine and the results reveal measurable performance gains in wafer processing time. The results also highlight the impact of environmental responsiveness and hardware capacity on the effectiveness of parallel execution. These findings underscore the value of our refined workload model not only for realistic performance evaluation, but also for enabling what-if analyses of application workloads that complements platform and mapping exploration in large-scale industrial dCPS.

Chapter 7

Summary, Discussion, and Future Directions

7.1 Summary of Challenges and Contributions

This thesis addresses the growing need for a scalable and automated DSE methodology tailored to complex industrial dCPS such as the ASML TWINSCAN machine used as a case study in the thesis. To this end, the proposed CompDSE methodology systematically tackles five core scientific challenges.

First, to overcome the lack of a coherent and scalable framework for efficient DSE focused on performance evaluation in complex industrial dCPS (challenge **CHL1**), we introduce a structured methodology that combines system-level modeling, automated model derivation and transformation, and discrete-event simulation for evaluation.

Second, to support abstract yet accurate system modeling (challenge **CHL2**), we define models for the application workload, the computing hardware platform, and the workload-to-platform mapping. These models are sufficiently abstract to enable efficient exploration while remaining accurate enough to yield meaningful insights.

Third, we identify the necessary execution traces and develop algorithms to automatically derive the aforementioned models from traces collected during real system operation (challenge **CHL3**), thereby eliminating the need for software source code analysis, which is often impractical in industrial contexts.

Fourth, we implement model-to-model transformations and code generators (challenge **CHL4**) to produce simulation-ready entities, thereby enabling rapid performance analysis in a target simulation environment. In this thesis, the OMNeT++ environment has been targeted because it supports a hierarchical module structure, event scheduling, and message passing between components, making it suitable for simulation of large-scale distributed systems with complex interactions.

Finally, to capture intra-process parallelism alongside inter-process parallelism

(challenge **CHL5**), we propose a refined application workload model by grouping cohesive computation and communication events into semantically meaningful execution blocks as well as a dependency analysis approach to detect which blocks within a process can execute safely in parallel.

Through experiments on real trace data collected from the ASML TWINSCAN machine, we demonstrate the accuracy and efficiency of the proposed CompDSE methodology. It enables efficient performance evaluation of large-scale workloads and supports insightful system architecture exploration for realistic industrial settings.

7.2 Discussion and Future Directions

While the CompDSE methodology offers a practical and generalizable framework for the DSE of large-scale industrial dCPS, several limitations and open directions have emerged during its development and evaluation. These can be framed as key challenges for future work.

Partial Traceability of Subsystems

Due to system constraints, we may be unable to trace all subsystems to automatically derive models. Some subsystems are bare-metal and lack an OS or another infrastructure with tracing facilities. Thus, such untraceable subsystems have been grouped into the ENV model. In addition, some embedded subsystems, although they are traceable, they cannot store large traces locally, similar to most resource-constrained embedded systems. As a result, trace data has to be streamed over the communication network, which introduces bandwidth bottlenecks. In our ASML TWINSCAN case study, it is not feasible to trace more than seven subsystems without disrupting the system operation. Even tracing these seven subsystems requires careful configuration of local buffer sizes, streaming rates, and other parameters.

One possible direction to solve the aforementioned tracing problem could be to repeat the same system operation multiple times under the same configuration and operation mode, and then aggregate the resulting partial traces to automatically derive a more complete application workload model. However, such an approach poses challenges because according to ASML engineers, the system's operational determinism, i.e., executing the exact same workload order across several runs, cannot always be guaranteed. As a result, aggregating traces across executions may introduce inconsistencies. In practice, however, these variations are typically limited in magnitude

and appear as small execution-level differences, which may fall below the resolution of the abstracted workload model and therefore do not prevent meaningful comparative analysis across design alternatives within the defined design space.

Unmodeled Shared-Memory Communication

While the majority of inter-process communication in the ASML TWINSCAN machine is captured through message passing, a small subset of interactions rely on shared-memory communication. These interactions are not traced and modeled in this work. Although their impact on the system model accuracy is minimal in our case, other dCPS in which shared memory communication plays a more significant role may require explicit modeling of such communication to ensure accurate performance analysis.

Events Signatures in Application Workload Model

In our CompDSE methodology, we estimate the workload signature of the computation and communication events using CPU cycle counts as an approximation of execution cost. This abstraction captures only the computational demand of events on a CPU and omits important factors such as cache behavior, memory access latency, and other architectural interactions that may significantly affect actual execution time of events.

This simplification also limits the DSE to hardware platforms with CPU-based computing architectures. As a result, the CompDSE methodology cannot be directly applied to evaluate systems that include alternative computing architectures and components such as GPUs or FPGAs for example. A promising future direction is to develop a mapping approach based on empirical measurements that enables the translation of execution costs (signatures) of events across heterogeneous platforms. For example, by deriving mathematical functions or learning ML models that approximate how execution signatures of events change between different architecture components, CompDSE could be extended to explore a broader hardware platform design space.

Simplified Scheduling for Processes

In this work, we employ a FCFS scheduling policy to schedule the processes mapped onto a given host, while the real ASML TWINSCAN system employs a more complex, priority-based scheduling. Although this simplification does not significantly impact the accuracy of our performance analysis for this system, the same may not hold for other dCPS that operate under hard real-time constraints for example. In such

systems, accurately modeling the scheduling policy, potentially replicating the real-time scheduler used in deployment, may be essential to ensure reliable performance analysis.

Communication Network Model Simplification

Our communication network model within the hardware platform model relies on approximated communication delays associated with communication events. While this high level of abstraction is sufficient for our case study to explore computing hardware platforms and workload-to-platform mappings, it restricts the fidelity with which alternative network architectures can be evaluated. Although the design space defined in Section 3.6 allows modifications to the network topology, such variations are assessed at the abstraction level of the adopted communication model. Detailed protocol behavior, routing dynamics, and fine-grained contention effects are not explicitly represented. Consequently, the impact of topology changes on communication load and overall system performance can only be analyzed in terms of abstract latency and bandwidth parameters rather than detailed network-level effects.

Scope and Limitations of the Explored Design Space

The design space defined in this thesis is intentionally scoped to what can be evaluated efficiently using abstract models. This scoping reflects a trade-off between accuracy and scalability, i.e., increasing model details may improve fidelity for specific effects, but it also increases design-point evaluation time and can make DSE impractical for large and complex industrial distributed systems.

Some limitations arise directly from the chosen abstraction level and cannot be removed within the current methodology. In particular, design dimensions that require instruction-level or micro-architectural visibility are fundamentally outside the scope of the current methodology. This includes exploration of cache hierarchies and cache policies, pipeline-related effects, fine-grained memory-access behavior, and other instruction-level mechanisms. Supporting such variations would require instruction-level traces or cycle-accurate models, which conflict with the intended scalability constraints in large industrial environments.

There are additional limitations in this thesis that do not stem from the proposed methodology itself, but from the current instantiation of the models. For example, as discussed earlier, a more detailed communication network model would enable the network architecture and topology to be treated as explicit design dimensions, thereby allowing a broader exploration of design points. Similarly, heterogeneous

compute resources (e.g., GPUs or FPGAs) could be considered at an abstract level if an appropriate equivalence or mapping of computational capability can be established as discussed.

Memory-related design dimensions occupy an intermediate position with respect to these limitations. While detailed memory behavior is strongly influenced by micro-architectural mechanisms, memory effects can, in principle, be incorporated at a higher level of abstraction. For example, memory can be modeled as an abstract shared service characterized by parameters such as bandwidth, and latency, and application workloads can be augmented with abstract memory-demand attributes. Such representations do not aim to capture precise memory access timing, but rather to reflect the relative impact of memory pressure on execution behavior across alternative design points. However, introducing these abstractions requires additional assumptions regarding overlap between computation and memory access, as well as contention effects.

Algorithmic or functional changes in the application software are not treated as direct variation/design points within the defined design space. However, such changes can still be evaluated within the overall workflow by collecting new execution traces and re-deriving the workload model automatically for the modified software, after which the resulting model can be assessed under alternative platform and mapping configurations.

Generality vs. Accuracy in Parallelism Analysis

The refined application workload model enables the identification of inter-process parallelism, i.e., identifying dependencies between execution blocks within a process to reveal potential parallel execution of blocks. To identify such dependencies, we adopt a conservative dependency analysis approach in order to maintain generality, deadlock-free model execution, and avoid reliance on system domain knowledge. However, incorporating system-specific knowledge in the dependency analysis could make it less conservative, thereby revealing more accurately the potential parallelism between execution blocks. For example, in the ASML TWINSCAN machine, the measurement and exposure phases belong to two distinct and independent functional categories. Recognizing this separation could help distinguish independent execution blocks within a process more effectively.

Additionally, in the refined model, the ordering of events within each execution block is kept fixed to ensure correctness. With sufficient domain knowledge, however, this constraint could also be relaxed, allowing events within a block to be reordered or

executed in parallel, thereby revealing further opportunities for parallelism.

Evolving the Δ Parameter Model into Design Variables

The Δ-parameter approach is introduced to capture the timing impact of untraced embedded hosts that interact with the physical environment via sensors and actuators. Two questions arise: Can individual Δ parameters serve as independent design variables in DSE rather than scaling all Δ parameters, as done in the experiments presented in Chapter 6? If so, can the model be reformulated such that the Δ parameter values capture only delays due to the physical environment (sensors and actuators), while excluding execution and communication latency inside untraced subsystems?

In practice, for a complex dCPS, our current approach generates thousands of Δ parameters because each communication with an untraced host is represented individually. The very large number of Δ parameters creates a huge design space that is costly and infeasible to explore. In addition, many of these parameters are dependent, so varying them independently can yield unrealistic or infeasible design points.

A straightforward way to reduce the number of Δ parameters is to increase trace coverage so that fewer interactions remain untraced. However, as explained earlier, tracing additional subsystems has its own challenges, and moreover, some subsystems will remain untraceable anyway. In such cases, a more effective approach can be to incorporate system domain knowledge. The key idea is to identify a small subset of independent Δ parameters as `representative` parameters and to express the remaining parameters as functions of these representatives. Using only the representative parameters as design variables in DSE can reduce the size of the design space significantly and can prevent infeasible or unrealistic design points from being considered. This can decrease the number of Δ parameters and improve the effectiveness and efficiency of DSE when exploring changes in untraced hosts and in sensors and actuators.

Domain knowledge also helps separate Δ parameters that are tied directly to the physical environment from those that arise from execution or communication within untraced subsystems. For parameters that correspond to repeated interactions with the same sensor or actuator within an operation mode, the observed timings are often constant or nearly constant. In such cases a single characteristic value can serve as the representative for that component. If a different sensor or actuator is selected during DSE, only the corresponding representative value needs to change. Consequently, the portion of the Δ parameters that reflects the physical environment becomes a set of well-defined design variables to explore.

Simulation Speed vs. Model Detail Trade-off

By refining the application workload model, we improve the model accuracy but increase the model simulation time by approximately 10 times, with each design point taking around 10 minutes to evaluate. This fact highlights the need to carefully balance between the model accuracy and model simulation efficiency. Future work aiming for higher model accuracy, such as adding more detail to the ENV model, must consider the impact of such model refinement on the efficiency of DSE, especially the time needed for a single design point evaluation.

OMNet++ Scalability Limits

The OMNeT++ discrete-event simulation environment has been chosen in our work due to its strong capabilities in modeling large-scale networks and distributed systems. However, our practical experiments reveal that it has scalability limits in terms of the number of modules that can be simulated. This means that even if we are able to trace all subsystems in a dCPS, it may not be feasible to include all process entities as separate modules in the simulation environment. Therefore, a systematic study is needed to evaluate the suitability of different discrete-event simulation tools for large-scale workloads in complex dCPS, and to identify alternatives or extensions that can scale better.

7.3 Concluding Remark

This thesis concludes a very focused and industry-driven research effort towards enabling DSE for complex industrial dCPS. While this work marks the end of our contribution to this specific line of research, it lays a foundation that can be extended and applied beyond the system studied here.

The proposed CompDSE methodology is not limited to the ASML TWINSCAN machine. Its trace-based modeling approach, algorithms and tools, simulation infrastructure, and refinement techniques are relatively general and can be adapted to other large-scale systems. In this sense, the work takes a first step towards making DSE more practical and accessible in industrial contexts, supporting system architects in evaluating alternative designs early in the development process through efficient system simulation.

Bibliography

[1] Sezen Acur and Teun Hendriks. Vision and Outlook for Systems Architecting and Systems Engineering in the High-Tech Equipment Industry. Report TNO-2024-R10542, TNO-ESI, 2024.

[2] Shreya Adyanthaya, Hadi Alizadeh Ara, João Bastos, Amir Behrouzian, Róbinson Medina Sánchez, Joost van Pinxten, Bram van der Sanden, Umar Waqas, Twan Basten, Henk Corporaal, Raymond Frijns, Marc Geilen, Dip Goswami, Martijn Hendriks, Sander Stuijk, Michel Reniers, and Jeroen Voeten. xCPS: A tool to eXplore Cyber Physical Systems. *ACM SIGBED Review*, 14(1):81–95, 2017.

[3] Ilge Akkaya, Patricia Derler, Shuhei Emoto, and Edward A. Lee. Systems Engineering A Formal Approach to Power Optimization in CPSs with Delay-Workload Dependence Awareness. *Proceedings of the IEEE*, 104(5):997–1012, 2016.

[4] John R Allen and Ken Kennedy. Automatic Loop Interchange. In *SIGPLAN symposium on Compiler construction*, pages 233–246, 1984.

[5] Randy Allen and Ken Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *Transactions on Programming Languages and Systems (TOPLAS)*, 9(4):491–542, October 1987.

[6] Rajeev Alur. *Principles of Cyber-Physical Systems*. MIT Press, 2015.

[7] Maral Amir and Tony Givargis. Pareto Optimal Design Space Exploration of Cyber-Physical Systems. *Internet of Things*, 12:100308, 2020.

[8] Hyung-Chan An, Hoeseok Yang, and Soonhoi Ha. A Formal Approach to Power Optimization in CPSs With Delay-Workload Dependence Awareness. *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 35(5):750–763, 2016.

[9] Emad Arasteh Malekzadeh and Rainer Dömer. Improving Parallelism in System Level Models by Assessing PDES Performance. In *Forum on Specification and Design Languages (FDL)*, pages 1–7. IEEE, 2021.

[10] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, December 1994.

[11] J. Bakakeu, Jonathan Fuchs, Tallal Javied, Matthias Brossog, Jörg Franke, H. Klos, Werner Eberlein, Schirin Tolksdorf, Jörn Peschke, and Lars Jahn. Multi-Objective Design Space Exploration for the Integration of Advanced Analytics in Cyber-Physical Production Systems. In *Industrial Engineering and Engineering Management (IEEM)*, pages 1866–1873. IEEE, 2018.

[12] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *Computer*, 36(4):45–52, 2003.

[13] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based Performance Prediction in Software Development: a survey. *Transactions on Software Engineering*, 30(5):295–310, 2004.

[14] Utpal Banerjee. Dependence. In *Dependence Analysis for Supercomputing*, volume 60, pages 27–43. Springer, 1988.

[15] Twan Basten, Emiel van Benthum, Marc Geilen, Martijn Hendriks, Fred Houben, Georgeta Igna, Frans Reckers, Sebastian de Smet, Lou Somers, Egbert Teeselink, Nikola Trčka, Frits Vaandrager, Jacques Verriet, Marc Voorhoeve, and Yang Yang. Model-Driven Design-Space Exploration for Embedded Systems: The Octopus Toolset. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA)*, volume 6415 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2010.

[16] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 41–46, June 2005.

[17] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 Simulator. *SIGARCH computer architecture news*, 39(2):1–7, 2011.

[18] William Lloyd Bircher and Lizy K John. Complete System Power Estimation Using Processor Performance Events. *Transactions on Computers*, 61(4):563–577, 2011.

[19] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, and Thomas Lawrence. Parallel Programming with Polaris. *Computer*, 29(12):78–82, December 1996.

[20] Paul Bogdan and Radu Marculescu. Towards a Science of Cyber-Physical Systems Design. In *International Conference on Cyber-Physical Systems (ICCPS)*, pages 99–108. IEEE, 2011.

[21] Alex Boten and Charity Majors. *Cloud-Native Observability with OpenTelemetry: Learn to Gain Visibility into Systems by Combining Tracing, Metrics, and Logging with OpenTelemetry*. Packt Publishing Ltd, 2022.

[22] Sami Boukhechem and El-Bay Bourennane. TLM Platform Based on SystemC for STARSoC Design Space Exploration. In *Adaptive Hardware and Systems (AHS)*, pages 354–361, July 2008.

[23] Arquimedes Canedo and Jan H. Richter. Architectural Design Space Exploration of Cyber-Physical Systems using the Functional Modeling Compiler. *Procedia CIRP*, 21:46–51, 2014. 24th CIRP Design Conference 2014.

[24] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 15–28, June 2004.

[25] Jianjiang Ceng, Jerónimo Castrillón, Weihua Sheng, Hanno Scharwächter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Tsuyoshi Isshiki, and Hiroaki Kunieda. MAPS: an Integrated Framework for MPSoC Application Parallelization. In *Design Automation Conference (DAC)*, pages 754–759, 2008.

[26] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. A General Framework for Analysing System Properties in Platform-based Embedded System Designs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, volume 3, pages 190–195, 2003.

[27] Anirban DasGupta. The Trimmed Mean. In *Asymptotic Theory of Statistics and Probability*, pages 271–278. Springer New York, 2008.

[28] John II Davis, Christopher Hylands, Jörn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Steve Neuendorffer, Sonia Sachs, Mary Stewart, Kees Vissers, Paul Whitaker, and Yuhong Xiong. Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M01/11, Department of Electrical Engineering and Computer Science, University of California, Berkeley, March 2001.

[29] Patricia Derler, Thomas Huining Feng, Edward A. Lee, Slobodan Matic, Hiren D. Patel, Yang Zhao, and Jia Zou. PTIDES: A Programming Model for Distributed Real-Time Embedded Systems. Technical Report UCB/EECS-2008-72, EECS Department, University of California, Berkeley, May 2008.

[30] Mathieu Desnoyers and Michel R. Dagenais. The LTTng tracer: A Low Impact Performance and Behavior Monitor for GNU/Linux. In *Linux Symposium*, volume 1, pages 209–224, 2006.

[31] Rainer Dömer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel D. Gajski. System-on-Chip Environment: A SpecC-Based Framework for Heterogeneous MPSoC Design. *EURASIP Journal on Embedded Systems*, 2008:1001–1015, 2008.

[32] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing Domain-Specific Languages for Java. *SIGPLAN Not.*, 48(3):112–121, 2012.

[33] Cagkan Erbas, Andy D Pimentel, Mark Thompson, and Simon Polstra. A Framework for System-Level Modeling and Simulation of Embedded Systems Architectures. *EURASIP Journal on Embedded Systems*, 2007(1):82123, 2007.

[34] Thomas Huining Feng, Lynn Wang, Wei Zheng, Sri Kanajan, and Sanjit A. Seshia. Automatic Model Generation for Black Box Real-Time Systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2007.

[35] Paul Feutrier. Dataflow Analysis of Scalar and Array References. *International Journal of Parallel Programming*, 20(1):23–53, February 1991.

[36] John Fitzgerald, Carl Gamble, Richard Payne, and Benjamin Lam. Exploring the Cyber-Physical Design Space. In *INCOSE International Symposium (IS)*, volume 27, pages 371–385. Wiley, July 2017.

[37] Peter Fritzson, Adrian Pop, Karim Abdelhak, Adeel Ashgar, Bernhard Bachmann, Willi Braun, Daniel Bouskela, Robert Braun, Lena Buffoni, Francesco Casella, Rodrigo Castro, Rüdiger Franke, Dag Fritzson, Mahder Gebremedhin, Andreas Heuermann, Bernt Lie, Alachew Mengist, Lars Mikelsons, Kannan Moudgalya, Lennart Ochel, Arunkumar Palanisamy, Vitalij Ruge, Wladimir Schamai, Martin Sjölund, Bernhard Thiele, John Tinnerholm, and Per Östlund. The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development. *Modeling, Identification and Control*, 41(4):241–295, 2020.

[38] Johannes Geismann and Eric Bodden. A Systematic Literature Review of Model-Driven Security Engineering for Cyber-Physical Systems. *Journal of Systems and Software*, 169:110697, 2020.

[39] Daniela Genius, Ilias Bournias, Ludovic Apvrille, and Roselyne Chotin. High-Level Partitioning and Design Space Exploration for Cyber-Physical Systems. In *Model-Driven Engineering and Software Development (MODELSWARD)*, pages 84–91. SCITEPRESS, 2020.

[40] Andreas Gerstlauer, Christian Haubelt, Andy D. Pimentel, Todor Stefanov, Daniel D. Gajski, and Jürgen Teich. Electronic System-Level Synthesis Methodologies. *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 28(10):1517–1530, October 2009.

[41] Frank Ghenassia, editor. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, New York, USA, 1 edition, 2005.

[42] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical Dependence Testing. In *SIGPLAN Programming Language Design and Implementation (PLDI)*, pages 15–29. ACM, 1991.

[43] Imen Graja, Aicha Mechim, Slim Kallel, Nawal Guermouche, and Ahmed Hadj Kacem. Demonstrating BPMN4CPS: Modeling and Verification of Cyber-Physical Systems. In *Consumer Communications & Networking Conference (CCNC)*, page 593. IEEE, 2017.

[44] Jeff Gray and Bernhard Rumpe. Reflections on the Standardization of SysML 2. *Software and Systems Modeling*, 20(2):287–289, 2021.

[45] Lukas Gressl, Christian Steger, and Ulrich Neffe. Design Space Exploration for Secure IoT Devices and Cyber-Physical Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(4):1–24, 2021.

[46] Matthias Gries. Methods for Evaluating and Covering the Design Space During Early Design Development. *Integration, the VLSI Journal*, 38(2):131–183, 2004.

[47] Thomas Gross, David R O'Hallaron, and Jaspal Subhlok. Task Parallelism in a High Performance Fortran Framework. *Parallel and Distributed Technology*, 2(3):16–26, 1994.

[48] Soonhoi Ha, Sungchan Kim, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo. PeaCE: A Hardware-Software Codesign Environment for Multimedia Embedded Systems. *Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3):1–25, 2008.

[49] Christian Haubelt, Thomas Schlichter, Joachim Keinert, and Mike Meredith. SystemCoDesigner: Automatic Design Space Exploration and Rapid Prototyping from Behavioral Models. In *Design Automation Conference (DAC)*, pages 580–585, 2008.

[50] Marius Herget, Faezeh Sadat Saadatmand, Martin Bor, Ignacio González Alonso, Todor Stefanov, Benny Akesson, and Andy D. Pimentel. Design Space Exploration for Distributed Cyber-Physical Systems: State-of-the-art, Challenges, and Directions. In *Euromicro Conference on Digital System Design (DSD)*, pages 632–640. IEEE, 2022.

[51] Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion Detection using Sequences of System Calls. *Journal of computer security*, 6(3):151–180, 1998.

[52] Liang Huang and Kenny Wong. Assisting Failure Diagnosis through Filesystem Instrumentation. In *Center for Advanced Studies on Collaborative Research (CASCON)*, pages 160–174, November 2011.

[53] Tsuyoshi Isshiki, Dongju Li, Hiroaki Kunieda, Toshio Isomura, and Kazuo Satou. Trace-driven Workload Simulation Method for Multiprocessor System-On-Chips. In *Design Automation Conference (DAC)*, pages 232–237, July 2009.

[54] Ruben Jonk, Jeroen Voeten, Marc Geilen, Rolf Theunissen, Yuri Blankenstein, Twan Basten, and Ramon Schiffelers. Inferring Timed Message Sequence Charts from Execution Traces of Large-scale Component-based Software Systems. (ESR-2019-01), 2019.

[55] Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D. Hämäläinen, Jouni Riihimäki, and Kimmo Kuusilinna. UML-based Multiprocessor SoC Design Framework. *Transactions on Embedded Computing Systems (TECS)*, 5(2):281–320, 2006.

[56] Joachim Keinert, Martin Streubūhr, Thomas Schlichter, Joachim Falk, Jens Gladigau, Christian Haubelt, Jūrgen Teich, and Michael Meredith. SystemCoDesigner–an Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming applications. *Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):1:1–1:23, 2009.

[57] Bart Kienhuis, Ed F. Deprettere, Pieter van der Wolf, and Kees Vissers. A Methodology to Design Programmable Embedded Systems: The Y-Chart Approach. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, volume 2268 of *Lecture Notes in Computer Science*, pages 18–37. Springer, 2002.

[58] Dmitrii Kirov, Pierluigi Nuzzo, Roberto Passerone, and Alberto Sangiovanni-Vincentelli. ArchEx: An Extensible Framework for the Exploration of Cyber-Physical System Architectures. In *Design Automation Conference (DAC)*, pages 1–6. ACM, 2017.

[59] Tim Kogel, Neeraj Goel, and Andreas Wieferink. Method for Generating Workload Models from Execution Traces. US Patent Application US20160357516A1, December 2016.

[60] Akash Kumar, Shakith Fernando, Yajun Ha, Bart Mesman, and Henk Corporaal. Multi-Processor System-Level Synthesis for Multiple Applications on Platform FPGA. In *Field Programmable Logic and Applications (FPL)*, pages 92–97, 2007.

[61] Leslie Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17(2):83–93, 1974.

[62] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. In *Concurrency: The Works of Leslie Lamport*, pages 179–196. ACM, 2019.

[63] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. Enjoy Your Observability: an Industrial Survey of Microservice Tracing and Analysis. *Empirical Software Engineering*, 27, 2022.

[64] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic Generation of Software Behavioral Models. In *International Conference on Software Engineering (ICSE)*, pages 501–510. ACM, 2008.

[65] Federico Maggi, Matteo Matteucci, and Stefano Zanero. Detecting Intrusions through System Call Sequence and Argument Analysis. *Transactions on Dependable and Secure Computing*, 7(4):381–395, 2008.

[66] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, 2002.

[67] Radu Marculescu and Paul Bogdan. Cyberphysical Systems: Workload Modeling and Design Optimization. *Design & Test of Computers*, 28(4):78–87, 2011.

[68] Brit Meier, Mladen Skelin, Frans Beenker, and Wouter Leibbrandt. HTSM Systems Engineering Roadmap. Technical Report v1.0, Topsector HTSM, July 2020.

[69] Hugo Meyer, Uraz Odyurt, Simon Polstra, Evangelos Paradas, Ignacio Gonzalez Alonso, and Andy D. Pimentel. On the Effectiveness of Communication-Centric Modelling of Complex Embedded Systems. In *International Conference on Big Data and Cloud Computing (BDCloud)*, pages 979–986. IEEE, 2018.

[70] Ahmad Mizan and Greg Franks. An Automatic Trace Based Performance Evaluation Model Building for Parallel Distributed Systems. In *International Conference on Performance Engineering*, page 61–72. ACM, 2011.

[71] Mahtab Modaber, Martijn Hendriks, Marc Geilen, Twan Basten, and Jeroen Voeten. A Method for Building Trustworthy Hybrid Performance Models for Cyber-Physical Systems of Systems. *IEEE Access*, 12:92733–92752, 2024.

[72] Steven S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[73] Nina Mühleis, Michael Glaß, Liyuan Zhang, and Jürgen Teich. A Co-Simulation Approach for Control Performance Analysis during Design Space Exploration of Cyber-Physical Systems. *ACM SIGBED Review*, 8(2):23–26, 2011.

[74] Sanjai Narain, Emily Mak, Todd Huster, Jeremy Cohen, Kishore Pochiraju, Brendan Englot, and Niraf Jha. Design Space Exploration for Cyber Physical Systems. Technical report, Perspecta Labs, Basking Ridge, NJ, USA, 2019.

[75] Kai Neubauer, Philipp Wanko, Torsten Schaub, and Christian Haubelt. Exact Multi-Objective Design Space Exploration using ASPmT. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 257–260. IEEE, 2018.

[76] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Multi-Processor System Design with ESPAM. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 211–216, 2006.

[77] Pierluigi Nuzzo, Alberto L. Sangiovanni-Vincentelli, and Richard M. Murray. Methodology and Tools for Next Generation Cyber-Physical Systems: The iCyPhy Approach. In *INCOSE International Symposium*, volume 25, pages 235–249. Wiley, 2015.

[78] Andy D Pimentel. Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration. *IEEE Design & Test*, 34(1):77–90, 2016.

[79] Andy D. Pimentel. Methodologies for Design Space Exploration. In *Handbook of Computer Architecture*, pages 1–31. Springer Nature Singapore, 2022.

[80] Andy D. Pimentel, Simon Polstra, Frank Terpstra, A. W. van Halderen, Joseph E. Coffland, and Louis O. Hertzberger. Towards Efficient Design Space Exploration of Heterogeneous Embedded Media Systems. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, volume 2268 of *Lecture Notes in Computer Science*, pages 57–73. Springer Berlin Heidelberg, 2002.

[81] Riccardo Pinciroli, Connie U. Smith, and Catia Trubiani. Modeling More Software Performance Antipatterns in Cyber-Physical Systems. *Software and Systems Modeling*, 23(4):1003–1023, 2024.

[82] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[83] William Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *ACM/IEEE Conference on Supercomputing*, pages 4–13, 1991.

[84] Víctor Reyes, Tomás Bautista, Gustavo Marrero, Pedro P Carballo, and Wido Kruijtzer. CASSE: a System-level Modeling and Design-space Exploration Tool for Multiprocessor Systems-on-chip. In *Euromicro Symposium on Digital System Design (DSD)*, pages 476–483, January 2004.

[85] Valentina Richthammer, Fabian Fassnacht, and Michael Glaß. Search-Space Decomposition for System-Level Design Space Exploration of Embedded Systems. *Transactions on Design Automation of Electronic Systems (TODAES)*, 25(2):1–32, 2020.

[86] 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.

[87] 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.

[88] Faezeh Sadat Saadatmand, Todor Stefanov, Andy D. Pimentel, Benny Akesson, and Ignacio González Alonso. Unraveling Parallelism in Automated Workload Modeling for Distributed Cyber-Physical Systems. In *Euromicro Conference on Digital System Design (DSD)*, pages 24–33. IEEE, September 2025.

[89] Alberto Sangiovanni-Vincentelli. Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design. *Proceedings of the IEEE*, 95(3):467–506, 2007.

[90] Donghwan Shin, Domenico Bianculli, and Lionel Briand. PRINS: Scalable Model Inference for Component-based System Logs. *Empirical Software Engineering*, 27(4):87, 2022.

[91] Zhaogang Shu, Jiafu Wan, Daqiang Zhang, and Di Li. Cloud-Integrated Cyber-Physical Systems for Complex Industrial Applications. *Mobile Networks and Applications*, 21(5):865–878, 2016.

[92] Todor Stefanov, Andy Pimentel, and Hristo Nikolov. DAEDALUS: System-Level Design Methodology for Streaming Multiprocessor Embedded Systems on Chips. In *Handbook of Hardware/Software Codesign*, pages 1–36. Springer Netherlands, 2017.

[93] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Eclipse Series. Addison-Wesley, 2nd edition, 2008.

[94] Weiqi Sun and Wenbin Dai. An Automatic Software Behavior Model Generation Method for Industrial Cyber-Physical System. In *International Conference on Industrial Informatics (INDIN)*, pages 897–902. IEEE, 2020.

[95] Janos Sztipanovits, Ted Bapty, Sandeep Neema, Larry Howard, and Ethan Jackson. OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber-Physical Systems. In Saddek Bensalem, Yassine Lakhnech, and Axel Legay, editors, *From Programs to Systems: The Systems Perspective in Computing*, volume 8415 of *Lecture Notes in Computer Science*, pages 235–248. Springer, 2014.

[96] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. John Wiley & Sons, 2009.

[97] Giacomo Tanganelli, Luca Cassano, Antonio Miele, and Carlo Vallati. A Methodology for the Design and Deployment of Distributed Cyber-Physical Systems for Smart Environments. *Future Generation Computer Systems*, 109:420–430, 2020.

[98] Prerit Terway, Kenza Hamidouche, and Niraj K. Jha. DISPATCH: Design Space Exploration of Cyber-Physical Systems. *arXiv preprint arXiv:2009.10214*, 2020.

[99] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Application of Concurrency to System Design (ACSD)*, pages 29–40, 2007.

[100] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time Calculus for Scheduling Hard Real-time Systems. In *International Symposium on Circuits And Systems (ISCAS)*, volume 4, pages 101–104. IEEE, 2000.

[101] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere. A Framework for Rapid System-Level Exploration, Synthesis, and Programming of Multimedia MP-SoCs. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 9–14, 2007.

[102] Mark Thompson and Andy D. Pimentel. Exploiting Domain Knowledge in System-Level MPSoC Design Space Exploration. *Journal of Systems Architecture*, 59(7):351–360, 2013.

[103] Casper Thule, Kenneth Lausdahl, Cláudio Gomes, Gerd Meisl, and Peter Gorm Larsen. Maestro: The INTO-CPS co-simulation framework. *Simulation Modelling Practice and Theory*, 92:45–61, 2019.

[104] Trace Compass Developers. Trace Compass, 2024. [Software]. Available at: https://tracecompass.org/.

[105] Konstantinos Triantafyllidis, Yuri Blankenstein, Sobhan Niknam, and Jos Hegge. Platform Performance Suite (PPS): A Framework for Performance Analysis & Diagnosis of Complex Cyber-physical Systems. In *International Conference on Performance Engineering (ICPE)*, pages 168–180, May 2025.

[106] Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings. MCDS: AI Augmented Workflow Scheduling in Mobile Edge Cloud Computing Systems. *Transactions on Parallel and Distributed Systems (TPDS)*, 33(11):2794–2807, 2021.

[107] Bram van der Sanden, Yonghui Li, Joris van den Aker, Benny Akesson, Tjerk Bijlsma, Martijn Hendriks, Kostas Triantafyllidis, Jacques Verriet, Jeroen Voeten, and Twan Basten. Model-Driven System-Performance Engineering for Cyber-Physical Systems. In *International Conference on Embedded Software (EMSOFT)*, pages 11–22. ACM, 2021.

[108] Bram van Hoof, Arjan Holscher, Ralf Gommers, Jeroen Cottaar, Marcel Raas, Samah Khalek, Jan van Kemenade, Maarten Voncken, Roelof de Graaf, Elliot Oti, Stefan Weichselbaum, Richard Droste, Byeong Soo Lee, Chansam Chang, Young Seog Kang, Young Ha Kim, Jeong-Heung Kong, Jong Hoon Jang, Young Sun Nam, and Hyunwoo Hwang. Layout Independent Leveling (LIL) on NXT: 1980Di Immersion Scanners for Enhanced Productivity. In *Optical Microlithography XXX*, volume 10147, pages 374–381. SPIE, 2017.

[109] Peter Van Stralen and Andy D. Pimentel. Scenario-based Design Space Exploration of MPSoCs. In *International Conference on Computer Design (ICCD)*, pages 305–312, October 2010.

[110] Yon Vanommeslaeghe, Joachim Denil, Jasper De Viaene, David Ceulemans, Stijn Derammelaere, and Paul De Meulenaere. Leveraging Domain Knowledge for the Efficient Design-Space Exploration of Advanced Cyber-Physical Systems. In *Euromicro Conference on Digital System Design (DSD)*, pages 351–358. IEEE, 2019.

[111] Yon Vanommeslaeghe, Joachim Denil, Jasper De Viaene, David Ceulemans, Stijn Derammelaere, and Paul De Meulenaere. Ontological Reasoning in the Design Space Exploration of Advanced Cyber-Physical Systems. *Microprocessors and Microsystems*, 85:104151, 2021.

[112] Yon Vanommeslaeghe, Joachim Denil, Bert Van Acker, and Paul De Meulenaere. Automatic Generation of Workflows for Efficient Design Space Exploration for Cyber-Physical Systems. In *Internet of Things (iThings) and Green Computing and Communications (GreenCom) and Cyber, Physical and Social Computing (CPSCom) and SmartData and Cybermatics*, pages 346–351. IEEE, 2021.

[113] András Varga and Rudolf Hornig. An Overview of the OMNeT++ Simulation Environment. In *International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*. ICST, 2008.

[114] Jiang Wan, Arquimedes Canedo, and Mohammad Abdullah Al Faruque. Functional Model-Based Design Methodology for Automotive Cyber-Physical Systems. *IEEE Systems Journal*, 11(4):2028–2039, 2017.

[115] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System Architecture Evaluation Using Modular Performance Analysis: A Case Study. *Software Tools for Technology Transfer (STTT)*, 8(6):649–667, November 2006.

[116] Vincent M Weaver. Self-monitoring Overhead of the Linux perf_event Performance Counter Interface. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 102–111, 2015.

[117] Raphael Weber, Stefan Henkler, and Achim Rettberg. Multi-Objective Design Space Exploration for Cyber-Physical Systems satisfying hard Real-Time and Reliability Constraints. In *Proceedings of IDEAL'14 Workshop*. Springer, April 2014.

[118] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-Time Problem–overview of Methods and Survey of Tools. *Transactions on Embedded Computing Systems (TECS)*, 7(3):36:1–36:53, 2008.

[119] David Wilner. WindView: A Tool for Understanding Real-Time Embedded Software Through System Visualization. *SIGPLAN Not.*, 30(11):117–123, 1995.

[120] Yifeng Xiao, Chanwook Oh, Michele Lora, and Pierluigi Nuzzo. Efficient Exploration of Cyber-Physical System Architectures Using Contracts and Subgraph Isomorphism. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

English Summary

The increasing complexity and performance demands of industrial Distributed Cyber-Physical Systems (dCPS), such as lithography machines in semiconductor manufacturing, necessitate early-stage design validation techniques that are both efficient and accurate. Design Space Exploration (DSE) is essential in this context, as it enables the evaluation of alternative hardware and software configurations before deployment. However, existing DSE tools and methodologies fall short for industrial dCPS: they lack a global system model, rely heavily on manual and impractical model construction, and often lack scalable simulation and optimization support that can efficiently explore large system-level design spaces.

To overcome these limitations, this thesis introduces 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 automated tools for model derivation, model transformation, and simulation-ready code generation, resulting in a coherent workflow for system performance evaluation. CompDSE is particularly suited to existing complex industrial dCPS that are being explored for next-generation upgrades.

CompDSE is structured around three abstract models following the Y-chart approach: the application workload model, the hardware platform model, and the mapping model. The application workload model captures the behavior and interactions of distributed software processes; the hardware platform model describes the available computing infrastructure, including processing components and interconnection networks; and the mapping model specifies how software processes are assigned to hardware resources. In addition, CompDSE introduces the 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. All these models are defined at a high level of abstraction to enable efficient DSE while preserving sufficient accuracy for meaningful performance evaluation.

The CompDSE toolbox includes a set of algorithms that automatically derive the abstract models from execution traces collected from real-world industrial systems. This trace-based approach allows the modeling framework to generate a dedicated model for each operation mode of the system, accurately reflecting the distinct behaviors

and interactions of processes in that mode. The toolbox further includes model-transformation algorithms that convert these abstract models into simulation-ready entities for execution in a discrete-event simulation environment, as well as code-generation algorithms that produce the required simulation code. Together, these tools enable efficient and systematic performance analysis of each design point.

The effectiveness of CompDSE is demonstrated through an evaluation on ASML TWINSCAN lithography machine as a representative industrial dCPS case study. The results show that the models automatically derived by CompDSE closely reproduce the observed behavior of the real system. In particular, the simulated average wafer processing time differs by 2.7% from the actual system measurements. Despite the complexity and scale of the resulting models, each design point can be simulated in under one minute. These findings demonstrate that CompDSE achieves sufficient accuracy for meaningful DSE while maintaining the level of simulation efficiency required for exploring the large design spaces typical for industrial dCPS.

To further enhance concurrency analysis, the thesis introduces a refined application workload model capable of capturing both inter-process and intra-process execution dependencies. This refined model is automatically derived from execution traces and supports a method for identifying tasks within software processes that can be safely reordered or executed in parallel without modifying the existing software structure. The refinement enables deeper performance analysis and reveals parallelism opportunities that are crucial for improving the performance of software workloads on modern multi-core CPUs within the computing subsystems. Experimental validation confirms that simulations based on the refined model expose measurable performance improvements.

Nederlandse Samenvatting

De toenemende complexiteit en prestatie-eisen van industriële gedistribueerde cyber-fysieke systemen (dCPS), zoals lithografiemachines in de halfgeleiderproductie, vereisen ontwerpvalidatietechnieken in een vroeg stadium die zowel efficiënt als nauwkeurig zijn. Design Space Exploration (DSE) is in deze context essentieel, omdat hiermee alternatieve hardware- en softwareconfiguraties kunnen worden geëvalueerd vóór de daadwerkelijke implementatie. Bestaande DSE-tools en -methodologieën schieten echter tekort voor industriële dCPS: ze missen een globaal systeemmodel, zijn sterk afhankelijk van handmatige en in de praktijk onhaalbare modelconstructie, en bieden vaak onvoldoende schaalbare ondersteuning voor simulatie en optimalisatie om grote ontwerpruimten op systeemniveau efficiënt te verkennen.

Om deze beperkingen te overwinnen, introduceert dit proefschrift CompDSE, een gestructureerde methodologie voor het uitvoeren van DSE op systeemniveau, toegespitst op de cybercomponenten van industriële dCPS. De methodologie integreert efficiënte systeemmodellering met geautomatiseerde tools voor modelafleiding, modeltransformatie en het genereren van simulatieklare code, wat resulteert in een samenhangende workflow voor de evaluatie van systeemprestaties. CompDSE is met name geschikt voor bestaande complexe industriële dCPS die worden onderzocht met het oog op upgrades naar een volgende generatie.

CompDSE is opgebouwd rond drie abstracte modellen volgens de Y-chart-benadering: het applicatiewerklastmodel, het hardwareplatformmodel en het mappingmodel. Het applicatiewerklastmodel beschrijft het gedrag en de interacties van gedistribueerde softwareprocessen; het hardwareplatformmodel beschrijft de beschikbare rekeninfrastructuur, waaronder verwerkingscomponenten en interconnectienetwerken; en het mappingmodel specificeert hoe softwareprocessen aan hardwarebronnen worden toegewezen. Daarnaast introduceert CompDSE het omgevingsinvloedmodel, dat de impact op de timing vastlegt van niet-getraceerde embedded hosts die interacteren met de fysieke omgeving, bijvoorbeeld via sensoren en actuatoren, maar die niet expliciet kunnen worden gemodelleerd. Al deze modellen zijn gedefinieerd op een hoog abstractieniveau om efficiënte DSE mogelijk te maken, terwijl voldoende nauwkeurigheid behouden blijft voor betekenisvolle prestatie-evaluatie.

De CompDSE-toolbox omvat een reeks algoritmen die automatisch de abstracte

modellen afleiden uit executietraces die zijn verzameld uit industriële systemen in de praktijk. Deze op traces gebaseerde aanpak stelt het modelleringsraamwerk in staat om voor elke bedrijfsmodus van het systeem een specifiek model te genereren, waarin het onderscheidende gedrag en de interacties van processen in die modus nauwkeurig worden weerspiegeld. De toolbox bevat daarnaast modeltransformatiealgoritmen die deze abstracte modellen omzetten in simulatieklare entiteiten voor uitvoering in een discrete-event-simulatieomgeving, evenals codegeneratiealgoritmen die de vereiste simulatiecode produceren. Samen maken deze tools een efficiënte en systematische prestatieanalyse van elk ontwerppunt mogelijk.

De effectiviteit van CompDSE wordt aangetoond door middel van een evaluatie op een ASML TWINSCAN-lithografiemachine als representatieve industriële dCPS-casestudy. De resultaten laten zien dat de modellen die automatisch door CompDSE worden afgeleid, het waargenomen gedrag van het werkelijke systeem nauwkeurig reproduceren. In het bijzonder wijkt de gesimuleerde gemiddelde waferverwerkingstijd 2,7% af van de daadwerkelijke systeemmetingen. Ondanks de complexiteit en schaal van de resulterende modellen kan elk ontwerppunt in minder dan één minuut worden gesimuleerd. Deze bevindingen tonen aan dat CompDSE voldoende nauwkeurigheid bereikt voor betekenisvolle DSE, terwijl de simulatie-efficiëntie behouden blijft die nodig is voor het verkennen van de grote ontwerpruimten die kenmerkend zijn voor industriële dCPS.

Om de analyse van gelijktijdige uitvoering verder te verbeteren, introduceert dit proefschrift een verfijnd applicatiewerklastmodel dat zowel afhankelijkheden tussen processen als afhankelijkheden binnen processen kan vastleggen. Dit verfijnde model wordt automatisch afgeleid uit executietraces en ondersteunt een methode voor het identificeren van taken binnen softwareprocessen die veilig kunnen worden herschikt of parallel kunnen worden uitgevoerd zonder de bestaande softwarestructuur te wijzigen. De verfijning maakt een diepgaandere prestatieanalyse mogelijk en brengt mogelijkheden voor parallellisme aan het licht die cruciaal zijn voor het verbeteren van de prestaties van softwarewerklasten op moderne multicore-CPU's binnen de rekensubsystemen. Experimentele validatie bevestigt dat simulaties op basis van het verfijnde model meetbare prestatieverbeteringen zichtbaar maken.

List of Publications

Journal Article

- **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.

Peer-Reviewed Conference Proceedings

- Marius Herget, **Faezeh Sadat Saadatmand**, Martin Bor, Ignacio González Alonso, Todor Stefanov, Benny Akesson, and Andy D. Pimentel. Design Space Exploration for Distributed Cyber-Physical Systems: State-of-the-art, Challenges, and Directions. In *Euromicro Conference on Digital System Design (DSD)*, pages 632–640. IEEE, 2022.

- **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*.

- **Faezeh Sadat Saadatmand**, Todor Stefanov, Andy D. Pimentel, Benny Akesson, and Ignacio González Alonso. Unraveling Parallelism in Automated Workload Modeling for Distributed Cyber-Physical Systems. In *Euromicro Conference on Digital System Design (DSD)*, pages 24-33. IEEE, September 2025.

Curriculum Vitae

Faezeh Sadat Saadatmand was born on 3 October 1992 in Tehran, Iran. She completed her pre-university education in Physics and Mathematics in Tehran. In 2011, she began her B.Sc. studies in Computer Engineering (Hardware) at Amirkabir University of Technology and graduated in 2015. She was then admitted to the M.Sc. program in Computer Engineering (Computer System Architecture) at Sharif University of Technology on the basis of outstanding academic performance, completing the degree in 2017. Afterward, she worked for three years in a technology startup, where she designed and developed user-interface programs that interacted with embedded systems. In March 2021, she joined Leiden University to pursue her Ph.D. in Computer Science within the Leiden Institute of Advanced Computer Science (LIACS), under the supervision of Dr. Todor Stefanov. Her doctoral research is conducted in collaboration with TNO-ESI, the University of Amsterdam, and ASML B.V., and is funded by NWO. During her Ph.D. studies, she followed a course in scientific conduct as part of her doctoral training at Leiden University. Alongside her research activities, she has served as a teaching assistant for multiple undergraduate and graduate courses between 2016 and 2025.

Acknowledgments

As I reach the last chapter of my PhD journey, I look back on the path that brought me here, which was shaped by curiosity, hard work, and many moments that taught me more than I expected. These years have had their ups and downs, but they have also been deeply rewarding and full of personal growth. I know I would not have made it this far without the support and kindness of many wonderful people, and I am truly grateful to each of them.

First of all, I would like to express my sincere gratitude to my supervisor, **Todor Stefanov**, for giving me the opportunity to join his group and for his guidance throughout my PhD. I am particularly grateful for his support and for teaching me how to structure my ideas more effectively and improve the clarity of my scientific writing.

I would also like to thank my second promotor, **Nele Mentens**, for the time she dedicated to reading my thesis and for her valuable feedback and insightful comments, which helped improve the quality and clarity of this work. I am also grateful to **Marcello Bonsangue**, Scientific Director of LIACS, for his patience, kindness, and constructive support, especially during the final stage of my PhD.

It was a pleasure to carry out my PhD within the DSE2.0 project, which allowed me to collaborate with and learn from people across different organizations. I would like to express my sincere gratitude to **Andy Pimentel**, whose continuous encouragement helped keep me moving forward even during periods when our progress was slower or less clear than we had hoped. I am also thankful to **Benny Akesson**, whose resilient and optimistic mindset was inspiring and always reminded us that we could overcome any challenge. I would also like to thank **Ignacio Alonso**. Beyond the technical insights, working with and talking to him always gave me a sense of warmth and belonging, like being part of a family.

I would also like to thank our other colleagues at the University of Amsterdam, TNO-ESI, and ASML, including **Martin Bor**, **Marius Herget**, **Uraz Odyurt**, **Alok Lele**, **Ivo ter Horst**, **Rolf Theunissen**, **Jos Vaassen**, and **Yuri Blankenstein**. Many thanks as well to **Kostas Triantafyllidis**, both for his help at TNO-ESI and later at ASML. Kostas, your technical support was invaluable in helping me find my way, especially at the beginning of my PhD, and I sincerely appreciate your continued

support and efforts during the final stage of this journey.

Thanks go to my colleagues in the LERC Lab: **Svetlana**, **Xiaotian**, **Abolfazl**, **Roozbeh**, **Sumiran**, and **Fatemeh**. Our interactions and discussions, both technical and informal, were always engaging and pleasant. A special thanks goes to Fatemeh and Roozbeh, who have been valued friends and kindly agreed to be my paranymphs. I would also like to thank my other friends at LIACS, **Shima**, **Mahnaz**, **Samira**, **Jafar**, **Amin**, and **Arian**, whose friendly presence and shared moments were both enjoyable and memorable.

I am also grateful to my colleagues at LIACS, especially **Abdel** and **Marloes**, who were always welcoming, supportive, and willing to help. Their positive attitude and readiness to assist, even during the pandemic, were greatly appreciated.

My heartfelt thanks go to my first friends in the Netherlands, **Robert** and **Joyce**, who welcomed me and my husband, Mahdi, from the moment we arrived at Schiphol and supported us through the challenges of daily life during the pandemic and beyond. Robert and Joyce, I truly believe you are angels who make the world a better place.

I would like to express my thanks to my family, especially my mother, **Nahid**, whose unwavering support and care have always been a source of strength for me, and my father, **Sadegh**, whom I sadly lost during the final stage of my PhD. He always believed in me, encouraged me to pursue the engineering sciences as he did, and taught me how to face challenges with confidence. Mama and Baba, I am truly grateful and proud to be your child.

Last but not least, my deepest gratitude goes to my dear **Mahdi**, who has been by my side throughout this entire journey. Through moments of joy, exhaustion, uncertainty, and sadness, his support, understanding, and encouragement were always there, and his presence gave me strength during the most challenging times. Mahdi, I could not have reached this point without you, and I am grateful and happy to share this journey and my life with you.