



Universiteit
Leiden
The Netherlands

Towards design space exploration of industrial distributed cyber-physical systems

Saadatmand, F.S.

Citation

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

Version: Publisher's Version

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

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

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

Chapter 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 TWINSKAN 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 meta-heuristic 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;

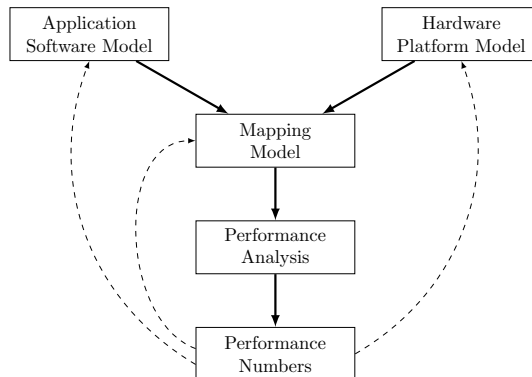


Figure 2.1: Illustration of the Y-chart modeling approach

- **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

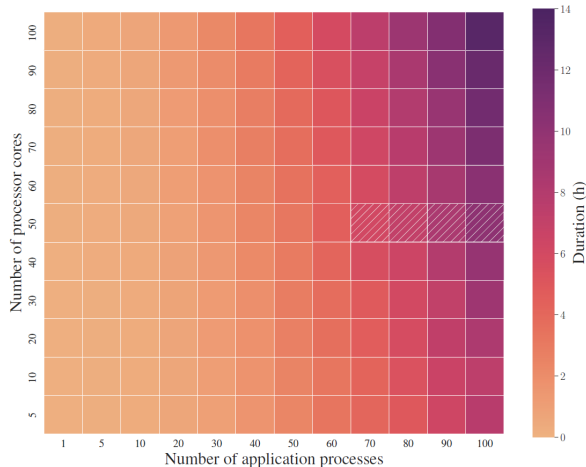


Figure 2.2: Heatmap of the DSE runtime (in hours) by varying the number of application processes and processing components

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 TWINSKAN 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 *TWINSKAN* 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 *TWINSKAN* 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 TWINSKAN 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++.