



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 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

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

4.4.1 Application-level Execution Traces

Each set $US_{h_i} = \{E_1, \dots, E_m\}$ 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)

<i>ts</i>	<i>pn</i>	<i>fn</i>	<i>l</i>	<i>c</i>	<i>id</i>	<i>λ</i>	<i>θ</i>
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

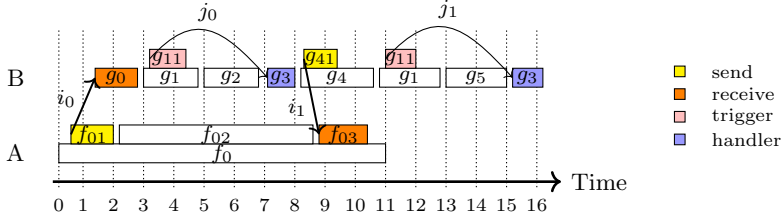


Figure 4.2: Visualization of the complete function executions, highlighting traced functions from Figure 4.1

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

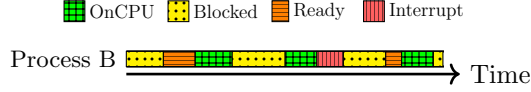


Figure 4.3: Process status over time

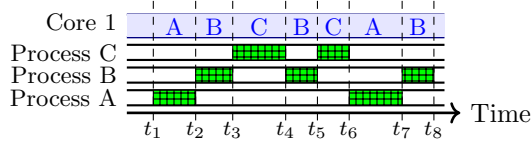


Figure 4.4: Deriving OnCPU timing by using context switch events

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

```

1  $P, C \leftarrow \emptyset; W_{tmp} \leftarrow \emptyset;$ 
2 foreach  $US_{h_i} \in US$  do
3    $W_{h_i} = WLM D(US_{h_i}, KS_{h_i});$ 
4    $W_{tmp} \leftarrow W_{tmp} + W_{h_i};$ 
5 foreach  $W_{h_i} \in W_{tmp}$  do
6   foreach  $ch_j \in W_{h_i}.C$  do
7      $h_{src} = ch_j.src.h; h_{des} = ch_j.des.h;$ 
8     if  $h_{src} \neq h_i$  then
9        $ch_j.src.p = \text{findPeerProc}(W_{h_{src}}, ch_j, S);$ 
10    if  $h_{des} \neq h_i$  then
11       $ch_j.des.p = \text{findPeerProc}(W_{h_{dst}}, ch_j, D);$ 
12 foreach  $W_{h_i} \in W_{tmp}$  do
13    $P \leftarrow P + W_{h_i}.P;$ 
14    $C \leftarrow C + W_{h_i}.C;$ 
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 : $US_h = \{E_1, \dots, E_m\}$, KS_h
Output : $W_h = (P, C)$

```

1   $P, C \leftarrow \emptyset;$ 
2  foreach  $E_i \in US_h$  do
3      foreach  $r_k \in E_i$  do
4          if  $\nexists p_{r_k.pn} \in P$  then
5               $p_{r_k.pn} \leftarrow \emptyset;$    $P \leftarrow P + p_{r_k.pn};$ 
6          if  $r_k.A.c = receive$  then
7               $r_j = \text{findREC}(E_i, send, start, r_k.A.id);$ 
8              if  $\nexists ch_{(r_j.pn, r_k.pn)} \in C$  then
9                   $ch_{(r_j.pn, r_k.pn)} = (p_{r_j.pn}, p_{r_k.pn});$ 
10                  $C \leftarrow C + ch_{(r_j.pn, r_k.pn)};$ 
11
12  foreach  $p_x \in P$  do
13       $H_x^i \leftarrow \emptyset;$   /* set of tuples  $(ev, T_s, T_e)$  */
14       $H_x^i \leftarrow H_x^i + (\emptyset, 0, 0);$ 
15
16  for  $k = 1$  to  $|E_i|$  do
17      if  $r_k.A.c = send \wedge r_k.l = start$  then
18           $r_{k'} = \text{findREC}(E_i, send, end, r_k.A.id);$ 
19           $sig = \text{getSIG}(KS_h^{E_i}, p_x.pn, r_k.ts, r_{k'}.ts);$ 
20           $r_j = \text{findREC}(E_i, receive, start, r_k.A.id);$ 
21           $e_k^W = (ch_{(r_k.pn, r_j.pn)}, r_k.A.\lambda, sig);$ 
22           $H_{r_k.pn}^i \leftarrow H_{r_k.pn}^i + (e_k^W, r_k.ts, r_{k'}.ts);$ 
23
24      if  $r_k.A.c = receive \wedge r_k.l = start$  then
25           $r_{k'} = \text{findREC}(E_i, receive, end, r_k.A.id);$ 
26           $sig = \text{getSIG}(KS_h^{E_i}, p_x.pn, r_k.ts, r_{k'}.ts);$ 
27           $r_j = \text{findREC}(E_i, send, start, r_k.A.id);$ 
28           $e_k^R = (ch_{(r_j.pn, r_k.pn)}, r_k.A.\lambda, sig);$ 
29           $H_{r_k.pn}^i \leftarrow H_{r_k.pn}^i + (e_k^R, r_k.ts, r_{k'}.ts);$ 
30
31      if  $r_k.A.c = trigger \wedge r_k.l = start$  then
32           $e_k^{TS} = (r_k.A.id, r_k.A.\theta);$ 
33           $H_{r_k.pn}^i \leftarrow H_{r_k.pn}^i + (e_k^{TS}, r_k.ts, r_k.ts);$ 
34
35      if  $r_k.A.c = handler \wedge r_k.l = start$  then
36           $e_k^{TH} = (r_k.A.id, r_k.A.\theta);$ 
37           $H_{r_k.pn}^i \leftarrow H_{r_k.pn}^i + (e_k^{TH}, r_k.ts, r_k.ts);$ 
38
39  foreach  $p_x \in P$  do
40      for  $i = 1$  to  $m$  do
41           $M_i \leftarrow \emptyset;$ 
42          if  $\exists H_x^i$  then
43               $t_s = r_k.ts : r_k \in E_i \wedge k = |E_i|;$ 
44               $H_x^i = H_x^i + (\emptyset, t_s, t_s);$ 
45              for  $y = 1$  to  $|H_x^i| - 1$  do
46                  if  $H_x^i[y].ev \neq \emptyset$  then
47                       $M_i \leftarrow M_i + H_x^i[y].ev;$ 
48                       $t_s = H_x^i[y].T_e;$    $t_e = H_x^i[y+1].T_s;$ 
49                       $sig = \text{getSIG}(KS_h^{E_i}, p_x.pn, t_s, t_e);$ 
50                       $e_y^C = (sig);$    $M_i \leftarrow M_i + e_y^C;$ 
51
52           $p_x \leftarrow p_x + M_i;$ 
53
54  return  $(P, C)$ 

```

Algorithm 3: findREC

Input : E, c, l, id
Output : r

```

1 foreach  $r_i \in E$  do
2   if  $r_i.A.c = c \wedge r_i.l = l \wedge r_i.A.id = id$  then
3     return ( $r \leftarrow r_i$ )

```

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

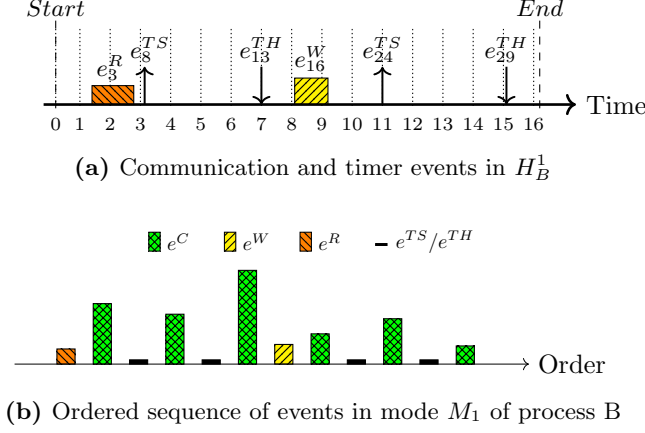


Figure 4.5: Derivation of the workload model of the first mode in process B

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.

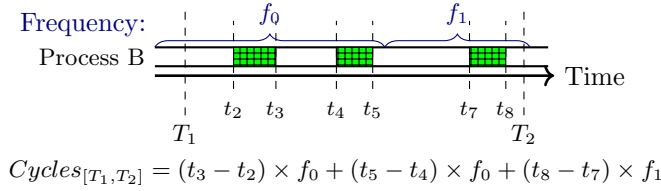
Algorithm 4: getSIG

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  $T_{start} < q.OnCPU \leq T_{end}$  then
4      $cycles \leftarrow cycles + q.OnCPU \times q.freq;$ 
5 return ( $sig = cycles$ )

```

**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.

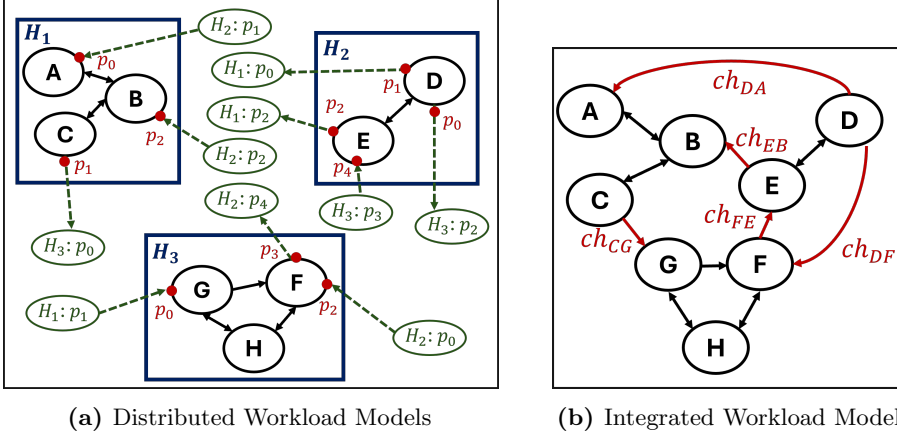


Figure 4.7: The workload model integration

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

```

Algorithm 6: Platform Model Derivation from Traces

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

```

1  $H, N, L \leftarrow \emptyset;$ 
2 foreach  $US_{h_i} \in US$  do
3    $ip_i = \text{getIPs}(US_{h_i});$ 
4    $nCr = \text{number of cores from } KS_{h_i}; crs_i \leftarrow \emptyset;$ 
5   for  $j = 1$  to  $nCr$  do
6      $f_j = \text{related frequency from } KS_{h_i};$ 
7      $cr_j = (f_j); crs_i \leftarrow crs_i + cr_j;$ 
8    $sch_i \leftarrow \emptyset;$ 
9    $h_i = (ip_i, sch_i, crs_i); H \leftarrow H + h_i;$ 
10 foreach  $h_i \in H$  do
11   foreach  $ip_j \in h_i.ip$  do
12      $subn_j = \text{subnetMask}(ip_j);$ 
13     if  $\nexists swch_j \in N \text{ such that } swch_j.subn = subn_j$  then
14        $bw_j = 0; swch_j = (subn_j, bw_j);$ 
15        $N \leftarrow N + swch_j;$ 
16      $l_j = (h_i, swch_j); L \leftarrow L + l_j;$ 
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  $ipSet \leftarrow \emptyset$ ;
2 foreach  $r_k \in US_h.E_1$  do
3   if  $r_k.A.f_n = send$  then
4      $ip_k = r_k.A.id.ip_{src}$ ;
5     if  $ip_k \notin ipSet$  then
6        $ipSet \leftarrow ipSet + ip_k$ ;
7 return  $ipSet$ 

```

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 sbm_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 sbm_j (Line 13). If such a switch does not exist, the algorithm creates a new switch $swch_j$ with sbm_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 : $US = \{US_{h_1}, \dots, US_{h_n}\}$
Output : $MP = \{MP_1, \dots, MP_m\}$

```

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

```

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

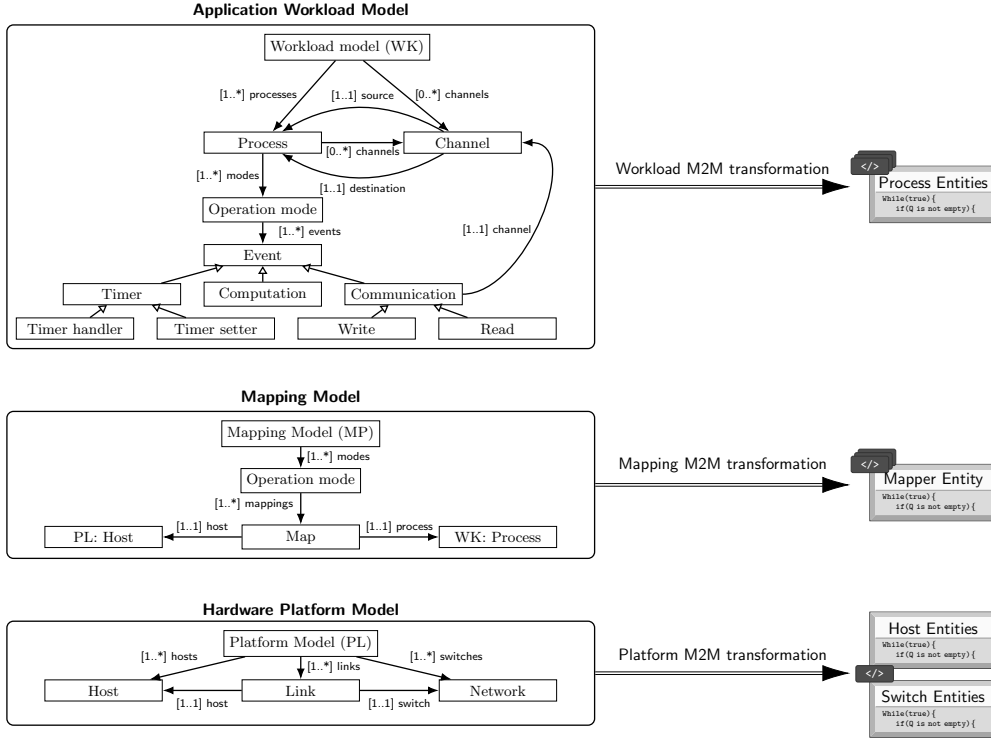


Figure 4.8: Model-to-Model transformation from formal models (left) to executable simulation entities (right)

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.

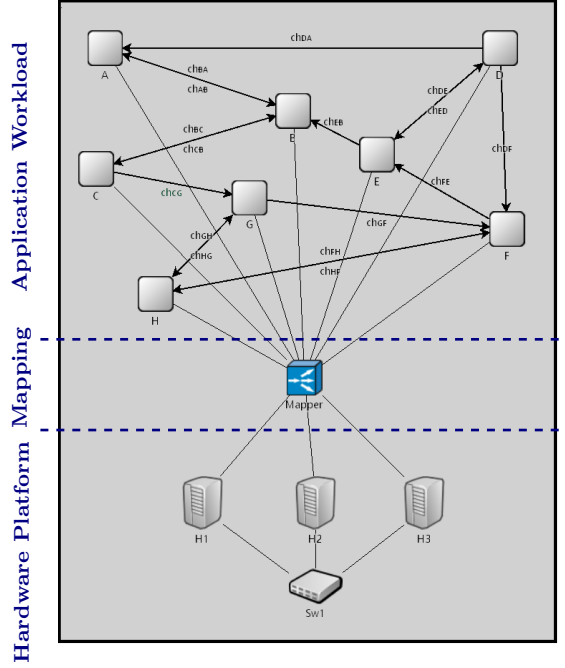
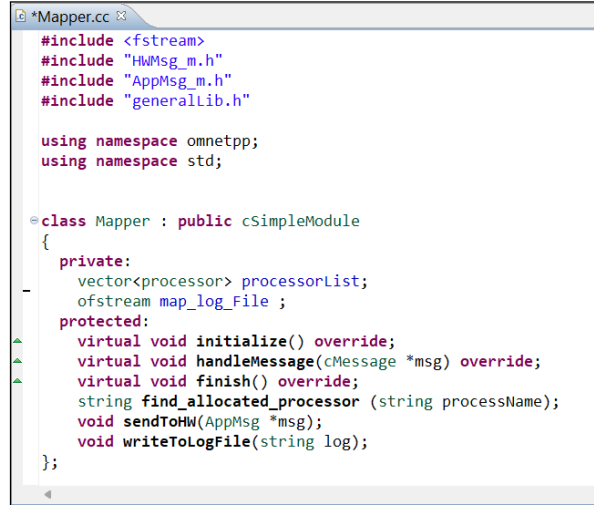


Figure 4.9: The application workload, computing hardware platform, and mapping layers in OMNet++

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.

```
1 <!-- Mapping File -->
2 <mapping>
3   <bind process="A" host="H1"/>
4   <bind process="B" host="H1"/>
5   <bind process="C" host="H1"/>
6   <bind process="D" host="H2"/>
7   <bind process="E" host="H2"/>
8   <bind process="F" host="H3"/>
9   <bind process="G" host="H3"/>
10  <bind process="H" host="H3"/>
11 </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.

