Execution Trace Analysis for a Precise Understanding of Latency Violations

Maysam Zoor[1], Ludovic Apvrille[1*], Renaud Pacalet[1] and Sophie Coudert[1]

[1*]LTCI, Télécom Paris, Institut Polytechnique de Paris, Sophia-Antipolis, France.

*Corresponding author(s). E-mail(s): ludovic.apvrille@telecom-paris.fr;
Contributing authors: maysam.zoor@telecom-paris.fr; renaud.pacalet@telecom-paris.fr;
sophie.coudert@telecom-paris.fr;

Abstract

Despite the amount of proposed works for the verification of embedded systems, understanding the root cause of violations of requirements in simulation or execution traces is still an open-issue, especially when dealing with temporal properties such as latencies. Is the violation due to an unfavorable real-time scheduling, to contentions on buses, to the characteristics of functional algorithms or hardware components? The paper introduces the Precise Latency ANalysis approach (PLAN), a new trace analysis technique whose objective is to classify execution transactions according to their impact on latency. To do so, we rely first on a model transformation that builds up a dependency graph from an allocation model, thus including hardware and software aspects of a system model. Then, from this graph and an execution trace, our analysis can highlight how software or hardware elements contributed to the latency violation. The paper first formalizes the problem before applying our approach to simulation traces of SysML models. A case study defined in the AQUAS European project illustrates the relevance of our approach. Last, a performance evaluation gives computation times for several models and requirements.

Keywords: Embedded Systems, Execution Trace Analysis, Dependency Graph, Model-Based Systems Engineering (MBSE), Timing analysis, Simulation

1 Introduction

The growing complexity of embedded systems makes their analysis challenging. In particular, better understanding how their mechanisms impact each other is a key aspect. Relying on *trace analysis* has been proposed as a promising solution as it provides relevant information about system execution [46]. Traces are collected by simulating a model or running the embedded system in real-time. Trace analysis is a powerful approach to understand and optimize the behaviors of a system [47], to debug it [47] [36], to perform model checking [32], to analyze timings, to detect data races [31] or perform other verifications [32]. References [35], [23], [33], [14], [13], [11] and [34] rely on simulation traces for performance analysis. Most of these approaches focus on verifying temporal properties, on statistical evaluation, on bottleneck analysis and on deadlock/fault detection. Thus, all these contributions focus on whether a property is satisfied or not, but not on the reasons *why* it is not satisfied. Yet, understanding the reasons for a property violation is difficult since a trace is the result of complex interactions between different processes running

on different hardware components, and communicating using communication paths of different nature (shared memories, Direct Memory Access (DMA) transfers, network sockets, ...).

Our contribution, named PLAN, can investigate a simulation or execution trace produced from a system-level model featuring an application, an architecture and the allocation of the application on the architecture. PLAN takes as input a model, a trace, two events of interest (e_1, e_2), and the maximum delay (also called "latency" in this paper) between the occurrences of these two events. PLAN can then automatically check the time delay between events, and can produce a categorization of the different transactions of the trace (obligatory, optional, contention, no contention, etc.) so as to guide designers on how to update their system if the latency property is not satisfied. Possible decisions are to change the application model (e.g., using another algorithm), to modify the system architecture (e.g., replacing a processor by a more efficient one, selecting another scheduling policy), or finally to change the allocations, e.g., allocating a function to a different processor, or using other communication facilities between processors.

In this paper, the main contributions provide a more accurate formalization with regards to the one presented in [54]: formal definitions of assumptions, enhancement of the system formal definitions and more refined discussions. Also, a performance evaluation gives, for four different models, the time it takes to generate a dependency graph and to classify transactions.

Section 2 reviews different execution traces analysis approaches. Then, Section 3 formally defines different inputs of PLAN. Section 4 presents the formal definition of the PLAN categories. The mapping of our formal model to the Systems Modeling Language (SysML) diagrams is presented in Section 5. The implementation of PLAN in the TTool toolkit and the use case taken from H2020 AQUAS project are presented in Section 6. Performance evaluation on four models gives an insight on the usability of our approach. Section 7 discusses the complexity of PLAN and potential future work before the conclusion (Section 8).

2 Related work

Embedded systems must comply with functional and nonfunctional requirements such as system safety, security, performance, reliability, etc. [5] [43]. These requirements can be verified using different approaches throughout a Product Life Cycle (PLC) from design time to runtime. Formal verification approaches use mathematical logic to prove properties [44] [48] [45] while runtime verification approaches detect property violations by monitoring the system during execution [48]. Runtime verification can be applied on traces collected as the system runs (on-line) or afterwards. In the design stage of a PLC, simulation is meant to represent system execution. [50] compares simulation and run time verification. Both rely on obtaining traces and then performing requirement verification on traces. Yet, while simulation is used to enhance the system at design stage, runtime verification is rather used to detect faults in the system during operation and take required actions.

2.1 Simulation traces analysis

Simulation is a very common technique for evaluating and validating designs as simulation traces record the behavior of an application allocated to an architecture and thus provide relevant information about the system execution. Obviously, this requires models, which are approximations of real systems, to be executable [44].

Simulation trace analysis is a technique to discover what happened during simulation [29]. System evaluation and validation using trace analysis is considered useful when engineers can manage and use the trace analysis tools to analyze complex requirements [52]. For instance, [4] describes a trace analysis approach that allows the designer to reason aboutthe model execution at the level of the SysML/UML model. The aim of this approach is to help the designer to explore and understand the model-based analysis results. Tools using simulation trace analysis techniques to analyze and verify time related requirements are discussed next.

The SoC-Trace Project [47] aims to develop an infrastructure to store and analyze traces regardless of their format or size. The objective of

building this infrastructure is to have tools built on top of it that can analyze the stored traces.

Traviando [51] is an example of a software tool used for the analysis of simulation traces. It provides qualitative (e.g., Linear Time Logic (LTL) model checking) and quantitative (e.g., statistical evaluation, bottleneck analysis, deadlock detection) trace analysis [51]. The analysis aims to attract the attention of designers to sections of traces that correspond to potentially abnormal model behaviors. Traces corresponding to these behaviors are highlighted with Message Sequence Chart (MSC) [37].

The RT-Simex [2] project uses a set of code instrumentation tools to analyze and verify timing constraints and locate faults of parallel embedded code [23]. Real time constraints on UML models are specified using MARTE time models and the Clock Constraint Specification Language (CCSL) library [21]. Simulation traces in Open Trace Format (OTF) are studied to check if the specified real time constraints are met. TimeSquare [22] has been used in RT-Simex. TimesSquare relies MARTE model and CCSL for designing systems. TimeSquare analyzes clock constraints and provides feedback during system simulation.

[33] presents SATM (**S**treaming **A**pplication **T**race **M**iner), an approach to help debugging real time applications such as streaming application and understand the violation reasons for quality of service (QoS) properties. SATM takes as input an execution trace and outputs a description of system activity indicating the origin of the temporal bug. To identify the origin of the QoS problem, SATM uses data mining. In [33], the execution trace is based on executing the embedded software on a real hardware —an already manufactured chip— however, the application of pattern mining algorithms on simulation traces is highlighted. The data mining algorithm is used to characterize simulation traces of program executions that corresponded to temporal properties violations.

Chen et al. [14] suggest to analyze simulation traces of systems, including hardware/software models, to check if functional and performance constraints expressed in Logic of Constraints (LoC) [15] are satisfied. A trace checker reports any constraint violation of a simulation trace. Constraints are specified at system level.

One of the verification techniques implemented in Metropolis—a system-level design framework for embedded systems—is based on simulation trace checking [13]. Functional and performance properties can be specified by the designer using LoC, mathematical logics and LTL. Trace analysis tools integrated into the Metropolis simulator automatically check for the specified properties. This verification can be performed off-line or during the simulation [13].

The TRAP tool [52] is a model-based framework that analyzes simulation traces to verify causal and temporal properties of embedded systems. Simulation traces are generated by Virtual Prototypes (VPs) simulators. An error is raised in case a property is violated. A trace file generated by a VP simulator often contains a lot of detailed information about the system. To minimize the trace size, a domain specific language, Simulation Trace Mapping Language (STML), is used to abstract trace data into symbolic information (logical clocks) and remove irrelevant information.

[34] provides a complete design flow (named Koski) to model multiprocessor system-on-chips in a UML profile with automated design space exploration. It uses simulation for functional verification and performance evaluation. Performance evaluation related to statistics obtained at simulation: process execution time, communication latency and communication throughput. From these metrics, alternative architectures can be compared.

2.2 Execution traces analysis

Runtime verification approaches detect property violations by monitoring the system during execution [48]. A runtime verification system with a decision procedure for the property under study is referred to as a *monitor*. Creating a monitor is the first step in the runtime verification process [25]. The monitor takes as input events from the system under analysis. To generate these events the system is instrumented. Thus, the second step in the runtime verification process is *system instrumentation*. Then, the system is executed and the monitor analyzes the generated events to produce a verdict [25]. An overview of a taxonomy of work in runtime verification is described in [26]. It presents seven major high-level concepts used to classify runtime verification approaches and classifies 20 runtime verification tools according to this taxonomy. For instance, a

property may be implicit or explicit. Implicit properties describe correct concurrent behavior and aim at avoiding runtime errors, e.g., absence of deadlocks while explicit properties express functional or nonfunctional requirements. However, detecting the violation of critical safety properties in operation is not acceptable [27]. Thus, runtime analysis must be used firstly for unexpected events while requirements are expected to be verified in an earlier stage of the PLC.

There are many proposals of specification languages for runtime verification [10] [20]. Temporal Stream-based Specification Language (TeSSLa) [17] is an example of a runtime verification language allowing to express timing properties and events along execution traces. Unlike traditional stream-based runtime verification approaches that process events in execution traces without considering timing information, a timestamp is associated to each event of an execution trace [17], thus enforcing event ordering and easing timing analysis between events.

The Copilot language [48] is a runtime verification framework for real-time embedded systems used in combination with NASA core flight system applications. The Copilot language supports a variety of temporal logics that can be used to express re-occurring patterns.

LOLA [19] is a specification language of synchronous systems that allows not only the monitoring of boolean temporal specifications but also of quantitative/statistical properties of the system. It has been successfully used to monitor synchronous, discrete time properties of autonomous aircrafts [9].

To the best of our knowledge, if some of the aforementioned works could detect violations of latency requirements of high-level allocation models, they do not explain *why* they are violated. Most of the analysis tools calculate the Worst-Case Execution Time (WCET) or Best-Case Execution Time (BCET) or latency and throughput values. While having the minimum and maximum latency can be beneficial for the designer, not understanding the cause of the latency and what elements are contributing to its value limits the designer's knowledge on how to enhance the model to further improve performance.

The approach introduced in this paper is based on the conversion of the model semantics into a directed graph and the study of the execution trace along the generated graph as explained in the next section.

3 Overview and problem formalization

This section presents the general approach of PLAN and formalizes input models.

3.1 Precise Latency Analysis Approach

PLAN takes as input a system-level model, a latency requirement, and an execution trace of the model, Figure 1. This trace can be obtained from a model simulation, or from a model-to-code generation and then code execution. Our method follows the Y-Chart approach [39] to partition the system between hardware and software: application and platform are modeled independently before the application is allocated to the platform. PLAN then builds a dependency graph to simplify model analysis, as explained in the next section. The execution trace analysis answers whether the latency requirement is satisfied. If not, then the analysis classifies the transactions along the traces into categories to support the developer in her efforts to determine the cause of the violation.

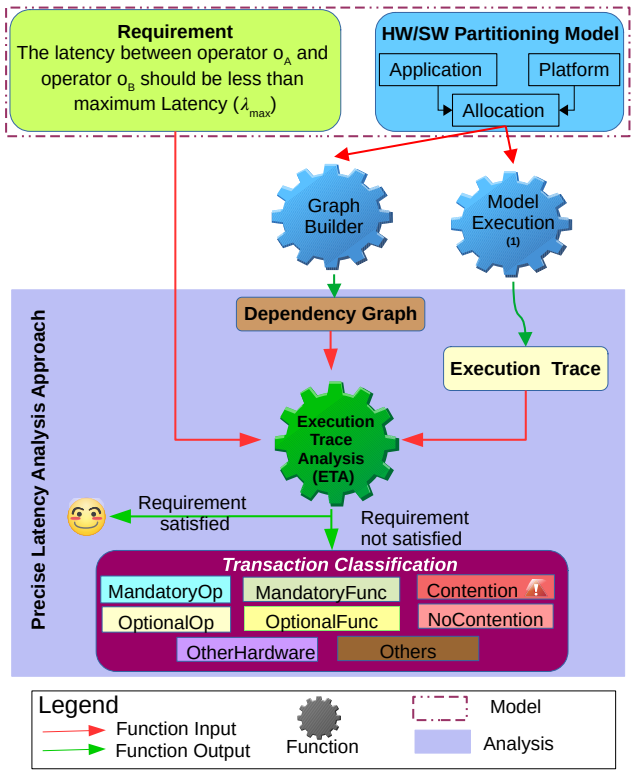
3.2 Formal definition of system models

The following definitions capture the application, the platform and the allocation of the considered system.

Definition 1 HW/SW partitioning model
A HW/SW partitioning model $m = \langle \mathcal{F}_m, \mathcal{P}_m, \mathcal{A}_m \rangle$ is a 3-tuple with $\mathcal{F}_m$ an application model, $\mathcal{P}_m$ a platform model and $\mathcal{A}_m$ an allocation model.

In the sequel we always consider one model at a time, thus indexes or parameters m will be omitted.

Notations: An element of a HW/SW partitioning is denoted x_{id} where x is a letter associated with its type (o for operator, h for hardware component...) and id is an integer that makes the identifier unique. Ordered sequences of elements are denoted $\langle e_i, \ldots, e_j \rangle$ and we sometimes handle



(1) Model execution may require a model transformation first. However this transformation and execution is not part of our contribution

Fig. 1 Overview of PLAN

them as sets, using symbols $\in, \subseteq, \ldots$ By notation abuse, $e_i \in seq$ means that e_i is an element of sequence seq, and $\overline{e_i e_j} \in seq$ means that e_i and e_j are consecutive in sequence seq.

3.2.1 Application

Definition 2 Application model
An application model is a 2-tuple $\mathcal{F} = \langle F, \mathcal{CC} \rangle$ of a set of functions F and a set of communication channels $\mathcal{CC}$.

Figure 2 gives the graphical representation of an application model with 5 functions $(f_1, \ldots, f_5)$ and 3 communication channels (dc_6, dc_7, sc_8).

Definition 3 Communication channel
A communication channel cc_{f_i,f_j} links a writing function f_i to a reading function f_j. $\{\mathcal{DC}, \mathcal{SC}\}$ is a partition of $\mathcal{CC}$. $\mathcal{DC}$ contains data channels and $\mathcal{SC}$ contains synchronization channels.

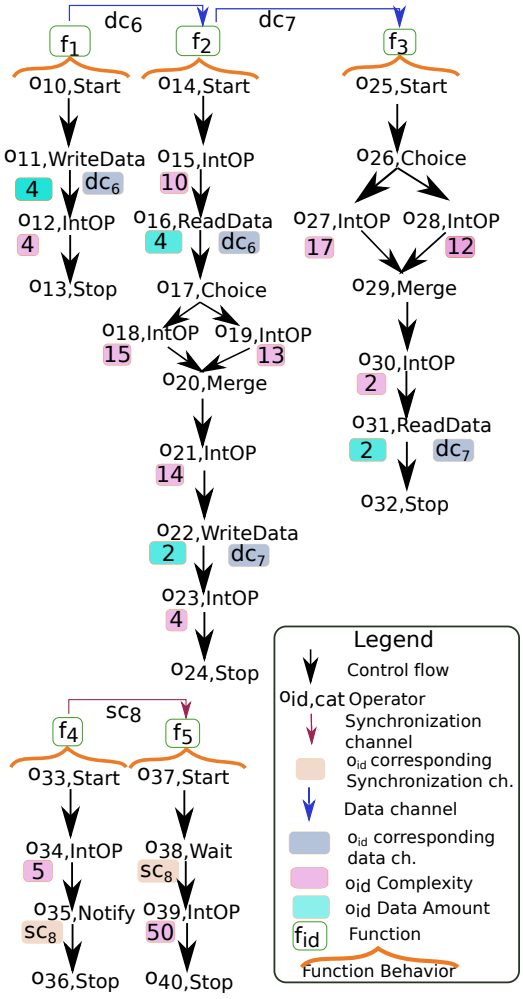


Fig. 2 Graphical representation of an application model

Figure 2 shows two data channels (dc_6, dc_7) and one synchronization channel (sc_8).

Definition 4 Synchronization channel
Following the semantics in [24], a synchronization channel $sc_{f_i,f_j} \in \mathcal{SC}_{f_i,f_j}$ supports the transmission of control messages between two functions.

Definition 5 Data channel
Data channels model the quantity of exchanged data between two functions, not the data values (they are abstracted).

Definition 6 Function
A function is a 2-tuple $f = \langle V_f, B_f \rangle$, where V_f is a set of variables and B_f is a behavior.

Definition 7 Function behavior
A behavior is a 2-tuple $B_f = \langle O_f, C_f \rangle$ where O_f is a set of operators and $C_f \subseteq \{(o_i, o_j) \in O_f^2 \mid o_i \neq o_j\}$ is the set of control flow connections of B_f.

Figure 2 shows the behavior of each function where an operator is represented together with its id and type (see Definition 12). Also, directed arrows between two operators represent control flow connections.

Definition 8 Set $\mathcal{O}$ of All Operators

$$\mathcal{O} = \bigcup_{f \in \mathcal{F}} O_f$$

Definition 9 Dependencies between operators
$\mathcal{D} \subseteq \mathcal{O}^2$ denotes the set of direct dependencies. $\overline{o_i o_j} \in \mathcal{D}$ means that o_j directly depends on o_i.
$\mathcal{D} = synChDep \cup dataChDep \cup controlFlowDep$, where *synChDep* (Definition 13) and *dataChDep* (Definition 14) relate to communication dependencies and *controlFlowDep* (Definition 11) relates to control flow dependencies.

Definition 10 Model dependency path
A dependency path is a finite sequence of operators such that for any pair of consecutive operators o_i and o_j in the sequence, $\overline{o_i o_j}$ is a direct dependency in $\mathcal{D}$. $\overrightarrow{o_i o_j}$ denotes a dependency path having o_i as first operator and o_j as last one. $DP_{\overrightarrow{o_i o_j}}$ denotes the set of all dependency paths between o_i and o_j.

Notations: by notation abuse $\forall \overrightarrow{o_i, o_j}$ and $\exists \overrightarrow{o_i, o_j}$ respectively abbreviate $\forall \overrightarrow{o_i, o_j} \in DP_{\overrightarrow{o_i, o_j}}$ and $\exists \overrightarrow{o_i, o_j} \in DP_{\overrightarrow{o_i, o_j}}$.

Definition 11 Control flow dependency (*controlFlowDep*)
$c = (o_i, o_j) \in C_f$ denotes an oriented control flow connection from o_i to o_j, as defined in Definition 3.2.1. In the application model, we say a control flow dependency $\overline{o_i o_j}$ exists from operator o_i to operator o_j if and only if there exists a control flow connection between them.

$$controlFlowDep = \bigcup_{f \in F} C_f$$

In Figure 2, $\overline{o_{10} o_{11}}$, $\overline{o_{11} o_{12}}$ and $\overline{o_{12} o_{13}}$ are example of control flow dependencies.

Definition 12 Types of operators
Operators belong to one of the following types: Start, Stop, Choice, Merge, IntOp, Set, WriteData, ReadData, Notify, Wait or Loop.

- Start: start of control flow. The Start operator is unique in a function behavior. This operator represents the first operator to be executed by function f and is denoted by St_f.
- Stop: end of control flow.
- Choice: selects one control flow among the next ones whose guard is true. If no guard is true, then the choice operator blocks. Only choice operators can have more than one next operator.
- Merge: merges together several execution flows. The merge operator is the only one that can have several previous operators.
- IntOp: abstracts computations by specifying the complexity of the operation in terms of, e.g., integer operations. Said differently, the computation steps of algorithms are not provided: only an estimation of the amount of corresponding integer operation is used.
- Set: assigns a value to a variable.
- WriteData, ReadData: writes/reads an amount of data to/from a data channel.
- Notify, Wait: sends a message or waits for a message in a synchronization channel.
- Fixed Iteration Loop: iterates a number of times on a set of operators called *insideLoop* operators. The fixed number of iterations is a parameter of this operator. Also, loops created without using the loop operator, that is created by using merge and choice operators are not supported. Said differently, cycles in models can only be created using the loop operator.

If o is a Notify or Wait operator, $getSyncCh(o)$ denotes the corresponding synchronization channel. Similarly, if o is a ReadData or WriteData operator, $getDataCh(o)$ denotes the data channel.

Property 1 Control flow connection constraint
When there is a control flow connection between two operators $(o_i, o_j) \in C_f$ then, $(o_j, o_i) \notin C_f$.

Property 2 Well-formed function For each $o \in O_f$, there must exist at least one control flow dependency path from the start operator St_f to o.

Definition 13 Synchronization channel dependency (*synChDep*)
If there exists a synchronization channel sc between two functions f_i and f_j, then for all operators o_i of f_i and o_j of f_j such that o_i sends messages on sc and o_j receives messages from sc, then $\overline{o_i o_j}$ is a synchronization channel dependency.

$$synChDep = \bigcup_{f_i,f_j \in F} \{(o_i, o_j) \in O_{f_i} \times O_{f_j} \mid type(o_i) = \text{Notify} \wedge type(o_j) = \text{Wait} \wedge getSyncCh(o_i) = getSyncCh(o_j)\}$$

In Figure 2, o_{35} sends one synchronization message on sc_8 and o_{38} receives one synchronization message from $sch8$. Thus, $\overline{o_{35}o_{38}}$ is a synchronization channel dependency.

Definition 14 Data channel dependency (*dataChDep*)
If there exists a data channel dc between two functions f_i and f_j, then for all operators o_i of f_i and o_j of f_j such that o_i writes on dc and o_j reads from dc, $\overline{o_i o_j}$ is a data channel dependency.

$$dataChDep = \bigcup_{f_i,f_j \in F} \{(o_i, o_j) \in O_{f_i} \times O_{f_j} \mid type(o_i) = \text{WriteData} \wedge type(o_j) = \text{ReadData} \wedge getDataCh(o_i) = getDataCh(o_j)\}$$

For instance, in Figure 2, o_{11} writes on dc_6 and o_{16} reads from dc_6. Thus, $\overline{o_{11}o_{16}}$ is a data channel dependency.

3.2.2 Platform

Definition 15 Platform model
A platform model $\mathcal{P} = \langle \mathcal{H}, \mathcal{L} \rangle$ is a set of hardware components $\mathcal{H}$ and a set of links $\mathcal{L}$. A hardware component represents the physical electronic component plus its support software, e.g., a processor and its operating system. We consider three different kinds of hardware components: execution, communication and storage. They respectively belong to subsets $\mathcal{H}_E$, $\mathcal{H}_C$ and $\mathcal{H}_S$. $\{\mathcal{H}_E, \mathcal{H}_C, \mathcal{H}_S\}$ is a partition of $\mathcal{H}$.

Figure 3 depicts three $\mathcal{H}_E$ hardware components h_{41}, h_{42} and h_{43}, one $\mathcal{H}_C$ hardware component h_{44}, one $\mathcal{H}_S$ hardware component h_{45} and links $l_{46} = (h_{41}, h_{44})$, $l_{47} = (h_{42}, h_{44})$, $l_{48} = (h_{43}, h_{44})$, $l_{49} = (h_{44}, h_{45})$.

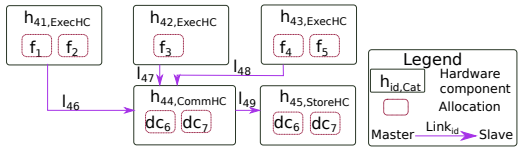


Fig. 3 Graphical representation of an allocation model

Definition 16 Links
A link is a 2-tuple (h_i, h_j) of hardware components, at least one of which is a communication component:

$$\mathcal{L} \subseteq \mathcal{H}_C \times \mathcal{H} \cup \mathcal{H} \times \mathcal{H}_C$$

In Figure 3, $\mathcal{L} = \{(h_{41}, h_{44}), (h_{42}, h_{44}), (h_{43}, h_{44}), (h_{44}, h_{45})\}$.

Definition 17 Communication path
A write path $\pi_w = \langle h_i, \ldots, h_j \rangle$ is a sequence of hardware components linked together, starting with an execution component and ending with a storage component. A read path $\pi_r = \langle h_i, \ldots, h_j \rangle$ is a sequence of hardware components linked together, starting with a storage component and ending with an execution component.

$$\pi_w = \langle h_i, \ldots, h_j \rangle, \forall \overline{h_k h_l} \in \pi_w, (h_k, h_l) \in \mathcal{L}, h_i \in \mathcal{H}_E, h_j \in \mathcal{H}_S, \forall h_{k \neq i,j} \in \pi_w, h_k \in \mathcal{H}_C$$

$$\pi_r = \langle h_i, \ldots, h_j \rangle, \forall \overline{h_k h_l} \in \pi_r, (h_k, h_l) \in \mathcal{L}, h_i \in \mathcal{H}_S, h_j \in \mathcal{H}_E, \forall h_{k \neq i,j} \in \pi_r, h_k \in \mathcal{H}_C$$

A write path π_w and a read path π_r form a communication path $c_P = \langle \pi_w, \pi_r \rangle \in \mathcal{C}_P$ if and only if the last component of π_w is the first component of π_r.

In Figure 3, $\pi_w = \langle h_{41}, h_{44}, h_{45} \rangle$ is a write path and $\pi_r = \langle h_{45}, h_{44}, h_{43} \rangle$ is a read path.

3.2.3 Allocation

Definition 18 Allocation
Functions and their communications must be allocated to hardware components. Functions are allocated to $\mathcal{H}_E$ hardware components while data channels are allocated to communication paths. We assume in our model that synchronization channels do not generate significant traffic and we thus ignore their allocation.

Formally, we define allocations as 2-tuples of total functions $\mathcal{A} = \langle \overrightarrow{\mathcal{A}_f}, \overrightarrow{\mathcal{A}_{dc}} \rangle$ with $\overrightarrow{\mathcal{A}_f} : F \rightarrow \mathcal{H}_E$ and $\overrightarrow{\mathcal{A}_{dc}} : \mathcal{DC} \rightarrow \mathcal{C}_P$.

Figure 3 shows an allocation of the application of Figure 2 to a platform with one communication, one storage and three execution components.

Functions f_1 and f_2 are allocated to h_{41}, function f_3 to h_{42} and functions f_4 and f_5 to h_{43} while data channels dc_6 and dc_7 are allocated to a communication path containing h_{44} and h_{45}.

Property 3 Valid allocation
Let $f_i, f_j \in F$ and $h_i = \overrightarrow{\mathcal{A}_f}(f_i), h_j = \overrightarrow{\mathcal{A}_f}(f_j)$ and $dc_{f_i,f_j} \in \mathcal{DC}$ a data channel between f_i and f_j. Then, dc_{f_i,f_j} can be allocated to a communication path $c_P = \langle \pi_w, \pi_r \rangle$ if and only if the first component of π_w is h_i and the last component of π_r is h_j.

3.3 Trace generation

We now define the notion of *traces* and *occurrences of operators* from a *model execution*. An execution engine must provide HW/SW partitioning models with an operational semantics, i.e. it has to provide the hardware components with all the necessary behaviours to execute the application, e.g., scheduling policies must be used by the engine to schedule tasks on processors and to perform transfers on buses while respecting the maximum capacity of these buses (i.e., throughput). A model execution generates one execution trace for a given model.

Definition 19 Execution trace
An execution trace for a time interval $[0, \tau]$ is a sequence $x = \langle t_i, \ldots, t_j \rangle$ where each $t \in x$ is an execution transaction with start and end times in $[0, \tau]$. Transactions in an execution trace are ordered. The ordering is based on a unique sequence number assigned to each transaction by the execution engine.

Table 1 gives one possible execution trace for the application model of Figure 2, with allocation model of Figure 3, for a time interval $[0, 50]$. This example will be referenced as "our main example" in the sequel. As the execution of Start, Stop, Choice, Merge and Set operators is assumed to take no time, their execution is not represented in the trace. The `id` field corresponds to the operator indexes of Figure 2. Other fields are explained below.

Definition 20 Execution transaction
An execution transaction $t = \langle seq_t, \tau_t^s, \tau_t^e, h_t, o_t \rangle$ represents an execution of operator o_t on a hardware component h_t. A transaction has a sequence number seq_t, a start time τ_t^s and an end time $\tau_t^e \geq \tau_t^s$.

The order defined by seq is strict and total on x. Moreover, a transaction must always be before any transaction with a higher start time and two transactions having the same start time must be ordered according to their end time. A transaction can have the same start time τ_t^s and end time τ_t^e.

Definition 21 i^{th} Occurrence of operator o in x
As transactions are provided with an order in the execution trace, it is clear to speak about the i^{th} occurrence $t_{o,i}$ of the operator o in the execution trace x.

Property 4 System execution constraints.
A trace $x \in EXEC(m, \tau)$ must fulfill the following constraints so we can analyze it:

- Loops are unrolled. Since they have a fixed number of iterations, each operator o inside a loop can be replaced by a set of equivalent operators $o_i, ..., o_j$.
- Only one operator at a time can be executed on a given hardware component.
- If a read or write path of a communication path is allocated to hardware components with different throughputs, the overall communication throughput is constrained by the hardware component having the minimum throughput.

Hypothesis 1 Execution engine. We assume that the execution engine complies with all the model dependencies and constraints we have defined before.

3.4 Requirements on model execution

Generally, a requirement expresses a property on the system. Usually, it is a goal or an anti-goal that the system must satisfy. Requirements are expected to be satisfied for all possible execution traces.

Definition 22 *Maximum latency requirement*
Latency requirements specify timing constraints on the execution of a system. A maximum latency requirement r specifies a maximum delay between elements of execution traces, characterized by their operators. We assume that a maximum latency requirement is expressed as "The maximum latency between

Table 1 Execution trace in tabular format

seq	5	6	7	8	11	12	14	15	17	20	21	23	25	27	28	29	32	33	34
id	34	15	27	35	38	39	11	11	12	16	16	18	30	21	22	22	31	31	23
h	43	41	42	43	43	43	41	44	41	44	41	41	42	41	41	44	44	42	41
τ_s	0	0	0	5	6	7	10	10	11	15	15	16	17	31	45	45	46	46	46
τ_e	5	10	17	6	7	57	11	11	15	16	16	31	19	45	46	46	47	47	50

operator o_A and operator o_B should be less than λ_{max}". A maximum latency requirement is denoted as: $r = \langle o_A, o_B, \lambda_{max} \rangle$.

$r = \langle o_{11}, o_{31}, 35 \rangle$ is a latency requirement which completes our main example. It is violated by execution trace in Table 1, as

$$\tau^e_{t_{o_{31},1}} - \tau^s_{t_{o_{11},1}} = 47 - 10 = 37 > 35$$

3.5 Valid execution traces

PLAN can analyze only valid execution traces, defined as follows.

Definition 23 Valid execution traces
$\mathcal{E}_m$ denotes the set of all possible execution traces of model m and $\mathcal{E}_V \subset \mathcal{E}_m$ the set of all valid execution traces, i.e. traces on which the analysis technique can be applied. Execution traces are valid only if they satisfy Hypothesis 2 and Hypothesis 3 given below. Some of these assumptions are discussed in Section 7.

In the scope of this paper, we are concerned with operators which depend on each other. In other words, we assume that the two operators of a given maximum latency requirement are dependent since a maximum latency requirement between 2 independent operators is of little interest. So, there must exist at least one dependency path between operators o_A and o_B. There may be several. The executed operators that are in dependency paths between o_A and o_B are one of the main parameters to classify transactions.

Hypothesis 2 Limitation on the occurrence of operators. Valid execution traces must have the two following properties.

First. The execution trace must contain exactly one transaction corresponding to operator o_A and one transaction corresponding to operator o_B. In fact, when these transactions rely on a channel that is mapped on a hardware path with several components, they may be decomposed into several transactions sharing a same timing: one per component (see for example $t_{o_{11},1}$ and $t_{o_{11},2}$ in Table 1). In this case, we consider the first of them. Thus, in the sequel, t_{o_A} and t_{o_B} will respectively abbreviate $t_{o_A,1}$ and $t_{o_B,1}$.

Second. A dependency path between o_A and o_B must have been selected by the execution engine and executed. More formally, let us define sequences of operators "included" in an execution trace x:

$$\begin{aligned} &opSeq(x) = \\ &\{\langle o_{i_1}, \cdots, o_{i_n} \rangle \mid \exists \{t_1, \cdots t_n\} \subseteq x, \\ &\forall k \in [1, n], o_{t_k} = o_{i_k} \\ &\wedge \forall k \in [1, n-1], seq_{t_k} < seq_{t_{k+1}}\} \end{aligned}$$

In other words, according Hypothesis 2, an execution trace is valid only if there is at least one dependency path from o_A to o_B included in x:

$$x \in \mathcal{E}_V \Rightarrow \exists \overrightarrow{o_A o_B}, \overrightarrow{o_A o_B} \in opSeq(x)$$

Thus, executions in which t_{o_B} is not linked to t_{o_A} by a dependency path are rejected. Moreover, to enforce a dependency path, two transactions corresponding to two consecutive operators in the same dependency path must not overlap. That is the end time of the first transaction must be smaller than the start time of the second one. Hypothesis 3 is an over approximation of this.

Hypothesis 3 Transactions in the same dependency path are not interleaved.

$$\begin{aligned} x \in \mathcal{E}_V \Rightarrow &\forall \overrightarrow{o_A o_B} \in opSeq(x), \forall \overrightarrow{o_i o_j} \in \overrightarrow{o_A o_B}, \\ &\forall (t_{o_i,n}, t_{o_j,m}) \in x^2, \\ &\tau^s_{t_{o_j,m}} > \tau^e_{t_{o_i,n}} \vee \tau^s_{t_{o_i,n}} > \tau^e_{t_{o_j,m}} \end{aligned}$$

From now on, we assume that Hypotheses 2 and 3 are fulfilled and that an execution trace x is thus always valid.

4 Precise Latency Analysis Approach: categorization

4.1 Execution trace analysis

When a requirement is not satisfied, human analysis of bulk transactions may be tedious if not unfeasible, especially when there are many transactions. Our execution trace analysis classifies transactions into categories in order to help understand the causes of the violation. Definition 24 lists the categories we identify. These categories are called impact sets as the classification relies on the impact of transactions on latency. As they form a partition, they are exclusive and exhaustive.

Definition 24 Execution trace partition

We classify transactions of an execution trace x into the following categories, called impact sets: MOP_x (mandatory operator), OOP_x (optional operator), MF_x (mandatory function), OF_x (optional function), C_x (contention), NDC_x (no direct contention), OH_x (other hardware), O_x (Others). $\{MOP_x, OOP_x, MF_x, OF_x, C_x, NDC_x, OH_x, O_x\}$ is a partition of the transactions of an execution trace x. For short we will denote MOP_x, OOP_x, ... as MOP, OOP, ...

The choice of these categories is a strong heuristic. In practice, the set of all transactions from the beginning of the trace that have an impact on latency may be very large. We therefore consider an impact as an effect on the date at which the impacted transition t can start. As soon at there are shared ressources (in particular scheduling), most transactions are delayed with respect to their ideal starting date (their ideal starting date is when there is no concurrence). When a transaction is delayed, it directly delays all transactions depending on it. And that goes double when considering indirect impact, where a transaction t delays a transaction t' which in turn delays another transaction t'', and so on. To avoid having to classify transactions acording to potentially longs chains of delaying dependencies, we decided to focus on the transactions that are the most "close" to the latency requirement under analysis. This choice has been relevant for the case study we present in section 6, but may obviously be questionable for larger or different systems. We intend to consider a longer set of delaying dependencies in our future work.

Intuitively, for a trace x and a requirement $\langle o_A, o_B, \lambda_{max}\rangle$, our focus is the following one (sets are formally defined in following sections):

- All categories except O only contain transactions that execute between t_{o_A} and t_{o_B}.
  O contains all other transactions.
- $DP = MOP \cup OOP \cup MF \cup OF$ contains transactions of x that execute after t_{o_A} and before t_{o_B} because of control flow or data communication dependency to t_{o_B}. Some of them (MOP and MF) are mandatory: they occur in any trace containing t_{o_A} and t_{o_B}. The other ones (OOP and OF) are optional: they occur because some choice operators branch has been taken in x but could be absent in another trace. They are relevant because delaying one of them automatically delays t_{o_B} and indentifying which ones have been strongly delayed may be interesting.
- C contains transactions that delay some transaction of DP because they concurrently execute on the same hardware, just before.
- NDC and OH contain transaction that do not directly delay any transaction of DP because they do not execute immediately before, or they execute on another hardware.

Thus analysis consists in first looking at DP and C, which is often sufficient to locate a problem. Once a disruptive transaction t has been identified, it is possible to search indirect impacts (in sets NDC, OH and O) that have delayed t in an indirect manner, as explained in section 4.2.8 (implementation of this feature is future work).

Notice also that we consider a direct dependency from any Read operator to any Write operator on the same channel although there is not always a one-to-one correspondence between Read and Write operators at execution. In figure 4, the dashed arrows represent *over approximations* of data dependencies. Let us first consider case (A), with two write operators: the first writes 2 samples, and the second 3 samples. The read operators reads 5 samples. Since we consider dependencies for the general case, i.e. all read operators of a channel depend on all write operators of the same channel, in our example, this leads us to consider the operator `0` as optional although it is mandatory (The read needs the two write operators to be

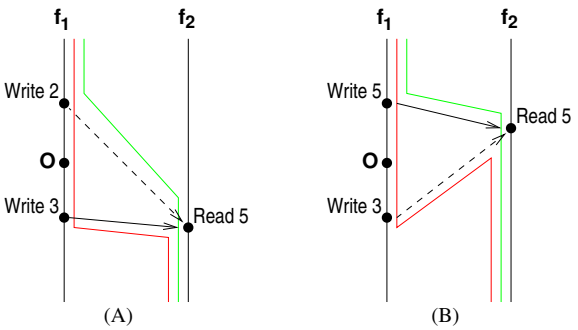


Fig. 4 Data dependency approximation

performed to complete), because there is a path (the green path) that does not contain O. In case (B), right, O is considered as optional because it is on a path (the red path, left): in this case, however, O should neither be mandatory nor optional for the considered paths.

Nonetheless, for numerous applications (in particular in signal processing), we do need this one-to-one correspondence, with the same amount of samples on both sides: in that case, there is no approximation. A way to avoid all these approximations is to have only one write and one read operator per channel, with the same amount of samples written/read.

Finally, as often in static analysis, our algorithms does not evaluate guards of choices operators. Thus we do not detect branches that can not be executed because guards are always false. Thus, some mandatory operators (from the execution point of view) are tagged as optional (from the logical dependency point of view).

4.2 Impact sets in Detail

This section provides the definitions that decide to which impact set a transaction t_o of a trace x belongs. We remind that our approach relies on a trace that has been generated by an execution engine: even though a lot of information of the input model is not used by the classification exposed in this section, this (semantical) information has been taken into account by the execution engine to generate the trace to be analyzed.

In the sequel, all definitions are parametrised by the same data which are the parameters provided to the execution engine and the resulting trace. To avoid repeating them many times, we rely on the following convention.

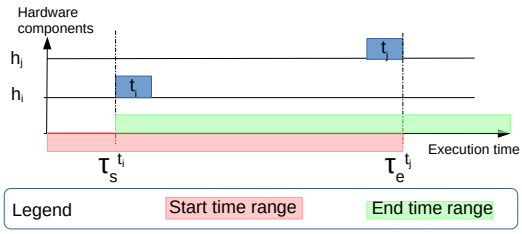


Fig. 5 Illustration of the *cross* function

Convention All following definitions rely on

- a model m
- a latency requirement with operators o_A and o_B
- a trace x containing t_{o_A} and t_{o_B}
- a transaction t_o in x for which an Impact set must be selected.
  o is the operator of t_o

We use the following notation:

Definition 25 Function of an operator
Function $getF$ takes an operator o as input and returns the unique function f to which operator o belongs:

$$getF(o) = f \mid o \in O_f$$

Also, since we focus on transactions that are executed between o_A and o_B, we rely on the following definition:

Definition 26 Transaction Executes Between Two Transactions
A transaction t is said to execute (at least partially) between two transactions t_i and t_j if t ends after the start time of t_i and t starts before the end time of t_j. Formally, function *cross* characterizes sets of such transactions.

$$cross(t_i, t_j) = \{t \in x \mid \tau_t^e > \tau_{t_i}^s \wedge \tau_t^s < \tau_{t_j}^e\}$$

Figure 5 shows two transactions t_i and t_j. Transactions that start in the pink range and end in the green range execute between t_i and t_j.

4.2.1 On Path sets

MOP and *OOP* are called On Path sets since dependency paths between o_A and o_B are of

utmost importance to identify transactions of these sets. Transactions with operators in dependency paths $\overrightarrow{o_A o_B}$ executed between t_{o_A} and t_{o_B} are in one of the On Path sets. Some of these operators are mandatory, as at least one path containing them must be executed between o_A and o_B. The other ones are optional.

Definition 27 In-Dependency Between Operators
Let o_i and o_j be operators, and let $ST = \bigcup_{f \in F} St_f$ be the set of start operators of functions.

Operators that *may have to* be executed between o_i and o_j are characterized by *Dop*:

if $o_i = o_j$ *then* $Dop(o_i, o_j) = \{o_i\}$
else if $o_j \in ST$ *then* $Dop(o_i, o_j) = \emptyset$
else $Dop(o_i, o_j) =$
$\bigcup_{o' \in pred(o_j), Dop(o_i, o') \neq \emptyset} (\{o_j\} \cup Dop(o_i, o'))$

Some of them are *mandatory*, characterized by *Mop*:

if $o_i = o_j$ *then* $Mop(o_i, o_j) = \{o_i\}$
else if $o_j \in ST$ *then* $Mop(o_i, o_j) = \emptyset$
else
  if o_j *is a merge operator then*
    let $\Theta = \bigcap_{o' \in pred(o_j), Dop(o_i, o') \neq \emptyset} Mop(o_i, o')$.
    if $\Theta = \emptyset$
    then $Mop(o_i, o_j) = \emptyset$
    else $Mop(o_i, o_j) = \Theta \cup \{o_j\}$
  else $Mop(o_i, o_j) =$
  $\bigcup_{o' \in pred(o_j), Mop(o_i, o') \neq \emptyset} (\{o_j\} \cup Mop(o_i, o'))\}$

The other ones are *optional*, characterized by *Oop*:
$Oop(o_i, o_j) = Dop(o_i, o_j) \backslash Mop(o_i, o_j)$.

Lemma: for any o_i, o_j, $Mop(o_i, o_j) \subseteq Dop(o_i, o_j)$ and thus, $(Mop(o_i, o_j), Oop(o_i, o_j))$ is a partition of $Dop(o_i, o_j)$.

In requirement $r = \langle o_{11}, o_{31}, 35 \rangle$ (operators are defined in Figure 2), we are interested in the delay between sending data in f_1 and receiving data from f_2 in f_3. o_{11} writes data to dc_6 and o_{16} reads from dc_6. Similarly, o_{22} writes to dc_7 and o_{31} reads from dc_7. As o_{17} is a choice there are two dependency paths between o_{11} and o_{31} denoted as $\overrightarrow{o_{11} o_{31}}^1$ and $\overrightarrow{o_{11} o_{31}}^2$.

$$\overrightarrow{o_{11}, o_{31}}^1 = o_{11}, o_{16}, o_{17}, o_{18}, o_{20}, o_{21}, o_{22}, o_{31}$$
$$\overrightarrow{o_{11}, o_{31}}^2 = o_{11}, o_{16}, o_{17}, o_{19}, o_{20}, o_{21}, o_{22}, o_{31}$$

So, between o_{11} and o_{31}, operators o_{11}, o_{16}, o_{17}, o_{20}, o_{21}, o_{22} and o_{31} are mandatory as they *must* execute, while operators o_{18} and o_{19} are optional as they *may* execute.

Definition 28 *MOP* (Mandatory Operator transaction set)
t_o belongs to *MOP* if and only if operator o is a mandatory operator between operators o_A and o_B and t_o executes between o_A and o_B.

$$t_o \in cross(t_{o_A}, t_{o_B}) \wedge o \in Mop(o_A, o_B) \iff t_o \in MOP$$

Definition 29 *OOP* (Optional Operator transaction set)
t_o belongs to *OOP* if and only if operator o is an optional operator between o_A and o_B and t_o occurs in the execution of a dependency path between o_A and o_B.

$$t_o \in cross(t_{o_A}, t_{o_B}) \backslash MOP \wedge (\exists \overrightarrow{o_A o_B} \in opSeq(x),\ o \in \overrightarrow{o_A o_B}) \iff t_o \in OOP$$

Thus, in the example in Figure 2, transactions between t_{11} and t_{31} executing $o_{11}, o_{16}, o_{17}, o_{20}$ or o_{21} are in *MOP* while transactions executing o_{18} and o_{19} may be in *OOP* (or not, depending on which one has been executed in the analysed trace). Thus, for the trace in Table. 1, $\{t_{16,1}, t_{16,2}, t_{21,1}, t_{22,1}, t_{22,2}\} \subset MOP$ and $t_{18,1} \in OOP$. Notice that Definition 29 is a little bit more complex than Definition 28 because the optional path between o_A and o_B that contains o must have been executed (o_2 in example of Figure 6, section 4.2.3 illustrates this).

4.2.2 In Functions sets

Operators which have transactions in x and are in an executed dependency path between o_A and o_B can not execute before their previous operators in their functions are executed. So, for each o_i in On Path sets, the execution trace x must contain transactions with operators in a dependency path from St_f to o_i, where $f = getF(o_i)$.

The same applies for synchronization and data channel dependencies. For instance, let f_i and f_j be two functions, and o_i an operator of f_i in an executed dependency path from o_A and o_B. Let o'_i be a Wait operator on an executed dependency

path $\overrightarrow{St_{f_i} o_i}$, depending on Notify operator o_j of f_j. Then o_j and its predecessors must have completed their execution before o_i starts executing. More generally this characterization relies on operators that must or may be executed before o_B with respect to dependency paths and function control flows.

Definition 30 In-Dependency Operators
Let o be an operator, and $pred(o) = \{o' | \overrightarrow{o'o} \in \mathcal{D}\}$

Operators that *may have to* be executed before o w.r.t. dependency are characterized by Df:
$Df(o) = \{o\} \cup \bigcup_{o' \in pred(o)} Df(o')$

Some of them are *mandatory,* characterized by Mf:
if o is a merge operator then
$Mf(o) = \{o\} \cup \bigcap_{o' \in pred(o)} Mf(o')$
else $Mf(o) = \{o\} \cup \bigcup_{o' \in pred(o)} Mf(o')$

The other ones are optional, characterized by Of:
$Of(o) = Df(o) \backslash Mf(o)$.

Thus, just like for On Path sets, In Function sets contain transactions with operators that may be mandatory or optional.

Definition 31 MF (Mandatory Function transactions set)
t_o belongs to MF if and only if it does not belong to MOP or OOP and o is a mandatory operator before o_B:

$$t_o \in cross(t_{o_A}, t_{o_B}) \backslash (MOP \cup OOP) \wedge o \in Mf(o_B) \iff t_o \in MF$$

For instance, let us consider requirement $\langle o_{11}, o_{31}, \lambda_{max} \rangle$, operators being taken from the model given in Figure 2. Operator o_{16} is mandatory between o_{11} and o_{31}. Also, operator o_{15} has to be executed since it belongs to the only dependency path between St_{f_2} and o_{16}. Thus $t_{o_{15},1} \in MF$.

Let us now consider $\langle o_{15}, o_{21}, \lambda_{max} \rangle$. o_{16} is mandatory between o_{15} and o_{21} and o_{10} and o_{11} are on all paths from o_{10} to o_{16}. Thus o_{10} and o_{11} are in MF.

Definition 32 OF (Optional Function transactions set)
t_o belongs to OF if and only if it does not belong to MOP, OOP or MF and there exists a dependency path from o to o_B executed in x:

$$t_o \in cross(t_{o_A}, t_{o_B}) \backslash (MOP \cup OOP \cup MF) \wedge \exists \overrightarrow{oo_B} \in opSeq(x) \iff t_o \in OF$$

For instance in Figure 2, let us consider requirement $\langle o_{22}, o_{31}, \lambda_{max} \rangle$ and the trace given in Table. 1. $o_{31} \in MOP$ and there are two paths from St_{f_3} to o_{31}. o_{26} belongs to both, thus $t_{o_{26},1}$ is in MF. o_{27} belongs to only one, thus $t_{o_{27},1}$ is in OF.

4.2.3 Dependency path transactions

This section summarizes the four previous impact sets, which classify transactions in x executed between o_A and o_B and that have at least a dependency path to o_B. We now propose to classify transactions that delay any of them. To simplify the definition of these delays, we consider a global set for these four impact sets:

Definition 33 DP (Dependency Path transactions)
$$DP = MOP \cup OOP \cup MF \cup OF$$

Lemma: respecting notation of convention p. 11,
$$DP = \{t_o \in x \mid DP_{\overrightarrow{o\,o_B}} \cap opSeq(x) \neq \emptyset\} \cap cross(o_A, o_B)$$

Figure 6 presents a simplified example to shortly illustrate the four impact sets. Operator o_3 is in all pathes from o_A to o_B thus in MOP. If they are executed before o_A in the executed execution trace, o_1 and o_4 are not in DP because they are not in $cross(o_A, o_B)$. Otherwise they are respectively in MF and OF because o_1 is mandatory before o_B but not o_4 . If the trace contains the left green path, o_2 is in MF if it is executed after o_A. Indeed, it is mandatory before o_B but not in any executed path from o_A to o_B. If the trace contains the right red path, o_2 is in OOP as it is in a dependency path that has been executed but it is not mandatory due to the existence of the left green path.

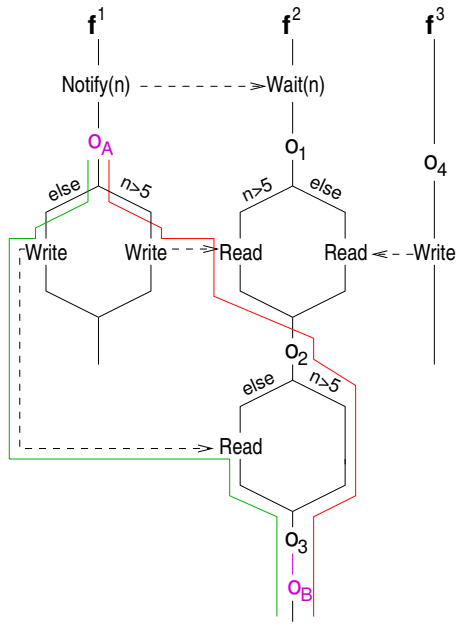


Fig. 6 A summary example for MOP/OOP/MF/OF sets

4.2.4 Contention delay

We are now interested in transactions that delay other transactions, in particular when this delay is caused by a contention. Contentions occur when transactions of two concurrent functions want to use the same execution hardware or the same communication hardware at the same time. In this situation, one of the two functions is delayed until the requested resource is available.

In order to identify transactions that have been delayed, we propose to compare actual execution dates with the ones obtained in an ideal model where each transaction has exactly its expected duration[1] and starts exactly when the last transaction it depends on terminates[2]. Our ideal model relies on the use of one processor for each function, and we assume buses can carry all the necessary samples without delay, i.e. their data width is set to the sum of the data widths of all channels mapped on them.

[1] computed from its specified complexity and hardware caracteristics (bus throughput, CPU frequency, ...)

[2] By exactly, we mean that there is no other transaction between.

Definition 34 Best Start Execution Date (BSED)
$BSED(t)$ is the earliest possible start time of transaction t, obtained by executing exactly the same operators on the same dependency paths but considering the ideal model.

Definition 35 Best End Execution Date (BEED)
$BEED(t)$ is the earliest possible end time of transaction t, obtained by executing exactly the same operators on the same dependency paths but considering the ideal model.

A transaction is delayed if its start and/or end time is higher than the one in the ideal model. A transaction delays another transaction if it uses the same hardware and its end time is between the ideal value and the obtained value of the delayed transaction. This leads to the following definitions[3].

Definition 36 Delayed Transactions
Let $T \subseteq x$ be a set of transactions of trace x,

- **start delay:**
  $sDelayed(T) = \{t \in T \mid BSED(t) < \tau_t^s\}$
  $sDelaying(T) = \{t' \in x \mid \exists t \in sDelayed(T), h^{t'} = h^t \wedge BSED(t) \leq \tau_{t'}^e \leq \tau_t^s\}$
- **end delay:**
  $eDelayed(T) = \{t \in T \mid BEED(t) < \tau_t^e\}$
  $eDelaying(T) = \{t' \in x \mid \exists t \in eDelayed(T), h^{t'} = h^t \wedge BEED(t) \leq \tau_{t'}^e \leq \tau_t^e\}$
- **delay:**
  $Delayed(T) = sDelayed(T) \cup eDelayed(T)$
  $Delaying(T) = sDelaying(T) \cup eDelaying(T)$

4.2.5 Contention set

A transaction is in the contention set C if it directly causes a delay in the execution of a transaction in DP (Definition 33).

Definition 37 Contention set (C)
t_o belongs to C if and only if it was executed between t_{o_A} and t_{o_B} and it delays some transaction in DP. Formally,

$$t_o \in cross(t_{o_A}, t_{o_B}) \backslash DP \wedge t_o \in Delaying(DP) \iff t_o \in C$$

[3] Reminder: definitions rely on notations provided in convention p. 11

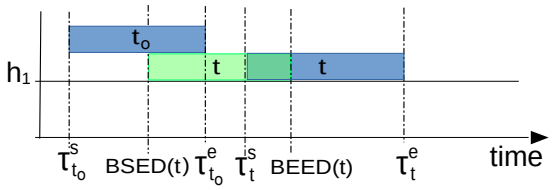


Fig. 7 Graphical representation of an example where a transaction t_0 is classified in the Contention set

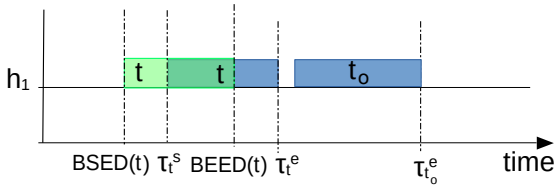


Fig. 8 Graphical representation of an example where a transaction t_0 is classified in the No Direct Contention set

Figure 7 shows a situation where a transaction t is delayed by t_o, both using the same hardware h_1. If transaction t is in DP but t_o does not belong to this set, then t_o is in the contention set.

In our main example (Table. 1), $t_{o_{12},1}$ does not belong to DP. Transaction $t_{o_{16},2}$ executes on the same hardware (h_{41}) and is delayed. We have $\tau^e_{t_{o_{12},1}} > BSED(t_{o_{16},2})$ and $\tau^e_{t_{o_{12},1}} = \tau^s_{t_{o_{16},2}}$. Thus $t_{o_{12},1}$ belongs to the contention set.

4.2.6 No Direct Contention set

There are also transactions that run on a hardware used by a transaction in DP but that do not directly delay transactions in DP. These transactions are said to cause no direct contention.

Definition 38 No Direct Contention set (NDC)
t_o belongs to NDC set if and only if it is executed between t_{o_A} and t_{o_B} on a hardware used by a transaction in DP and it is not in DP nor in C. Formally,

$$t_o \in cross(t_{o_A}, t_{o_B}) \backslash (DP \cup C) \wedge \exists t \in DP, h_{t_o} = h_t \iff t_o \in NDC$$

Figure 8 shows a case where a transaction t is delayed. However, t is not delayed by t_o although t_o runs on the same hardware.

Let us come back to our main example with the latency requirement $\langle o_{11}, o_{31}, 35\rangle$, see Table. 1. By computing dependency paths, one can prove

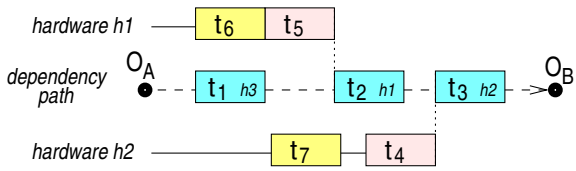


Fig. 9 Example of NDC transactions

that $t_{o_{23},1}$ does not belong to DP. It executes on h_{41} but does not delay any transaction in DP running on this hardware. Thus, it belongs to the No Direct Contention set. Another case for transactions being classified in NDC: transactions delaying at least one transaction in C but delaying no transactions in DP are in the No Direct Contention set (i.e., their impact is indirect).

Figure 9 illustrates two simple cases for transactions to be in NDC. In this example, blue transactions (t_1, t_2, t_3) are in DP. Pink transactions (t_4, t_5) are in C as t_5 delays t_2 on hardware $h1$ and t_4 delays t_3 on hardware $h2$. Yellow transactions (t_6, t_7) are in NDC as they execute between o_A and o_B on hardware used by blue transactions but do not directly delay any of these blue transactions. This illustrates that transactions in NDC may have indirect impact (t_6 delays t_5) or no indirect impact (t_7 does not delay t_4).

4.2.7 Other Hardware set (OH)

Other transactions executed between t_{o_A} and t_{o_B} are those whose corresponding hardware component is not used in DP. They are in the *Other Hardware* set.

Definition 39 Other Hardware set (OH)
t_o belongs to OH if and only if it was executed between t_{o_A} and t_{o_B} on a hardware that is not used in DP. Formally,

$$t_o \in cross(t_{o_A}, t_{o_B}) \wedge \forall t \in DP, h_{t_o} \neq h_t \iff t_o \in OH$$

In our main example (Table. 1), $t_{o_{39},1}$ is executed between o_{11} and o_{31} on h_{43} which does not support any transaction of DP (these transactions are in C or NDC). Thus, $t_{o_{39},1}$ is in OH.

4.2.8 Others set

The last impact set contains the remaining transactions, i.e. the transactions that are not executed

between t_{o_A} and t_{o_B}. In our example, only $t_{o_{23}}$ is in O.

Definition 40 Others set (O)
t_o belongs to O if and only if it does not execute between t_{o_A} and t_{o_B}. Formally,

$$t_o \notin cross(t_{o_A}, t_{o_B}) \iff t_o \in O$$

Clearly, transactions that execute after the latency interval do not have any impact on this latency. Transactions that execute before t_{o_A} may have some indirect impact. This is also the case for transactions in NDC and OH (which execute between t_{o_A} and t_{o_B}). Such transactions are numerous and often less relevant w.r.t. the goal of finding delay causes. Thus we do not classify them more finely. However, with some enhancement of the tool, they could be explored in a user-guided way. For this, we can rely on Definition36 on delaying transactions. Once we have identified a transaction t which directly impacts the latency, if t itself is delayed in a way that disturbs us, we can further investigate by looking at $Delaying(\{t\})$, for example, and continue recursively if necessary. More globally, we could build combined requests such as $Delaying(C)\backslash C$, or $Delaying(C) \cap cross(o_A, o_B)$,etc

Lemma.
$\{MOP_x, OOP_x, MF_x, OF_x, C_x, NDC_x, OH_x, O_x\}$ is a partition of the execution trace x, thus Definition 24 is well defined.

5 Application to UML/SysML

The objectives of this section is to show how PLAN was implemented to provide the categorization results to the designer in a user-friendly way. It discusses how our formal model maps to SysML diagrams.

5.1 Application to UML/SysML

For SysML diagrams, we selected TTool [3] and SysML-Sec [7] developed in our lab. SysML-Sec follows the Y-Chart approach [38]. In the application model, functions are defined as SysML blocks (green blocks) and variables of a function are attributes of blocks. Functions of the example given in Figure 2 are shown in Figure 10. Figure 10 also captures communications between functions

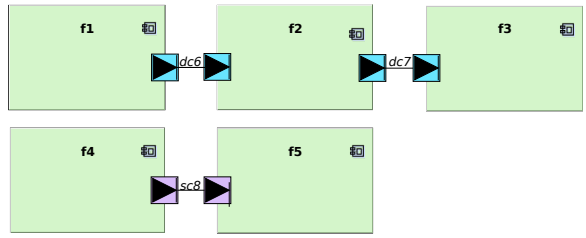


Fig. 10 Functions of the example in Figure 2

with an Internal Block Diagram. Synchronization ports are in purple while data ports are in blue.

The behavior of a function can be captured with extended SysML Activity diagrams, thanks to its built-in operators and control flow connections. Operators of functions are mapped to SysML elements as shown in Table 2. Operators IntOp and Set are both mapped to actions of activity diagrams. Notify and Wait operators are modeled in violet, WriteData and ReadData operators in blue, Choice, Merge, IntOp and Set operators in green, Start and Stop in black. The Merge operator is not directly supported by TTool, but loops or OrderedSequence can be used for this. OrderedSequence captures concurrent activities which are synchronized with their end / start from left to right. StaticForLoop executes a subactivity for a fixed number of times (Table 3). Finally, Figure 11 shows the behavior of the functions of Figure 10.

UML Deployment Diagrams or SysML allocations can be used for platform and allocation models. Execution, communication and storage hardware components are shown in blue, brown and green respectively (Figure 12, deployment diagram). Functions are allocated to execution hardware components and data channels are allocated to communication paths (communication and storage hardware components), for instance using artifacts, as shown in Figure 12.

5.2 Model simulation

Simulation is one particular case of execution using an environment that reproduces the behavior of a model to produce an execution trace that we call *simulation trace*.

Our HW/SW partitioning models can obviously be used as documentation but thanks to the execution semantics provided to functions and hardware components as explained in [41], we are able to generate simulation traces or to generate

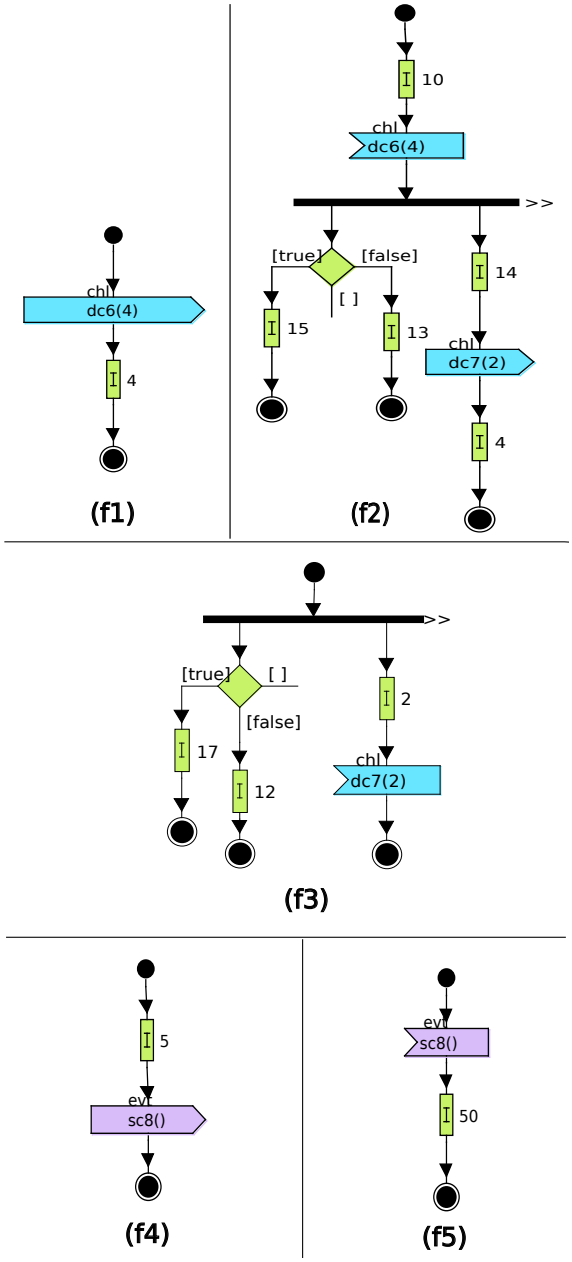


Fig. 11 Behaviors of the functions of Figure 2

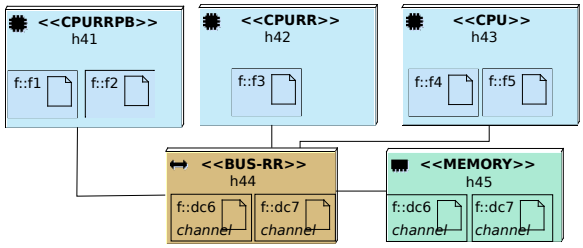


Fig. 12 Allocation of the functions given in Figure 3

Table 2 Operators in TTool

cat(c)	Graphical Representation
Start	
Stop	
Choice	[guard1] [guard2] [ ]
WriteData	chl data_channel(size)
ReadData	chl data_channel(size)
Notify	sync_ch
Wait	sync_ch
IntOp	complexity
Set	x=x+1

Table 3 OrderedSequence and StaticForLoop operators in TTool

Operator	Graphical Representation
OrderedSequence	>> branch1 branch2 branch3
StaticForLoop	Loop 10 times inside loop exit loop

reachability graphs. Getting a simulation trace is generally fast, thanks to our high abstraction level and to the advanced simulation techniques (e.g. cutting of transactions). Among the abstraction the simulator considers, data exchanges between functions are modeled only with the amount of exchanged data as explained in Definition 12. Also, algorithms are modeled using IntOp operators with a complexity attribute used to specify the processing complexity of the algorithm. This is referred to as *algorithm abstraction*. Control operators that are Start, Stop, Choice, Merge and Set categories do not consume any clock cycles. Also, in the platform model, the hardware components are modeled as parameterized hardware components. The throughput of a data channel is determined by (1) the slowest hardware resource along the communication path on which this data channel has been allocated, (2) the quantity of data to read or write and (3) the contentions on execution, communication and storage hardware components involved in a communication path.

These abstraction levels in the application and platform models do not allow us to have a simulation in which we can execute in a precise way just like it can be done for the lowest level of abstraction, e.g., cycle-accurate, bit accurate (CABA) [12].

In the scope of this paper, we assume that the simulation produces one simulation trace (in XML), and then we check in this trace whether a given requirement is satisfied or not. In case the requirement is not satisfied, the classification of transactions is performed.

6 Tooling and evaluation

This section shows on a use case how our approach can be efficiently applied to the analysis of simulation traces obtained from SysML models.

6.1 Integration in TTool

TTool is a free and open-source tool supporting several UML profiles, including SysML-Sec. TTool can simulate a model, can perform safety or security formal verifications, and can generate executable code. We have added PLAN to TTool, both in its graphical interface and in its command line interface.

Basically, TTool-PLAN takes as input a model and a TTool simulation trace. In the model, we assume that a designer has selected two operators between which the latency can be studied. In Figure 13, we chose the WriteData operator in f_1 and ReadData operator in f_3. Then, PLAN first generates a dependency graph, and then outputs the transaction categories, see Figure 13.

As shown in previous sections, dependency paths play an important role for the classification of transactions. Thus, each time we analyze an execution transaction to know whether it is related to another transaction, we need to refer to the SysML partitioning model to search for dependency paths. Unfortunately, computing dependency paths directly from the SysML model is costly since many elements are involved (control flows, data channels, ...) As this is done for other domains just like compilers [42], we rely on a dependency graph built from SysML models. $G_m = (\mathcal{O}_m, \mathcal{D}_m)$ is the dependency graph associated to model m, with $\mathcal{O}_m$ and $\mathcal{D}_m$ the vertexes and directed edges of G_m defined with respect to Definition 8 and Definition 9. This graph features all logical dependencies thus making it possible to apply well-known graph algorithms, such as the shortest path algorithm. Thus, in TTool, the classification of transactions relies on G_m. [54] gives the full algorithm on how to build G_m from a SysML model. When relying on the graph rather than the SysML model, transactions are classified exactly as stated before. The only difference is in how dependencies are searched for.

Several facilities help designers to analyze the trace. For instance, with the *Check Path Between Operators* button the designer can check if a dependency path exists in the dependency graph between two operators selected in the dropdown lists. *Show Directed Graph* opens a separate window showing the generated graph. The *Compute Latency* button starts the computation of latency values, stored in the *Latency Values* table. Figure 13 corresponds to the latency in our example in Figure 12. The designer can then run Execution Trace Analysis (ETA) to analyze how transactions impacted latency. Another window displays the result of this analysis as a table of classified transactions, as shown in Figure 14. In this table, each row corresponds to one hardware component in the system and each column represents a one-time slot in the simulation. The

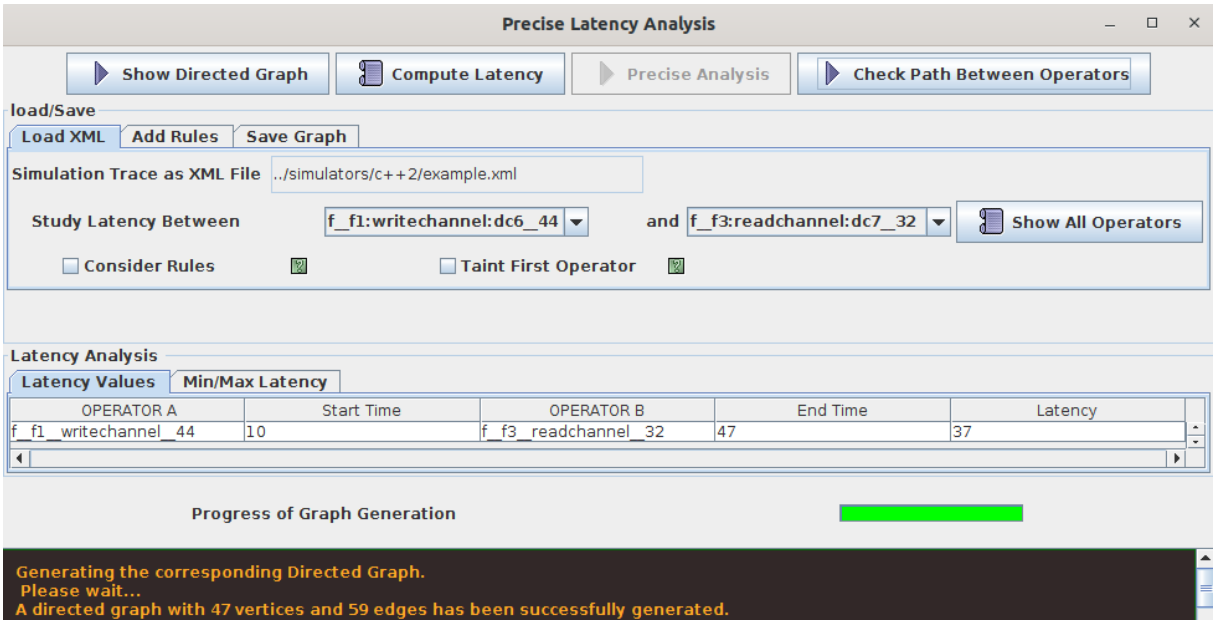


Fig. 13 PLAN window

Fig. 14 PLAN classification output for a latency value

3 sub-figures show different time slots. Transactions are placed according to when and where they were executed. The Mandatory OP (MOP) transactions are displayed in green, the Optional OP (OOP) transactions are displayed in bright purple, the Mandatory Func (MF) transactions are displayed in bright blue (cyan), the Optional Func (OF) transactions are displayed in light gray, transactions that contributed to increasing the latency value due to Contention (C) transactions are colored in red, the NoDirectContention (NDC) transactions are colored in orange and finally those of OtherHardware (OH) are colored in dark blue.

In our example, the IntOp operator in f_1 caused contentions on the hardware component h_{41}.

6.2 Description of the use case

A specification of the industrial drive system—defined in the scope of the H2020 AQUAS project [1] [49]—is shown in Figure 15. The system consists of 3 main components: *Client*, *Motor Control*, and *Motor*. The *Motor Control* is further split into 3 sub components: *Server Control*, *Main Loop* and *Motor Control Power*. The *Motor Control* receives speed and direction data signals from the *Client* through the *Server Control* and sends them to the *Main Loop*. Once the data signals have been read, the *Main Loop* notifies the *Client* through *Server Control* by sending an acknowledgment and runs an algorithm to generate PWM (Pulse Width Modulation) signals. The PWM signals are then sent to the *Motor Control Power*. The *Motor Control Power* transforms these signals into supply voltages and sends them to the *Motor*. *Main Loop* periodically runs an algorithm to monitor the speed and direction of the *Motor* after reading the position data and current value signals sent from the *Motor* via *Motor Control Power*. In case an adjustment is needed, the *Main Loop* sends updated PWM signals to the *Motor Control Power*.

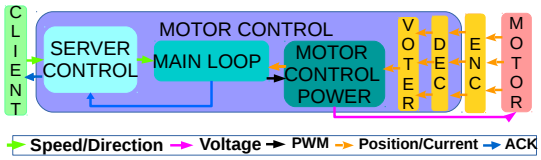


Fig. 15 Specification of the Use Case: An Industrial Drive

Also, a *Voter* ensures safety by receiving redundant position signals from the *Motor*, then calculating their average. This average value is sent to *Motor Control Power*. To ensure confidentiality, position signals are encrypted. The system must ensure that the latency between starting a new iteration of the *Main Loop* and the *Motor* receiving the supply voltages from the *Motor Control Power* is always below 55 µs; this constraint is our latency requirement.

6.3 Model simulation and trace analysis

In Figure 16, x is a variable in function $PWMtoPS$. In Figure 16, a synchronization channel run_Inter connects $Interrupt$ to $MainLoop$, and a synchronization channel $PhaseSig$ and a data channel $PStoM$ connect $PWMtoPS$ to $MotorF$.

Figure 17 shows the allocation of the $MotorF$ function and the $PStoM$ data channel. 56 µs of the industrial drive execution have been simulated since this duration corresponds to the latency requirement. Hardware components run at 200MHz. The obtained simulation trace contains 11888 transactions.

o_A is the first operator of the main loop and o_B is the operator receiving of the voltage value of the motor. These two operators and their respective functions are shown in Figure 16: o_A is the Wait operator named $run\_Inter()$ in $MainLoop$ and o_B is the reading data operator named $PStoM$ in $MotorF$.

In the simulation trace, the start time of $t_{o_A,1}$ is 2 and the end time of the $t_{o_B,1}$ is 11115. The latency in this case is thus 11113 cycles (i.e., 55.56 µs) and the requirement is not satisfied. Let us use PLAN to better understand this situation. Since transactions are colored according to their category, transactions involved in contentions are easy to identify. In our use case, after generating a dependency graph of 552 vertexes and 965 edges

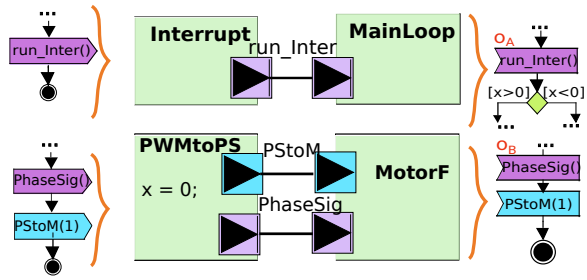


Fig. 16 An Excerpt of the Application Model of The Use Case

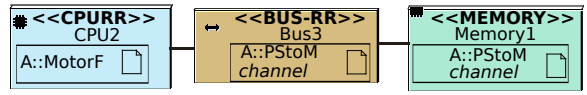


Fig. 17 An Excerpt of the Allocation Model of The Use Case

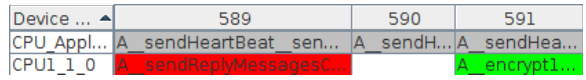


Fig. 18 ETA Output Showing Contention

and running the execution trace analysis, contentions were spotted on the execution hardware component on which the *Motor Control* functions are allocated (Figure 18). These contentions are due to the *Server Control* function: the latter processes data to be written to the *Client* while the encryption function was ready to execute.

To resolve this contention, an execution hardware component is added to the platform and the *Server Control* function is now allocated to it. Running PLAN again, the end time of $t_{o_B,1}$ is now 10606, that is, a latency equal to 10604 cycles (i.e., 53.02 µs). The latency requirement is satisfied. To see how the transaction classifications changed between the two models, we can use PLAN even though the requirement is satisfied. The output in Figure 19 reveals that no contention was detected and that the *Server Control* function could process its data while the decryption function was executing.

6.4 Performance evaluation

As explained before, the two main features of PLAN are the dependency graph generation and the classification of transactions. The dependency graph must be generated for each model, while the classification must be performed for each couple (o_A, o_B) of a simulation trace.

Model	Trans.	D.G.	o_A	o_B	Latency	Class. (s)
Industrial drive	15k	277/436/105ms	sendCommandData	phaseSignal	9989	4.8
			speedDirection	phaseSignal	9825	4.4
			executeFOC	phaseSignal	5610	2.4
			sendCommandData	executeFOC	4379	1.9
Copilot	13k	208/378/43ms	sendPos1	loggingIn	208 004	0.3
			sendDoor1Frame	send_position2	6 419 149	4.7
Rovers	1.5k	191/308/32ms	fromSAtoDT	newMotorPower	2079	1.2
			newLeaderSocketData	injection	14	0.04
			fromSAtoDT	fromTCtoSSO	2212	1.4
testPlan1k	33k	57/83/8ms	trigger	appEnd	22 000	2.2
testPlan10k	330k	57/83/8ms	trigger	appEnd	220 000	30
testPlan20k	660k	57/83/8ms	trigger	appEnd	440 000	66
testPlan60k	1650k	57/83/8ms	trigger	appEnd	1 100 002	285

Table 4 Performance evaluation of PLAN

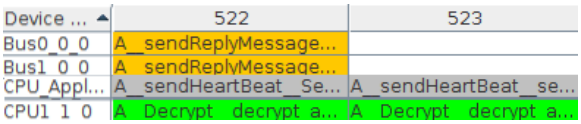


Fig. 19 ETA Output Showing No Contention

We have performed evaluation tests using a macbook pro 2019 running at 2.3GHz, i9 core, 32 GB of RAM. Oracle Java, version 11.0.15.1, has been used to execute most recent version of TTool (Sept. 2022)[4]. Performance results are given in Table 4. The first column gives the evaluated model. Then, "Trans." represents the number of transactions in the whole simulation trace of this model. The third column represents information on the dependency graph, separated with "/", i.e., respectively: the number of vertices, the number of edges and the time in milliseconds to generate the dependency graph from the considered model. The two next columns give the selected o_A and o_B (the name of the event or channel of the operator is given). The latency (in the number of cycles) between o_A and o_B is given in column "Latency". Last, column "Class." gives, in seconds, the time TTool took to classify transactions between o_A and o_B. For both computation times (the graph generation and the classification), the graphical rendering time is not included.

The two first models (Industrial drive, Copilot) have been developed from public use cases of the AQUAS project [5] and used in the scope of AQUAS. The copilot system manages doors of automatic subway systems. The third model was made in the scope of the SPARTA European project (CAPE program), Task 5.2: "Securing the connected vehicle". This model captures how a platoon of rovers works. The fourth model was built in order to stress our trace analysis feature (testing with millions of transactions): a sender emits data to an intermediate task that dispatches received data to a receiver. A fourth task performs data transfers to create contentions on buses and memories. The number of times this data sending occurs is configurable in the model: in Table 4, testPlan1k means 1k loops of data sending, etc. The third and fourth model are available in the model repository of TTool, accessible from within TTool (menu "File", then "open model from network repository").

The results demonstrate that the generation of the dependency graph is straightforward: a few dozens of milliseconds in the worst case. Computing the classification is done in a few seconds or minutes even on large sets of transactions. As explained in next section, both the graph size and the number of transactions have an impact on the PLAN computation time.

7 Discussion

By understanding the reasons why a requirement is not satisfied, designers are expected to be guided

[4]The help integrated in TTool, section Diplodocus, then Mapping, then PLAN, gives a step-by-step process to generate a dependency graph and perform a transaction classification

[5]Both uses cases are described in the public deliverable D5.3, see https://aquas-project.eu/documents/

on how to improve their model. This section discusses the complexity and perspectives of our contributions.

7.1 Complexity

The complexity of the implementation of PLAN in TTool is linked to the size of a system model and the size of an execution trace (the number of transactions in the trace). Computing dependency paths is at the root of many categories. To check if there exists a dependency path between o_A and o_B, TTool checks for the shortest path between these operators: if the shortest path is non empty then these operators are dependent. The complexity of Dijkstra's Shortest Path algorithm is given in [8] as: $O(|E|+|V|log|V|)$, where $|E|$ is the number of edges and $|V|$ is the number of vertexes in a graph. Also, to check if an operator o is in the dependency path between o_A and o_B, TTool computes the shortest path between o_A and o and the shortest path between o and o_B.

Since PLAN iterates over all the transactions in an execution trace x, checking if the operator of each transaction is in the dependency path between o_A and o_B results in a complexity of $O\big((\mid x \mid) * (|E| + |V|log|V|)\big)$.

Further enhancement of our implementation could avoid computing several times the same elements: for instance, pre-computing all possible dependency paths between o_A and o_B and classifying the operators of these paths as optional or mandatory. Yet, the number of the dependency paths to study at the beginning of the analysis might be large. Finally, all systems to which PLAN has been applied were analyzed in a few seconds.

7.2 Extensions:

7.2.1 *One-to-one* relation between operators

For a model, a trace and a requirement $r = \langle o_A, o_B, \lambda_{max} \rangle$, PLAN assumes the trace has exactly one transaction for o_A and one transaction for o_B. This is allowed because Hypothesis 2 guarantees that all loops have a fixed number of iterations, have been unrolled, and the operators have been renamed accordingly.

A possible extension could be to relax the constraint that loops have a fixed iteration count, still offering designers the possibility to relate specific instances of o_A and o_B. A further extension could be, for each occurrence of o_B, to identify to which occurrence of o_A it corresponds. For this, we are currently working on dependency graph tainting so as to trace execution flows.

7.2.2 Dependencies between channel operators

As stated by Definition 14, and underlined in Section 4.1, all read operators of a channel c are considered as depending on all write operators of channel c. This property is meant to cover all possible communication semantics, but it is an over approximation that can lead to identify some of the write / read operators as optional, when they could be in fact mandatory, or even totally unrelated. Taking into account in PLAN the communication semantics used by the execution engine would help reducing this approximation. Note that the same remark applies to synchronization channels. A static analysis taking into account the communication semantics could also help reducing this over approximation. More generally, i.e., for all operators, because of over approximation, transactions could be classified in the wrong category. Since our approach over-approximates dependencies, this means that transactions could get an orange flag when a green flag could be used (thus raising a false warning), and an orange flag when a red flag could be used. A first improvement would be to highlight differently transactions linked to over approximations in order to better inform users of PLAN.

7.2.3 Multi-trace analysis

In this paper PLAN is used to analyze one execution trace at a time. A typical PLAN-based work flow consists in first analyzing the trace with the highest latency, and to accordingly update the model. In some cases this could improve the analyzed trace while worsening other traces, and maybe resulting in a even higher worst case latency. An interesting extension would be to concurrently analyze several execution traces for the same model and requirement.

Extending PLAN to support traces of symbolic execution could be a way to achieve multi-trace analysis. Symbolic execution executes a system using symbols as inputs instead of concrete inputs

(e.g., numbers, integers or strings) and symbolic expressions instead of program variables [40] [16]. The output of the system symbolic execution is expressed in terms of the input symbols [16]. Yet, while a symbolic execution can replace a large number of regular system executions [40], it may result in a large number of symbolic expressions [18], which could increase the complexity of PLAN.

7.2.4 Enriched model

Several extensions could enrich the modeling.

Currently PLAN does not really consider the complete semantics of communication channels. Instead, this is mostly left to the execution engine. What PLAN considers is the dependencies between write and read operators: a data channel is modeled as a vertex in the dependency graph to which the ReadData and WriteData operators are connected. However, we already assign attributes to this data channel vertex including the current buffer size attribute per hardware component. The later attribute returns the amount of data already stored in the buffer on a hardware component. The classification definitions could thus be adapted to check the current buffer size attribute. Indeed, accurately considering how many messages are stored in a buffer could help to more accurately compute latencies and categorize transactions. The same could also apply to synchronization channels.

The categories of operators presented in Section 3.2.1 allow the modeling of algorithms having a statically known complexity. It is also possible to model an algorithm with two possible complexity values (value #1 or #2), thanks to the use of a Choice operator and, for each branch of the Choice operator, of an IntOp representing one of the possible complexity values. A possible extension would be to allow IntOp operators with a complexity interval instead of a fixed value. Adding more control flow operators could also facilitate the modeling.

Finally, as currently defined, PLAN rather targets system-level design: data channels do not convey values but a quantity of data, algorithms can be abstracted with complexity operators,... Handling lower level operations such as operations on boolean, integer and float, and more fine-grain control operations (memory manipulation with locations or pointers) would allow lower-level models, like, for instance, assembly languages running on processor models.

7.2.5 Proof of correctness

Since we have formally defined the different classification sets of transactions of PLAN, a proof of correctness would be an interesting extension of this work.

7.2.6 Implementation

Currently, TTool already supports all categories defined in PLAN. We could extend TTool to support the analysis of transactions outside of the interval defined by maximum latency requirement. Also, the ideal model used to compute BSED and BEED could be improved.

7.2.7 Applying PLAN for a trade-off between safety and security mechanisms

One of the challenges when designing embedded systems is to satisfy altogether its safety, security and performance requirements. The advantages of designing embedded systems while taking the interactions of safety, security and performance requirements into consideration early in the design cycle are highlighted in several approaches e.g. [30] [28].

A model update consists in adding or removing software or hardware components, updating hardware resource parameters or changing a functional behavior. These modifications may occur when adding new safety or security mechanisms, or when improving performance. For example, the addition of encryption/decryption mechanisms can support more secure message exchanges or a new security algorithm can increase the system security level. Such modifications may result in extra computations and communications. In addition, extra contentions on resources may result from the transfer of longer messages [53].

An interesting application of PLAN is to analyze the impact on timing when modifying a SysML model. In this paper, PLAN can be applied to a use case model and an updated version of this model separately. However, we have started implementing an extension of PLAN where the

designer can compare the latency and the classification of transactions for two models. This aims to further assist the designer to compare and determine how the modifications applied to a model impacted its timing requirements. Further work on large sets of security and safety mechanisms is needed to prove our claim and show that our approach always works. However, since safety and security aspect can be easily implemented and verified [6] in TTool, future work intends to apply PLAN on safety and security-related models and evaluate how it supports the interaction between these two aspects.

8 Conclusion

A common way to design an embedded system is to make a model of this system, to simulate or to execute it to obtain traces, and then to check in these traces whether functional and non functional requirements are satisfied or not. The automated trace analysis technique introduced in this paper helps designers to go beyond the yes/no result of requirement verification. To deeper understand the reasons of a maximum latency requirement violation, we capture dependencies in the system model and we accordingly categorize transactions of an execution trace according to their impact on latency. By identifying dependencies between model elements and classifying transactions of an execution trace with respect to latency requirements, we can identify and highlight which software functions and/or hardware components contributed to this delay.

We now target to address the limitations mentioned in the category or discussion sections. In particular, as underlined before, our classification approach deals only the first cause when a contention occurs, and can thus not yet investigate what caused the contention before its first cause: investigating further traces to identify the possible root causes for contentions is part of our future work. Our ultimate goal is to provide designers with automated suggestions for enhancing models in order to ensure that timing constraints are all met.

Acknowledgment

The AQUAS project is funded by ECSEL JU under grant agreement No 737475.

References

[1] Aggregated quality assurance for systems (aquas). https://aquas-project.eu, 2013. Accessed: 2019-09-24.

[2] Retro-ingénerie de traces d'analyse de simulation et d'exécution de systèmes temps-réel – rt-simex. https://anr.fr/Projet-ANR-08-SEGI-0015, 2013. Accessed: 2019-09-24.

[3] TTool, 2013.

[4] El Arbi Aboussoror, Ileana Ober, and Iulian Ober. Significantly increasing the usability of model analysis tools through visual feedback. In *International SDL Forum*, pages 107–123. Springer, 2013.

[5] Nada Alhirabi, Omer Rana, and Charith Perera. Security and privacy requirements for the internet of things: A survey. *ACM Transactions on Internet of Things*, 2(1):1–37, 2021.

[6] L. Apvrille and L. W. Li. Harmonizing safety, security and performance requirements in embedded systems. In *Design, Automation and Test in Europe (DATE'2019)*, Florence, Italy, 2019.

[7] Ludovic Apvrille and Yves Roudier. SysML-Sec: A SysML environment for the design and development of secure embedded systems. *APCOSEC, Asia-Pacific Council on Systems Engineering*, pages 8–11, 2013.

[8] Michael Barbehenn. A note on the complexity of dijkstra's algorithm for graphs with weighted vertices. *IEEE transactions on computers*, 47(2):263, 1998.

[9] Jan Baumeister, Bernd Finkbeiner, Sebastian Schirmer, Maximilian Schwenger, and Christoph Torens. Rtlola cleared for takeoff: monitoring autonomous aircraft. In *International Conference on Computer Aided Verification*, pages 28–39. Springer, 2020.

[10] Yoann Blein. *ParTraP: A Language for the Specification and Runtime Verification of Parametric Properties*. PhD thesis, Université Grenoble Alpes, 2019.

[11] Jens Brandenburg and Benno Stabernack. Simulation-based hw/sw co-exploration of the concurrent execution of hevc intra encoding algorithms for heterogeneous multi-core architectures. *Journal of Systems Architecture*, 77:26–42, 2017.

[12] Richard Buchmann, Frédéric Pétrot, and

Alain Greiner. Fast cycle accurate simulator to simulate event-driven behavior. In *International Conference on Electrical, Electronic and Computer Engineering, 2004. ICEEC'04.*, pages 35–38. IEEE, 2004.

[13] Xi Chen, Harry Hsieh, and Felice Balarin. Verification approach of metropolis design framework for embedded systems. *International Journal of Parallel Programming*, 34(1):3–27, 2006.

[14] Xi Chen, Harry Hsieh, Felice Balarin, and Yosinori Watanabe. Automatic trace analysis for logic of constraints. In *Proceedings of the 40th annual Design Automation Conference*, pages 460–465, 2003.

[15] Xi Chen, Harry Hsieh, Felice Balarin, and Yosinori Watanabe. Logic of constraints: A quantitative performance and functional constraint formalism. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(8):1243–1255, 2004.

[16] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. Using symbolic execution for verifying safety-critical systems. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 142–151, 2001.

[17] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Tessla: temporal stream-based specification language. In *Brazilian Symposium on Formal Methods*, pages 144–162. Springer, 2018.

[18] David Currie, Xiushan Feng, Masahiro Fujita, Alan J Hu, Mark Kwan, and Sreeranga Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *International Journal of Parallel Programming*, 34(1):61–91, 2006.

[19] Ben d'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. Lola: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174. IEEE, 2005.

[20] Joshua Heneage Dawes. *Towards Automated Performance Analysis of Programs by Runtime Verification*. PhD thesis, Manchester U., 2021.

[21] Julien Deantoni. *Towards Formal System Modeling: Making Explicit and Formal the Concurrent and Timed Operational Semantics to Better Understand Heterogeneous Models*. PhD thesis, Université Côte d'Azur, CNRS, I3S, France, 2019.

[22] Julien DeAntoni and Frédéric Mallet. Timesquare: Treat your models with logical time. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 34–41. Springer, 2012.

[23] Julien DeAntoni, Frédéric Mallet, Frédéric Thomas, Gonzague Reydet, Jean-Philippe Babau, Chokri Mraidha, Ludovic Gauthier, Laurent Rioux, and Nicolas Sordon. Rt-simex: retro-analysis of execution traces. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 377–378, 2010.

[24] Andrea Enrici, Letitia Li, Ludovic Apvrille, and Dominique Blouin. A tutorial on ttool. *DIPLODOCUS: an Open-source Toolkit for the Design of Data-flow Embedded Systems*, 2018.

[25] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. *Engineering dependable software systems*, pages 141–175, 2013.

[26] Yliès Falcone, Srdjan Krstić, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. *International Journal on Software Tools for Technology Transfer*, 23(2):255–284, 2021.

[27] Michael Fisher, Viviana Mascardi, Kristin Yvonne Rozier, Bernd-Holger Schlingloff, Michael Winikoff, and Neil Yorke-Smith. Towards a framework for certification of reliable autonomous systems. *Autonomous Agents and Multi-Agent Systems*, 35(1):1–65, 2021.

[28] Radek Fujdiak, Petr Mlynek, Petr Blazek, Maros Barabas, and Pavel Mrnustik. Seeking the relation between performance and security in modern systems: Metrics and measures. In *2018 41st International Conference*

on Telecommunications and Signal Processing (TSP), pages 1–5. IEEE, 2018.

[29] Daniel M Gordon and Peter Kemper. On clustering simulation traces. In *Proceedings Eighth International Workshop on Performability Modelling of Computer and Communication Systems (PMCCS-8 2007). Edinburgh, Scotland, UK*, 2007.

[30] Thomas Gruber, Christoph Schmittner, Martin Matschnig, and Bernhard Fischer. Co-engineering-in-the-loop. In *International Conference on Computer Safety, Reliability, and Security*, pages 151–163. Springer, 2018.

[31] Damien Hedde and Frédéric Pétrot. A non intrusive simulation-based trace system to analyse multiprocessor systems-on-chip software. In *2011 22nd IEEE International Symposium on Rapid System Prototyping*, pages 106–112. IEEE, 2011.

[32] Fazilat Hojaji, Tanja Mayerhofer, Bahman Zamani, Abdelwahab Hamou-Lhadj, and Erwan Bousse. Model execution tracing: a systematic mapping study. *Software and Systems Modeling*, 18(6):3461–3485, 2019.

[33] Oleg Iegorov, Vincent Leroy, Alexandre Termier, Jean-François Méhaut, and Miguel Santana. Data mining approach to temporal debugging of embedded streaming applications. In *2015 International Conference on Embedded Software (EMSOFT)*, pages 167–176. IEEE, 2015.

[34] Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D Hämäläinen, Jouni Riihimäki, and Kimmo Kuusilinna. Uml-based multiprocessor soc design framework. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2):281–320, 2006.

[35] Peter Kemper and Carsten Tepper. Trace based analysis of process interaction models. In *Proceedings of the Winter Simulation Conference, 2005.*, pages 10–pp. IEEE, 2005.

[36] Peter Kemper and Carsten Tepper. Automated analysis of simulation traces-separating progress from repetitive behavior. In *Fourth International Conference on the Quantitative Evaluation of Systems (QEST 2007)*, pages 101–110. IEEE, 2007.

[37] Volker Kemper and Carsten Tepper. Trace analysis-gain insight through modelchecking and cycle reduction. Technical report, SFB 559, 2006.

[38] Bart Kienhuis, Ed F Deprettere, Pieter Van der Wolf, and Kees Vissers. A methodology to design programmable embedded systems. In *International Workshop on Embedded Computer Systems*, pages 18–37. Springer, 2001.

[39] Bart Kienhuis, F Deprettere, Pieter van der Wolf, and Kees Vissers. The y-chart approach. In *Embedded processor design challenges*, page 18. Springer, 2002.

[40] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[41] Daniel Knorreck, Ludovic Apvrille, and Renaud Pacalet. Fast simulation techniques for design space exploration. In *International Conference on Objects, Components, Models and Patterns*, pages 308–327. Springer, 2009.

[42] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, page 207–218, New York, NY, USA, 1981. Association for Computing Machinery.

[43] Ruth Malan and Dana Bredemeyer. Defining non-functional requirements, 2001.

[44] Peter Marwedel. *Evaluation and Validation*, pages 239–293. Springer International Publishing, Cham, 2021.

[45] Aleksandar Matović. Case studies on modeling security implications on safety. Independent thesis Advanced level (degree of Master (One Year)), Malardalen University, School of Innovation, Design and Engineering., 2019.

[46] Generoso Pagano, Damien Dosimont, Guillaume Huard, Vania Marangozova-Martin, and Jean-Marc Vincent. Trace management and analysis for embedded systems. In *2013 IEEE 7th International Symposium on Embedded Multicore Socs*, pages 119–122. IEEE, 2013.

[47] Generoso Pagano and Vania Marangozova-Martin. Soc-trace infrastructure. 2012.

[48] Ivan Perez, Frank Dedden, and Alwyn Goodloe. Copilot 3. Technical report, Technical Report NASA/TM-2020-220587, National Aeronautics and Space . . . , 2020.

[49] Luigi Pomante, Vittoriano Muttillo, Bohuslav Krena, Tomás Vojnar, Filip

Veljkovic, Pacome Magnin, Martin Matschnig, Bernhard Fischer, Jabier Martinez, and Thomas Gruber. The AQUAS ECSEL project aggregated quality assurance for systems: Co-engineering inside and across the product life cycle. *Microprocess. Microsystems*, 69:54–67, 2019.

[50] Kristin Yvonne Rozier. From simulation to runtime verification and back: Connecting single-run verification techniques. In *2019 Spring Simulation Conference (SpringSim)*, pages 1–10. IEEE, 2019.

[51] G Tepper and P Kemper. Traviando-debugging simulation traces with message sequence charts. In *Third International Conference on the Quantitative Evaluation of Systems-(QEST'06)*, pages 135–136. IEEE, 2006.

[52] Daian Yue, Vania Joloboff, and Frédéric Mallet. Trap: trace runtime analysis of properties. *Frontiers of Computer Science*, 14(3):143201, 2020.

[53] Bowen Zheng, Peng Deng, Rajasekhar Anguluri, Qi Zhu, and Fabio Pasqualetti. Cross-layer codesign for secure cyber-physical systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(5):699–711, 2016.

[54] Maysam Zoor, Ludovic Apvrille, and Renaud Pacalet. Execution trace analysis for a precise understanding of latency violations. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 123–133. IEEE, 2021.