Execution Trace Analysis for a Precise Understanding of Latency Violations

Maysam Zoor, Ludovic Apvrille, Renaud Pacalet

Execution Trace Analysis for a Precise Understanding of Latency Violations

Maysam Zoor
LTCI, Télécom Paris,
Institut Polytechnique de Paris
Sophia-Antipolis, France
maysam.zoor@telecom-paris.fr

Ludovic Apvrille
LTCI, Télécom Paris,
Institut Polytechnique de Paris
Sophia-Antipolis, France
ludovic.apvrille@telecom-paris.fr

Renaud Pacalet
LTCI,Télécom Paris,
Institut Polytechnique de Paris
Sophia-Antipolis, France
renaud.pacalet@telecom-paris.fr

***Abstract*—Despite the amount of proposed works for the verification of diverse model properties, understanding the root cause of latency requirements violation in execution traces is still an open-issue especially for complex HW/SW system-level designs: is it due to an unfavorable real-time scheduling, to contentions on buses, to the characteristics of functional algorithms or hardware components? This identification is particularly at stake when adding new features in a model, e.g., a new security countermeasure. The paper introduces 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 interest of our approach.**

***Index Terms*—Embedded Systems, Execution Trace Analysis, Dependency Graph, MBSE, Timing analysis, Simulation**

I. 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 [1]. 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 [2], to debug it [2] [3], to perform model checking [4], to analyze timings, to detect data races [5] or perform other verifications [4]. References [6], [7], [8], [9], [10], [11] and [12] rely on simulation traces for performance analysis. Most of these approaches focus on verifying timing constraints and properties, statistical evaluation, bottleneck analysis and deadlock/fault detection.

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 (memory copies, 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 classification of the different transactions of the trace (obligatory, optional, contention, no contention, etc.) so as to help designers to decide how to update the system if a property (e.g., latency) 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 model allocation (to execute a function on a different processor, to use other communication facilities between processors). Again, PLAN helps spotting the property violation reasons so it helps taking model update decisions.

Section II discusses different execution traces analysis approaches. Then, Section III and Section IV formally define different stages of PLAN. A motor drive use case studied in the scope of the H2020 AQUAS project illustrates our contribution in Section V before the conclusion in Section VI.

II. Related work

Embedded systems must comply with functional and nonfunctional requirements like system safety, security, performance, reliability, etc. [13] [14]. 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 [15] [16] [17] while runtime verification approaches detect property violations by monitoring the system during execution [16]. 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. A comparison between simulation and runtime verification is presented in [18]. According to [18], simulation takes as input a model and outputs a set of executions on which statistics can be computed or requirements verified while runtime verification takes an execution trace and a requirement as inputs and outputs a verdict (true or false) based on the evaluation of the requirement over the trace. Thus, the purpose of simulation is different from the one of runtime verification [18]. While simulation is used to enhance the system in the design stage before deployment, runtime verification is used to detect faults in the system during operation and take required actions.

A. Simulation traces analysis

Traviando [19] is an example of a software tool used for simulation traces analysis. It provides qualitative (e.g., Linear Time Logic (LTL) model checking) and quantitative (e.g., statistical evaluation, bottleneck analysis, deadlock detection) trace analysis [19]. The analysis aims to attract the attention of the designer to sections of traces that correspond to extraordinary model behaviors. The traces corresponding to these behaviors are highlighted in the Message Sequence Chart (MSC) output [20].

The RT-Simex [21] project uses a set of code instrumentation tools to analyze and verify timing constraints and locate faults of parallel embedded code [7]. Real time constraints on UML models are specified using MARTE time models and the Clock Constraint Specification Language (CCSL) library [22]. Simulation traces in Open Trace Format (OTF) are studied to check if the specified real time constraints are met. TimeSquare [23] has been used in RT-Simex. It targets system designs based on MARTE model and CCSL. TimeSquare analyzes clock constraints and provides feedback during the simulation.

Chen et al. [9] suggest to analyze simulation traces of systems, including hardware/software models, to check if functional and performance constraints expressed in Logic of Constraints (LoC) [24] 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 [10]. 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 [10].

The TRAP tool [25] 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 [25]. 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.

B. Execution traces analysis

An extensive survey of runtime verification approaches applied to hard real-time distributed avionics is presented in [26]. Temporal Stream-based Specification Language (TeSSLa) [27] is an example of 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 [27], thus enforcing events ordering and easing timing analysis between events.

The Copilot language [16] 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 [28] 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 [29].

However, detecting the violation of critical safety properties in operation is not acceptable [30]. Thus, runtime analysis must be used firstly for unexpected events while requirements are expected to be verified in an earlier stage of the PLC.

Nevertheless, to the best of our knowledge, if some of the aforementioned works can detect violations of latency requirements of high-level allocation models, they do not explain why they are violated. The approach introduced in this paper does not require the use of instrumentation, data mining or any external tool. It 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. This approach can be used to advise a designer on how to enhance a high-level allocation model to satisfy a latency requirement. This is achieved by indicating which software/hardware components in the model contributed to the latency between events.

III. Precise latency analysis approach

This section presents the general approach and formalizes our models.

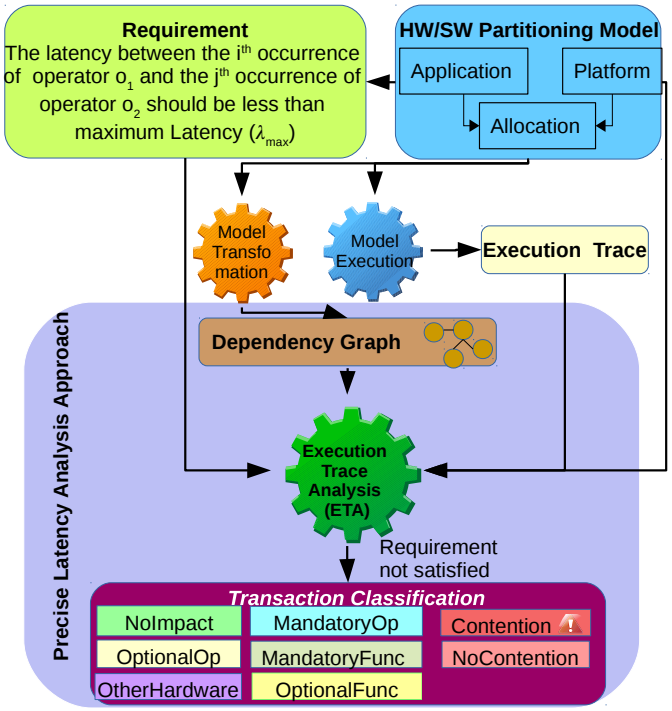

Fig. 1: Overview of the PLAN method

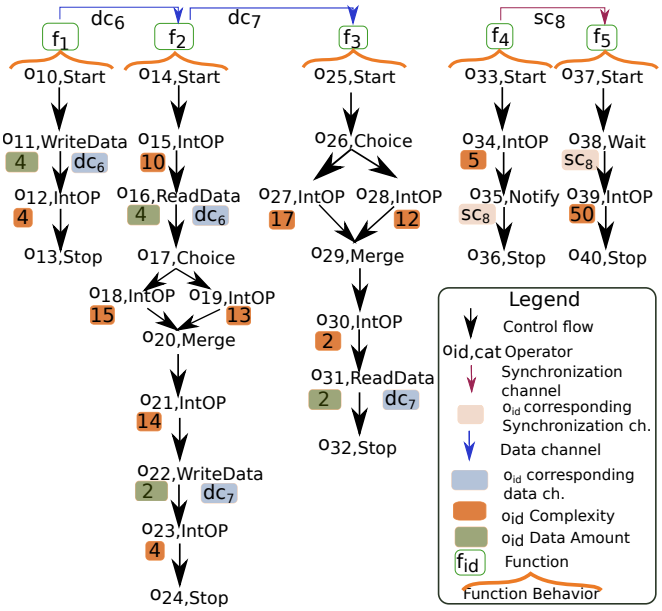

Fig. 2: Graphical representation of an application model

A. General approach

Our trace analysis approach is given in Fig. 1. PLAN takes as input a system-level model, a latency requirement, and an execution trace of the model. 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 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 produces a classification of the transactions of the execution trace.

B. System model

Throughout this section, we use $\mathrm{id}(p)$ and $\mathrm{name}(p)$ functions to obtain respectively the unique identifier and name of a given parameter p. All elements of a HW/SW partitioning model belong to a unique category obtained with $\mathrm{cat}(e)$ where e is an element.

Definition 1. HW/SW partitioning model

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

1) Application:

Definition 2. Application model

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

Fig. 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 links a writing function f_1 to a reading function f_2 and is denoted $cc_{f_1,f_2} \in \mathcal{CC},$. A communication channel can be of type "data channel" or "synchronization channel".

The set of all data channels of a system is denoted $\mathcal{DC}$ and the set of all synchronization channels of a system is denoted $\mathcal{SC}$ ($\mathcal{CC} = \mathcal{DC} \cup \mathcal{SC}$). Fig. 2 shows two data channels (dc_6, dc_7) and one synchronization channel (sc_8).

Synchronization channels $sc_{f_1,f_2} \in \mathcal{SC}$ are blocking notify - blocking wait finite FIFOs. Data channels $dc_{f_1,f_2} \in \mathcal{DC}$ are blocking read and blocking write finite FIFOs; they model the quantity of exchanged data, not the data values which are abstracted.

Definition 4. Function

A function $f = \langle V_f, O_f, C_f \rangle \in F$ is defined by a set of variables V_f, a set of operators O_f and a set of unidirectional control flows connections C_f between operators. Fig. 2 shows function operators and control flow connections between them (arrows).

Property 1. Control flow connection. There can be at most one control flow connection between two operators.

Definition 5. Operator

Operators belong to one of the following categories: Start, Stop, Choice, Merge, IntOp, Set, WriteData, ReadData, Notifyor Wait.

- Start, Stop: start/end of control flow.

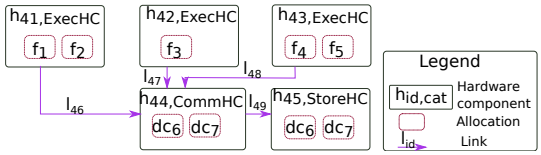

Fig. 3: Graphical representation of an allocation model

- Choice: selects one next control flow among the one whose guard is true. If no guard is true, then the choice operator blocks. A choice operator is the only category that 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: specifies the complexity of an algorithm e.g., a given number of integer operations.
- Set: sets a variable to a new value.
- 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.

If o is a Notify or Wait operator we denote $o.sc$ the corresponding synchronization channel. Similarly, if o is a ReadData or WriteDataoperator we denote $o.dc$ the corresponding data channel.

Property 2. Restriction on O_f. The set of operators O_f of a function f must contain one and only one operator whose category is Start: $\forall f \in F, \exists! o \in O_f, \text{cat}(o) = \text{Start}$.

Definition 6. Execution flow

An execution flow e_f of a function f is a sequence of operators $\langle o_1, \ldots, o_n \rangle$ starting with a Start operator ($\text{cat}(o_1) = \text{Start}$). Between all adjacent operators of an execution flow there must be a control flow. For example, in Fig. 2, $e_{f1} = \langle o_{10}, o_{11}, o_{12}, o_{13} \rangle$.

Property 3. Well-formed execution flows. For each $o \in O_f$, there must exist at least one execution flow with o.

2) Platform:

Definition 7. Platform model

A platform model $\mathcal{P} = \langle \mathcal{H}, \mathcal{L}, \mathcal{C}_P \rangle$ is a set of hardware components $\mathcal{H}$, a set of links $\mathcal{L}$ and a set of communication paths $\mathcal{C}_P$. A hardware component represents the physical electronic component plus its support software, e.g., an operating system for a processor. A hardware component is either an execution, a communication or a storage component: $\mathcal{H} = \mathcal{H}_E \cup \mathcal{H}_C \cup \mathcal{H}_S$, where $\mathcal{H}_E$, $\mathcal{H}_C$ and $\mathcal{H}_S$ denote the set of hardware components of execution, communication and storage types, respectively. Fig. 3 depicts 3 $\mathcal{H}_E$ 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 the links l_{46}, l_{47},l_{48} and l_{49}.

Definition 8. Links

A link is a pair (h_1, h_2) of hardware components, one of which is a communication one: $\mathcal{L} \subseteq \mathcal{H}_C \times \mathcal{H} \cup \mathcal{H} \times \mathcal{H}_C$.

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

Definition 9. Communication path

A write path π_w is an ordered sequence of hardware components linked together, starting with an execution component and ending with a storage component. A read path π_r is an ordered sequence of hardware components linked together, starting with a storage component and ending with an execution component.

$$\pi_w = \langle h_1, \ldots, h_m \rangle, \forall 1 \leq i \leq m-1, (h_i, h_{i+1}) \in \mathcal{L},$$
$$h_1 \in \mathcal{H}_E, h_m \in \mathcal{H}_S, h_{2 \leq j \leq m-1} \in \mathcal{H}_C$$
$$\pi_r = \langle h_1, \ldots, h_n \rangle, \forall 1 \leq i \leq n-1, (h_i, h_{i+1}) \in \mathcal{L},$$
$$h_1 \in \mathcal{H}_S, h_n \in \mathcal{H}_E, h_{2 \leq j \leq n-1} \in \mathcal{H}_C$$

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

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 they have the same ending and starting storage component.

3) Allocation:

Definition 10. 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 $\mathcal{A} = \langle \overrightarrow{\mathcal{A}_f}, \overrightarrow{\mathcal{A}_{dc}} \rangle$ where $\overrightarrow{\mathcal{A}_f} : \mathcal{F} \mapsto \mathcal{H}_E$ is a function that maps each function $f \in \mathcal{F}$ to a hardware component $h \in \mathcal{H}_E$ and $\overrightarrow{\mathcal{A}_{dc}} : \mathcal{DC} \mapsto \mathcal{C}_P$ is a function that maps each data channel $dc \in \mathcal{DC}$ to a communication path $c_P \in \mathcal{C}_P$.

Fig. 3 suggests an allocation of the application of Fig. 2 to a platform with one communication, one storage and 3 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_{41} while data channels dc_6 and dc_7 are allocated to h_{44} and h_{45} of a c_P.

Property 4. Valid allocation. Let $f_1, f_2 \in \mathcal{F}$ and $h_1, h_2 \in \mathcal{H}_E$ such that $\overrightarrow{\mathcal{A}_f}(f_1) = h_1, \overrightarrow{\mathcal{A}_f}(f_2) = h_2$ and there is a data channel $dc_{f_1,f_2} \in \mathcal{DC}$ between f_1 and f_2. Then, dc_{f_1,f_2} can be allocated to a communication path $c_P = \langle \pi_w, \pi_r \rangle$ if and only if the starting execution component of π_w is h_1 and the ending execution component of π_r is h_2.

C. Trace generation

We now define the notion of traces and occurrences of operators from a model execution.

Definition 11. System execution, execution trace

A system execution of a HW/SW partitioning model m for a time interval $[0, \tau]$ returns an execution trace $x = \langle t_1, \ldots, t_k \rangle$ where t_i is an execution transaction.

Table. I gives one possible execution trace for the application model of Fig. 2, with allocation model of Fig. 3, for a time interval $[0, 50]$. The `id` field corresponds to the operator indexes of Fig. 2. The execution of Start, Stop, Choice, Merge and Set operators is assumed to take no time. Thus, their execution is not represented in the execution trace.

Definition 12. Execution transaction

An execution transaction $t = \langle \tau_s^t, \tau_e^t, h^t, o^t \rangle$ represents the execution of an operator o^t on a hardware component h^t, with the start time τ_s^t and the end time τ_e^t.

Definition 13. i^{th} occurrence of operator o in execution trace x

To retrieve all the transactions corresponding to a specific operator in an execution trace x, a function AllTransWithOp takes as an input an execution trace and an operator and returns a set of transactions containing all the transaction of operator o. Formally, $\text{AllTransWithOp}(x, o) = \{t \in x \mid o^t = o\}$.

We denote by $t_{o,i}^x$ the transaction of the i^{th} occurrence, in increasing order of start times, of operator o in an execution trace x. For simplicity $t_{o,i}^x$ is abbreviated as $t_{o,i}$.

TABLE I: Execution trace in tabular format

hc	41							
id	15	11	12	16	18	21	22	23
starttime	0	10	11	15	16	31	45	46
endtime	10	11	15	16	31	45	46	50

hc	43				42			44			
id	34	35	38	39	27	30	31	11	16	22	31
τ_s^t	0	5	6	7	0	17	46	10	15	45	46
τ_e^t	5	6	7	57	17	19	47	11	16	46	47

D. Requirements on model execution

Generally speaking, a requirement expresses a property representing a goal or an anti-goal that the system must satisfy. A requirement can be explicitly linked to execution traces of a systems.

Definition 14. Maximum latency requirement

A maximum latency requirement $r = \langle o_1, i, o_2, j, \lambda_{max} \rangle$ specifies a maximum delay between the start and end times of two operator occurrences: $\tau_e^{t_{o_2,j}} - \tau_e^{t_{o_1,i}} \leq \lambda_{max}$.

For instance, with the execution trace of Table. I, latency requirement $r = \langle o_{11}, 1, o_{31}, 1, 35 \rangle$ is violated because $\tau_e^{t_{o_{31},1}} - \tau_s^{t_{o_{11},1}} = 47 - 10 = 37 > 35$.

E. Execution trace analysis

When a requirement is not satisfied, execution trace analysis classifies transactions in order to help understanding the root causes of the violation. For this, the relations between transactions, and thus between operators, has to be computed.

Definition 15. Dependencies between model elements

For any model elements m_1 and m_2, $\overline{m_1 m_2} \in D_M$ denotes that $m2$ depends on m_1. Function $\text{getDepType}(\overline{m_1 m_2})$ returns the type of dependency between m_1 and m_2. Dependency types can be synChDep, writeDataChDep, readDataChDep, controlFlowDep, startDep, linkDep, dataChAllocDepor funAllocDep.

This list relates to communication dependencies (synChDep, writeDataChDep or readDataChDep), to control flow dependencies (controlFlowDep), to behavioral dependencies of functions (startDep), to architecture dependencies e.g., connection between hardware components (linkDep) and to allocation dependencies of functions and communications (dataChAllocDep, funAllocDep). Table II gives the formal definitions of these dependencies, and further explains them below.

TABLE II: Dependencies between model elements

Dependencies	***Formal definition***	***Example***
synChDep between operators o_1 and o_2	$\text{cat}(o_i) = \text{Notify} \wedge$ $\text{cat}(o_j) = \text{Wait} \wedge$ $o_i.sc = o_j.sc$	$\overline{o_{35} o_{38}}$
writeDataChDep between operator o and data channel d	$\text{cat}(o) = \text{WriteData} \wedge$ $o.dc = d$	$\overline{o_{11} dc_6}$
readDataChDep between data channel d and operator o	$\text{cat}(o) = \text{ReadData} \wedge$ $o.dc = d$	$\overline{dc_6 o_{16}}$
controlFlowDep between operators o_1 and o_2	$(o_1, o_2) \in C_f$	$\overline{o_{11} o_{12}}$
startDep between function f and Start operator o_s	$o_s \in O_f \wedge$ $\text{cat}(o_s) = \text{Start}$	$\overline{f_1 o_{10}}$
linkDep between hardware components h_1 and h_2	$(h_1, h_2) \in \mathcal{L}$	$\overline{h_{41} h_{44}}$
dataChAllocDep between hardware component h and data channel d	$h \in \overrightarrow{\mathcal{A}_{dc}}(d)$	$\overline{h_{44} dc_6}$
funAllocDep between hardware component h and function f	$\overrightarrow{\mathcal{A}_f}(f) = h$	$\overline{h_{41} f_1}$

Definition 16. Model dependency path

The dependency relation is transitive. $\overrightarrow{m_1 m_2}$ denotes a dependency path between m_1 and m_2. We denote DP the set of all dependency paths.

Problem 1. Execution trace analysis

To understand the reasons of a maximum latency re-

quirement violation we classify all transactions of an execution trace based on their impact on the latency between the two operators of the requirement.

Definition 17. Execution trace partition

To solve Problem 1, we classify transactions in an execution trace into the following sets: *no impact* (NI), *mandatory operator* (MOP), *optional operator* (OOP), *mandatory function* (MF), *optional function* (OF), *contention* (C), *no contention* (NC) and *other hardware* (OH) where, $\{NI, MOP, OOP, MF, OF, C, NC, OH\}$ is a partition of the execution trace.

Also, our analysis technique assumes the following assumptions:

Hypothesis 1. Limitation on the occurrence of operators. Since our analysis depends on the notion of dependency path, we cannot analyze execution traces in which not all operators of the dependency path $\overrightarrow{o_1 o_2}$ selected by the execution engine were executed.

Hypothesis 2. No cycles in function control flows.

Hypothesis 3. No interleaving between transactions. Our analysis works if and only if all transactions corresponding to operators in the same dependency path are not interleaved.

Given an execution trace x and a maximum latency requirement $r = \langle o_1, i, o_2, j, \lambda_{max} \rangle$, Table III gives the classification conditions for a transaction $t_{o,k}$. The examples presented in Table III correspond to the maximum latency requirement $r = \langle o_{11}, 1, o_{31}, 1, 35 \rangle$ applied to the example system of Fig. 2 and Fig. 3.

A transaction $t_{o,k}$ belongs to the NI set if the end time of $t_{o,k}$ is less than the start time of $t_{o_1,i}$ or the start time of $t_{o,k}$ is greater than the end time of $t_{o_2,j}$. In case $t_{o,k}$ has delayed another transaction $t_{q,n}$ that will create a contention between $o_{1,i}$ and $o_{2,j}$, the transaction $t_{q,n}$ is classified in C set.

A transaction $t_{o,k}$ belongs to MOP set if it does not belong to the NI set and if all dependency paths between o_1 and o_2 contain two dependency paths: one from o_1 to o and the other from o to o_2. However, if there exist dependency paths between o_1 and o_2 that does not contain $\overrightarrow{o_1 o}$ or $\overrightarrow{o o_2}$ while others contain $\overrightarrow{o_1 o}$ and $\overrightarrow{o o_2}$ then $t_{o,k}$ belongs to the OOP set.

In Fig. 2, assuming that o_{11} writes and o_{16} reads from dc_6 and o_{22} writes and o_{32} reads from to dc_7, there are two dependency paths between o_{11} and o_{31}, $\overrightarrow{o_{11} o_{31}}^1$ and $\overrightarrow{o_{11} o_{31}}^2$.

$$\overrightarrow{o_{11} o_{31}}^1 = \overline{o_{11} dc_6}, \overline{dc_6 o_{16}}, \overline{o_{16} o_{17}}, \overline{o_{17} o_{18}}, \overline{o_{18} o_{20}}, \overline{o_{20} o_{21}}, \overline{o_{21} o_{22}}, \overline{o_{22} dc_7}, \overline{dc_7 o_{31}} \quad (1)$$

$$\overrightarrow{o_{11} o_{31}}^2 = \overline{o_{11} dc_6}, \overline{dc_6 o_{16}}, \overline{o_{16} o_{17}}, \overline{o_{17} o_{19}}, \overline{o_{19} o_{20}}, \overline{o_{20} o_{21}}, \overline{o_{21} o_{22}}, \overline{o_{22} dc_7}, \overline{dc_7 o_{31}} \quad (2)$$

TABLE III: Definition of impact sets assuming a trace x and a maximum latency requirement $\langle o_1, i, o_2, j, \lambda_{max} \rangle$

Name	***Formal definition***	***Example***
$t_{o,k} \in NI$	$\tau_e^{t_{o,k}} < \tau_s^{t_{o_1,i}} \vee \tau_s^{t_{o,k}} > \tau_e^{t_{o_2,j}}$	$t_{o_{34},1}$
$t_{o,k} \in MOP$	$\forall \overrightarrow{o_1 o_2} \in DP, \overrightarrow{o_1 o} \subset \overrightarrow{o_1 o_2} \wedge$ $\overrightarrow{o o_2} \subset \overrightarrow{o_1 o_2} \wedge t_{o,k} \notin NI$	$t_{o_{16},1}$
$t_{o,k} \in OOP$	$\exists \overrightarrow{o_1 o_2}^1, \overrightarrow{o_1 o_2}^2 \in DP$ $\Big(\big(\overrightarrow{o_1 o} \subset \overrightarrow{o_1 o_2}^1 \wedge \overrightarrow{o_1 o} \not\subset \overrightarrow{o_1 o_2}^2 \big) \vee$ $\big(\overrightarrow{o o_2} \in \overrightarrow{o_1 o_2}^1 \wedge \overrightarrow{o o_2} \notin \overrightarrow{o_1 o_2}^2 \big) \Big) \wedge$ $t_{o,k} \notin NI$	$t_{o_{18},1}$
$t_{o,k} \in MF$	$\forall \overrightarrow{o_s o_2} \in DP,$ $\overrightarrow{o_s o} \subset \overrightarrow{o_s o_2} \wedge \overrightarrow{o o_2} \subset \overrightarrow{o_s o_2}$ $\wedge \text{cat}(o_s) = \text{Start}$ $\wedge t_{o,k} \notin NI \cup MOP \cup OOP$	$t_{o_{30},1}$
$t_{o,k} \in OF$	$\exists \overrightarrow{o_s o_2}^1, \overrightarrow{o_s o_2}^2 \in DP,$ $\big(\overrightarrow{o_s o} \subset \overrightarrow{o_s o_2}^1 \wedge \overrightarrow{o_s o} \not\subset \overrightarrow{o_s o_2}^2 \big)$ $\vee \big(\overrightarrow{o o_2} \subset \overrightarrow{o_s o_2}^1 \wedge \overrightarrow{o o_2} \not\subset \overrightarrow{o_s o_2}^2 \big)$ $\wedge \text{cat}(o_s) = \text{Start}$ $\wedge t_{o,k} \notin NI \cup MOP \cup OOP$	$t_{o_{27},1}$
$t_{o,k} \in C$	$\forall (\beta, \tau_s) \in \text{getCT}(m, x, h^{t_{o,k}})$ $h^{t_{o,k}} \in \text{getDPHC}(m, o_1, o_2) \wedge$ $\tau_s^{t_{o,k}} >= \beta \wedge \tau_s^{t_{o,k}} <= \tau_s \wedge t_{o,k} \notin$ $NI \cup MOP \cup OOP \cup MF \cup OF$	$t_{o_{12},1}$
$t_{o,k} \in NC$	$\forall (\beta, \tau_s) \in \text{getCT}(m, x, h^{t_{o,k}})$ $h^{t_{o,k}} \in \text{getDPHC}(m, o_1, o_2) \wedge$ $!(\tau_s^{t_{o,k}} >= \beta \wedge \tau_s^{t_{o,k}} <= \tau_s) \wedge t_{o,k} \notin$ $NI \cup MOP \cup OOP \cup MF \cup OF$	$t_{o_{23},1}$
$t_{o,k} \in OH$	Other	$t_{o_{39},1}$

Thus, $t_{16,1}, t_{16,2}, t_{21,1}, t_{22,1}, t_{22,2} \in MOP$ and $t_{18,1} \in OOP$.

A transaction $t_{o,k}$ belongs to MF set if it does not belong to NI, MOP or OOP sets and if all dependency paths between o_s and o_2, where o_s is the Start operator of function f, contain two dependency paths: one from o_s to o and the other from o to o_2. However, if there exist dependency paths between o_s and o_2 that does not contain $\overrightarrow{o_s o}$ or $\overrightarrow{o o_2}$ while others contain $\overrightarrow{o_s o}$ and $\overrightarrow{o o_2}$ then $t_{o,k}$ belongs to OF set.

Thus, in Table. I, and according to $\overrightarrow{o_{25} o_{31}}^1$ and $\overrightarrow{o_{25} o_{31}}^2$, $t_{27,1} \in OF$ and $t_{30,1} \in MF$.

To identify transactions that belong to contention set, we need to define two functions.

Definition 18. Dependency path hardware components

Let us consider all dependency paths between two operators o_1 and o_2. Function getDPHC takes as argument a model m and two operators o_1 and o_2 and returns a subset H_{Dep} of hardware components in a platform model. This subset H_{Dep} contains execution hardware components on which functions containing operators on dependency paths between o_1 and o_2 are allocated and communication/storage hardware components on which

data channels on dependency paths between o_1 and o_2 are allocated. Formally, getDPHC : $(m, o_1, o_2) \mapsto H_{Dep}$. So, for an IntOp operator of a dependency path $\overrightarrow{o_1 o_2}$, an execution hardware component is added to the list and for a data channel of a dependency path $\overrightarrow{o_1 o_2}$, the hardware components of a communication path are added to the list.

Definition 19. Best start execution date

The Best Start Execution Date (BSED) is the earliest possible time that would have been obtained by executing exactly the same operators on the same dependency path but considering execution hardware components with an infinite number of cores and communication hardware components with unlimited bandwidth. Function getBSED(m, x, t) returns the BSED of a transaction t. For instance, in the model in Fig. 3, the BSED of the transactions $t_{o_{15},1}$ and transactions $t_{o_{11},1}$ is 0 since they correspond to the first operators to execute after the start of functions f_1 and f_2 respectively. The BSED of the transactions $t_{o_{16},2}$ on h_{41} is 11 since to execute operator $o_{16,2}$ on h_{41}, operators o_{14} and o_{15} in function f_2 must execute. The BSED of o_{14} is zero and thus the BSED of o_{15} is also zero. The complexity of o_{15} is 10. So, operator $o_{16,2}$ can start at 11. However, in our example, for $o_{16,2}$ to execute, $o_{11,1}$ must execute too. The best case execution of $o_{11,1}$ is 0 with duration of 1 cycle on h_{41}. Thus, $o_{11,1}$ does not impact the BSED of $o_{16,2}$ in this case. So, the BSED of operator $o_{16,2}$ is 11.

Function getCT(m, x, h) returns the list of (BSED, τ_s) pairs of all transactions that were executed for a hardware component h. Formally, getCT : $(m, x, h) \mapsto \{(\text{getBSED}(m, x, t), \tau_s^t)) \mid h^t = h\}$

A pair returned by getCT(m, x, h) such that getBSED$(m, x, t) < \tau_s^t$ indicates that the transaction is delayed.

A transaction $t_{o,k}$ belongs to C if and only if it satisfies three conditions. First, $t_{o,k}$ should not belong to NI, MOP, OOP, MF or OF sets. Second, the hardware $h^{t_{o,k}}$ should belong to the set of hardware returned by getDPHC(m, o_1, o_2). Third, $\tau_s^{t_{o,k}}$ must fall between a pair of (BSED,τ_s) in the list returned by getCT$(m, x, h^{t_{o,k}})$.

In Table. I, $\tau_s(t_{12,1}) = 11$ and $\tau_s(t_{16,2}) = 15$ and according to Definition 19, getBSED$(t_{16,2}) = 11$. Thus, a pair $(11, 15) \in$ getCT$(M, E_{M,\tau}, h(t_{16,1}))$. In our example (Fig. 3), getDPHC$(m, o_1, o_2) = \{h_{41}, h_{42}, h_{44}, h_{45}\}$ and $h(t_{12,1}) = h(t_{16,2}) = h_{41}$. Thus, $t_{12,1}$ satisfies the conditions of contention set.

When only the third condition is not satisfied, then, $t_{o,k}$ belongs to NC set. A transaction $t_{o,k}$ belongs to OH set if it does not satisfy any of the previous conditions.

IV. Dependency Graph Construction

As shown in previous section, many aspects of the partitioning models can play a role in the delaying of transactions. In particular, dependency paths are strongly involved in the classification. Thus, each time we analyze an execution transaction to know whether it is related to another transaction, we need to refer to the HW/SW partitioning model to search for dependency paths. Unfortunately, computing dependency paths is complex since many elements are involved (control flows, allocations, communication paths, ...).

As this is done for other domains just like compilers [31] [32], we suggest to rely on a dependency graph of partitioning models. This graph features all logical dependencies thus making it possible to apply well-known graph algorithms, such as the shortest path algorithm. Thus, Problem 1 can be redefined taking into consideration dependency graph G.

A. Graph definitions

Definition 20. Dependency graph

G is a dependency graph $G = (V, E)$ with V and E the vertexes and directed edges of G.

Vertex $v_{id} = \langle id, \text{functionID} \rangle$ has an id referencing a model element and a functionID referencing the related function of the model, if applicable.

Function addVertex(G, id) adds a vertex v_{id} to G. Function getVertex(G, id) returns $v_{id} \in G$ if this vertex exists, otherwise $\emptyset$. Function setFID(G, id, id_f) sets functionID of v_{id} to value id_f and getFID(G, v_{id}) gets the value of functionID of vertex v_{id}.

An edge is a couple $(v_{id}, v_{id'})$ that connects vertex v_{id} to vertex $v_{id'}$. The set of all edges E is defined as: $E = \{(v_{id}, v_{id'}) \mid v_{id}, v_{id'} \in V \wedge v_{id} \neq v_{id'}\}$.

Function addEdge$(G, v_{id}, v_{id'})$ adds a directed edge from vertex v_{id} to vertex $v_{id'}$ in graph G. Function $getEdge(G, v_{id}, v_{id'})$ returns true in case an edge exits from vertex v_{id} to vertex $v_{id'}$ in G else it returns false.

Definition 21. Graph path

In graph G, we say a path $\overrightarrow{v_{id} v_{id'}}$ exists from a vertex v_{id} to a vertex $v_{id'}$ if and only if:

$$\exists v_1, v_2, \ldots, v_n \mid \forall i = 1, \ldots, n, (v_i, v_{i+1}) \in E \wedge v_1 = v_{id} \wedge v_n = v_{id'} \quad (3)$$

The set of all paths between two vertexes v_{id} and $v_{id'}$ in G is denoted as $AP_{\overrightarrow{v_{id} v_{id'}}}$.

B. Graph generation

Algorithm 1 takes as input a partitioning model m and returns a dependency graph $G = generateGraph(m)$. The algorithm first adds vertexes for all hardware components of the platform model $\mathcal{P}$ and edges to represent links between them. The algorithm then iterates over hardware components that belong to $\mathcal{H}_C$ and $\mathcal{H}_S$ to consider data channel allocations. The algorithm finally iterates over $\mathcal{H}_E$ and adds vertexes to represent functions and operators. There, function addEdges(G, f, o) adds edges according to the operator category and to the control flow dependencies between operators. For a Start vertex corresponding to a Start operator in function f behavior, an edge is added

from the vertex corresponding to function f to the Start vertex. Also, at this step, edges to represent reading from and writing to the data channels are added. If the operator is a WriteData operator then the WriteData vertex is connected by an edge to the vertex corresponding to the data channel to which the operator writes data. Same principle applied to ReadData. For Notify and Wait operators, an edge is added from the Notify vertex to the Wait vertex to represent a synchronization channel. Synchronization channels are represented as edges in the graph while data channels are represented as vertexes since for the latter "edges to hardware components vertexes" must be added to represent their allocation to a communication path.

Algorithm 1: Generate Graph

Data: $m = \langle \mathcal{F}, \mathcal{P}, \mathcal{A} \rangle$
Result: Dependency graph G

foreach $h \in \mathcal{H}$ **do**
  addVertex$(G, \text{id}(h))$
end
▷ Links between hardware components
foreach $(h1, h2) \in \mathcal{L}$ **do**
  $v_{h1} \leftarrow$ getVertex$(G, \text{id}(h1))$
  $v_{h2} \leftarrow$ getVertex$(G, \text{id}(h2))$
  addEdge(v_{h2}, v_{h1}) ;
  addEdge(v_{h1}, v_{h2})
end
▷ Data channel allocation dependencies
foreach $h \in \{\mathcal{H}_C \cup \mathcal{H}_S\}$ **do**
  foreach $dt \in \mathcal{DC}$ ***such that*** $h \in \overrightarrow{\mathcal{A}_{dc}}(dt)$ **do**
    if *getVertex*$(G, id(dt))=\emptyset$ **then**
      addVertex$(G, \text{id}(dt))$
      $v_{dt} \leftarrow$ getVertex$(G, \text{id}(dt))$
      $v_h \leftarrow$ getVertex$(G, \text{id}(h))$
    addEdge(v_h, v_{dt})
  end
end
▷ Communication dependencies, control flow dependencies between, application dependencies
foreach $h \in \mathcal{H}_E$ **do**
  foreach $f' \in F$ ***such that*** $\overrightarrow{\mathcal{A}_f}(f') = h$ **do**
    addVertex$(G, \text{id}(f'))$
    $v_{f'} \leftarrow$ getVertex$(G, \text{id}(f'))$
    $v_h \leftarrow$ getVertex$(G, \text{id}(h))$
    addEdge$(v_h, v_{f'})$
    foreach $o \in O_{f'}$ **do**
      addVertex$(G, \text{id}(o))$;
      addEdges(G, f', o)
    end
  end
end

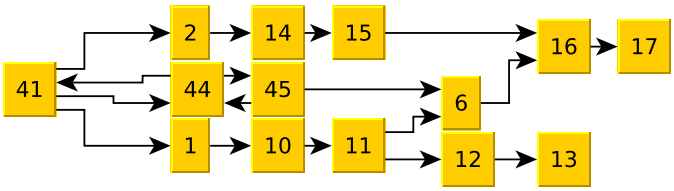

Fig. 4: An Excerpt of the Dependency Graph Corresponding to the Models in Fig. 2 and Fig. 3

C. Execution trace analysis using dependency graph

We now show how a dependency graph simplifies the classification of transactions. As a starting example, to check if there is a dependency between o_{11} and o_{16} (Figure Fig. 2), without using the graph, we had to check the behavior of the two functions f_1 and f_2 and the data channels between them. With the graph we can easily identify the dependency because an edge connects v_{11} to v_6 and another edge connects v_6 to v_{16} (Fig. 4).

Throughout this section we denote by t_1, t_2 and t, the transaction $t_{o_1,i}$, $t_{o_2,j}$ and $t_{o,k}$ respectively and by v_1,v_2 and v the corresponding vertexes. Similar to function getDPHC(m, o_1, o_2) defined in the system model, we define Function getDPHCV(G, v_1, v_2) to get a list of all vertexes corresponding to hardware components to which a vertex on the path between v_1 and v_2 is allocated. This list can be "easily" retrieved from the graph as all allocation dependencies are represented by edges.

To take the BSED into consideration in our analysis, delayTime attribute is added to the vertex attributes defined previously. The delayTime attribute is nonempty for vertexes corresponding to hardware components. It is a list of pairs of values corresponding to the BSED and the start time for every transaction executed on that hardware. Function getDelayTime(G, v_h) returns the list of (BSED,τ_s) pairs for a hardware component h.

The classification of transactions is now expressed as shown in Table IV. While transactions are still classified according to the same conditions stated previously, the difference is in how the dependencies are expressed, e.g., using vertexes. For example, a transaction t belongs to MOP if and only if it does not belong to NI set and vertex v belongs to all paths from v_1 to v_2. However, if there exist paths from v_1 to v_2 that does not contain v while other paths from v_1 to v_2 contain v, then t belongs to OOP set. A transaction t belongs to MF set if and only if vertex v belongs to all paths from v_s to v_2 where v_s corresponds to a Start operator and t does not belong to NI, MOP or OOP sets.

V. Application to UML/SysML and a Use Case

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

TABLE IV: Impact Sets Classification Using Dependency Graph

Name	*Formal Description*
$t \in NI$	$\tau_e^t < \tau_s^{t_1} \vee \tau_s^t > \tau_e^{t_2}$
$t \in MOP$	$\forall \overrightarrow{v_1v_2} \in AP_{\underset{v_1v_2}{\longrightarrow}}, v \in \overrightarrow{v_1v_2} \wedge t \notin NI$
$t \in OOP$	$\exists \overrightarrow{v_1v_2}^1, \overrightarrow{v_1v_2}^2 \in AP_{\underset{v_1v_2}{\longrightarrow}}, v \in \overrightarrow{v_1v_2}^1 \wedge$ $v \notin \overrightarrow{v_1v_2}^2 \wedge t \notin NI$
$t \in MF$	$\forall \overrightarrow{v_sv_2} \in AP_{\underset{v_sv_2}{\longrightarrow}}, v \in \overrightarrow{v_sv_2} \wedge$ $cat(o_s) = Start \wedge$ $t \notin NI \cup MOP \cup OOP$
$t \in OF$	$\exists \overrightarrow{v_sv_2}^1, \overrightarrow{v_sv_2}^2 \in AP_{\underset{v_sv_2}{\longrightarrow}}, v \in \overrightarrow{v_sv_2}^1 \wedge$ $v \notin \overrightarrow{v_sv_2}^2 \wedge cat(o_s) = Start \wedge$ $t \notin NI \cup MOP \cup OOP$
$t \in C$	$\forall (\beta, \tau_s) \in \text{getDelayTime}(G, v_{h^t}), v_{h^t} \in$ $getDPHCV(G, v_1, v_2) \wedge \tau_s^t >= \beta \wedge \tau_s^t <= \tau_s \wedge$ $t \notin NI \cup MOP \cup OOP \cup MF \cup OF$
$t \in NC$	$\forall (\beta, \tau_s) \in \text{getDelayTime}(G, v_{h^t}), v_{h^t} \in$ $getDPHCV(G, v_1, v_2) \wedge !(\tau_s^t >= \beta \wedge \tau_s^t <= \tau_s) \wedge$ $t \notin NI \cup MOP \cup OOP \cup MF \cup OF \cup C$

A. Description of the use case

A high-level view of the industrial drive system—defined in the scope of the H2020 AQUAS project [33] [34]—is shown in Fig. 5. 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* runs periodically 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*.

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 $55\mu s$.

B. SysML models

Our formal model easily maps to SysML diagrams. Functions can be defined as SysML blocks in SysML Block Definition diagrams and their communications can be captured with Internal Block Diagrams, and their behavior can be captured with SysML Activity diagrams (e.g., start, stop, choices, ...). Operators *IntOp* and *Set* are both mapped to action element in the activity diagram. UML Deployment Diagrams or SysML allocations can be used for platform and allocations.

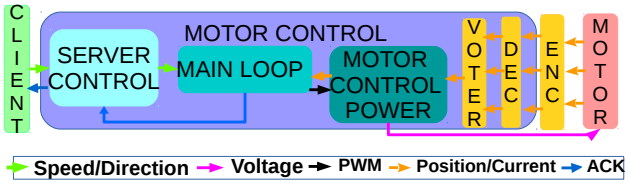

Fig. 5: Specification of the Use Case: An Industrial Drive

We selected TTool [35] and SysML-Sec [36] for modeling and simulation trace generation. SysML-Sec follows the Y-Chart approach [37]. In the application model (e.g., Fig.6), functions are modeled as blocks colored green and variables of a function are displayed inside the green block (in Fig.6, "x" is a variable in function $PWMtoPS$). Operators in the function behavior are modeled as follows: $Notify$ and $Wait$ operators are modeled in violet, $WriteData$ and $ReaData$ operators in blue, *Choice*, *merge*, *IntOp* and *Set* operators in green, *Start* and *Stop* in black. Synchronization ports are in purple while data ports are in blue. In Fig.6, a synchronization channel named run_Inter is shown between *Interrupt* and *MainLoop* functions and a synchronization channel and a data channel are shown between $PWMtoPS$ and $MotorF$ functions named $PhaseSig$ and $PStoM$ respectively.

In the platform model (Fig.7), execution, communication and storage hardware component are shown in blue, brown and green respectively. Functions are allocated to execution hardware components and data channels are allocated to communication paths. Fig.7 shows the allocation of the $MotorF$ function and the $PStoM$ data channel.

C. Model simulation and trace analysis

Simulation is one of the verification techniques available in SysML-Sec [38]. $56\mu s$ of the industrial drive execution have been simulated. $56\mu s$ has been chosen since it is the minimum duration to validate the latency requirement. The simulated hardware components run at 200MHz. The obtained simulation trace contains 11888 transactions.

We denote by o_γ the operator corresponding to the start of the main loop and by $o_{\gamma'}$ the operator corresponding to the receiving of the voltage in the motor. The two operators and their two functions are shown in Fig. 6. Operator o_γ is the Wait operator named $run\_Inter()$ in violet in $MainLoop$ and operator $o_{\gamma'}$ is reading data operator named $PMStoM$ in blue in $MotorF$.

The start time of $t_{\gamma,1}$ in the simulation trace is "2" and the end time of the $t_{\gamma',1}$ is "11115". Thus, the latency in

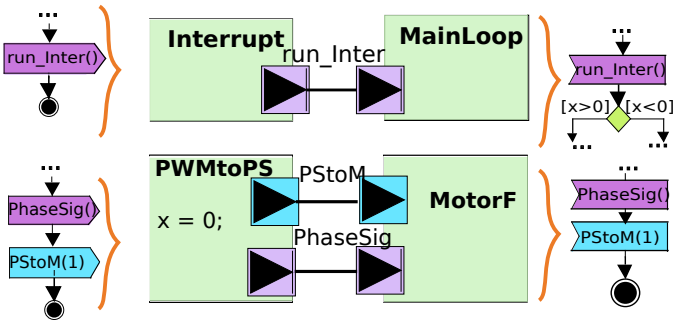

Fig. 6: An Excerpt of the Application Model of The Use Case

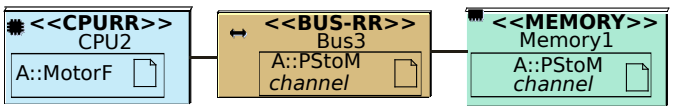

Fig. 7: An Excerpt of the Allocation Model of The Use Case

this case is 11113 cycles (i.e., 55.56μs). Thus, the requirement is not satisfied, thus leading to use PLAN. PLAN is implemented in TTool: the transactions are classified and displayed in a table. Transactions in the graphical table (e.g., Fig. 8) are displayed according to the hardware that executed them and the time of execution. Since transactions are colored according to their category, contention transactions are easy to identify. For example, transactions in *MandatoryOp* set are colored in green, those in *Contention*, *NoContention* and *MandatoryFunc* sets are colored in red, orange and gray respectively. In our use case, after generating a dependency graph of 552 vertexes and 965 edges and running the execution trace analysis, contentions were spotted on the execution hardware component on which the *Motor Control* functions are allocated (Fig. 8). The contention is due to the *Server Control* function processing data to write acknowledgment to the *Client* while the encryption function was ready to execute but its the resource was busy.

To resolve this execution contention, an execution hardware component is added to the platform and the *Server Control* function is now allocated to it. Running PLAN again, the latency is now equal to 10604 cycles (i.e., 53.02μs) since the start time of $t_{\gamma,1}$ is noted as 2 and the end time of the $t_{\gamma',1}$ is 10606. Thus, the latency requirement is satisfied. To see how the transaction classifications changed between the two models, we used PLAN even though the requirement was satisfied. The output in Fig. 9 reveals that no contention was detected and that the *Server Control* function could process data to write acknowledgment to the *Client* while the decryption function was executing.

Device ...	589	590	591
CPU_Appl...	A__sendHeartBeat__sen...	A__sendH...	A__sendHea...
CPU1_1_0	A__sendReplyMessagesC...		A__encrypt1...

Fig. 8: ETA Output Showing Contention

Device ...	522	523
Bus0_0_0	A__sendReplyMessage...	
Bus1_0_0	A__sendReplyMessage...	
CPU_Appl...	A__sendHeartBeat__Se...	A__sendHeartBeat__se...
CPU1_1_0	A__Decrypt__decrypt_a...	A__Decrypt__decrypt_a...

Fig. 9: ETA Output Showing No Contention

VI. Conclusion and Perspectives

The automated trace analysis technique introduced in this paper allows system designers to study latency requirements and accordingly classify transactions. This can be used to study the impact of e.g., new safety/security measures on performance and thus better adjust the system models during early design phases. The approach is now implemented in SysML-Sec.

We now target to address the following limitations. At the functional-level, we will add more communication semantics. For communication semantics, we intend to take buffer sizes and other communication constraint into account. Enhancements can also be done by handling any requirement that can be checked with our tracing facility, while in this paper, we considered that the occurrence of both operators is given in the requirement. We also intend to automatically relate operators occurrences using tainting in the dependency graph. Tainting will also help in targeting the hypotheses presented in this paper.

Our ultimate goal is to provide designers with automated suggestions for enhancing the model such that the timing constraints are all met.

Acknowledgment

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

References

[1] G. Pagano, D. Dosimont, G. Huard, V. Marangozova-Martin, and J.-M. Vincent, "Trace management and analysis for embedded systems," in *2013 IEEE 7th International Symposium on Embedded Multicore Socs*. IEEE, 2013, pp. 119–122.

[2] G. Pagano and V. Marangozova-Martin, "Soc-trace infrastructure," 2012.

[3] P. Kemper and C. Tepper, "Automated analysis of simulation traces-separating progress from repetitive behavior," in *Fourth International Conference on the Quantitative Evaluation of Systems (QEST 2007)*. IEEE, 2007, pp. 101–110.

[4] F. Hojaji, T. Mayerhofer, B. Zamani, A. Hamou-Lhadj, and E. Bousse, "Model execution tracing: a systematic mapping study," *Software and Systems Modeling*, vol. 18, no. 6, pp. 3461–3485, 2019.

[5] D. Hedde and F. 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*. IEEE, 2011, pp. 106–112.

[6] P. Kemper and C. Tepper, "Trace based analysis of process interaction models," in *Proceedings of the Winter Simulation Conference, 2005.* IEEE, 2005, pp. 10–pp.

[7] J. DeAntoni, F. Mallet, F. Thomas, G. Reydet, J.-P. Babau, C. Mraidha, L. Gauthier, L. Rioux, and N. Sordon, "Rt-simex: retro-analysis of execution traces," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 377–378.

[8] O. Iegorov, V. Leroy, A. Termier, J.-F. Méhaut, and M. Santana, "Data mining approach to temporal debugging of embedded streaming applications," in *2015 International Conference on Embedded Software (EMSOFT).* IEEE, 2015, pp. 167–176.

[9] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, "Automatic trace analysis for logic of constraints," in *Proceedings of the 40th annual Design Automation Conference*, 2003, pp. 460–465.

[10] X. Chen, H. Hsieh, and F. Balarin, "Verification approach of metropolis design framework for embedded systems," *International Journal of Parallel Programming*, vol. 34, no. 1, pp. 3–27, 2006.

[11] J. Brandenburg and B. 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*, vol. 77, pp. 26–42, 2017.

[12] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki, and K. Kuusilinna, "Uml-based multiprocessor soc design framework," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 5, no. 2, pp. 281–320, 2006.

[13] N. Alhirabi, O. Rana, and C. Perera, "Security and privacy requirements for the internet of things: A survey," *ACM Transactions on Internet of Things*, vol. 2, no. 1, pp. 1–37, 2021.

[14] R. Malan and D. Bredemeyer, "Defining non-functional requirements," 2001.

[15] P. Marwedel, *Evaluation and Validation.* Cham: Springer International Publishing, 2021, pp. 239–293.

[16] I. Perez, F. Dedden, and A. Goodloe, "Copilot 3," Technical Report NASA/TM-2020-220587, National Aeronautics and Space ..., Tech. Rep., 2020.

[17] A. Matović, "Case studies on modeling security implications on safety," 2019.

[18] K. Y. Rozier, "From simulation to runtime verification and back: Connecting single-run verification techniques," in *2019 Spring Simulation Conference (SpringSim).* IEEE, 2019, pp. 1–10.

[19] 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).* IEEE, 2006, pp. 135–136.

[20] V. Kemper and C. Tepper, "Trace analysis-gain insight through modelchecking and cycle reduction," SFB 559, Tech. Rep., 2006.

[21] "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.

[22] J. Deantoni, "Towards formal system modeling: Making explicit and formal the concurrent and timed operational semantics to better understand heterogeneous models," Ph.D. dissertation, Université Côte d'Azur, CNRS, I3S, France, 2019.

[23] J. DeAntoni and F. Mallet, "Timesquare: Treat your models with logical time," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation.* Springer, 2012, pp. 34–41.

[24] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, "Logic of constraints: A quantitative performance and functional constraint formalism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 8, pp. 1243–1255, 2004.

[25] D. Yue, V. Joloboff, and F. Mallet, "Trap: trace runtime analysis of properties," *Frontiers of Computer Science*, vol. 14, no. 3, p. 143201, 2020.

[26] A. E. Goodloe and L. Pike, *Monitoring distributed real-time systems: A survey and future directions.* National Aeronautics and Space Administration, Langley Research Center, 2010.

[27] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma, "Tessla: temporal stream-based specification language," in *Brazilian Symposium on Formal Methods.* Springer, 2018, pp. 144–162.

[28] B. d'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna, "Lola: runtime monitoring of synchronous systems," in *12th International Symposium on Temporal Representation and Reasoning (TIME'05).* IEEE, 2005, pp. 166–174.

[29] J. Baumeister, B. Finkbeiner, S. Schirmer, M. Schwenger, and C. Torens, "Rtlola cleared for take-off: monitoring autonomous aircraft," in *International Conference on Computer Aided Verification.* Springer, 2020, pp. 28–39.

[30] M. Fisher, V. Mascardi, K. Y. Rozier, B.-H. Schlingloff, M. Winikoff, and N. Yorke-Smith, "Towards a framework for certification of reliable autonomous systems," *Autonomous Agents and Multi-Agent Systems*, vol. 35, no. 1, pp. 1–65, 2021.

[31] S. Rehab and A. Chaoui, "Tgg-based process for automating the transformation of uml models towards b specifications," *International Journal of Computer Aided Engineering and Technology*, vol. 7, no. 3, pp. 378–400, 2015.

[32] E. Kerkouche, K. Khalfaoui, A. Chaoui, and A. Aldahoud, "Uml activity diagrams and maude integrated modeling and analysis approach using graph transformation," in *Proceedings of the 7th International Conference on Information Technology (ICIT 2015) doi*, vol. 10, 2015.

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

[34] L. Pomante, V. Muttillo, B. Krena, T. Vojnar, F. Veljkovic, P. Magnin, M. Matschnig, B. Fischer, J. Martinez, and T. Gruber, "The AQUAS ECSEL project aggregated quality assurance for systems: Co-engineering inside and across the product life cycle," *Microprocess. Microsystems*, vol. 69, pp. 54–67, 2019. [Online]. Available: https://doi.org/10.1016/j.micpro.2019.05.013

[35] "TTool," 2013. [Online]. Available: https://ttool.telecom-paris.fr

[36] L. Apvrille and Y. Roudier, "SysML-Sec: A SysML environment for the design and development of secure embedded systems," *APCOSEC, Asia-Pacific Council on Systems Engineering*, pp. 8–11, 2013.

[37] B. Kienhuis, E. F. Deprettere, P. Van der Wolf, and K. Vissers, "A methodology to design programmable embedded systems," in *International Workshop on Embedded Computer Systems.* Springer, 2001, pp. 18–37.

[38] D. Knorreck, "UML-based design space exploration, fast simulation and static analysis," Ph.D. dissertation, Telecom ParisTech, 2011.