SysML Models: Studying Safety and Security Measures Impact on Performance Using Graph Tainting

Maysam Zoor
maysam.zoor@telecom-paris.fr
LTCI, Télécom Paris, Institutpolytechnique de Paris
France

Ludovic Apvrille
ludovic.apvrille@telecom-paris.fr
LTCI, Télécom Paris, Institutpolytechnique de Paris
France

Renaud Pacalet
renaud.pacalet@telecom-paris.fr
LTCI, Télécom Paris, Institutpolytechnique de Paris
France

ABSTRACT
Designing safe, secure and efficient embedded systems implies understanding interdependencies between safety, security and performance requirements and mechanisms. In this paper, we introduce a new technique for analyzing the performance impact of safety/security implemented as hardware and software mechanisms and described in SysML models. Our analysis approach extracts a dependency graph from a SysML model. The SysML model is then simulated to obtain a list of simulation transactions. Then, to study the latency between two events of interest, we progressively taint the dependency graph according to simulation transactions and to dependencies between all software and hardware components. The simulation transactions are finally classified according to which vertex taint they correspond, and are displayed according to their timing and related hardware device. Thus, a designer can easily spot which components need to be re-modeled in order to meet the performance requirement. A Rail Carriage use case studied in the scope of the H2020 AQUAS project illustrates our approach, in particular how tainting can handle the multiple occurrences of the same event.

1 INTRODUCTION
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 is highlighted in several approaches [18][16][28]. To study the requirements dependencies, simulation and verification shall be used as early as possible in the design process. Estimating performance at this stage of the design process is considered as “very valuable approach in the area of SoC design” [32] as it results in updating the model in a cost efficient manner [30].

Thus, the paper introduces a new technique for analyzing the impact on performance when changing a SysML model. A model change consists in adding or removing safety and security mechanisms. Mechanisms are based on a set of software or hardware components. For example, adding encryption may result in additional computation and communication time due to encryption—and decryption—functions. Also, longer messages to transfer may create additional contentions on shared resources [33]. This new analysis approach can assist the designer in tuning and adapting the model by indicating which hardware components or software functions provoke an extra latency between selected events. It can also report how much a new function is involved in this latency.

This paper is organized as follows. Section 2 discusses different model verification approaches with a focus on performance verification and on approaches with security and performance dependency. Then, Section 3 presents the SysML—Sec modeling and verification approach upon which our new contribution is based. In Section 4, our performance analysis algorithm is detailed. A Rail Carriage use case studied in the scope of the H2020 AQUAS project illustrates our contribution in Section 5. Finally, Section 6 concludes the paper.

2 RELATED WORK
Several tools can analyze and verify the timing properties of real-time systems. These tools are based on static or dynamic analysis and verification methods [19]. Static methods don’t require model execution [19]. Static methods are used in hard real-time systems to guarantee that deadlines are met and calculate worst-case execution time [26]. Dynamic methods require the execution of the model, they are applied on soft real-time systems and are further divided into formal and simulation methods. Tools like Metropolis from Berkeley [10] implements simulation and formal verification methods while TimeSquare [12] simulate a design based on MARTE model where clock constraints are specified based on Clock Constraint Specification Language (CCSL) [9]. Some tools like Time4Sys
connect system modeling editors and real-time analysis tools. TTool [5] is a free and open source framework for the design and verification of embedded systems. SysML-Sec is one of the modeling profiles supported by TTool. In the first stage of SysML-Sec (Figure 1), requirements are identified and explicitly tagged as safety, security or performance. At this step, requirements are textual specifications regarding important properties of the system, defined informally with an identifier and a text. The formal semantics of properties is defined within TEPE [20] Parametric Diagrams (PDs). Also, in this step, attacks that could target the system and faults that could occur in the system are modeled in attack and fault trees respectively. Next, the architecture and high-level functional behavior are modeled before being linked in the mapping phase: this step helps deciding how functions should be split between hardware and software mechanisms, and how communications between functions are realized using physical elements. Second, the design of the software elements can be performed in the software design stage: functions mapped to processors are expected to be refined as software components. Verification can be performed with a press-button approach from most views so as to check that all requirements are satisfied. TTool can perform verifications using formal techniques (e.g., model-checking) and simulations. Safety verification relies on the TTool model checker or on UPPAAL. Security verification relies on the ProVerif [11] external toolkit. Performance verification relies on a System-C like simulator provided by TTool. Once a model has been verified, C code generation can be performed from partitioning models or from software design.

### 3.2 HW/SW partitioning

A HW/SW partitioning is formally defined as the composition of a Functional view, an Architecture Model and a Mapping Model [21].

In the functional view, composite components (colored in yellow) serve as containers for primitive components.Primitive components (green), also referred to as tasks, have attributes and behaviors assigned to them. The behavior is described by an activity diagram built upon a set of operators. Operators can be divided into 3 categories.

1. **Control operators**: handle the execution flow of a task e.g., loops.
2. **Complexity operators**: intend to facilitate the modeling of algorithms’ complexity in terms of, e.g., integer operations (Excl).
3. **Communication operators**: Channels, Events or Requests.
Channels: model data exchange. As we are considering a high level of abstraction, only the amount of data is considered not the data values. There are 3 possible types of channels [14]:
(a) Blocking Read — Non Blocking Write (BR-NBW): this is equivalent to an infinite FIFO buffer between the sender and receiver task. The sender can infinite write and the receiver can write infinite times while the receiver task blocks when attempting to read from an empty channel.
(b) Non Blocking Read — Non Blocking Write (NBR-NBW): this is equivalent to a shared memory of infinite size between the sender and receiver task. The sender can write infinite times while the receiver never blocks when attempting to read.
(c) Blocking Read — Blocking Write (BR-BW): this is equivalent to a finite FIFO buffer between the sender and receiver. The sender blocks when attempting to write to a full channel and the receiver task blocks when attempting to read from an empty one.

Events: used for synchronization between two tasks. Events arriving at a given task can be managed in 3 ways:
(a) Infinite FIFO: events are never lost.
(b) Non Blocking finite FIFO: when the FIFO is full, the first (oldest) element is removed from the FIFO and the new one is added.
(c) Blocking finite FIFO: when the FIFO is full, no event is added until the FIFO is not full. The event sender is blocked until the event is added to the FIFO.

Requests: used to model task spawning. Requests arriving at a given task are stored in an infinite FIFO; they are never lost. Requests are never blocking for the sender task.

An Architecture Model is built upon a set of parametrized hardware nodes and physical links between nodes. Hardware nodes are split into three categories:
(1) Execution nodes: Hardware Accelerators, CPUs, FPGAs...
(2) Communication nodes: Buses, Bridges...
(3) Storage nodes: Memories

A mapping model allocates tasks and communications to hardware components. Tasks mapped to processors are software implemented while tasks mapped to Hardware Accelerators or FPGAs are hardware implemented. The semantics of hardware nodes can be customized with parameters. The high level semantics of these nodes makes it possible to perform formal verifications or fast transaction-based simulations.

3.3 Performance Evaluation
Performance evaluation mostly consists in generating simulation traces from a given SysML mapping, and then analyzing these traces. Trace analysis helps figuring out performance parameters of the hardware nodes (e.g. processor and bus load) but also how the application behaves. There, one important application metrics is the latency between two events executing within the application, as shown in [21]. While in some simple cases having the min/max latency can be beneficial for the designer, in other complex cases, especially when new safety and security measures are added to the model, having only the min/max delay between operators doesn’t help much the designer on the precise cause of the latencies or on how to enhance the model to further improve performance. For this, the performance evaluation technique of TTool has been updated in [34].

The enhanced performance analysis technique analyzes the simulation traces of a SysML mapping model to show which elements of the platform contributed to the latency value. The main algorithm named Simulation Trace Analysis (STA) takes as input (Figure 2): (1) the simulation trace (2) a generated latency graph corresponding to the mapping model and (3) two operators — named $\theta_A$ and $\theta_B$ — selected by the user to study the latency across them. Operators are defined in Section 3.2. The output of STA is two arrays of transactions: mandatory transactions and non-mandatory transactions. The mandatory transactions are the transactions that should be executed after the first operator and are mandatory for the second operator to execute. Non-mandatory transactions are those related to the same hardware as either one of the two operators, they are not mandatory to execute and might contribute to an additional latency that can be eliminated between the two operators. More details on mandatory and non-mandatory transactions is given in Section 4.1.

4 DETAILED LATENCY ANALYSIS TECHNIQUE (DLAT)

4.1 Simulation Trace
A mapping model $p$ is simulated for a time interval using TTool simulator. TTool simulator [19] is transaction-based. A transaction represents a computation or communication operation in the task activity diagram. Control flow operators do not have a corresponding transaction since we assume that they are executed in zero
time. After simulating the model, the executed transactions can be saved in a simulation trace \( sp \) according to their start times. Thus a simulation trace is defined as a set of simulation transactions where each simulation transaction contains the following attributes:

- **device, task and operator**: defines to which task/operator the transaction belongs and on which hardware node it was executed.
- **runnableTime** (in clock cycles): defines the absolute time at which the transaction is ready to be executed. This attribute is independent of shared resource contention.
- **length**: number of clock cycles needed to execute the transaction.
- **startTime** and **endTime** (in clock cycles): define the time at which the hardware node started and ended the transaction execution. In case of hardware congestion, a transaction may be postponed, thus delaying its start time. **endTime** is calculated as: \( \text{startTime} + \text{length} + \text{Penalties} \). Penalties represent the time taken by the OS and the CPU hardware to go idle and the time taken by the OS for a context switch (task switching time).

Simulating the mapping of Figure 2 for 402 cycles results in 90 transactions. An excerpt of the simulation trace showing six transactions is shown in Figure 6(a).

### 4.2 Latency Analysis Using Graph Tainting

The Detailed Latency Analysis Technique (Figure 2) already implemented in TTool [34] helps the designer to investigate the model performance and the cause of delay between two operators \( \theta_A \) and \( \theta_B \). However, it is based on the assumption that the two operators have a one-to-one relation. In other words, it assumes that the \( i^{th} \) occurrence of the second operator \( \theta_B \) corresponds to the \( i^{th} \) occurrence of the first operator \( \theta_A \). While this assumption holds for some use cases as shown in [34], removing this assumption opens new avenues. To overcome the one-to-one limitation, a new analysis technique is defined in this section. This analysis is now based on graph tainting. In addition, our new contribution takes into account contentions on communication and storage nodes and identifies in its output the transactions that caused extra delays due to contentions on communication and storage nodes.

Similar to Figure 2, the Detailed Latency Analysis based on graph tainting has a main algorithm named Simulation Trace Analysis-Graph Tainting (STA-GT). STA-GT (detailed in Section 4.4) takes as inputs:

1. **A Latency Graph**: To analyze the dependencies and relation between the transactions in a simulation trace, a SysML mapping model is translated into a directed graph. Vertices of this graph are tainted with the STA-GT algorithm (Section 4.3).
2. **A Simulation Trace**: TTool simulator generates a simulation trace of the considered mapping model (Section 4.1).
3. **2 operators**: The designer selects two operators (\( \theta_A \) and \( \theta_B \)) between which (s)he wishes to study the latency. These operators must be part of the activity diagram of the considered tasks (Section 3.2).

![Figure 3: Section of the Metamodel Diagram of SysML-Sec Methodology](image-url)

#### 4.3 Latency Graph

The first step in DLAT is generating a directed graph from the mapping model.

As shown in figure 3, the architecture model is a UML Deployment Diagram built upon a set of connected nodes that represent resources. These nodes are divided into 3 categories: computation, communication and storage nodes. The functional view is built on a set of tasks interconnected by data and control ports and channels. It is defined by SysML Block Definition and Internal Block Diagrams. Each task is defined by a SysMLBlock and its internal behavior is a sequence of actions (activity diagram) defined in a SysML Activity diagram. In the mapping model, tasks along with their communication channels are allocated on the Nodes of the architecture model.

Throughout this section, the mapping displayed at the top of Figure 2 is considered. The functional view corresponding to this mapping is shown in Figure 4, with block instances at the top and related activity diagrams below blocks. This toy example illustrates sending an event then data from one task (STASK) to another task (RTASK). In RTASK, the data is received, a computation is done and data is sent to a third task (CTASK). TIMER is a "toy competing task" that runs a delay of 1ns added to create contentions on its host CPU. The latter task was omitted from Figure 4 to keep it readable.

A Latency Graph (\( G \)) is a directed graph consisting of a set of vertices \( v \) and a set of directed edges \( e \): \( G = (v, e) \). \( G \) is built from a mapping model. For instance, the directed graph of the toy example is shown in Figure 5 (TIMER task omitted). A vertex is added to \( G \) for each hardware node i.e. for computation, communication or storage node (\( \text{Bus}_0, \text{CPU}_R, \text{CPU}_S, \text{Memory}_0 \)). Then for every task mapped to a node, a vertex is added. Moreover, a directed edge is added from the corresponding node vertex to the added task vertex to represent the mapping. The same is applied for mapped communication channels. For example, considering the mapping in Figure 2: CTASK vertex is added and CPU_R vertex connected.
to it, S2R_CHANNEL vertex is added and Bus0 vertex connected to it in Figure 5. This approach is applied to all model elements including activity diagrams.

For every action in the activity diagram, a vertex is added along with the required edges to preserve the sequence on the control flow. In addition, directed edges are added to represent the logical connections between tasks (i.e. events and requests. For example: Send Event "STASK.send(S2R_EVENT)" is directly connected to Receive Event "RTASK.wait(S2R_EVENT)". For the channels, directed edges are added between the read/write channel action and the vertex that correspond to the communication channel. In figure 5, STASK.write(S2R_CHANNEL, 4) vertex is connected to S2R_CHANNEL vertex.

In the mapping model a unique ID is given to each element. This ID combined with the name of the element serve as the key value of each vertex in $G$. To implement the tainting within $G$, in addition to the vertex ID, basic attributes are added to vertices. These attributes are:

1. **type**: identifies to which kind of element of a mapping model each vertex corresponds: Node, task, for ever loop, for loop, control, channel, start, end, choice, sequence, unordered sequence or transaction operator...

2. **taintValues**: stores the taint values of a vertex. Every taint value is unique within a DLAT. An 8-byte unique taintValue is generated whenever a transaction related to the first operator ($\theta_A$) is encountered in the simulation trace. The generated taintValue is added to the taintValues attribute of the vertex corresponding the first operator. The taintValue is propagated to other vertices as discussed in section 4.4.

3. **taintFixedNumber** (fixedNbr): is the default number of times this vertex is considered in calculating the delay between two operators per taintValue. The fixedNbr is 1 for all vertex types except for "for loops" where it is equal to the number of iterations and for "for ever loop" where it is equal to integer maximum value. For example in figure 5, if the vertex "RTASK.forLoop(5)" is tainted, its fixedNbr is 5. The fixedNbr is used to determine the maxNbr introduced next.

4. **taintMaxNumber** (maxNbr): stores the maximum number of times this vertex is considered in calculating the delay between two operators per taintValue. The maxNbr is 1 for all vertex types except for "for loops" where it is equal to the number of iterations and for "for ever loop" where it is equal to integer maximum value. For example in figure 5, the vertex "RTASK_forLoop(5)"'s maxNbr is 5. The maxNbr is used to determine the maxNbr introduced next.

5. **taintConsideredNumber** (conNbr): identifies the number of times this vertex is already considered in calculating the delay between two operators per taintValue. The STA-GT algorithm detailed in the following section shows the use of these different attributes. Note that the type and fixedNbr...
attributes are set during graph generation when a vertex is added. In the scope of this paper, the communication semantics of finite FIFO buffer where read/write channels, send/receive events or request can be overwritten in the buffer are not (yet) handled in the graph vertices. We intend to address this limitation in future work.

4.4 Simulation Trace Analysis-Graph Tainting (STA-GT)

Let’s consider a simulation trace \( s_p \). To study in \( s_p \) the latency between two operators \( \theta_A \) and \( \theta_B \) defined in the mapping model \( p \), the simulation transactions in \( s_p \) are ordered according to their start time. In case several transactions have the same start time, the transactions are further ordered according to their end time. The order of considering transactions that have the same start and end time is indifferent as in the simulator only transactions with earlier end time may alter later transactions due to the cause and effect policy [19].

Algorithm 1 performs the simulation trace analysis using graph tainting to calculate the latency between two operators \( \theta_A \) and \( \theta_B \). The delay between the occurrence of \( \theta_A \) and \( \theta_B \) is calculated based on the propagation of a taint value along the generated directed graph. In addition, algorithm 1 highlights for the designer which hardware component or software function contributed in increasing the delay between \( \theta_A \) and \( \theta_B \). In this algorithm, the transactions in \( s_p \) are considered sequentially. For each transaction its corresponding operator, startTime, endTime, device and runnTime are used to determine if the transaction contributed to extra delay between \( \theta_A \) and \( \theta_B \) execution and whether its corresponding vertex should be tainted with a taintValue.

For every taintValue (t) added to the vertex corresponding to \( \theta_A \) (\( V_{\theta_A} \)) we should have:

- a simulation transaction (\( st_{A_i} \)) where \( st.\text{operator} = \theta_A \): this situation leads to adding (t) to \( V_{\theta_A} \).
- a simulation transaction (\( st_{B_i} \)) where \( st.\text{operator} = \theta_B \): this simulation transaction is encountered after (t) is propagated to \( V_{\theta_B} \).
- an array of simulation transactions that were executed after the occurrence of the \( st_{A_i} \), and are mandatory for \( st_{B_i} \) occurrence (\( Arr\text{ayMandatory} \)).
- an array of simulation transactions that don’t belong to \( Arr\text{ayMandatory} \), but that are in the simulation trace \( s_p \) and occurred between \( st_{A_i} \) and \( st_{B_i} \). This array (\( Arr\text{ayNonMandatory} \)) is further split into two sub arrays:
  - \( Arr\text{ayContention} \): contains transactions that have delayed the execution of transactions from \( Arr\text{ayMandatory} \) because of a contention on a shared and common execution node.
  - \( Arr\text{ayNonContention} \): contains transactions from \( Arr\text{ayNonMandatory} \) that don’t belong to \( Arr\text{ayContention} \), i.e. transactions that have used hardware execution nodes without impacting the delay between the operators under study.

Using the output of algorithm 1, the latency \( \lambda_t \) between \( \theta_A \) and \( \theta_B \) for a taint value t is computed as:

\[
\lambda_t = end\text{Time}_{st_{B_i}} - start\text{Time}_{st_{A_i}}
\]  

To show how a taint value propagates along a generated graph, lets consider the example given previously. Let \( \theta_A \) be the writing channel \( S2R\text{CHANNEL} \), \( \theta_B \) be the reading channel \( R2C\text{CHANNEL} \) operator in \( C\text{TASK} \) task and \( \theta_g \) be the reading channel \( R2C\text{CHANNEL} \) operator in \( C\text{TASK} \) task. \( V_{\theta_A} \) and \( V_{\theta_B} \) are colored green in Figure 5. In this example, the one-to-one relation between \( \theta_A \) and \( \theta_B \) doesn’t hold since the data channels \( S2R\text{CHANNEL} \) and \( R2C\text{CHANNEL} \) are Non Blocking Read — Non Blocking Write (Section 3). Thus, in the simulation trace multiple simulation transactions corresponding to reading these channels may exist before writing them (first 2 transactions in Figure 6(a)). To calculate the latency algorithm 1 is used.

A taintValue is generated whenever a simulation transaction (st) where \( st.\text{operator} = \theta_A \) is encountered. The generated taintValue is added to the taintValues attribute of \( V_{\theta_A} \). The taintValues attribute of \( V_{\theta_A} \) contains a unique taintValue for every occurrence of \( st.\text{operator} = \theta_A \) in \( s_p \). In Figure 6, a taintValue (t), presented in purple circle, is added to \( V_{\theta_A} \), once a simulation transaction with \( st.\text{operator} = \theta_A \) is encountered in \( s_p \) (second simulation transaction in Figure 6(a)). In addition to adding t to \( V_{\theta_A} \), t is propagated to all successor(s) vertices of \( V_{\theta_A} \) (shown in red arrows in Figure 6(b)) and maxNbr determined for each. A successor vertex is a vertex connected by one incoming edge from \( V_{\theta_A} \). In Figure 6, t is propagated to \( V_{S2R\text{CHANNEL}} \) corresponding to communication on \( BUS0 \) to \( V_{STASK\text{STOP}} \). The \( conNbr \) for a vertex is incremented by one once its successors are tainted. \( conNbr \) of a vertex is compared to its \( maxNbr \) to check if this vertex can still be considered for t. Once \( st.\text{operator} = V_{S2R\text{CHANNEL}} \) is encountered after \( V_{\theta_A} \), \( V_{S2R\text{CHANNEL}} \) are tainted with t (third simulation transaction in Figure 6(a)). t is propagated to \( V_{RTASK\text{STOP}(S2R\text{CHANNEL}4)} \) (shown in green arrow in Figure 6(b)). When a simulation transaction corresponding to \( V_{RTASK\text{STOP}(S2R\text{CHANNEL}4)} \) is encountered (fifth simulation transaction in Figure 6(a)), and \( V_{RTASK\text{STOP}(S2R\text{CHANNEL}4)} \) is tainted with t, then this simulation trace corresponds to reading the tainted data. t is propagated to \( V_{RTASK\text{FORLOOP}} \) (navy color arrow in figure 6(b)). As no transactions corresponds to control operators including “for loops”, t is propagated to \( V_{RTASK\text{action}(x+=1)} \) where the \( maxNbr \) is also set to 5 since it falls inside a for loop.

The \( conNbr \) for the \( RTASK\text{FORLOOP} \) will be incremented after all the vertex inside the loop are considered once. Verteres corresponding to control operators are tainted and their \( conNbr \) is updated in order to maintain the progress of a taint value across the graph according to the functional logic in the mapping. The vertex corresponding to the exit of the loop (\( RTASK\text{WRITE}(R2C\text{CHANNEL}4) \) in our case) will be tainted after the \( conNbr \) for the \( RTASK\text{FORLOOP} \) equals its \( maxNbr \).

We consider reading channel \( R2C\text{CHANNEL} \) in \( C\text{TASK} \) task to be \( \theta_B \). t will be transmitted to \( V_{C\text{TASK}\_\text{READ}(R2C\text{CHANNEL}A)} \) after \( R2C\text{CHANNEL} \) is tainted. The simulation transaction where \( st.\text{operator} = \theta_B \), encountered after t is propagated to \( V_{\theta_B} \), is used to calculate the latency \( \lambda_t \).

Algorithm 1 (lines 13 — 34) is executed to fill \( Arr\text{ayMandatory} \) and \( Arr\text{ayNonMandatory} \). The first step is to check if \( V_{\theta_A} \) and \( V_{\theta_B} \) are connected by at least one path in the graph G. A path is defined as a sequence of vertices such that each vertex in the sequence is connected with directed edge to vertex next to it. This sequence
Table 1: Time Values of Two Simulation Transaction in Figure 6

<table>
<thead>
<tr>
<th></th>
<th>runnableTime</th>
<th>startime</th>
<th>endtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>st₁</td>
<td>1</td>
<td>200</td>
<td>202</td>
</tr>
<tr>
<td>st₁₁</td>
<td>200</td>
<td>300</td>
<td>302</td>
</tr>
</tbody>
</table>

Table 2: The Tainting Progress

<table>
<thead>
<tr>
<th>V₀₁ is V₀₂?</th>
<th>V₀₂</th>
<th>V₀₃</th>
<th>V₅₂</th>
<th>V₀₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = null?</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>vertexHasSuccessors(g, V₀ᵢ)?</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>V₀ᵢ ∈ path?</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>add(st) to ArrayMandatory?</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>add(st) to ArrayNonMandatory?</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>

should start with V₀₁, and end with V₀₆, to say that we have a path between V₀₁ and V₀₆. For every simulation transaction added to ArrayMandatory, its runnableTime and startime are saved to be used to divide ArrayNonMandatory between ArrayConteion and ArrayNonConteion (Algorithm 1: lines 35 – 44). Generally speaking, if the runnableTime and startime values of a transaction don’t match, it means that the simulator scheduled a transaction but didn’t execute it since the resource was busy.

Let’s consider the fourth and sixth transaction in the excerpt of the simulation trace of the example introduced before (Figure 6). We refer to the fourth simulation transaction as st₁ and the sixth as st₁₁. The simulation transactions st₁ and st₁₁ belong to ArrayNonMandatory and ArrayMandatory respectively. st₁ and st₁₁ are executed on the same hardware. st₁ is executed at cycle 200 between the runnableTime (200) and startime (300) of st₁₁ (Table 1). Thus st₁ caused a delay in a mandatory transaction st₁₁, so st₁ is added to ArrayConteion. Table 2 summarizes the main points that the tainting algorithm checks for each simulation transaction. The second column corresponds to \(V_{\text{CTASK.read(R2C_CHANNEL,4)}}(V₀₂)\) encountered before \(V₀₁\). Then \(V₀₁\), \(V_{\text{SIR_CHANNEL}}\) and \(V_{\text{TIMER}}\) are considered in columns 3, 4 and 5. Column 6 represents a simulation trace where \(st\_operator = V_{\text{CTASK.read(R2C_CHANNEL,4)}}\) however this time \(V_{\text{R2C_CHANNEL}}\) is considered to be tainted.

4.5 Graphical Interface

DLAT is implemented within TTool [5]. TTool enables the designer to simulate the model through graphical interface. The simulation trace can be saved in xml format. DLAT can be initiated on the saved simulation trace within TTool with a mouse click on the simulation trace name. Once DLAT is initiated on the simulation trace, the graph corresponding to the model is automatically generated in the background. The designer is informed of the number of edges and vertices of the graph. \(V₀₁\) and \(V₀₆\) are then chosen from drop down list to run \(\text{STA_GT}\). Thanks to \(\text{STA_GT}\) output, the latency \(\lambda₁\) for each taint t is calculated and displayed along \(\text{startTime}_{\text{STA}}\) and \(\text{endTime}_{\text{STA}}\) in tabular format. Figure 7 shows us the latency between writing channel S2R_CHANNEL and reading channel R2C_CHANNEL operator in CTASK task and reading channel R2C_CHANNEL operator in CTASK task of the previous example. The latency \(\lambda₂\) in this case is 281 cycles. In addition to \(\lambda₁\), the arrays ArrayMandatory, ArrayContention and ArrayNonContention for each \(\lambda₂\) are also displayed in a tabular format.

Transactions in these arrays are placed according to their execution time and device, and colored according to which array they belong. Those that belong to ArrayMandatory are colored green since they are essential for \(\text{st₆}\) execution, others that belong to ArrayNonContention are colored orange since in this simulation they didn’t delay other transactions while the ones that belong to ArrayContention are colored red as they caused contentions on hardware nodes. Figure 8 shows simulation traces in ArrayNonContention and ArrayMandatory. However in time-slot 201 (figure 9) the RTASK function was scheduled to execute but found the resource CPU_R busy executing TIMER task, thus the simulation trace corresponding to the TIMER operator is colored red. Thanks to this display, the designer can directly identify which transactions are causing an increase in the latency between the execution of two operators and quickly spot contention on hardware nodes.

As mentioned previously, ideally we should have \(\text{st₆}\) for each taintValue (t) of \(V₀₁\). However if the simulation was stopped before \(V₀₁\) is tainted, a message indicating “no transaction was found for this taint” is shown to the user.

5 CASE STUDY

We illustrate the benefits of graph tainting with the Rail Carriage Mechanisms use case defined in the scope of the H2020 AQUAS project [4], with a focus on the control of automatic platform gates.

The system consists of Lidars with their processing units, a main computing unit, a relay and a PSD (Platform Screen Doors) controller. The Lidars are divided in two categories. Positioning
Algorithm 1: Simulation trace analysis with Graph tainting

Data: $\theta_A, \theta_B, sp, g$

Result: Tainted Detailed time analysis between $\theta_A, \theta_B$

foreach SimulationTransaction $st$ in $sp$ do
  if $V_{\theta_A}st$ is $V_{\theta_A}$ then
    $t$ = generateTaintValue()
    addTaintValue($V_{\theta_A}t$)
    $st_{\theta_A} = st$;
  end
  else if !$V_{\theta_A}.getTaintValue()$.isEmpty() then
    $t$ = $V_{\theta_A}.getTaintValue()$;
  end
  if vertexHasSuccessors($g, V_{\theta_A}$) && $t! = Null$ then
    addTaintValueToSuccessors();
  end
  if $\exists path(V_{\theta_A} \rightarrow V_{\theta_B})$ in $g$ then
    if $V_{\theta_B}e$ in $path$ && $t! = Null$ then
      if $V_{\theta_B}is V_{\theta_B}$ then
        $st_{\theta_B} = st$;
      end
      ArrayMandatory$st$.add(st);
      addRunnableTimePerDevice();
    end
    else if $st.deviceName == hardware\theta_A||\theta_B||\theta_X|V_{\theta_X}\epsilon path$ then
      ArrayNonMandatory$st$.add(st);
    end
  end
  else if $\exists path(V_{\theta_A} \rightarrow V_{\theta_A}) || \exists path(V_{\theta_A} \rightarrow V_{\theta_B})$ then
    if $V_{\theta_A}e$ in $path$ && $t! = Null$ then
      ArrayMandatory$st$.add(st);
      addRunnableTimePerDevice();
    end
    else if $st.deviceName == hardware\theta_A||\theta_B||\theta_X|\theta_X\epsilon path$ then
      ArrayNonMandatory$st$.add(st);
    end
  end
end

foreach SimulationTransaction $st_E$ in ArrayNonMandatory$st$ do
  foreach SimulationTransaction $st_R$ in ArrayMandatory$st$ do
    if $st_E.startTime >= st_R.runnableTime &&
      st_E.startTime <= st_R.startTime && $st_E.deviceName
      == st_R.deviceName then
      ArrayContention$st_E$.add(st_E);
    end
    else
      ArrayNonContention$st_E$.add(st_E);
    end
  end
end

Lidars scan for a train presence and door Lidars scan the train doors to determine their status. The processing unit of the positioning Lidars calculates the position and the speed of the train once it is present while the processing unit of the door Lidars detects the state of doors e.g. opening, open, closing and closed. The main computing unit gathers data from the Lidars processing units and issues orders to relays to open or close the platform screening doors. This open/close authorization is sent to the PSD controller through the relay.

Our design captures four Lidars (2 positioning Lidars and 2 door Lidars). We also consider the four following requirements:

1. **Req. 1**: The delay between sending the data from the positioning Lidar and the relay receiving the order from the main computing unit shall be less than 130ms (safety requirement)
2. **Req. 2**: The delay between sending the data from the positioning Lidar and processing it in the corresponding processing unit shall be less than 85ms. (safety requirement)
3. **Req. 3**: Data sent from the Lidars processing units (speed and direction, or door status) to the main computing unit should remain authentic (security requirement)
4. **Req. 4**: Data sent from the Lidars to their corresponding processing units should remain confidential (security requirement)

### 5.1 HW/SW partitioning models

Figure 10 shows the functional view of the use case where only one Lidar is presented. The primitive component $PL_1$ is used to represent sending data by the first positioning Lidar. $PL_1$ sends 1 frame of data once triggered by $triggerPL1$ every 67ms. This frame is received by another primitive component named $F_1\_land2\_PL1$ where the frame is copied to the algorithm buffer then checked for validity by checking its length and CRC calculation. After being checked, a detection algorithm is run that includes rotational mapping, filters and pattern detection. The computation complexity of this algorithm is modeled in the activity diagram using complexity operators (Section 3). $F_1\_3\_PL1$ reads the output of the detection algorithm, runs CRC calculation and sends a message to the $F_3\_1\_MsgAcquisition$ component. $F_1\_3\_PL1$ is triggered every 50ms. $F_1\_land2\_PL1$ and $F_3\_3\_PL1$ represent the functionality of the positioning lidar processing unit. All these blocks are duplicated for the 3 remaining Lidars. The door Lidars are triggered every 20ms.

$F_3\_1\_MsgAcquisition$ is a primitive component in a composite component named $SafetyComponent$. $F_3\_1\_MsgAcquisition$ reads data from $F_1\_3\_PL1$. The same applies for the data received from the other 3 Lidars processing unit functions. In the composite component $SafetyComponent$, another primitive component named $F_3\_2\_MsgAcquisition\_SafePart$ is a redundant function added to the model to ensure safety. $F_3\_2\_MsgAcquisition\_SafePart$ is triggered every 50ms. It runs a validity check and a sequence algorithm (represented by computation complexity) to compute the adequate result to be sent to Relay. The later is triggered every 33 ms.

The architecture of the system is as follows. Each Lidar is captured by its own set of processors, buses, memories, while the safety platform is built upon a CPU (MainCPU) and 2 memories: Main-Memory and RelayMemory. The mapping model associates LIDAR...
blocks triggerPL1, PL1, F1_1and2_PL1 and F1_3_PL1 and their communications to their corresponding hardware while safety blocks are mapped to MainCPU. A share memory helps exchanging data between a MainCPU and Relay.

5.2 System verification

The System Under Analysis (SUA) is supposed to run at 80 MHz. TTool was used to simulate it on a Intel Core i7-7820HQ CPU running at 2.9 GHz. 150 ms of the SUA execution have been simulated; the simulation trace contains 19575 transactions and is saved in xml format. A duration of 150 ms is chosen since it is the minimum duration that permit us to validate Req_1 using DLAT. DLAT is used to validate Req_1 since the computationResult — ControlData channel in Figure 10 is Non Blocking Read — Non Blocking Write (NBR-NBW). This means — as mentioned previously — that it is equivalent to a shared memory between the sender and the receiver. In other words, the receiver task is not blocked if the sender didn’t send data on the channel. Thus, tainting should be used to trace when the control data is computed based on the position frame input. So data sent from PL1 should be tainted to calculate the exact time delay between \( \theta_A \) and \( \theta_B \). Sending a frame from the Positioning Lidar (request “SendPos1” in triggerPL1) is \( \theta_A \) in Req_1 and the relay receiving a control signal to send to the PSD (channel “controlData” in Relay) is \( \theta_B \). STA_GT requires as input: the simulation trace, the generated latency graph of the model, and the 2 operators \( \theta_A \) and \( \theta_B \). The latency graph \( g \) corresponding to the model is generated based on the algorithm presented in [34]. In this use case, the latency graph is composed of 244 vertices and 393 edges.

The latency between \( \theta_A \) and \( \theta_B \) can be calculated whenever \( V_{\theta_B} \) is tainted with the same taint value as \( \theta_A \) and the conNbr of \( V_{\theta_B} \) is greater than 0. Based on algorithm 1, the latency between \( \theta_A \) and \( \theta_B \) is 10170380 cycles (127.1 ms). Thus Req_1 is satisfied. The latency corresponding to Req_2 is 681372 cycles (8.51 ms) thus Req_2 is not satisfied.

To validate the authenticity of the data sent from F1_3_PL1 to F3_1_MsgAcquisition and from F3_1_MsgAcquisition to F3_2_MsgAcquisition_SafePart (Req_3), and the confidentiality between PL1 and F1_3_PL1 (Req_4), the formal security verification of TTool/ProVerif is used. The latter proves that Req_3 and Req_4 are not satisfied and shows it to the user by adding a red lock on the concerned data channels. To ensure the authenticity property on these channels, CRC is replaced by HMAC-SHA256 in F3_1_PL1. [21] describes how a security operator can be added in TTool to represent HMAC-SHA256. To determine the computation complexity of HMAC-SHA256 (i.e. 8322 clock cycles), we have used the technique described in [15] and relying on SSDLC (Secure Software Development Life Cycle). The overhead of the message is set to 256 bits.

To ensure the confidentiality property on the channel between PL1 and F1_3_PL1 (Req_4) encryption/decryption operators are added. We chose the AES algorithm in Cipher Block Chaining (CBC) mode and set the computational complexity to 3000 as indicated in [15].

By adding authenticity and confidentiality mechanisms, we could formally prove that (Req_3) and (Req_4) are now satisfied. The concerned data channels are annotated with green locks in figure 10. In TTool, channels can be either private or public and only attacks on public channels are considered [23].

We run again DLAT along with its new model and new simulation trace. The time delay corresponding to Req_1 is now 10249025 cycles (128.1 ms) while the time delay corresponding to Req_2 is 683551 cycles (8.54 ms). The increase of the time delay of Req_2 is due to the added encryption/decryption operators and the increase of the time delay of Req_1 is due to the scheduling policy of the mainCPU. The details corresponding to the increase or decrease of the time delay are displayed in the output table of DLAT.

To satisfy Req_2 while keeping the confidentiality property of Req_4 valid, we replace AES CBC with AES CTR (counter mode). The computational complexity is now set to 428 cycles. This value is obtained by applying the same interaction as indicated in [15]. The security verification indicates that the confidentiality property still holds. The latency was recalculated in a similar manner as mentioned before. The maximum delay corresponding to Req_2 now is 678029 cycles (8.47 ms). The maximum delay of Req_1 wasn’t effected as the latency for Req_1 depend on the trigger time and scheduling policy of mainCPU.

Table 3 summarizes the result of each requirement along each tested model in this use case. While replacing AES CBC with AES CTR mode enhanced performance by decreasing the latency proportionally to the decrease in the computational complexity cycles, several other methods can be tested in case further performance enhancement is required, e.g. by adding hardware accelerators for cryptographic functions, by using other security algorithms, by trying a different mapping, by adjusting the scheduling policy of CPUs or buses, of by using more powerful processing units, ....
After applying the required enhancements, the designer can simulate the model and run the verification process again to test if the requirements still hold.

6 CONCLUSION AND PERSPECTIVES
To accurately study the impact of safety/security measures on performance when designing an embedded system, an approach named Detailed Latency Analysis Technique based on graph tainting has been described. It is intended to be used at a high level of abstraction, thus giving early design guarantees. Its main idea is to model systems at a high level of abstraction and then simulating them. Simulation traces are then used to gradually taint a generated directed graph that corresponds to the model under investigation. Depending on the tainting progress, the latency between two events in the model can be evaluated. Studying this latency using tainting not only reveals the delay between the events under study but also highlights in a clear way which model components are involved in this delay. Last but not least, the approach is now implemented in SysML-Sec.

The object of our future work is to enhance the graph generation to consider other functional-level communication semantics and to settle an automated search for a solution satisfying safety/security requirements while minimizing system latency.

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

REFERENCES