Incremental Latency Analysis of Heterogeneous Cyber-Physical Systems

Julien Delange and Peter Feiler
Carnegie Mellon Software Engineering Institute
4500 5th Avenue
Pittsburgh, PA15213
Email: {jdelange,phf}@sei.cmu.edu

***Abstract*—Cyber-Physical Systems, as used in automotive, avionics, or aerospace domains, have critical real-time requirements. Time-related issues might have important impacts and, as these systems are becoming extremely software-reliant, validate and enforcing timing constraints is becoming difficult.**

Current techniques are mainly focused on validating these constraints late by using integration tests and tracing the system execution. Such methods are time-consuming and labor-intensive and, discovering timing issue late in the development process might incur significant rework efforts.

In this paper, we propose an incremental model-based approach to analyze and validate timing requirements of cyber-physical systems. We first capture the system functions, its related latency requirements and validate the end-to-end latency at a high level. This functional architecture is then refined into an implementation deployed on an execution platform. As system description is evolving, the latency analysis is being refined with more precise values.

Such an approach provide latency analysis from a high level specification without having to implement the system, saving potential re-engineering efforts. It also helps engineers to select appropriate execution platform components or change the deployment strategy of system functions to ensure that latency requirements will be met when implementing the system.

I. Introduction

Cyber-Physical Systems (CPS), used in domains such as avionics, automotive or aerospace, must comply with strong real-time requirements. A data arriving too early or too late can have an significant impacts on system behavior (for example, arrival of an acceleration or brake command in a self-driving car). For that reason, validating end-to-end latency of data flows is of particular importance.

A. Problem

Validating system latency is mostly done late, after implementation and integration efforts, by analyzing the system using various tests. Thus, fixing time-related issues requires to update the system specification and modify the implementation, incurring potential rework costs (design and implementation changes).

Such an approach has important limitations. First of all, it validates system latency at the end of the development process, after design and implementation efforts. Second, it does not provide any insight during the design process to select and design an appropriate architecture that will enforce real-time constraints. Finally, as timing issues must be fixed, discovering them late incurs an important rework cost that can be more than 1000 times more expensive as if it was addressed earlier in the development process[1].

B. Approach

We propose an incremental approach to specify and validate latency using architecture models. First, we capture the functional aspects of the system and allow a latency budget on the end-to-end data flows between connected components. This first model does not contain any realization detail but is sufficient to make preliminary verification (for example, if the end-to-end latency of the data flow is correct according to the latency specified on each functional components). Then, this latency analysis is refined while the architecture is refined and realization details are emerging. This approach can also support the analysis of several implementation candidates, supporting engineers to select the best architecture candidate to meet their latency requirements.

This approach is supported by specifying the system architecture with Architecture and Analysis Design Language (AADL) [2] and analysis tools implemented within OSATE [3], an Open-Source AADL toolsuite.

C. Related Work

Previous work already support latency analysis from architecture models designed with AADL [4]. This existing validation approach does not address the following aspects:

1) **Incremental analysis**: existing tools analyze one system without considering its potential refinement into different implementation alternatives.
2) **Specific CPS execution environment**: existing methods did not take into account many specific aspects of Cyber-Physical Systems such as processor, operating system, partitioning policy or communication protocols. The proposed methods supported partitioned operating systems, heterogeneous networks as well as traditional (integrated) systems.

Also, this analysis enforces that data flows latency is validated accross the architecture. It can be associated with AADL-related scheduling analysis tools (such as Cheddar or AADL Inspector [5]) to check that other timing requirements are also enforced in the model.

II. Data Flow Specification with AADL

Our approach uses AADL to specify the system architecture, its end-to-end flows as well as their latency requirements. This section introduces the modeling language and details how to define data flows between inter-connected components and specify latency requirements.

A. The Architecture Analysis and Design Language

The Architecture Analysis and Design Language (AADL) [2] is a modeling language standardized by SAE International. It defines a notation for describing system concerns and its interaction with its operating environment (i.e. processors, bus, devices).

The core language specifies several categories of components with well-defined semantics. For each component the modeler defines a component type to represent its external interface, and one or more component implementations to represent a blue print in terms of subcomponents. For example, the task and communication architecture of the embedded software is modeled with `thread` and `process` components interconnected with `port connections`, shared `data access` and remote service call. The hardware platform is modeled as an interconnected set of `processor`, `bus`, and `memory` components, with `virtual processor` representing partitions and hierarchical schedulers, and `virtual bus` representing virtual channels and protocol layers. A `device` component represents a physical subsystem with both logical and physical interfaces to the embedded software system and its hardware platform. The system component is used to organize the architecture into a multi-level hierarchy. Users model the dynamics of the architecture in terms of operational modes and different runtime configurations through the mode concept. Users further characterize components through standardized properties, e.g., by specifying the period, deadline, worst-case execution time for threads.

The language is extensible; users may adapt it to their needs using two mechanisms:

1) **User-defined properties**. New properties can be defined by users to extend the characteristics of the component. This is a convenient way to add specific architecture criteria into the model (for example, criticality of a subprogram or task)
2) **Annex languages**. Specialized languages [6] can be attached to AADL components to augment the component description and specify additional characteristics and requirements (for example, specifying the component behavior [6] by attaching a state-machine). They are referred to as annex languages, meaning that they are added as an additional piece of the component. In this paper we will discuss the Error Model Annex language.

AADL provides two views to represent models:

1) The **graphical view** outlines components hierarchy and dependencies (bindings, connection, bus access, etc.). While it does not provide all details, this view is very useful when using the architecture for communication and documentation purposes.
2) The **textual view** shows the complete model description, with component interfaces, properties and languages annexes. It is appropriate for users to capture system internals details and for tools to process and analyze the system architecture from models.

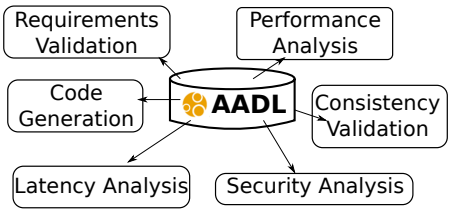

Fig. 1. AADL Ecosystem for Software System Design and Implementation

The AADL model, annotated with properties and annex language clauses is the basis for analysis of functional and non-functional properties along multiple dimensions from the same source, and for generating implementations, as shown in figure 1. AADL has already been successfully used to validate several quality attributes such as Security [7], [8], Performance or Latency [4]. Supporting analysis functions have been developed in the Open Source AADL Tool Environment (OSATE) [3], an AADL tool for architecture design and analysis.

B. Data Flows Definition

The AADL provides the appropriate semantics to define data flows within the architecture. It distinguishes three types of flows:

1) **Flow specification** specifies the externally visible flow through the component interfaces **within a component type**. A flow traversing a component is called a flow path. A flow originating in a component is called the flow source. A flow ending in a component is called the flow sink.
2) **Flow implementation** specifies how the flow is realized **within a component implementation**. It defines how the flows is concretely implemented by detailing how data is passed from one component incoming interface, transfer it to the subcomponents through connections and eventually send it through outgoing interfaces.
3) **End-to-end flow** is a logical flow through a sequence of components **within a component implementation**. It list all the elements of the flow, starting from the starting point of the end-to-end flow (the flow source), going through all the connections and components where the data is transported and ending in the component that use the data (flow sink).

The flow latency budget is specified by associating the AADL property `Latency` to an AADL flow element (flow specification or implementation) or an AADL end-to-end flow. When being associated with a flow specification or implementation, it defines the budget allocated for this flow element. When being associated with an end-to-end flow, it represents the min/max latency expected from the data originating from the source to its use by the flow sink.

III. Latency Analysis

A. Latency Contributors

Architecture design impacts data flow latency. Many factors must be taken into account. From an AADL perspective, they are represented by AADL components and their associated properties. Table I lists all contributors as AADL components, the following paragraphs provide more detail about the modeling rules that impact system latency.

For a software task (an AADL `thread` component) or a hardware component (an AADL`device` component) we consider the following timing characteristics: the execution time (AADL property `Compute_Execution_Time`), the period (rate at which the function is executed, specified with the AADL property `Period`) and the deadline (time at which the component must have finish its execution cycle, specified with the AADL properti `Deadline`.

When components are being connected, the implementation of the connection is an important latency contributor. We distinguish three types of connections (illustrated in figure 2):

1) **immediate**: data is received as soon it is sent - there is no additional delay.
2) **delayed**: data is received at the next execution period of the receiver.
3) **sampled**: data is received as soon as it is dispatched, according to the constraints of the execution environment (scheduling of the execution platform, bus or protocol latency, etc.).

The connection type is specified in AADL using the `Timing` property (legit values are `immediate`, `delayed` and `sampled`). Unless the designers explicitly specifies a type, a connection is considered `sampled`.

The deployment strategy is an important variability factor: if communicating tasks are not located on the same processor, this will require to analyze the time require to transfer the data. In AADL, this can be analyzed by analyzing how AADL `thread` and their associated `process` components are associated AADL `processor`.

In Cyber-Physical Systems (and especially in avionics architecture), tasks can also be deployed on different partitions within the same processor. In AADL, this is captured with `processor` and `virtual processor` components: a `processor` component represents the partitioning operating system and contains many `virtual processor` subcomponents, each one representing partitions execution environment. However, as partitioning operating systems enforce time isolation across partition, it also incurs additional latency for cross-partition latency, depending on the partitioning policy (the number of time slots and their allocation to partitions). This partitioning policy is specified using the AADL property `ARINC653::Schedule_Windows` property on `processor` components.

In addition, distributed tasks use a bus in order to transfer data across `processor`, adding additional latency (the time required to transfer the data over the bus). This characteristics is also captured by AADL using the property `Transmission_Time` on a `bus` component. It represents the time to transfer a data on the bus, as well as the time required to acquire the bus. Combining this property and the size of the transfered data is enough to evaluate the latency incurred by distributed connection.

Finally, connected tasks communicate using communication protocols (TCP, UDP, etc.) which might add potential latency. To specify the use of a particular protocol, an AADL connection should then define the property `Required_Virtual_Bus_Class`

Component	Related properties
`thread` or `device`	`Period` (rate at which the component do something), `Compute_Execution_Time` (how much time it takes to perform its job), `Deadline` (when the job is supposed to be completed)
`bus`	`Transmission_Time` (time required to transfer data between two communication points)
`virtual bus`	`Latency` (latency of a protocol)
`processor`	`ARINC653::Schedule_Window` (partitions scheduling)
`connection`	`Timing` (type of connection), `Required_Virtual_Bus` (protocol used to implement the connection)
`data`	`Data_Size` (to compute transmission time)

TABLE I. AADL components that contributes to system latency with their related properties

B. Support for Incremental Development and Architecture Selection

Our approach supports different levels of abstraction of the system architecture. It can be used at a very high description level (i.e. system function without any implementation specified) to have latency estimates and will produce more accurate results when the designers add implementation details.

The main benefits of this approach is that while the system architecture is evolving with more details, the latency estimates become closer to what engineers would expect at runtime. The analysis can be continuously being generated at each development iteration to analyze impact of changes.

This characteristic of our validation process enables an early validation of latency concerns without having to specify all system concerns in a single model. It can also support architecture selection: engineers can then experience and try different implementation choices (program a function using hardware component - such as AADL `device` - or software - AADL `process` or `thread`) deployment strategies (function allocation over the processing resources) or configuration mechanisms (use of a specific protocol via the use of AADL `virtual bus` components), see the impact on the end-to-end latency in order to take design decision before freezing the architecture design.

C. End-to-End Flow Analysis

The end-to-end flow analysis consists of analyzing each AADL end-to-end flow declared within each component and

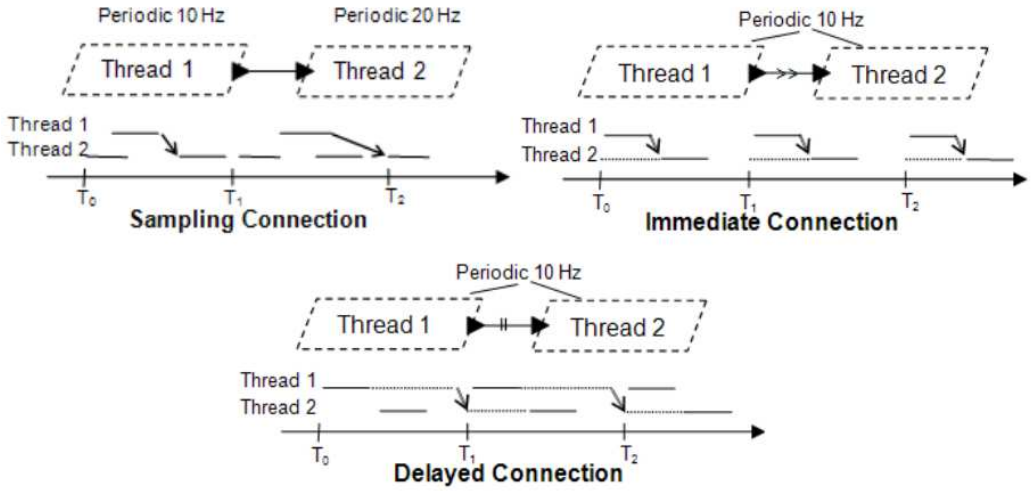

Fig. 2. AADL Connection Type: sampling, immediate and delayed

check that the latency value is great than the latency of the sum of each latency of the flow elements. This validation is done for both the lower (minimum) and upper (maximum) range of the latency value.

For each flow elements, the analysis tool behaves differently if the element is a component (e.g. a flow source or sink) or a connection. If the element is a component, the analysis calculates the Best and Worst Case Execution Time (BCET and WCET).

Also, when two components are being connected, the tool gets the connection latency. This value depends on the connection type (immediate, sampled or delayed) and the deployment policy (components collocated on the same processor or distributed, etc.):

- When the components are colocated on the same processor, the connection latency depends on the components period and the connection type. An immediate connection means that the receiver get the data as soon as this is sent while a delayed connection will wait for the next period.
- When being distributed, the connection latency depends on the physical bus and the underlying protocol to transport the data. In addition, as the receiver component samples incoming data, it might receive it at the beginning (best case) or the end (worst case) of its period.
- When being in a partitioned architecture, connection latency between components located in different partitions depend on the partitioning policy: the receiving partition have to wait that inter-partitions communications buffers are flushed. As for now, our analysis tool considers two different policies, as described in section IV-D.

The rules used by the latency analysis tool are available within the OSATE documentation[3]. It describes each case and precisely indicates how it defines the latency for each component and connection type.

IV. Case-Study

We demonstrate our latency analysis method on a usual architecture pattern in safety-critical systems. The system consists of three main functions:

1) **sensing**: acquire raw data from devices, sensors, etc.
2) **processing**: receives the sensed data, process it (e.g. filter, remove inconsistent values, convert it to a particular format, etc.).
3) **actuating**: get the processed data and use it to control a device (e.g. display information on a screen, adjust speed of a motor).

The data flow originates in the **sensing** function, traverses the `processing` and eventually ends in the **actuating** function. This example assumes that the end-to-end flow from the **sensing** function to the **actuating** function will be less than 100 ms.

Through this case-study, we will first establish a functional architecture that defines the various functions, their latency budgets as well as the end-to-end flows specification and their latency requirements. Then, this preliminary architecture is refined into a runtime architecture, replacing AADL `abstract` components by a software implementation using AADL `process` and `thread` components. This software implementation is then integrated on two different deployment alternatives:

1) **distributed**: each function is executed on separate processors connected using a bus.
2) **integrated**: all functions are executed on the same processor but isolated from each other using a partitioning system.

This case-study will then highlight how deployment (distributed or integrated) and configuration (partitions scheduling policy) impact the end-to-end latency. The next sections detail the functional architecture and its refinement into the several runtime architectures with different configuration alternatives. Finally, interested readers can get the AADL model on the official OSATE github repository [9].

A. Functional Architecture

We capture the functional specification of our system in an AADL model with `abstract` components, as shown in figure 3. This model declares the flow of every component:

- The **sensing** function has a `flow source` with a latency requirement between 3 ms and 4 ms.
- The **processing** function has two `flow paths` (one for each incoming sensor value) with a latency requirement between 4 ms and 5 ms.
- The **actuating** function has a `flow sink` with a latency requirement between 5 ms and 7 ms.

Finally, the top-level system defines two `end-to-end` flows from the **sensing** functions to the `actuating` function with a latency budget lower than 30 ms. The end-to-end flow from the first **sensing** function to the **actuated** is highlighted with a line in figure 3.

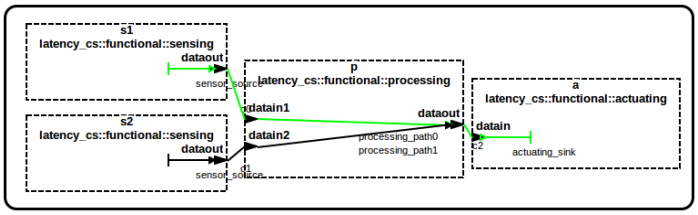

Fig. 3. Functional Architecture

When running our analysis tools, it shows that the latency budget allocated to each component is sufficient to enforce the end-to-end latency requirements. As shown in table II, the minimal total latency is 17 ms (as the minimal expected is 20 ms) while the total max is 25 ms (30 ms expected). This functional analysis is a high-level validation without implementation details. Next sections discuss how deployment consideration impact end-to-end flows latency and might break requirements enforcement.

Component	Min Latency	Max Latency
`Sensing`	5 ms	7 ms
`Processing`	5 ms	8 ms
`Actuating`	7 ms	10 ms
`Total`	17 ms	25 ms

TABLE II. LATENCY FOR THE FUNCTIONAL ARCHITECTURE

B. Runtime Architecture

Each function is implemented in software components by refining the AADL `abstract` components into AADL `process` components containing `threads`. We also define the flow implementation of the initial specification by specifying data flows other the `thread` components.

The **sensing** function is realized by an AADL `process` containing one periodic `thread`. The `thread` execution time is a range between 1 ms and 2 ms.

The **processing** function is realized by two `thread` components:

1) `tf`: receives the raw data from the **sensing** function, filter them (removing inconsistent values) and send it on the process outgoing interfaces. Its execution time ranges from 2 ms to 3 ms.
2) `ts`: receives the data and produces some statistics but does not contribute to the end-to-end flow. Its execution time ranges from 3 ms to 4 ms.

Figure 4 shows the implementation of the **processing** function with these internal components and connections. It also highlight the internal data flow originating from the component incoming interface `datain1`, traversing `tf` and eventually ending on the outgoing interface `dataout`.

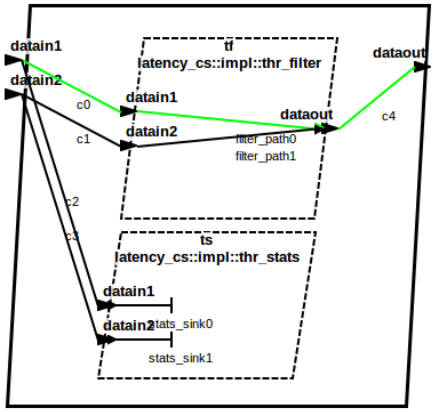

Fig. 4. Flow within the processing function

The **actuating** function is realized by one AADL `process` containing two inter-connected `thread` components:

1) `tc` receives the data, collects the last received values and send the commands to be display on its outgoing interfaces. Its execution time ranges from 1 ms to 3 ms.
2) `td` receives and display the data. Its execution time ranges from 1 ms to 2 ms.

Figure 5 shows the implementation of this function. The line represents the data flow within this component, originating from the incoming external interfaces and ending in the `td thread`.

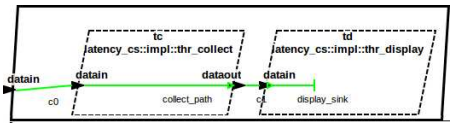

Fig. 5. Flow within the actuating function

This software architecture is then deployed on different execution platform. Next paragraphs present two deployment strategies with different configuration and discuss their impact on system latency.

C. Distributed Deployment

The distributed deployment (as illustrated in figure 6) allocates each AADL `process` to a dedicated AADL `processor`. Inter-process connections are associated to an AADL `bus`.

These `bus` components require between 1 ms and 2 ms to acquire the medium and between 1 us and 10 us to transfer one byte.

In addition, inter-process communications use a dedicated transfer protocol. It adds 1 ms to 2 ms to the total actual latency (required to connect nodes, acknowledge data transfers, etc.).

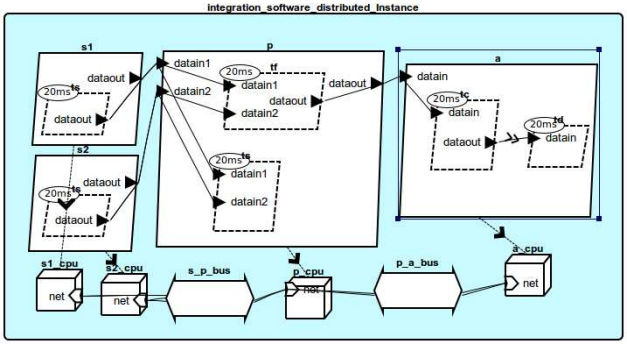

Fig. 6. Distributed Deployment, functions executed on separate processors connected over a bus

D. Integrated Deployment

The integrated deployment (as illustrated in figure 7) allocates each AADL `process` to an isolated partition (AADL `virtual processor` component). All partitions are co-located on the same physical processor. Such a deployment strategy avoids the cost of adding more hardware component (processor and buses), saving integration efforts and costs.

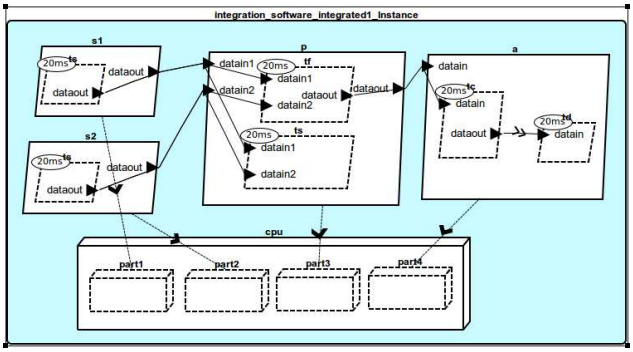

Fig. 7. Integrated Deployment, functions executed in isolated partitions

The processor scheduling policy defines four partition time slots executed periodically at a major frame (MF) of 20 ms:

1) one of 3 ms dedicated to the first **sensing** function
2) another one of 3 ms for the second **sensing** function
3) one of 8 ms for the **processing** function
4) one of 6 ms for the **actuating** function

Partitioned systems have different policies to flush inter-partitions communication. Initially, inter-partitions communications are realized and buffers are flushed after completing an execution cycle. In other words, a data send during a cycle is available to the other partitions after the major frame. This is illustrated in figure 8: a data sent by sensor1 will be available to the processing partition at the next execution cycle. This might

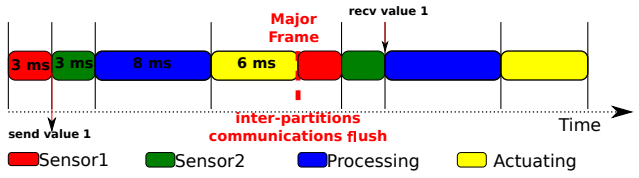

Fig. 8. Inter-Partitions Communications flushed at Major Frame

incur unexpected delays and enforcing latency requirements can be challenging with such architectures.

On the other hand, other execution platforms provide an optimization which consists of flushing inter-partitions communications buffers during partitions switching. As shown in figure 9, this reduces potential inter-partition delays: values sent by a sensor is available immediately to the processing partition.

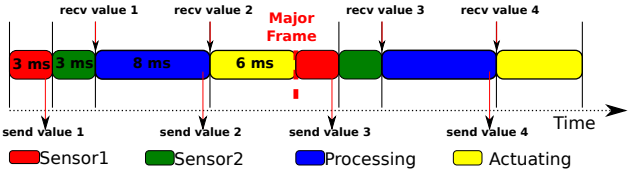

Fig. 9. Inter-Partitions Communication flush after executing each partition

Component	Min Latency	Max Latency
`Sensing`	1 ms	2 ms
`Connection`	2 ms	24 ms
`Processing`	2 ms	3 ms
`Connection`	2 ms	24 ms
`Actuating`	2 ms	5 ms
`Total`	9 ms	58 ms

TABLE III. LATENCY FOR THE DISTRIBUTED ARCHITECTURE

E. Latency Analysis Results

We apply our latency analysis tool to evaluate the end-to-end latency for each the distributed and integrated architectures (considering also both inter-partitions communication policy). We report the results related to the first end-to-end flow, from the first **sensing** function to the **actuating**. Table III shows the latency values for the distributed deployment while tables IV and table V represent the one for the integrated configuration (with different inter-communication policy: flushed at major-frame or at partition switch).

In these different deployment and configuration settings, the software components are the same, so, there is no variation in their related latency (execution time). On the other hand, inter-components communications incur an additional latency. In the distributed deployment, it corresponds to the delay required to sample and transfer data (1 ms bus and 1 ms protocol latency in the best case and 2 ms bus and 2 ms respectively for the worst case). As the components are being distributed over several processors, the connection latency may vary significantly: in the best case, the component receives new data as soon it gets dispatched while in the worst case, the data will be available at the next execution period (20 ms). That is

why in this example, the minimum latency estimate enforces the requirements while the maximum is too high.

In the integrated deployment, the additional latency is added by the inter-partition communications policy and the partitions execution order.

From these results, we can see that part of distributed configuration and one version of the integrated configuration meet the end-to-end latency requirements. On the other hand, the integrated configuration with the communications flushed at major frame fails to meet the requirements, due to the additional delay required to flush inter-partitions communications buffers.

These results highlight the impact of deployment and configuration choices and how they can incur unexpected resources overhead. In the case of the distributed deployment, extra time is required to transport data over system nodes (bus and protocol transfer time). In the case of the integrated deployment, there is a cost related to time isolation (partitions are executed during a fixed amount of time regardless the execution time of their tasks) but also to the inter-partitions communication policy.

Component	Min Latency	Max Latency
Sensing	1 ms	2 ms
Connection	25 ms	24 ms
Processing	2 ms	3 ms
Connection	26 ms	25 ms
Actuating	2 ms	5 ms
Total	56 ms	59 ms

TABLE IV. LATENCY FOR THE INTEGRATED ARCHITECTURE WITH INTER-PARTITION COMMUNICATION FLUSHED AT MAJOR-FRAME

In this case-study, we see that having a fixed execution time for each partition is not a problem as long as inter-partitions communications are flushed when switching partitions. On the other hand, flushing them at the major frame introduces too much delay so that latency requirements could not be met.

These results show that beyond the functional system description, designers must take into consideration all aspects of the architecture, including specific runtime (communication policy, buses, protocols, etc) concerns. Having them in a model support system analysis and avoid potential rework costs when being discovered late in the design process.

Component	Min Latency	Max Latency
Sensing	1 ms	2 ms
Connection	5 ms	4 ms
Processing	2 ms	3 ms
Connection	6 ms	5 ms
Actuating	2 ms	5 ms
Total	16 ms	19 ms

TABLE V. LATENCY FOR THE INTEGRATED ARCHITECTURE WITH INTER-PARTITION COMMUNICATION FLUSHED AFTER EACH PARTITION

V. CONCLUSION

Analyze and evaluate end-to-end latency is of primary importance for Cyber-Physical Systems. Unfortunately, such verification are usually done at the end of the development process, adding eventually a significant rework and reengineering efforts.

In this article, we present an incremental model-based approach to evaluate the end-to-end latency using system specification, avoid potential expensive rework costs. It relies on the Architecture Analysis and Design Language (AADL) and can analyze an abstract functional architecture without execution environment that can be later enriched with specialized execution components (such as processor, buses, etc.). It also offers the opportunity to experience and evaluate the impact of different deployment strategies and configurations, supporting design decision before implementing the system.

Our approach shows that, by using an appropriate level of abstraction, model-based techniques find many potential design defects without having to specify components internals. The case-study demonstrates that, changing the scheduling and deployment concerns impact the enforcement of an end-to-end latency. As CPS are becoming more software-reliant and complex, such issues become typical as well. Using model-based analysis techniques, such as the one of this article discover them early and avoid late rework and product postponement.

ACKNOWLEDGMENT

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. Carnegie Mellon© is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM-0001652

REFERENCES

[1] National Institute of Standards and Technology (NIST), "The Economic Impacts of Inadequate Infrastructure for Software Testing - http://www.nist.gov/director/prog-ofc/report02-3.pdf," Tech. Rep., 2002.

[2] SAE International, *AS5506 - Architecture Analysis and Design Language (AADL)*, 2012.

[3] Carnegie Mellon Software Engineering Institute, "Open Source AADL Tool Environment - http://www.aadl.info," Tech. Rep., 2006.

[4] P. Feiler and J. Hansson, "Flow latency analysis with the Architecture Analysis and Design Language (AADL) - TN CMU/SEI-2007-TN-010," Carnegie Mellon Software Engineering Institute, Tech. Rep., December 2007.

[5] P. Dissaux, O. Marc, S. Rubini, C. Fotsing, V. Gaudel, F. Singhoff, A. Plantec, V. Nguyen-Hong, and H.-N. Tran, "The SMART Project: Multi-Agent Scheduling Simulation of Real-time Architectures," in *Proceedings of the ERTSS 2014 conference*, Toulouse, France, Feb. 2014, pp. –. [Online]. Available: http://hal.univ-brest.fr/hal-00983724

[6] R. Frana, J.-P. Bodeveix, M. Filali, and J.-F. Rolland, "The AADL behaviour annex – experiments and roadmap," *Engineering Complex Computer Systems*, pp. 377–382, July 2007.

[7] J. Hansson and A. Greenhouse, "Modeling and validating security and confidentiality in system architectures," Carnegie Mellon Software Engineering Institute, Tech. Rep., 2008.

[8] P. Feiler, "Challenges in validating safety-critical embedded systems," in *AEROTECH Congress*. SAE, Nov 2009.

[9] Carnegie Mellon Software Engineering Institute, "AADL examples repository," https://github.com/osate/examples/.