Pragmatic cyber-physical systems design based on parametric models

Marisol García-Valls[a], Diego Perez-Palacin[b], Raffaela Mirandola[c]

[a]*Dept. of Telematic Engineering, Universidad Carlos III de Madrid, Leganés, Spain*
[b]*Dept. of Computer Science, Linnaeus University, Växjö, Sweden*
[c]*Dip. di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy*

Abstract

The adaptive nature of *cyber-physical systems* (CPS) comes from the fact that they are deeply immersed in the physical environments that is inherently dynamic. CPS also have stringent requirements on real-time operation and safety that are fulfilled by rigorous model design and verification. In the real-time literature, adaptation is mostly limited to off-line modeling of predicted system transitions. In the adaptive systems literature, adaptation solutions are silent about timely execution and about the underlying hardware possibilities that can potentially speed up execution. This paper presents a solution for designing adaptive cyber-physical systems by using parametric models that are verified during the system execution (i.e., online), so that adaptation decisions are made based on the timing requirements of each particular adaptation event. Our approach allows the system to undergo timely adaptations that exploit the potential parallelism of the software and its execution over multicore processors. We exemplify the approach on a specific use case with autonomous vehicles communication, showing its applicability for situations that require time-bounded online adaptations.

Keywords: CPS, autonomous systems, adaptive systems, verification

1. Introduction

Cyber Physical Systems (CPS) emerged as a new generation of systems that combine properties from real-time, embedded, control, and wireless sen-

Email addresses: `mvalls@it.uc3m.es` (Marisol García-Valls), `diego.perez@lnu.se` (Diego Perez-Palacin), `raffaela.mirandola@polimi.it` (Raffaela Mirandola)

sor network systems. This resulted in a combination of requirements that makes CPS extremely challenging to design and develop. For example, they must reconcile their *real-time* and *open* nature. On the one hand, some of their activities must finish before a given time deadline, and this needs good knowledge of the system activities including their execution time. But, on the other hand, CPS are open to the environment influence [25], and that is a source of unpredictability.

Cyber-physical systems can be autonomous, long lived sytems, mean-ing that they cannot be stoped, modified, and later restarted as this could have unaffordable consequences in terms of cost, data loss, or environmental threats. As a result, CPS will have to evolve during their lifetime to react to the external influence caused by the environment, by other systems, or by human operators. In the simplest form, this evolution can be viewed as a sequence of transitions where the system adapts the behavior and possi-bly the structure to handle the changes in the environment appropriately.

Given their stringent requirements, cyber-physical systems must preserve *correct operation* at all times, also across their transitions. *Correctness* is achieved by applying verification mechanisms to the system model. It is widely accepted nowadays that the model of a cyber-physical system cannot be fully verified off-line because it is deeply immersed in the environment that it monitors and actuates on; and the environment is highly dynamic. This means that during the system operation, new requirements will appear; this forces designers to move from the notion of predictability to the notion of *uncertainty.* To achieve the vision of CPS, new mechanisms need to be developed to support online autonomous decisions by mixing off-line correctness checks with some level of run-time analysis of tentative future transitions.

This paper presents an approach to achieve the evolution of CPS that require timely adaptations, i.e., the system must change as required by the frequency of the execution which is dependent on the environment. For example, if two nodes must exchange information while in motion, the system must adapt its operation to meet a given time deadline for the data exchange that is related to the speed of motion and the communication range. When the system has to adapt, the current model of the system will be enhanced by means of incremental updates that will be verified for correctness. The goal is to obtain a new verified incremental model that preserves the global system properties and addresses well the change in the environment that triggered the system adaptation. All the previous steps must be finished before the time deadline.

The evolution of a cyber-physical system along time is related to the principles of *autonomic computing* [1], also referred to as *self-adaptive* [24, 39]. We take as baseline the idea of MAPE-K loop [24] for autonomic systems; this loop was initially slightly modified in [14] to suit the needs of time-sensitive adaptations as required by CPS. We extend this work by supporting concurrent and parallel execution of activities considering the specific characteristics of current processors that are based on multi-core technology.

1.1. Technical problem and requirements

The technical problem is that of supporting flexible execution of a scenario that contains moving objects and control nodes. The problem is inspired on a small scale laboratory case used for study and validation of cyber-physical systems work. Our scenario has a set of autonomous moving nodes that can have a number of attached sensors. The nodes are toy cars equipped with Raspberri Pi processors connected to an on board IP camera and luminosity sensors that for surveillance of given preselected collision-free areas by video recording; they are equipped with 802.11n wireless connection and bluetooth. Nodes can reach a maximum speed of 52 km/h. There are a number of servers in the track grid equipped with a double core symetric processor that receive the video data from the moving nodes. The general requirements of our target systems (i.e., the autonomous nodes) are:

- Flexible execution. A node should be capable of modifying the set of activities that it runs and/or how activities are run; a node should be capable of finishing the execution of selected activities in shorter time when needed.

- Timely execution. Some activities are time sensitive, i.e., they must finish before a given time deadline. Then, resource management mechanisms are needed within the nodes to ensure timely operation.

- Dynamic behaviour. The current deployment and operation conditions may vary at any time. A moving node records video with some resolution; when entering a different area or upon the receipt of a remote command, it may have to change the resolution dynamically.

- Timely reconfiguration. Moving nodes have to adapt to the changing surrounding environment. For example, if node `a` has to transfer data to `b`, `a` will have to select a given transfer policy (or transfer rate) based on the speed of both moving nodes `a` and `b`.

- Accelerate execution by exploiting the capacity of the underlying multicore processors. We consider symetric processors with various cores.

1.2. Contribution

An initial contribution using Petri nets for fast online decisions in reduced complexity models was provided in [14]; also, [5] analyzed the suitability of CLTLoc for scenarios of the same complexity. However, these works did not at all consider some important aspects that are contributed in this paper:

- Timely adaptation process that considers the execution time of system activities. Cyber-physical systems have to explicitly consider the execution times of activities as well as the maximum deadline for the adaptation process that are application dependent values. In our approach, these are specified as reference values that the adaptation process considers automatically to modify the system model.Also, the adaptation process is more precisely decomposed into a set of steps based on the MAPE-K loop for autonomic systems. The adaptation logic can be embedded in an enhanced middleware as proposed in [12, 13].

- Supporting *scalable execution* to improve application response times. Since at least a decade, single core processors are not available in the market anymore. The existence of multicore systems is an important characteristic that is exploited in our approach to improve response times of activities and of the adaptation logic. We use the term *scalable execution* to indicate the possibility of accelerating the execution of some specific activities, that will yield shorter response times.

- Usage of parametric models. Parametric models are instantiated and evaluated at run-time to decide how the system will be modified to handle the change in the environment. These parametric models are instantiated and analyzed online based on the temporal requirements (execution times) of the activities and of the adaptation process.

- Providing a pragmatic view on the design of a CPS to handle the inherent dynamic behavior and effectively achieve timely adaptation. We exemplify our parametric model construction and evaluation from an applied perspective, on a use case inspired in a real lab setting and use modeling and verification based on Petri nets. As the complexity of CPS design and development is so high, we strongly believe that pragmatic contributions and fundamental theory have to proceed in parallel. Pragmatic contributions introduce more novel ideas that can be faster explored; they can later be used to improve the fundamental theory like formal language enhancements [4] to raise the expression power of models or research on verification mechanisms and tools.

1.3. Paper structure

Section 2 explains related work on adaptation. Section 3 describes the model of the target systems, the requirements for timely adaptation, and how execution can be scaled. Section 4 describes our approach for online adaptation Section 5 describes a use case inspired on a real lab setting for which its activity-based model and adaptation parameters are obtained. Section 6 applies our adaptation approach to the use case, showing how the model is modified by instantiation according to the conditions detected online and how it complies with the specified timing requirements. Section 7 summarizes the work and draws some conclusions.

2. Background and related work

This section presents the contributions that related to ours separated into three perspectives: self-adaptation approaches, modeling the adaptation process, and time sensitive adaptation. At the end of the section, Table 1 offers an overall comparison of the related approaches.
Self-adaptive systems. Several methodologies have been proposed to build self-adaptive systems, like the IBM *MAPE-K* autonomic control loop [24] (later revised in [1]), the *CADA* approach [9], the *Rainbow* architecture [15], the high-level *architectural approach* firstly proposed in [32], to cite a few. These approaches consider the fact that systems are located within a broader environment whose behavior is out of the control of the system[38]. The environment may have effects on the system and may probably require it to take some action; these effects are observed and evaluated by the system itself. These approaches are based on the premise that implementing self-managing attributes involves an intelligent control loop with a common general structure. Systems that adhere to this loop collect information from the system itself, make decisions and then adjust the system itself as necessary. This

structure involves two major elements: a *managed subsystem* and an *autonomic manager or managing subsystem*. The managing subsystem controls the managed subsystem. The managing subsystem comprises the *adaptation logic* that deals with one or more concerns and the *control loop* that is implemented as consisting of four stages that share knowledge.

A different reference model for a self-adaptive software system, called *3-Layers-Architecture* [26], suggests to architect the system along three different layers, that interact with each other by reporting status information to the above layer and issuing adaptation directives to the layer below. The bottom layer (component control) is concerned with adaptation at the level of single components (i.e., services in the SOA domain). The middle layer (change management) reactively uses a pre-specified set of plans to adapt the system consisting of components at the lower layer. When these plans are no longer able to meet the system goals, or when new goals are introduced, the upper layer (goal management) determines new adaptation plans. This architecture can be mapped onto the managed/managing subsystems schema as well, just considering two different levels of analysis and plan corresponding to the change and goal management, respectively.

In what follows, we refer to the specific implementation of the autonomic control loop called *MAPE-K* (Monitor, Analyze, Plan, Execute and Knowledge) [24]. The *Monitor component* observes the relevant properties of the managed subsystem and feeds the *Knowledge*; the *Analyze component* checks the status of the managed subsystem (stored within the knowledge) against the stated QoS and functional requirements; the *Plan component* decides the set of needed reconfiguration actions based on the *Knowledge* as well as on the current models; and finally the *Execute component* enacts the reconfiguration implementing the plan output into the managed subsystem.

As indicated in [30], around 60% of the contributions on adaptive sys-tems use MAPE control loop for the adaptation; moreover, around 66% of the works that apply self-adaptation to cyber-physical systems address efficiency/performance applied to particular domains such as transportation (e.g. vehicular networks [37]), robot navigation [18], energy [36], or manufacturing [27]). [33] applies MAPE-K loop for performing efficient resource allocation (i.e., mostly memory and processor) applying different learning techniques that processed the systems' monitored data. Optimization of the behavior is also a key aspect; there are a number of strategies such as those described in [29] concerning decision making for autonomic systems.
Modeling. Model-driven techniques are a popular approach to CPS de-

sign as reflected in [7]. Among the first ones to outline a methodology for model-based design of CPS, we find [23]; the adaptation here is limited to the control loop operation and its integration in a static component-based software design. Managing the scale and complexity of CPS benefits from component-based modeling contributions that provide reuse, encapsulation, and separation of concerns, and that also support adaptation of the components and their relations. In [6], component-based development is utilized as ground to apply selected software engineering approaches for designing cyber-physical systems. In [21], it is evidenced that a number of techniques for self adaptation ranging from component ensembles to representational languages can be applied to CPS.

Following the claim that component-based software engineering is not sufficient for modeling ensembles (or a collection of autonomic entity that collaborates for some global goal), [19] proposes a formal approach to model the structural aspects of collaborative entities and labeled transition systems to specify their dynamic behavior. This work targets at pervasive types of structures based on the roles played by components where timeliness is not a driver. Overall, the above techniques remain at a high abstraction level mainly focusing at the component specification and their interactions, being mostly silent about real-time adaptation.

Among the alternatives to represent adaptation concerns in CPS modeling. One of them is [40] that uses timing analysis to provide CPS with a sufficient temporal bound to anticipate adaptation decisions. Work in[35] uses formal models to verify probabilistic reliability properties in dynamic cyber-physical systems and smart cyber-physical spaces. The formalisms used are Markov Decision Processes, probabilistic Computation Tree Logic and model checking.

Time bounded adaptation. To achieve timely adaptations, traditional real-time systems (predecessors of CPS) used mode change protocols. An execution based on mode changes consists of a set of operation modes and a set of transitions among them, and in each mode only a given set of activities can run. This technique is very restrictive and static, but allowed for schedulability analysis [34] to guarantee temporal bounds on the mode change transitions. General self-adaptive systems also suffer the time consuming reasoning process that is carried out at runtime. To mitigate it, work in [10] proposes a time efficient method for applying model checking at runtime.

Adaptation and reconfiguration applied to distributed systems has been studied since decades, but contributions are silent about temporal guarantees for the adaptation transitions. For instance, [20] supports changes in the software based on the identification of a set of safe adaptation points. Recent work [13] has enhanced the logic of distribution middleware to embed adaptation and reconfiguration logic to support time bounded transitions in service oriented systems. Overall, these contributions also do not target CPS.

Table 1 shows selected works on adaptation. It lays down the main characteristics of these works: adaptation type; verification model; whether it explicitly targets CPS; support for real-time adaptation; and the main concept of the work. It has the goal of laying down their main characteristics easy the comparison with our contribution.

Table 1: Classification of state of the art contributions on adaptation

Work	**Type**	**Verif.**	**Target**	**RT**	**Concept**
[38]	Online	-	Generic	-	MAPE patterns
[4]	Offline	Model check	Embed. sys.	-	Contract language
[7]	Offline	-	Generic	-	Components
[34]	Offline	RT analysis	RT sys.	✓	Operation modes
[19, 20]	Offline	-	Generic	-	Components
[20]	Online	-	Generic	-	SW modules
[13]	Online	RT analysis	SOA RT sys.	✓	Distrib. RT reconf.
[14]	Online	Petri net	CPS	-	Service selection
[5]	Offline	CLTLoc	CPS	-	Activity adaptation
[23]	Offline	-	CPS	-	Components
[18]	Offline	-	Swarms	-	Components
[33]	Online	Learning	CPS	-	Resource alloc.
[27]	Offline	-	CPS-Grid	-	Cloud
[36]	Offline	-	CPS-Production	-	Multi agents
[40, 37]	Online	-	CPS-Vehicular	✓	Control theory
[10]	Online	Model check	Generic	-	Probability

From the above selected works it can be seen that none of them considers tasks with dynamic priorities in a processor multicore structure; our contribution uses this characteristic to speed up the execution of parallelizable activities. Moreover, our work supports online adaptation, including the verification of the overall model behavior as well as temporal verification that analyses the temporal guarantees.

3. System model

The system model that supports timely adaptations includes: (*i*) a software structure based on activities, (*ii*) the temporal requirements of adaptations, and (*iii*) mapping between software and hardware parallelism.

3.1. Software structure

The system is structured in *activities*. In the literature, similar concepts are referred to as activities, components, or services. Activities are light-weight self-contained code units with a well defined interface that can communicate with other activities only via message exchanges. Then, the set of n activities of a system j is indicated as A_i^j (where $i = 1, .., n$).

The timing characteristics of an activity are: computation time (et) is the time to finish its computation; *deadline* (d) that is the maximum time (relative to the activation of the activity) that must not be exceeded by the computation of an activity; *priority* (p) is a value showing the relative urgency with respect to the other activities in the system.

A system is viewed as a graph (G) of activities that can incrementally evolve during the execution lifetime: $G = \{A, R, gt, L\}$, where G represents a generic graph composed of a set of nodes that are the activities A; a set of directed arcs (R) connecting pairs of nodes; the end-to-end deadline (gt) or maximum time to execute the worst case or critical path of the graph and the set of limitations or restrictions to the graph (L).

Verification of a cyber-physical system is typically done from a schedulability analysis perspective that is the main target in real-time theory; the rest of properties are not contemplated explicitly. Considering that the set of complex properties of CPS can be synthesized as a worst case execution *time* parameter is a very strong assumption and it is often not a realistic one. In our approach, the restrictions (L) express the set of properties of different nature (i.e., not only *time*) that have to be checked jointly.

3.2. Timely adaptation

The changing environmental conditions trigger adaptation *events* that are detected by the system. As a result of an adaptation event and in order to handle the new external conditions, the system model may have to be modified. In summary, upon the detection of an adaptation event:

- a decision process is run that analyses the situation and creates a modified model capable of handling the new external conditions;
- the adaptation process has an associated deadline (d_a), i.e., a maximum time to complete the adaptation that is an application dependant value.

The time for the system to complete an adaptation is the *adaptation time* (*at*). *at* includes different temporal costs: (1) the execution time of the activities, (2) possible synchronization penalties, and (3) execution overlaps.

Changing the model implies changing some of the running activities. Since a cyber-physical system must be correct by construction, a model modification must be preceded by an online analysis to guarantee that the new model that will be applied is also correct. Formal methods are used for this purpose. The adaptation process introduces extra overhead as it is implemented with additional code. This overhead can be absorbed building parallel software models in which activities can be run simultaneously in the available hardware cores. The actual impact of this approach on the execution can only be analysed when the adaptation process is triggered and it will depend on the status of the system at the precise instant.

3.3. Scalability

Currently, there are no single core processors available in the market. Our approach exploits the multicore nature of current processors. *Parallelizable activities* at software level are mapped to a specific processor *core* at hardware level, so different activities can run simultaneously. Identifying parallelizable activities is key; these are the activities that do not have synchronization requirements during their execution although they may need to synchronize at the end of their operation.

The proposed model can combine, both, activities that need to run sequentially and others that can run in parallel. Then, application graphs (see Figure 1) will have concurrent parts for parallelizable tasks that will be possibly allocated to different cores, and sequential ones for non parallelizable activities. The allocation of parallelizable activities to different cores simultaneously will speed up execution and shorten the response time to an event. Parallellizable activities such as A_2 and A_3 of Figure 1, can run in different cores simultaneously, in the same core sequentially, or in either way depending on what is most convenient given the current situation.

A *decision function* is applied to determine which activities can be parallelized, taking as input the set of parallellizable activities and a set of criteria

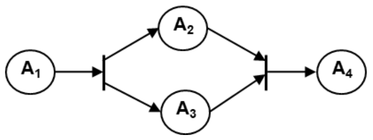

Figure 1: *Adaptable execution through parallelizable activities*

parameters as follows: the *activity set*; the *current system state*, and a set of *reference values*. The *activity set* of the system comprises all the activities whether parallelizable or sequential. The *current system state* indicates the resource availability. To obtain a result, a graph set will be generated, that contains the different execution possibilities over the candidate activities. The criteria for selecting a feasible solution, i.e., what model to apply is expressed as a set of *reference values*; these are known a priori and express the suitability of a solution.

The decision function is applied to the current model, i.e., the initial graph. The obtained results are compared with respect to the set of provided reference values. Then, a solution/model (i.e., a specific graph) is selected.

It should be considered that the integration of a parallelizable activity into a running model may have negative effects on the calculation of the end to end timeliness of other activities. Consequently, it is needed to perform an online analysis of the situation as it is the only means to determine the current system availability and associated execution possibilities.

4. Adaptation by model instantiation at run-time

The adaptation process, sketched in Figure 2, relies on the basis of an *original parametric model* of the system. This is a model that can be instantiated to adapt the system behavior to the needs of the changing environment. The adaptation process is done as the system is in execution, i.e., at run time. As a result, all the steps in the adaptation process have to be efficient in time to meet the overall application dependent deadline.

4.1. Steps in the adaptation process

Upon the occurrence of a transition event, G^{init} (the original system model or graph) must be changed to G^{target} (the target model). The target model must be one that complies to the new situation of the environment.

After the change to the target model, the set of activities will not be the same.

Once an *adaptation event* has been triggered, the adaptation process follows the steps given below :

1. *Read the event information* to determine the type of needed change, its inherent urgency, and any other context information that is relevant for handling the event. This step is associated with the *Monitor* component in the MAPE-K loop.
2. *Create incremental models* that are also named *tentative models*, that have the goal of satisfying the new environment conditions. A tentative model G^{tent} is derived from the current model G^{init} with the needed additions to handle the new situation. This step is associated with the query to the *Knowledge* of the *Analyze* activity in the MAPE-K loop.
3. *Analyze tentative model* individually and collect results with respect to the expected properties of the system as the result of the simulated execution of these models. This step is associated with the *Analyze* step in the MAPE-K loop.
4. *Decide* to what G^{tent} the system should evolve towards in order to adapt to the new environment situation. For this, the expected property values of the tentative models are compared against the set of *reference values* that are the values to satisfy. This step is associated with a simple *Plan* step in the MAPE-K loop.
5. Adapt the system to execute as represented by the chosen model G^{tent}. This phase is associated with a simple *Execute* step in the MAPE-K loop.

If there is not any G^{tent} whose properties accomplish the reference values, a back up decision must be taken, such as transitioning to an *emergency configuration*. Section 4.3 explains this parameterization.

4.2. Scalable execution

The scalable execution of a system depends on two main factors:

1. *The software/application nature.* As an example consider the video processing algorithms, they are typically scalable, providing a range of output qualities depending on the assigned computational resources such as time, i.e., the more time to execute, the better the quality of the processed image. Consider two video activities that are in execution, if

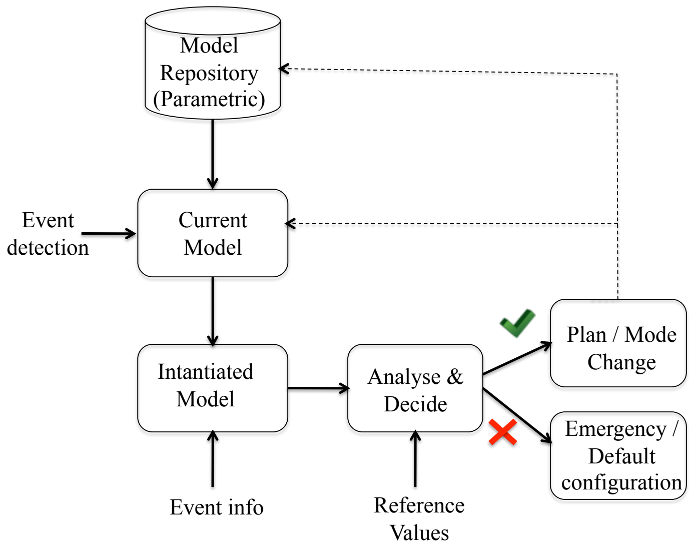

Figure 2: *Overview of adaptation process*

one is assigned more processor cycles/time, the other one will have to scale down its delivered quality, and vice versa. For instance, work in [14] addressed this type of factor.

2. *The hardware characteristics.* Current processor architectures are multicore, capable of running several actions in parallel. The basic assumption with respect to the software design is that actions that are parallelizable can execute at the same time, speeding the overall computation time which may benefit processor intensive activities, and improve the response time of applications.

In the presence of scalable applications, some activities have variable execution times depending on the specific execution conditions such as the volume of the processed data. Let us consider a system that must execute a video file transfer action that is scalable: it can be provided by a set of y different variants of this activity, each delivering various output qualities. The selection of the activity variant will depend on the current situation of the system and will have to be decided on-line by generating a *tentative future model* G^{tent} that will be analyzed and verified to guarantee that the system properties are preserved, i.e., the model is correct. If the tentative model is successfully verified, then it will become the new model of the system.

4.3. Parametric model creation and execution

The system model G contains a set of nodes or activities that are activated by means of transitions triggered by event occurrences such as completion of previous activities.

To support the evolution of the system behavior, the system adapts by transitioning from a model G to a model G^{tent}. The adaptation is guided by a set of rules of the form: `if [`c`] then [`G^{tent}`]`, where the adaptation conditions are c, and are parametric. Adaptation conditions are expressions containing variables, and they are evaluated at run-time. The structure of an adaptation condition c is an expression of the form: $f(PV) \oplus R$, where:

- *parametric variables* PV are represented by $PV = \{v_0, ..., v_n\}$ and their values depend on the current system situation and environment status;
- *logic operation* ($\oplus$) indicates a comparison with respect to a value of reference;
- *reference values* are represented by the set $R = \{r_0, ..., r_m\}$; these are application dependent values (expected values) that guide the transition of the system to a specific model;
- f is a function over the parametric variables that returns model-specific values to be compared to the reference values: $f : PV \rightarrow R$

Therefore, in the adaptation condition the expression determines a specific global evaluation of the system behavior whereas the reference values represent the limits that determine whether a transition is to be fired.

As an example, let us imagine that the system has the possibility of running two different computation models G^1 and G^2 which contain activities A_1 and A_2, respectively. These activities differ only on the precision of the output result (A_1 delivers higher precision results compared to A_2) and on the time that they consume to deliver such output (A_1 consumes more time than A_2). The highest precision operation is the preferred one, but it is not always possible to execute it since the system may have consumed more time on other actions, having less time to complete these activities before the deadline. Therefore, the default choice will be to run A_1 as it provides the best possible output quality; however, the decision on which action will be run depends on the remaining time before the deadline ($remT$) that is actually the used reference value in this case: $R = \{r_0\}$ and $r_0 = remT$.

This is shown by the following rule: `if` [$f(A_1) > remT$] ***then*** [A_2], where f in this case is the function that returns the execution time *et* of an action, i.e., $f(A_1)$ returns the required execution time of action A_1.

The decision function f can be complex as it sets the relation between the observed *parametric variables* whose values are compared to the *reference values*. The *Analyze* activity of the MAPE-K self-adaptive control loop runs the decision function. In our approach, f is implemented by a model verification engine that verifies models that are instantiated with the set of *parametric values*; these models are evaluated yielding values that are compared to the *reference values*. From the current model G, instantiations can be obtained as tentative models G^{tent} that will undergo online verification.

5. Use case

We validate our approach through a prototype of an autonomous system that coordinates the operation of vehicles that move over predefined areas performing video surveillance activities. Video data collected by such vehicles are transmitted to servers that process and upload them to a storage system.

Servers are located at specific locations, and the video file transmission occurs when the moving vehicles are within the allowed communication range. Servers are also responsible for handling ***alarm events*** that are sporadic occurrences triggered by the detection of some special environmental condition. Such conditions may have very important consequences on the system, e.g., disable some vehicle routes or consider the presence of additional vehicles.

Servers run in two different operational modes: *normal* and *alarm* handling. This section describes how transition across modes done at the server making use of the online adaptation approach explained in sections 3 and 4.

5.1. Server normal operation model

In the normal operation mode, the server receives requests from vehicles to transmit and upload video files as they pass close to the server. The *moving vehicles* act as *client nodes*. Then, the deadline for file transmission depends on the communications reachability range and the speed of the moving clients. Assuming that every client has the same reachability range (and that every client provides its speed to the server upon communication set up), the duration of the file transmission will have a time deadline D^t that is calculated as the reachability range divided by the client speed. Then, the interaction between client and server should finish within D^t time units since the start.

Figure 3 shows the interactions between server and vehicle through a UML activity diagram extended with MARTE-like [31] performance notation. Precisely, the set of server activities in the *normal operation mode* are $A = \{A_0, ..., A_5\} = \{InformClient, Transmit, Write, Read, Upload, Compute\}$.

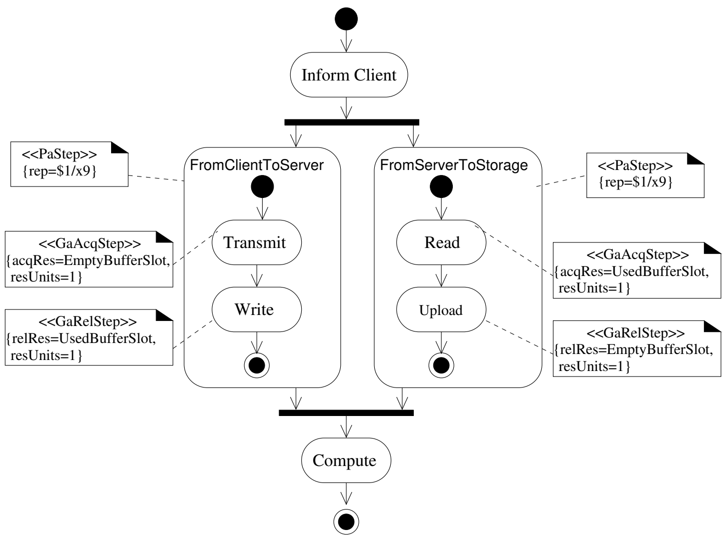

Figure 3: *Server behavior in the normal operation mode.*

When a vehicle is detected, the server runs activity `Inform Client` (with an execution time of x_1 time units) to first inform the vehicle client that it can start the transmission. The client then sends the data to the server, and the server uploads the data to the storage system. This is indicated by activities `Transmit, Write, Read` and `Upload` of Figure 3. After the data has been uploaded, the server runs activity `compute` on the vehicle information.

Detailed considerations of the data sending are:

1. Transmitted data size ranges from x_7MB to x_8MB. Data is split in independently processed and transmitted blocks of x_9KB. For each block size, activities `Transmit, Write, Read` and `Upload` have an execu-tion time of x_2, x_3, x_4, or x_5 time units, respectively. This is mod-eled as $rep = \$1/x_9$ iterations over activities `From Client to Server` and `From Server to Storage`, where \$1 is data length (in the range [x_7MB,x_8MB]). The last iteration of activity `From Client to Server` must finish before D^t to fulfill the deadline.

2. Figure 3 shows the parallel workflow that is supported by the multicore processor. Then, a data block can be transmitted concurrently to the upload of a received block to the data storage system. To synchronize transmission and uploading, a buffer size of x_6MB is kept at the server that can store up to $1024x_6/x_9$ blocks.
3. `Transmit` activity can suffer from environmental interference resulting in deadline miss. If so, vehicles take actions such as stop movement to keep in the reachability range. The approach supports the accumulation of errors in the transmission of data blocks. For this, the deadline is relaxed: proper operation is achieved if the transmission deadline is fulfilled in 99.5% of the vehicle-server interactions. Probability distributions and random transmission times for each data blocks adjust to this characteristic.

5.2. Uncertainty management: Alarm handling

Their unpredictable surrounding environment, forces cyber-physical systems to face unexpected situations that may trigger alarms requiring timely reaction. In the proposed use case, this is illustrated as follows. The server detects alarms raised due to some problem encountered by the moving clients along their circulation path that can affect the plant safety (e.g., changes in environmental conditions, meeting additional vehicles over the same path, etc.). Among other actions, this alarm requires the server to run a `Path Recalculation` activity, which is the most time consuming activity along the alarm management process. There are a number of contributions in the literature regarding path calculation algorithms. Examples are shortest path algorithms of Dijkstra [8] and Floyd [11]; the latter has been revisited also for negative cycles in [22].

This implies that the set of activities, A, is enhanced with this additional activity in the *alarm* mode (see Figure 4(a)) provided sequentially as run:

- `Read environmental conditions`: It runs on the first place to collect data of the current environment status and physical restrictions for the moving clients.

- `Recalculate the vehicle path`: Path calculation can be very complex depending on the number of restrictions over the environment. The time taken by this activity is assumed to be bounded to a worst case scenario where the upper bound of the restrictions is known.

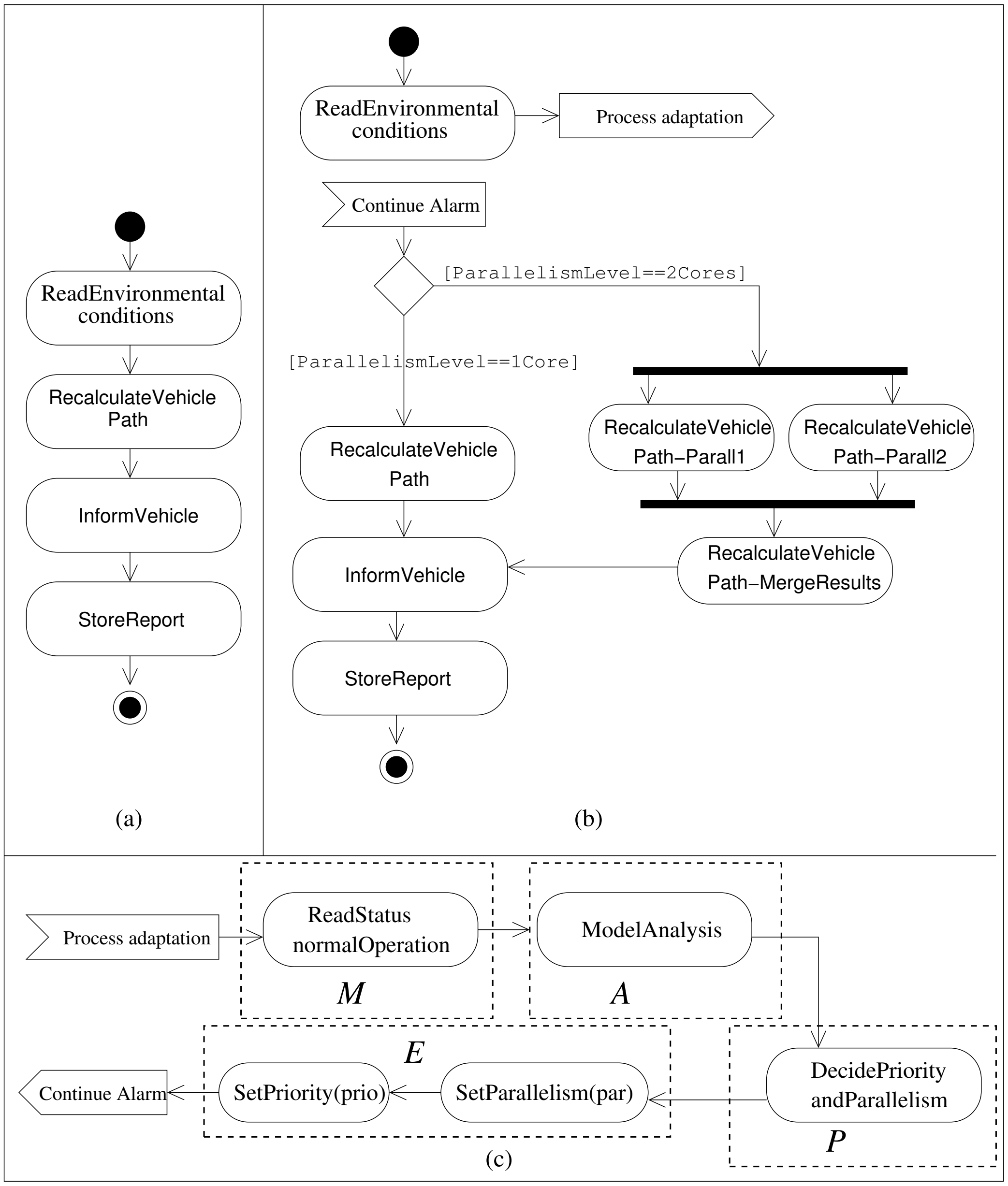
ReadEnvironmental conditions
RecalculateVehicle Path
InformVehicle
StoreReport
(a)
ReadEnvironmental conditions
Process adaptation
Continue Alarm
[ParallelismLevel==2Cores]
[ParallelismLevel==1Core]
RecalculateVehicle Path
RecalculateVehicle Path−Parall1
RecalculateVehicle Path−Parall2
RecalculateVehicle Path−MergeResults
InformVehicle
StoreReport
(b)
Process adaptation
ReadStatus normalOperation
M
ModelAnalysis
A
E
DecidePriority andParallelism
P
Continue Alarm
SetPriority(prio)
SetParallelism(par)
(c)

Figure 4: *Server alarm management: a) operations to perform, b) operations to perform with self-adaptation process c) detail of the activities in the self-adaptation process*

- `Inform vehicle`: It communicates the new path to be followed by the moving client.

- `Store report`: It registers the alarm in a log file.

The alarm handling activities have a deadline, D^a, that should be fulfilled, at least, 99.999% of the time.

The multicore nature of processors and the existence of parallelizable activities are exploited in our approach to yield the following benefits:

- The available processor is dual core; the alarm handling process can use both of them to speed up the response to alarm.

- Activity `recalculate a vehicle path`, which is the most computationally expensive activity in the process, is *parallelizable*; therefore, the system can run either a serial version of this activity or a parallel one. The rest of activities are not parallelizable, so they have to execute either in one core or in the other, but not on both simultaneously.

5.3. System self-adaptation

Both the normal operation and the alarm handling processes will meet the deadline if run in isolation. However, they can miss their deadline if an alarm event is triggered while the server running in normal mode, i.e., receiving a video file transmission from a moving client.

A first analysis might result in deciding that the alarm mode should have higher priority over the normal mode. The reason is that the alarm is likely to have a shorter deadline. However, this cannot be inferred for all cases since it may result in normal mode activities missing their deadline; as Section 6 will show. For instance, operation `Transmit`, which is also a real-time activity and has an associated deadline, would be preempted by the alarm and could fail to meet its deadline.

The possible workflows when an alarm event arrives to the server are:

- The alarm mode activities run after the normal mode activities have finished. Scenario 1 in Appendix A exemplifies need for this workflow in order to meet the deadlines of all processes.

- Activity `from server to storage` of the normal mode is preempted. Here, the alarm activities are executed in one core while the transmission from the moving client to the server continues in the other core until the buffer is full. Scenario 2 in Appendix A shows its utility.

- The normal mode activities of the system are preempted. The alarm mode activities are run in both cores and, when they are finished, the normal mode activities continue execution. Scenario 3 in Appendix A illustrates its utility.

When an alarm is detected, the server reads the data about the environmental conditions that can affect the duration of the handling process, and it launches the adaptation. The alarm mode activities set (A set) is enhanced so that the adaptation process is integrated into this set; this results in a modified adaptation management workflow.

Figure 4(b) illustrates the workflow of activities given that the alarm process can parallelize its `recalculate vehicle Path` depending on the decision taken by the adaptation managing process. The first activity `read environmental conditions` is equivalent to its homonym in part (a). First, the adaptation process is launched using the data on the environmental conditions. When the adaptation process finishes, the alarm process continues to run. Depending on the decision taken by the adaptation process, the alarm will implicitly run with the decided priority and it will explicitly split its `read environmental conditions` into two parallel activities.

In turn, the adaptation process, which is depicted in Figure 4(c) waits until the arrival of the signal to start its execution. Then, it proceeds as follows: (*i*) it reads the status and progress of the normal operation, which corresponds to the *Monitor* activity of MAPE-K; (*ii*) its instantiates and analyzes model G and different alternative G^{tent} to obtain the expected properties that are comparable with the reference values, which corresponds to the *Analyze* activity of MAPE-K; (*iii*) it decides the priority and parallelization level with which the alarm handling activities should run, corresponding to *Plan* activity and is further explained in the next paragraphs; and (*iv*) it sets the decided parallelism and priority, corresponding to the *Execute* phase. After, it notifies the alarm mode to continue execution.

Table 2: Variables to be set in the parameterizable model for timely handling of the normal operation of the system and the alarm event

Parameter	**Reference values**
`Priority`	$[0, 1, 2]$
`ParalelismLevel`	*[1 core, 2 cores]*

Table 2 shows the model parameters. Their values are decided by activity *Decide Priority and Parallelism* according to the provided reference values. Setting these parameters is an actual instantiation of the parametric model, that is a mechanism for supporting adaptation and system *evolution*. Activity *Decide Priority and Parallelism* is run online by the server providing the following results:

- the *parallelization level* of `recalculate vehicle path` activity, with possible values:
  - *sequentially*, if the adaptation analysis results yield a *parallelismLevel=1Core*, or
  - *parallelized*, if the adaptation analysis yield a *parallelismLevel=2Core*),
- the *priority* of the rest of the workflow activities; possible values are:
  - 0, the alarm mode activities will wait until the execution of the normal mode activities finishes;
  - 1, starting immediately the sequential execution of the alarm mode activities by preempting the core that uploads the data to the storage system;
  - 2, starting immediately the parallel execution of the alarm by preempting both cores.

  If the decision is to parallelize, the alarm process will split activity `recalculate vehicle path` in two activities that will run in parallel (i.e,. `parall1` and `parall2`) and a final activity `mergeResults` to integrate their results.

Table 3 shows an overview of the activities run by the server and their priority and parallelization level properties.

Priorities of `Transmit` and `Upload` activities reflect the previous description. `Upload` is best effort, so its priority value is the lowest one: 0. The priority of `Transmit` is higher than the previous one: 1. The priority of activity `RecalculateVehiclePath` is a *variable of the parametric model*; it is decided within the adaptation process.

In case that the adaptation analysis results show that the requirements of both normal and alarm mode activities cannot be fulfilled (e.g., the deadlines are not met within the given reference values and none of the adaptation

Table 3: Description of the server activities per execution mode

Normal mode actitivies					
Activity	**Priority**	**Activity**	**Priority**	**Activity**	**Priority**
Inform Client	1	Transmit	1	Write	1
Read	0	Upload	0	Compute	1

Alarm mode activities		
Activity	**Priority**	**Parallel**
Read Environmental conditions	2	-
Recalculate Vehicle Path	`prio_var`	✓
Recalculate Vehicle Path-Merge Results	`prio_var`	-
Inform Vehicle	`prio_var`	-
Store Report	`prio_var`	-
Read Status Normal Operation	2	-
Model Analysis	2	-
Decide Priority and Parallelism	2	-
Set Priority - Set Parallelism	2	-

conditions c in `if` `[`c`]` `then` `[`G^{tent}`]` is true), according to Figure 2 the system switches to an *Emergency configuration*. In this example, the emergency configuration corresponds to the one that ensures plant safety (i.e., the alarm process is immediately executed using both cores although the video transmission operation could not be completed).

The properties and particularities of the use case conform a target sys-tem with the following general characteristics: (i) *open* as it may dynamically modify the functionality and code that it executes, (ii) *real-time* as there are some activities that must finish before a specified deadline expires,(iii) *correctly designed* because it must always execute under a model that has been previously verified and checked against the set of desired properties; (iv) *priority-based*, as it can also handle the occurrence of events that should be handled with different urgency (e.g., an alarm); (v) *concurrent* at both intra and internode level: modern processors are multicore systems and they support execution speed up and increased throughput for parallelizable activities.

Table 4: Reference values

Name	**Value**	**Description**
D^a **fulfillment level**	99.999%	Probability of finishing alarm mode activities
D^t **fulfillment level**	99.5%	Probability of finishing normal mode activities

6. Validation

This section validates the proposed approach through its application to the self-adaptive case study presented in previous section.It also reports the performance results obtained from the execution of the model analysis, and discusses the limitations of the approach.

6.1. Adaptation process at work

The adaptation process must deal with adaptation analysis and decisions at run-time. The system will then adapt by assigning the appropriate computational resources (cores) to the `RecalculateVehiclePath` activity in a way that guarantees that deadlines are met even in the case that there is an on-going transmission from a moving client to the server.

Since the status and situation of vehicles when an alarm occurs is not pre-dictable, the computation time required by `RecalculateVehiclePath` will only be known at execution time concretely, only after `ReadEnvironmentalConditions` activity of the alarm process has executed. Next subsections describe how we applied the model instantiation and adaptation analysis activity to the use case.

6.1.1. Tentative model generation.

Figure 2 illustrated the adaptation process in which a repository stores parametric models models for each behavior of the system. In our case study, there are three tentative behaviors, and thus three parametric models or target system configurations, namely G^{tent1}, G^{tent2}, and G^{tent3}. For convenience, models are stored in the same language that will be used for their execution and analysis, *Generalized Stochastic Petri nets* (GSPN) [2].

Figure 5 depicts these three models. Part (a) shows G^{tent1}, the system behavior when the alarm mode activities wait until the execution of the transmission between the moving client and server in the normal mode has finished. Part (b) shows G^{tent2}, the system behavior when the alarm mode activities are immediately executed using one core and leaving the other core for the normal mode activities. Part (c) shows G^{tent3}, the system behavior when the alarm mode activities are immediately executed in both cores by preempting the normal mode activities, which can resume when the alarm has been completely handled.

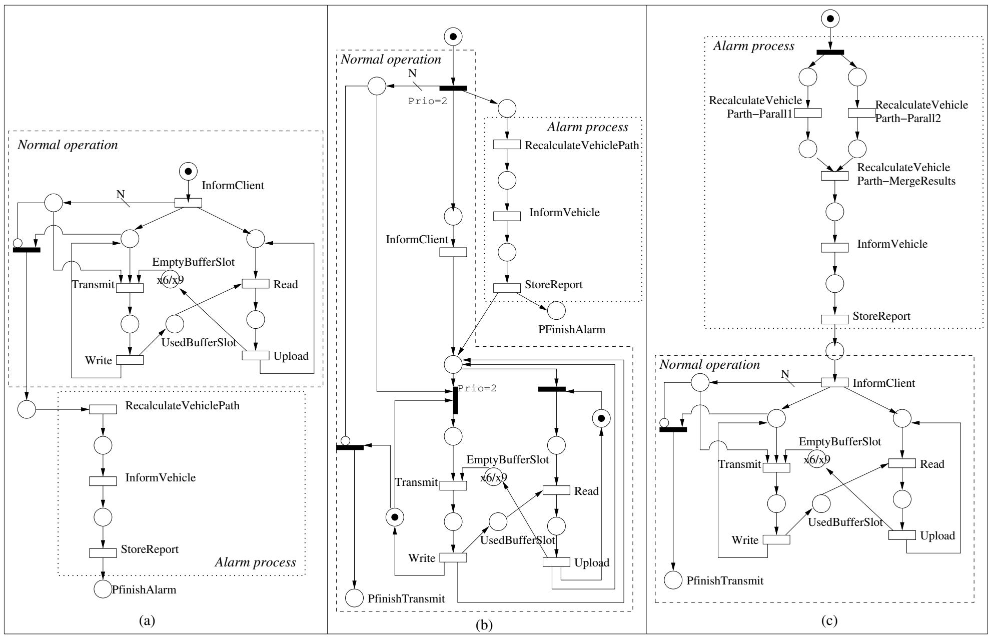

Figure 5: *Petri nets to analyze the appropriate level of parallelism and priority of the alarm process. These models are stored in the Model Repository. (a) represents system behavior G^{tent1}, (b) G^{tent2} and (c) G^{tent3}*

The *model instantiation* activity of the adaptation process in Figure 2 gives values to the Petri net model parameters. The environmental conditions that are used to instantiate in the Petri nets and which are represented as the *event info* in Figure 2, are: the expected execution time of *Inform-Client* activity, the number of remaining blocks to be sent from the vehicle to the server, the number of buffer slots currently used, and the number of elements that the vehicle path recalculation activity will need to consider (i.e. the size of the problem for the path recalculation). Each of these environmental conditions only affects a single parameter in the Petri net model; for instance, the number of remaining blocks to be sent affects only te arc weight parameterized as N in the Petri net models in Figure 5. More details on these parameters are provided in Appendix A.

Conversely, parameters in the Petri nets that are independent of the current environmental conditions are the mean execution times of activities: *mergeResults* and *InformVehicle* in the alarm process; and *read, upload trans-mit*, and *write* of a data block in the normal operation. These parameters do not need to be instantiated and can be set as constants in the model.

6.1.2. Role of model results in the adaptation decision

The *Analyse&decide* activity of the adaptation process analyses the Petri net models to obtain information regarding the expected completion times of both, the alarm mode and the normal mode activities. The logic of *Analyse&decide* activity is contained in Table 5. It relates the results of the model evaluation with the adaptation decisions.

Table 5: Incremental model creation: *Rules modification*

`if` $[P(\#FinishAlarm(D^a) = 1) \geq 0.99999 \ \mid \ G^{tent1}]$ `then`   [`prio_var` $= 0 \wedge$ `Paralelism_level` = one core]
`else if` $[(P(\#FinishAlarm(D^a)) = 1 \geq 0.99999 \ \mid \ G^{tent2}) \wedge$ $(P(\#FinishTransmit(D^t_{remain}) = 1) \geq 0.995 \ \mid \ G^{tent2})]$ `then`   [`prio_var` $= 1 \wedge$ `Paralelism_level` = one core]
`else if` $[P(\#FinishTransmit(D^t_{remain}) = 1) \geq 0.995 \ \mid \ G^{tent3})]$ `then`   [`prio_var` $= 2 \wedge$ `Paralelism_level` = two cores]
`else` # emergency configuration   [`prio_var` $= 2 \wedge$ `Paralelism_level` = two cores]

The expression $P(\#FinishAlarm(D^a) = 1)$ refers to the results of the Petri net analysis, specifically to the probability of being a token in place *FinishAlarm* after D^a time has elapsed. Similarly, $P(\#FinishTransmit(D^t_{remain}) = 1)$ refers to the probability of completing the client-server communication within time D^t_{remain}. Utilization and meaning of D^t_{remain} is explained later. Expression $(P\ (\#FinishAlarm(D^a)) = 1 \geq 0.99999 \mid G^{tent2})$ refers to the computation of the aforementioned probability in the tentative model G^{tent2} and checking whether its value is higher than the *reference value* 0.99999.

The amount of time available to finish the alarm mode activities is D^a, while the amount of time available to finish the transmission, called D^t_{remain} in the table, is not immediate but it requires the following calculation: being D^t the deadline to finish the transmission computed from the reachability range and the communicated vehicle speed, being $t_{current}$ the current time, and being t_{start} the moment in which the execution of the normal mode activities started, the *remaining time to finish the transmission* is $D^t_{remain} = D^t - (t_{current} - t_{start})$.

Looking at Table 5 as a workflow with the logic for the adaptation decision, we obtain the decision process in Figure 6, where each of its decision activity contains the condition in one of the rows in Table 5.

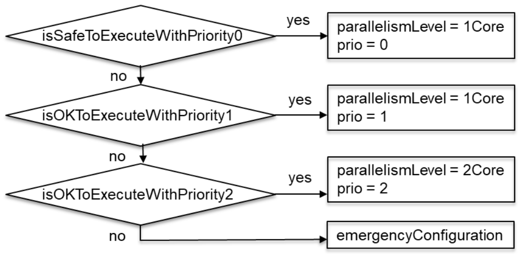

Figure 6: *Adaptation logic*

6.1.3. Model evaluation and Decision

The decision on `isSafeToExecuteWithPriority0` in Figure 6 is obtained from the evaluation of the condition in the first row in Table 5, which required the analysis of the instantiated Petri net G^{tent1}.

In turn, the decision on `isOKToExecuteWithPriority1` is obtained from the evaluation of the condition in the second row in Table 5, which involved the analysis of the Petri net G^{tent2}. In this case, it is checked whether both the alarm process without parallelism and the vehicle-server transmission can finish within their deadlines.

Finally, the value of `isOKToExecuteWithPriority2` is obtained from the evaluation of the condition in the third row in Table 5, which requires the analysis of the Petri net G^{tent3}. Since this execution allows the alarm mode activities to finish on time with the required probability, it is only checked the probability of finishing the normal mode activities within D^t_{remain} time units.

After executing this decision logic, it is known the target configuration to which the system should adapt actuating on parameters `priority` and `paralelismLevel`. The *Mode change* activity of the adaptation process in Figure 2 sets up the parallelism level of the *recalculate vehicle path* activity and the priority of the alarm management behavior.

The *Emergency configuration* activity of the adaptation process in Figure 2, which executes when the decision on `isOKToExecuteWithPriority2` is 'no', sets up the parallelism level of the alarm to use both cores and its priority to the highest. In this case the alarm execution will satisfy its requirements although the risk of the normal operation of the system to fail is higher than allowed in this case.

6.2. Experimental results

In order to evaluate the feasibility and quality of the approach, we have experimented the case study in different scenarios. These scenarios cover the three possible system configurations and require the analysis of the three tentative models. Table 6 provides the values of model parameters that are independent of the particularities of each scenario.

Table 6: Values of scenario-independent parameters

Parameter	**Value**	**Parameter**	**Value**
mergeResults	100ms	*InformVehicle*	10ms
read	5ms	*upload*	100ms
transmit	30ms	*write*	10ms
Buffer size	2MB	*Block size*	50KB

From these data, we easily obtain the maximum buffer capacity of 2MB/50KB=40 slots. Moreover, message sizes range from 1MB to 6MB; the reachability radius from the server is 25m which makes a reachability range of 50m; the deadline for managing the alarm process is 5 seconds; and operation *recalculatePath* can take from 500ms –in case of minimum complexity from the read environmental conditions– to 3 seconds –in case of maximum complexity of the problem to compute–. We assume that, once the number of environmental conditions to use for a path recalculation is known, the time required to compute the *recalculateVehiclePath* can be known with considerable precision. We assume a relative standard deviation value between the expected and the actual computation time of $\sqrt{0.2}$, which allows us to model the computation time with the Gamma $\Gamma(5, \theta)$ or with Erlang-5 distributions.

Appendix A provides, for each of the scenarios considered, a detailed description of the status of the normal operation execution when an alarm event was received by the system, and provides the Petri net evaluation results and the outcome of the adaptation decision.

As performance indicator for the feasibility of the approach, the analysis of the Petri nets for the example scenarios took around 200 ms; this execution time indicates that the analysis can be made at run time. For the analysis we have used the command-line scripts provided by the *GreatSPN*[16] Petri net analysis tool. These scripts implement a randomization method to efficiently calculate the transient solution [28, 17, 3]. GreatSPN tool provides a syntax very similar to the used in Table 5. In the definition of the Petri net model, it is specified as `result: p{#Px=1};` to mark that GreatSPN analysis engine must calculate the probability of place `Px` having exactly one token. However, the time horizon of the transient analysis (i.e., the deadline value) is not represented in the definition of the Petri net but it is passed as argument in the transient analysis command. Finally, *GreatSPN* stores the results of the analysis in a text file (with extension *.sta*), which is read and its value compared with the reference values for adapting the system. The execution of the Petri nets analysis was carried out in a MacBook Pro 2011 with 8GB of RAM which executed VirtualBox and virtualized a Linux Debian with GreatSPN 2 installed.

Nevertheless, if the execution time of the Petri nets would have been larger (and, therefore, not affordable to be executed online), other alternatives that can speed up the output of results could be applied. For example, a pre-analyis of the Petri net can be done parametrically without actual values for transition rates, leaving a set of formulas that need to be evaluated with the actual values; this technique that was proposed in [10] for the reliability analysis using Markov chains. As we are evaluating performance, this technique cannot be directly applied, but it appears as a research direction in case that results need be obtained within shorter deadlines.

Enforcing the execution of specific activities on given cores requires to use both *core affinity* and *priority assignment* to activities. Establishing a core affinity is done per activity and it consists of instructing the kernel to run the activity on a specific core of the processor. Cores are numbered and receive unique identifiers. Each activity can set the processor bitmask reflecting its execution preference that determines which is the core (or cores) that it will run on. In case that several activities wish to run on the same core, the ties are solved by using their priority values.

6.3. Discussion

This section provides a summary discussion of the approach benefits, pointing also at the limitations and current challenges that leave the door open to future contributions. For discussing benefits and limitations, we consider the range of situations in which it can be applied.
Benefits of contribution claims. Contributions enumerated in section 1.2 – i.
e., *Timely adaptation, supporting scalable execution, usage of parametric models, and pragmatic view of the design*– have been put at work in the use case description and validation. Precisely, their benefits are:

- The *Pragmatic view of the design* as a graph of activities that represent the system in a particular configuration; system adaptations that are triggered by the satisfaction of a predefined set of adaptation rules that imply the evolution of the graph towards a different target configuration.

- The analysis of the system in different configurations has been performed through *parametric analyzable models*, precisely the use case utilized a set of three parameterized Petri nets. Each of these Petri nets models the system behavior in a different configuration. Parameters of the Petri net refer to the state of the system execution and its environment, and they are instantiated from information from the monitoring just before the model analysis is launched. Results of the analysis are used within the rules that define the adaptation conditions.

- The goal of the self-adaptive system is to execute *timely adaptations* to keep the system in configurations that ensure that deadline of all its tasks are met, both in the normal behavior and in the sporadic alarm process. The use case proved through examples that a self-adaptive system was necessary since it did not exist any static configuration of the system under study that could satisfy the deadlines in every possible situation. The proposed timely adaptation process considers a key aspect such as the *execution times* of activities and the specification of *reference values* for the adaptation times that are required by the system. We also measured the overhead of the adaptation process, and of its most time consuming activity of analyzing the Petri nets of the case study took around 200ms, which showed feasible to execute such analysis at runtime.

- The illustration of the application of the approach has considered a system whose allowed configurations exploited a *Scalable* execution; precisely in the case study, an activity could be split up and executed in parallel due to a hardware structure based on multiple processing cores. This scalability allows to improve the timeliness of the response of the parallelized activities.

The combination of these benefits allows to relax some assumptions that are usually needed for the operation of real-time and cyber-physical systems, such as the assumption of known execution time of tasks (or planning the scheduling for the worst-case execution time). Under this assumption, when an event arrives, it is immediately known the remaining computing demand of tasks. This work partially relaxes this assumption by computing their remaining demand from their actual status when events that may require adaptation arrive. This simplifies the management of tasks that show different computing demand and different deadline each time they execute (e.g., such as the transmission of files of different sizes at different vehicle speeds). This runtime evaluation of the performance properties of processes is well-known in the self-adaptive software field, however the amount of time that is required for computing such evaluation is not usually considered critical in this field. This work mixes these two concepts, evaluating at runtime the execution time while paying attention to the time required by such evaluation. This has been possible thanks to both a pre-generated parameterized model for each alternative in the graph of configurations that only needs to receive the actual values and limiting the possible configuration alternatives. It is worth noting that these benefits are in trade-off with the generality of the approach. Next paragraphs present its identified limitations.
Limitations of the approach. The next list of limitations considers the defined system behaviors, the definition of system configurations, and the

physical environment characteristics.

- *Known possible system configurations*: since each parametric model represents the system in a given configuration, the models of new possible configurations have to be defined by the engineer before running the system. This means that new system behaviors that emerge at runtime are not supported by the approach because, at present, the structure of the parametric model is not created automatically. In the presented case study, this situation would happen if, for instance, a third additional process beyond the normal behavior and alarm handling is installed in the server; the engineer should manually extend the parametric models for considering this new process.

- *Finite set of configurations*: since the adaptation decisions are defined as rules that contain condition and action, the adaptation action to take when a condition is satisfied must exist in a predefined finite set of configurations. In our case study, a configuration was defined by the parallelism level and the priority value that are natural numbers. However, the applicability of the proposed approach is restricted to autonomous decisions. Therefore, the approach is difficult to apply if the server could also decide about the speed of the vehicle (e.g., decreasing its speed if there is a risk of deadline miss) that would mean that it acts as coordinator of the behavior of other nodes. Also, the approach does not consider an infinite number of known configurations, e.g., speed ranges that are real number between an interval such as [5km/h ... 15km/h]. Although the illustrated limitation could be overcome by discretizing the range of possible speeds, the applicability of the approach would be more complex due to the number of possibilities that may arise from discretization.

- *Finite state space of the parametric analyzable model*: it should be possible to represent the system behavior in a configuration using a Petri net model with finite reachability graph. Due to the Petri net transient analysis technique used in this work, at present it is not possible to analyze unbounded Petri nets.

7. Conclusion

This paper has presented an approach to support on-line adaptation in cyber-physical systems that have to adhere to timing requirements. The approach considers the inherent immersion of cyber-physical systems in the environment that they monitor and actuate on. We describe the software structure of a cyber-physical system based on its constituent entities or *activities* that can be either parallelizable or sequential. We provide a pragmatic approach towards supporting adaptation while respecting the inherent timing constraints of cyber-physical systems. The system is designed as an activity graph with a number of possible different configurations, all of them are translated into activity graphs; and the adaptation of the system is the transition between two different configurations which are fired when the specified set of adaptation rules are satisfied. To ensure correctness, system configurations are analyzed through parametric models that we exemplify with parameterized Petri nets. We exemplify the need for adaptation with a use case based on a real laboratory prototype, since there is no static configuration in our specific case that can satisfy the operation deadlines in all situations. We have proposed a timely adaptation process based on two key aspects such as the *execution times* of activities and the specification of *reference values* for the adaptation times that are required by the system. The measurement of the overhead of the adaptation process yields 200ms, which proves that it can be run online.

Parametric analyzable models have been used to analyze the system under various configurations. Precisely, three parameterized Petri nets are elaborated for our use case. The parameters of the Petri nets model the execution of the system and environment conditions. The monitoring performed at runtime derives a set of values that are fed to these parameterized Petri nets when an adaptation need is detected, i.e., just before launching the model analysis. The results of the analysis are used to obtain the rules that define the adaptation conditions. We show that the system performs *timely adaptation* satisfying the deadline of all its activities in the different operation modes. To improve timeliness and obtain shorter response times, we have exploited *scalable execution* by splitting sequential activities into other parallel activities that can execute simultaneously in different processing cores.

Acknowledgement

This work has been primarily funded by the Salvador de Madariaga research grant by the Spanish Ministry of Education (PRX2012-00252) and partly funded by the project REM4VSS (TIN2011-28339) and M2C2 funded by the Spanish Ministry of Economy and Competitiveness.

[1] An architectural blueprint for autonomic computing. Technical report, IBM, June 2005.

[2] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* John Wiley Series in Parallel Computing - Chichester, 1995.

[3] G. Balbo and M. Silva, editors. *Performance Models for Discrete Event Systems with Synchronizations: Formalisms and Analysis Techniques.* Editorial KRONOS, Zaragoza, Spain, 1998.

[4] A. Benveniste, B. Caillaud, and R. Passerone. A generic model of contracts for embedded systems. *CoRR*, abs/0706.1456, 2007.

[5] M. M. Bersani and M. García-Valls. Online verification in cyber-physical systems: Practical bounds for meaningful temporal costs. *Journal of Software: Evolution and Process*, pages n/a–n/a, 2018.

[6] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. Deeco: An ensemble-based component system. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, pages 81–90, New York, NY, USA, 2013. ACM.

[7] I. Crnkovic, I. Malavolta, H. Muccini, and M. Sharaf. On the use of component-based principles and practices for architecting cyber-physical systems. In *2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*, pages 23–32, April 2016.

[8] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959.

[9] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Trans. on Auton. and Adap. Syst.*, 1(2):223–259, 2006.

[10] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 341–350, New York, NY, USA, 2011. ACM.

[11] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345–, June 1962.

[12] M. García-Valls and R. Baldoni. Adaptive middleware design for CPS: Considerations on the os, resource managers, and the network runtime. In *International Workshop on Adaptive and Reflective Middleware (ARM)*, 2015.

[13] M. García-Valls, I. R. Lopez, and L. Fernández-Villar. iLAND: An Enhanced Middleware for Real-Time Reconfiguration of Service Oriented Distributed Real-Time Systems. *IEEE Trans. Industrial Informatics*, 9(1):228–236, 2013.

[14] M. García-Valls, D. Perez-Palacin, and R. Mirandola. Time-sensitive adaptation in CPS through run-time configuration generation and verification. In *IEEE 38th Annual Computer Software and Applications Conference (COMPSAC), 2014*, pages 332–337, July 2014.

[15] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[16] GreatSPN. Dipartamento di informatica, Universita di Torino. GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets, Dec., 2015. URL: `www.di.unito.it/~greatspn/index.html`.

[17] D. Gross and D. R. Miller. The randomization technique as a modeling tool and solution procedure for transient markov processes. *Oper. Res.*, 32(2):343–361, Apr. 1984.

[18] A. Gupta, O. J. Pandey, M. Shukla, A. Dadhich, A. Ingle, and P. Gawande. Towards context-aware smart mechatronics networks: Integrating swarm intelligence and ambient intelligence. In *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, pages 64–69, Feb 2014.

[19] R. Hennicker and A. Klarl. *Foundations for Ensemble Modeling – The Helena Approach*, pages 359–381. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[20] C. Hofmeister and J. Purtilo. Dynamic reconfiguration in distributed systems: adapting software modules for replacement. In *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, pages 101–110, May 1993.

[21] M. Hölzl and T. Gabor. *Reasoning and Learning for Awareness and Adaptation*, pages 249–290. Springer International Publishing, 2015.

[22] S. Hougardy. The Floyd-Warshall algorithm on graphs with negative cycles. *Information Processing Letters*, 110(8):279 – 281, 2010.

[23] J. C. Jensen, D. H. Chang, and E. A. Lee. A model-based design methodology for cyber-physical systems. In *2011 7th International Wireless Communications and Mobile Computing Conference*, pages 1666–1671, July 2011.

[24] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[25] K. D. Kim and P. R. Kumar. Cyber physical systems: A perspective at the centennial. *Proceedings of the IEEE*, 100(Special Centennial Issue):1287–1308, May 2012.

[26] J. Kramer and J. Magee. Self-managed systems: An architectural challenge. In *2007 Future of Software Engineering*, FOSE '07, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[27] N. Kumar, M. Singh, S. Zeadally, J. J. P. C. Rodrigues, and S. Rho. Cloud-assisted context-aware vehicular cyber-physical system for phevs in smart grid. *IEEE Systems Journal*, 11(1):140–151, March 2017.

[28] C. Lindemann. On efficiently calculating transient solutions of generalized stochastic petri net models. In *[1991] Proceedings of the 34th Midwest Symposium on Circuits and Systems*, pages 1001–1004 vol.2, May 1991.

[29] M. Maggio, H. Hoffmann, A. V. Papadopoulos, J. Panerati, M. D. Santambrogio, A. Agarwal, and A. Leva. Comparison of decision-making strategies for self-optimization in autonomic computing systems. *TAAS*, 7(4):36:1–36:32, 2012.

[30] H. Muccini, M. Sharaf, and D. Weyns. Self-adaptation for cyber-physical systems: A systematic literature review. In *2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 75–81, May 2016.

[31] Object Management Group. A UML Profile for MARTE. `http://www.omg.org/omgmarte`, 2012.

[32] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[33] J. Panerati, F. Sironi, M. Carminati, M. Maggio, G. Beltrame, P. J. Gmytrasiewicz, D. Sciuto, and M. D. Santambrogio. On self-adaptive resource allocation through reinforcement learning. In *2013 NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2013, Torino, Italy, June 24-27, 2013*, pages 23–30, 2013.

[34] K. Tindell, A. Burns, and A. J. Wellings. Mode changes in priority preemptively scheduled systems. In *Proceedings of the Real-Time Systems Symposium - 1992, Phoenix, Arizona, USA, December 1992*, pages 100–109. IEEE Computer Society, 1992.

[35] C. Tsigkanos, T. Kehrer, C. Ghezzi, L. Pasquale, and B. Nuseibeh. Adding static and dynamic semantics to building information models. In *2016 IEEE/ACM 2nd International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*, pages 1–7, May 2016.

[36] B. Vogel-Heuser, C. Diedrich, D. Pantfrder, and P. Ghner. Coupling heterogeneous production systems by a multi-agent based cyber-physical production system. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 713–719, July 2014.

[37] J. Wan, D. Zhang, S. Zhao, L. T. Yang, and J. Lloret. Context-aware vehicular cyber-physical systems with cloud support: architecture, challenges, and solutions. *IEEE Communications Magazine*, 52(8):106–113, Aug 2014.

[38] D. Weyns and et al. On patterns for decentralized control in self-adaptive systems. In R. De Lemos, H. Giese, H. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *LNCS*, pages 76–107. 2013.

[39] M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 2(10), 1995.

[40] Y. Zhao, S. Li, S. Hu, L. Su, S. Yao, H. Shao, H. Wang, and T. Abdelzaher. Greendrive: A smartphone-based intelligent speed adaptation system with real-time traffic signal prediction. In *2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems (ICCPS)*, pages 229–238, April 2017.

Appendix A. Validated scenarios

We have experimented the approach using the case study described in Section 5 in multiple situations, varying the available remaining time for completing the transmission, the remaining data in the vehicle to transmit to the server, and the percentage of utilization of the buffer when the alarm event occurs. Among the obtained results, we chose three different representative situations (i.e., a combination of parameters) to report. These three situations are described in the following three execution scenarios

Scenario 1. In this scenario, an alarm arrives when the normal mode activities are running and in the following situation:

- having still 1MB of data to transmit from the vehicle to the node,
- having already finished its first activity *InformClient*, with a half empty buffer, and
- being the vehicle speed 12km/h, which makes the deadline to finishing the normal mode activities of the node be $\frac{50m}{12km/h} = 15s$, and

- having elapsed already 13 seconds of transmission.

In this situation, the *readEnvironmentalConditions* operation of the alarm process is immediately executed because it has the highest priority (as shown in Table 3). According to the number of current environment conditions, the *recalculateVehiclePath* is expected to have an execution time of 1 second that will yield to selecting a new path.

In this case, it is safe to execute with priority 0 because the Petri net analysis of the model in Figure 5(a) returns that the probability of a token being at place *PfinishAlarm* in less than 5 seconds is 0.99999.

Furthermore, we have executed a further *what-if* analysis for approach validation only (i.e., it would not be executed in the real case) of this scenario. We have analyzed the expected results if the alarm process had received maximum priority, i.e., we have instantiated a Petri net model as in Figure 5(c). In this analysis that, we have obtained that if the alarm process had received maximum priority and parallelism, the normal behavior would have finished within its deadline with a probability of 0.933, then failing to satisfy its deadline requirement. This analysis demonstrates that the straightforward solution of giving maximum priority and resources to the alarm process is not appropriate in our case study, and hence it is necessary that the system performs the analysis of its situation in each alarm event and self-adapts to its best configuration.

Scenario 2. In this scenario, an alarm arrives when the node is running the normal mode activities and with the following situation:

- still has 2MB of data to transmit,
- the vehicle speed is 12km/h, thus giving again a deadline of 15 seconds, and
- the normal mode operations have already consumed 10.5 seconds of transmission, thus remaining 4.5 seconds until its deadline,and
- being the buffer again half full.

In this situation,the *readEnvironmentalConditions* operation of the alarm process executes and, according to the current environment conditions, it is estimated that the *recalculateVehiclePath* will its required execution time will be 1.2 seconds to select a new path.

In this case, the `isSafeToExecuteWithPriority0` returns false since the analysis of the Petri net in Figure 5(a), G^{tent1}, returns that the probability of a token being at place *PfinishAlarm* after 5 seconds is 0.97923. In turn, the analysis of the Petri net representing G^{tent2} yields that it is appropriate to execute the alarm process with priority 1. Its analysis shows that the probability of a token being at place *PfinishAlarm* after 5 seconds is 0.99999, whereas the probability of a token being at place *PfinishTransmit* after 4.5 seconds is 0.99660 –which satisfies the requirement of 99.5%. Therefore, `isOKToExecuteWithPriority1` returns true and the system executes with the configuration of `parallelismLevel` of the alarm process is `1Core` and the priority of the alarm process is `prio=1`.

Scenario 3. In this scenario, an alarm arrives when the node has just initiated the transmission of 6MB of data, with the following situation:

- the buffer is still empty, and
- the vehicle moves at the same 12km/h as in previous scenarios, and
- the process has used 0.5 seconds, then remaining $D^t_{remain} = 14.5$ seconds until the deadline.

In this situation, the *readEnvironmentalConditions* operation of the alarm process executes and, according to the current environment conditions, it is estimated that the *recalculateVehiclePath* activity will require 2.2 seconds to execute and provide a new path.

In this case, the analysis of the Petri net representing G^{tent1} returns that the probability of a token being at place *PfinishTransmit* after 14.5 seconds is ≈ 1; but also it yields that the probability of a token being at place *PfinishAlarm* after 5 seconds is ≈ 0. Therefore, `isSafeToExecuteWithPriority0` returns false. The analysis of the Petri net representing G^{tent2} are used to to assess whether `isOKToExecuteWithPriority1`. Its analysis yields that the probability of a token being at place *PfinishTransmit* after 14.5 seconds is 0.999977 (that satisfies the deadline requirement of the normal mode activities); and the probability of a token being at place *PfinishAlarm* after 5 seconds is 0.98756. Since $0.98756 < 0.99999$, it is not appropriate to execute with priority 1. Finally, the analysis of the Petri net G^{tent3} returns that the probability of a token being at place *PfinishTransmit* after 14.5 seconds is 0.99993, and the probability of a token being at place *PfinishAlarm* after 5 seconds is 0.999993. Thus, `isOKToExecuteWithPriority2` is true and hence the alarm process executes with `parallelismLevel=2core` and maximum priority. After the alarm process has finished, the normal mode continues and it is expected to satisfy its deadline with a probability of $0.99993 > 0.995$.