

A methodology for mapping functional blocks into earliest deadline scheduled threads

Cesare Bartolini

I. INTRODUCTION

A. Executive summary

Increasing complexity and short time to market demand for higher productivity and reduced error rates in the production of reactive embedded software. Formal techniques for verification of functional and non-functional properties and a number of design and modeling methodologies as well as formal languages have been proposed in the last decades in order to provide support for formal or simulation-based verification of a least a subset of the requirements. Automatic generation of the code implementation of the model is also advocated as necessary for reducing the probability of fault injection in the manual coding phase.

Collectively, these practices go under the name of model-based design and development. Unfortunately, an encompassing methodology that can support all phases of the design stage and provide formal verification of all system properties is still out of reach for most systems of practical interest and most designers must necessarily settle for lower expectations.

The most common assumption is the separation of the two main concerns of functional and architectural specification. Layering of functional and architectural-level design is among the founding principles that inspired the Ptolemy framework [14], the Metropolis methodology and tool set [3], emerging standards and recommendations, such as the UML Profile for Schedulability, Performance, and Time from the Object Management Group, [1] and industry best practices, such as the V-cycle of software development common in the automotive industry [5] (Figure 1).

Functional design is concerned with the development of logically correct software. Models and languages apt at representing the abstractions used by application domain experts are often used and formal verification or simulation of system proper-

ties is performed. In order to reduce programming errors, tools for automatic generation of functional code can also be used at this stage. For example, in the automotive domain the Simulink toolkit (based on a synchronous reactive semantics) supports the design and simulation of control algorithms and the Stateflow plug-in (and the associated statecharts-like semantics) captures the evolution of the systems state with the arrival of external events and the possibly subsequent mode changes (Control algorithms design and sign-off stages of Figure 1). UML and the object-oriented paradigm are other very popular choices (albeit often in different application domains) and, more recently, the real-time UML profile (implemented by commercial products such as Rational Rose Real-Time) and a subset of the 2.0 revision of the language assume a (transitional) model where active and passive components cooperate in the implementation of the system behavior and each call action upon a component's method results in the transition of the statechart automata describing the object behavior.

Requirements for logical-level resources, timing constraints and timing assumptions may be formally defined at this level, but timing analysis, when and if performed at this stage, only deals with abstract specification entities, typically assuming infinite availability of physical resources (such as memory or CPU speed).

At the *Architecture* level, the designer defines the concrete model on which the functional abstractions must be mapped and constrains the generic specification by defining an implementation on an architectural platform. At this level, design choices define (among others) the levels of concurrency and the resource management policies. The definition of the architecture of the software threads and the selection of the resource management policies is a non-trivial task, with possible implementations ranging from single thread implementations to networks of concurrent processes (Software design stage in the

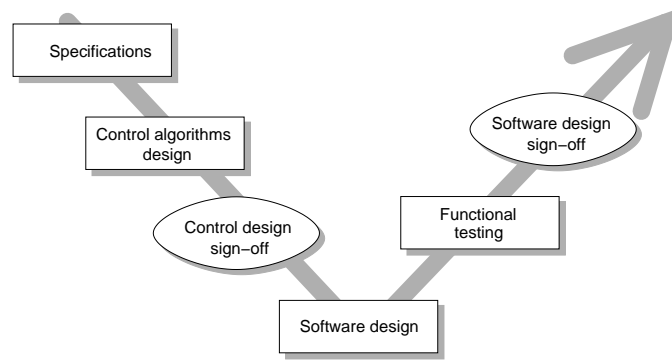


Fig. 1. Software development in the V-cycle methodology.

v-cycle of Figure 1).

Single thread implementations are quite common (see the related work section) and simple: the entire functional specification is executed in the context of a single task, which performs a never ending cycle where it serves events in a non-interruptable fashion according to the run-to-completion paradigm. The thread waits for an event (either external, like an interrupt from an I/O interface, or internal, like a call from one object to another); fetches the event and the associated parameters and, finally, it executes the corresponding code. On the other extreme, we could define one software task for every functional block. Each task can be assigned its own priority, depending on the criticality and on the deadline of the corresponding activity. At run time, the operating system scheduler properly synchronizes and sequentializes the tasks so that the order of execution respects the functional specification.

Both approaches may easily prove inefficient. The single thread implementation suffers from large priority inversion due to the need of completing the processing of each event before fetching the next event in the global queue. The one-to-one mapping of functions or actions to threads suffers from excessive scheduler overhead caused by the need for a context switch at each reaction. Considering that the action specified in a functional block can be very short and that the number of functional blocks is usually quite high (in most application it is in the order of hundreds), the overhead of the operating system could easily prove unbearable. Furthermore, standard Rate Monotonic analysis of this model is only possible when cooperation among active objects is restricted to purely asynchronous communication or interaction through semaphore-protected mailboxes [9].

The designer essentially tries to achieve a compromise between these two extreme, balancing responsiveness with schedulability, flexibility of the implementation and performance overhead.

The lack of a tool to support this development process is also an issue we try to address. At the current state of the art, no commercial tool is able to support a developer through all stages of system design. While there is a lot of available tools to describe a system, the mapping process currently lacks support.

To solve this type of problems, from many sides, several attempts are being made to incorporate already at the modeling stage appropriate notations for expressing the desired non-functional properties, in a way that can be subsequently analysed and transformed, via successive refinements, into a conforming implementation. The idea of using and annotating models as a first class element in software development is the philosophy behind the emerging Model Driven Engineering (MDE) approach [7], [8], [23].

MDE is a breach in software engineering concerning application modeling and implementation. In this paradigm, models provide abstractions of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on the relevant ones [8]. This is not actually different from traditionally recommended practice in any software development process since the earliest days of programming. The main difference between usual practice and the MDE stays in the fact that in the new vision models are not only used for documentation purposes, but they become well defined input/output elements for computer-based tools that implement precise operations [7].

Hence, models used for reasoning and analysis

are then directly transformed into the desired implementation. This usage of models as a baseline for implementation permits to rise the abstraction level at which solutions are described, and opens a realm of novel possibilities for creating, analyzing, and manipulating systems through various types of tools [23]. However, not to deceive these expectations, the customary attitude to system development based on the usage of individual tools that tend to only operate on specific models conforming to their own internal format has to change in favour of standard techniques and languages [23].

Currently, the standardization process of the MDE is strongly oriented towards the Unified Modeling Language (UML) as the primary modeling notation [8]. UML provides a variety of instruments to describe the characteristics of a generic system in corresponding models. However, it is not complete, in the sense that the basic elements of the language cannot cover all potential needs for describing specific systems from any domain. Hence in some cases the definition of domain-specific variants of the UML may be required. The UML however has already been conceived for extensibility, for which purpose it provides a built-in *profiling* mechanism to extend the language with elements and constructs apt to describe specialized features, though remaining compliant with its standard definition.

B. Scope and context

The goals of this research are centered around the design of real-time systems, particularly focusing on the mapping phase. The main goal is to provide a complete methodology for generating an optimal or sub-optimal task set ready for execution on a real-time scheduler. To achieve this goal, we have provided two algorithms, called LA and JLA respectively, which analyze the functional design and generate the task set. We have proven that the task set thus generated is optimal from a feasibility point of view, thanks to the use of a dynamic priority scheduling algorithm.

This methodology, though efficient and easily flexible and adaptable to specific requirements, needed to be supported by a universal language instead of being based on an ad-hoc model. For this reason, the work has been extended to be UML-compliant. In particular, an extension of the SPT-profile has been developed to include the concepts

needed to apply the proposed methodology. With this extension, the task generation algorithms could be easily applied to a UML design. While it would be interesting to see an actual implementation of this methodology in a UML design support tool, the lack of such a tool featuring all the novelties introduced in the recent UML 2.0 specification currently prohibits a similar development. However, the methodology can nonetheless be used by first describing the system according to the notation and structure introduced later.

C. Related work

The number of works devoted to the specification of real-time systems at the logical level is simply too high to try even a short survey of the existing methodologies and languages. We will divide them into the two broad categories of (strictly) synchronous models and languages and asynchronous (or general) models.

Synchronous reactive languages, such as Esterel, Lustre or Signal [6] define the system as a netlist (graph) of processing blocks and assume the computation of the response or reaction to events by each block takes zero time. External events are sampled and computation proceeds according to the discrete time model. A formal semantics and mathematical results allow verification of logical properties such as liveness or correctness with respect to constraints or invariants. In systems developed according to this model, the scheduling problem is solved statically at compile time (equivalently, the operating system level components are automatically synthesized) and the aim of the timing analysis is making sure that the longest chain of reactions or computations initiated by any event fits in the time interval between any two ticks. Unfortunately, these systems suffer from multiple inefficiencies, arising from two main sources: the quality of the code generated by the compiler and the priority inversions and limited flexibility of the scheduling strategy. However, in those cases where a discrete time model is applicable and concerns for safety overcome the additional costs caused by compiler and scheduling overheads, these models provide a satisfactory solution encompassing both stages of the development. A synchronous reactive model (although based on a looser, semi-formal semantics) is also at the base of the widely used commercial tool Simulink from Mathworks.

In this case, the accompanying code generation tools (such as Embedded coder) generally offer the designer three options for generating the thread architecture. The first option simply consists in a single thread implementation, with the associated drawbacks. In the multi-thread option (available only for discrete time models where the rates of external and internal events are strictly harmonic) all the functions activated with the same rate are grouped in one single task and executed in lexicographic order. Threads are then simply scheduled according to the Rate Monotonic rule. Possible race conditions when accessing variables shared among blocks with different rates can be solved according to a deterministic (fixed delay) or nondeterministic (semaphore-protected) synchronization model. Unfortunately, in both cases, the semantics of the implementation differs from the semantics of the simulated model [15].

In the case of asynchronous models and generic languages, such as UML (similar reasoning applies to SDL), a mapping or synthesis procedure including formal verification of logical and timing properties is currently not available and the logical and architecture models are developed in stages. In UML, the logical and architecture designs are both drawn using objects (at different levels of granularity).

The recent development of the OMG UML profile for schedulability, performance and time provided modeling constructs for specifying logical and physical resources, concurrent active objects (threads), binding of logical objects into architecture objects and representation of timing attributes and constraints, thus paving the way for schedulability analysis of UML architecture models.

Application of schedulability analysis techniques to the design problem can provably reduce the total composition time and the round implementation time when tuning the design of the architecture level by as much as 50% [22].

In the context of this work, a survey of the existing scheduling methodologies is clearly impossible for space constraints, however, an exception is necessary for previous work done in the context of architectural model synthesis, which bears a strong connection to our research.

An early attempt at task synthesis, mainly aimed at solving the problem of synthesizing a graph of cooperating tasks where architecture-level proper-

ties are expressed in a fundamental way, such as rates of execution and relative deadlines has been presented in [13]. As a further example, in [16] the authors proposed the synthesis of a cyclic executive scheduling starting from a Modechart specification of the system.

Finally, the problem of matching the logical level to the software architecture level and the corresponding synthesis of the task model has been discussed in length in the UML case by Saksena and others [20]. In their work, the authors assume a more general model of active objects freely interacting by exchanging events resulting in (asynchronous) signals or (synchronous) method calls. Each active object executes according to the base server model: awaits messages in the input queue and processes them. Incoming messages to each object are queued. The mapping and scheduling problem needs to account for multiple constraints and design options. First, object methods need to be executed according to a run-to-completion paradigm in order not to compromise the consistency of the object's state. Second, the scheduling problem becomes a dual problem: scheduling the messages in the input queue of each object and scheduling the threads on the processor. Furthermore, a method for mapping the execution of active object methods (actions) into threads is needed.

Based on this assumption, the authors discuss a single threaded solution, where all actions are implemented by a single thread and events are queued by priority (only the message scheduling problem needs to be considered) and a multi-threaded solution where threads inherit their dynamic priorities from the priorities of the messages that activate them. A mechanism based on priority ceilings protects against multiple priority inversions when mutual exclusion on objects needs to be guaranteed.

Unfortunately, while the single threaded implementation is analyzable and practically applicable (although with some restrictions and overheads), the multi-threaded implementations are either almost impossible to analyze for the worst case timing scenario (when thread priorities are static, as implemented by most commercial tools) or analyzable for worst case behavior but inapplicable to the code generated by existing tools.

II. FUNCTIONAL MODEL

Most real-time embedded systems have a simple structure: some event (generated for example by a sensor) or user input triggers the system's execution, the system does some operations, it processes the input signal, updates the internal state and then generates the outputs.

One natural model for this kind of structures is the *dataflow model*. Of course, since this model identifies a particular structure, not all systems (especially not general-purpose systems) can be adequately represented using dataflow.

A dataflow model can be represented with a directed graph. Such a graph will have inputs from the environment, which represent the external events that can trigger the execution. At the other end of the graph the outputs mark the end of the execution of the dataflow. In the middle, the collection of edges and vertexes represents the system's functionality.

In this research, we use a restriction: no cycles must exist inside the structure of the graph. This is a common restriction in the dataflow model, since it largely reduces the complexity of the analysis while still leaving enough expressive power to define the behavior of most embedded systems [6].

Our model of execution can be represented through a *Directed Acyclic Graph* (DAG). Formally speaking, the *functional model* used in this research is a t-uple $\{\mathcal{V}, \mathcal{E}, \mathcal{R}\}$, where \mathcal{V} is the set of vertices, \mathcal{E} the set of edges, and \mathcal{R} is a set of mutually exclusive (logical) resources, representing shared data.

- $\mathcal{V} = \{F_1, \dots, F_n\}$ is the set of *functional blocks*. They represent the basic operations of the system. A functional block F_i is characterized by a maximum computation time γ_i and a set of used resources $R_{i,1}, \dots, R_{i,n(i)}$. A block F_i has one *input port* and one *output port*. An input port is a *buffer of infinite size* that may contain *activation tokens*. When at least one *activation token* is present on the input port, the functional block is active. A functional block can execute only if it is active and all resources in the resource set are available. Each functional block is executed once for each activation token; therefore, if multiple tokens are input to a block, the activation semantics is of OR type. AND type activation semantics, as allowed by other modeling languages, is not

considered at the current stage of this research and will be the subject of future extensions.

If the block is scheduled to execute, it first acquires all the needed resources; then, it executes consuming exactly one token. The duration of the execution is a random variable comprised between 0 and γ_i . At the end of the execution, it first releases all resources, and then fires its output port. If no other token is present on the input port, the block becomes *inactive*. Otherwise it remains active.

- $\mathcal{E} = \{l_1, \dots, l_m\}$ is the set of *functional links*. A functional link $l_i = (F_h, F_k)$ connects the output port of functional block F_h (the source block) to the input port of block F_k (the sink). One functional block can be the source or sink of many links. When a functional block completes execution, it fires its output port: this means that tokens are sent on all the links starting from that output port. When fired, a token is instantaneously received on the input port of the sink.

In the following, the source and the sink of link l_i will also be denoted by $src(l_i)$ and $snk(l_i)$, respectively.

In our model, we focus on the flow of control among blocks. Although our tokens are pure activation signals (i.e. they do not carry data), an extension to typed signals is possible.

- $\mathcal{R} = \{R_1, \dots, R_z\}$ is the set of logical resources that can be used by the functional blocks to carry out their computations. They are used to model shared data structures that can contain state information or shared data and that need to be executed in mutual exclusion.

An *external event* results from the execution of a special functional block e_i with no input links, representing one entity in the external environment. Since external events are produced by the environment, we assume a minimum interarrival time between successive activations, denoted by T_i . Therefore, an external event can be *periodic* (i.e. with a constant interval of time between events) or *sporadic*.

An *output* o_j is a special functional block with no output link. It represents a consumption by the environment of the data produced by the system and sent to some actuator. For our purposes, an output is merely a stub where execution ends.

It is important to make some considerations on the execution time of a block. In practice, execution times strongly depend on the hardware platform on which the application is implemented. Therefore, we assume that the hardware architecture is known and that computation times can be estimated before actually deploying the system. In reality, the development process will follow a spiral model in which, at each incremental step, the execution time of each functional block can be more accurately estimated.

A *functional chain* from F_i to F_j , or $P(i, j)$, is an ordered sequence $P = [l_1, \dots, l_n]$ of links that, starting from $F_i = \text{src}(l_1)$, reach $F_j = \text{snk}(l_n)$ crossing $n+1$ functional blocks such that $\text{snk}(l_k) = \text{src}(l_{k+1})$. F_i will be the chain's source and F_j its sink. Clearly, a sequence of functional links is in strict relation with a unique sequence of functional blocks. Thus a functional chain also (and most importantly) denotes a sequence of blocks.

There can be more than one functional chain between two blocks i and j . The k -th functional chain between F_i and F_j is denoted with $P_k(i, j)$. If a block $F_s \in P_k(i, j)$ is activated before $F_p \in P(i, j)$, the former is a *predecessor* in respect to the latter, which is *successor*; the notation used is $F_p \prec F_s$. To simplify the notation, without loss of generality, a chain from F_i to F_j will be simply denoted by $P(i, j)$.

A *path* $P_k(e_i, o_j)$ is a chain with e_i as source and o_j as sink. Substantially, a path is a chain from one end of the DAG to the other. A path represents one end-to-end execution of the system, from the triggering of the external event to the generation of the output. More than one path can be originated by one external event.

Being the application described with a DAG, no chain must exist which has the same block as both source and sink; i.e., no cycles must exist in the graph. Clearly, this limitation does not exclude programming constructs like conditional loops from our modeling; simply, any loop must be entirely contained inside a functional block.

The *path deadline* for P_i , denoted by Δ_i , is the end-to-end constraint for the computation performed in the path. The implementation and the run-time mechanisms must ensure that all the functional blocks in the path must be completed at most Δ_i time units since the arrival of the event.

$e_{i,j}$ is the j -th *activation instance* of event e_i , and $P_{i,j}$ is the j -th activation of path P_i , since every

TABLE I
PARAMETERS OF THE EXAMPLE.

e_1	$T_1 = 200$	F_1	$\gamma_1 = 10$
e_2	$T_2 = 300$	F_2	$\gamma_2 = 20$
e_3	$T_3 = 700$	F_3	$\gamma_3 = 15$
e_4	$T_4 = 1500$	F_4	$\gamma_4 = 25$
$P(e_1, F_6)$	$\Delta = 125$	F_5	$\gamma_5 = 25$
$P(e_2, F_8)$	$\Delta = 200$	F_6	$\gamma_6 = 5$
$P(e_3, F_9)$	$\Delta = 700$	F_7	$\gamma_7 = 90$
$P(e_3, F_{10})$	$\Delta = 700$	F_8	$\gamma_8 = 25$
$P(e_4, F_{12})$	$\Delta = 1500$	F_9	$\gamma_9 = 90$
		F_{10}	$\gamma_{10} = 25$
		F_{11}	$\gamma_{11} = 25$
		F_{12}	$\gamma_{12} = 50$

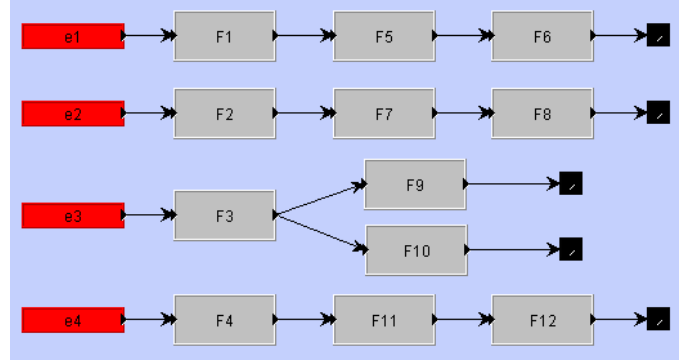


Fig. 2. Graphical representation of the example of Section II-A.

instance of a given event triggers a new instance of the path. The activation time for event $e_{i,j}$ is denoted with $A_{i,j}$.

While the path deadline is a relative time constraint, independent of the actual path instance, each path instance will have its absolute deadline: $\delta_{i,j} = A_{i,j} + \Delta_i$.

A. Example of a functional model

The expressiveness of the proposed functional model is demonstrated through a simple example equivalent to the sample (UML) object model in [20]. The DAG is shown in Figure 2. The example consists of 4 external events that activate 5 different paths. A total of 12 functional blocks are defined. The periods of the events, the computation times of the functional blocks and the deadlines of the paths are summarized in Table I.

One main difference between our model and the model in [20] is the possibility to set deadlines for paths. In the model presented in [20], deadlines are assigned to the external events. This means that all paths starting from the event must be completed before the same deadline.

However, in many cases, the functional blocks activated by an external event can have very different criticalities. As an example, consider the external event e_3 that activates two paths, one consisting of blocks F_3 and F_9 , the other one consisting of blocks F_3 and F_{10} . Suppose that block F_9 is critical and must be completed before the next instance of e_3 is triggered. Therefore, the deadline for path $P(e_3, F_9)$ is 700. Now, suppose that block F_{10} is used to log some data on a secondary storage. This activity is not critical, so we could set a long relative deadline for path $P(e_3, F_{10})$. Expressing these constraints with the model proposed in [20], is not easy (if possible), while we can naturally do it in our model.

Another difference is that in our model we do not need to associate a *priority* to each event. In [20], a customized scheduler, based on fixed priority, is assumed. Therefore, it is necessary to specify the priority of every event, introducing one more (unnecessary) design parameter. Instead, our scheduling model is based on EDF, as will be described in Section IV-A. Therefore, we do not need to specify any additional priority.

III. EXTENDED UML PROFILE

The functional model described in II is very practical and simple, suiting perfectly the methodology for generating task sets developed in our research. However, since it is a custom model, it has several defects. Such a model cannot be designed using a widely known formalism such as UML, and the description of a system would need a custom tool. For this reason, an effort has been made in this research to extend the UML Profile for SPT; this extension would allow a developer to design a system using a UML notation while still being able to create the task set using the proposed methodology.

The usual structure in the UML community followed for presenting a profile is to introduce first the basic abstractions we need and their semantics (which is called the “Domain Viewpoint”) and then to define how these abstractions can be modeled in UML (which is called the “UML Viewpoint”).

The Domain Viewpoint has been broadly discussed in Section II. The main abstractions we need to model focus on the notions of *resources*, *functional blocks*, *external events* and *paths*. Even though something quite similar to the concepts of resources, functional blocks and external events are

present and expressed in a very detailed way in the UML extensions proposed in [18], there are no references to the notion of a path as intended here.

The rest of this section focuses on the UML viewpoint, i.e., we explain how to model the above notions into a UML-compliant notation. In particular, as explained below, we have identified a specific type of diagram, the Activity Diagram, as the central element of our profile. Activity Diagrams [12] were introduced in UML to model the procedural flow of actions that are part of a larger activity. Since the concepts and the scheduling algorithm we refer in this research are based on a *dataflow model* (in particular on a *Direct Acyclic Graph* representation), we note a natural correspondence with these kind of diagrams.

Resource. To model the resources, in principle we would inherit the whole set of characteristics proposed in the SPT-Profile within the «SAResource» stereotype definition. In particular, this stereotype identifies a kind of resource that can be contended by multiple concurrent actions and whose access is protected by some mechanism. Several tagged values are associated with the definition of resource, but these are too detailed for our purposes, so they will be not considered at this stage. Unfortunately, such stereotype cannot be applied directly to the ObjectNodes of an Activity Diagrams [17], since this concept does not specialize any element of the stereotype base class list [18]. In order to exploit the definition of resource as we utilize it in our profile, without infringing the semantics of SPT-Profile elements, we introduce a new stereotype («Resource»), which is defined in identical way as «SAResource», but can be applied also to ObjectNodes.

Functional block. A functional block is basically a unit of work with a defined execution pattern, which contends the use of some resources with other functional blocks. Such definition corresponds to the specification of «SAAction» used for the scheduling domain description in the SPT [18]. The SPT Profile again introduces a lot of characteristics in order to clarify the role of an action, but in our case only the SAworstCase tagged value will be considered at the present stage. Such value refers to the maximum computation time γ of the functional block as defined in Section II. This stereotype can be directly applied to an Action element of an Activity Diagram, since it specializes the concept of an

Stereotype	Base Class	Parent	Used Tags
«Resource»	ObjectNodes	«SResource»	
«SAction»	Action		SAworstCase
«ExtEvent»	InitialNode	«SAttrigger»	SAoccurrence
«Path»	ActivityPartition		Deadline

TABLE II
STEREOTYPES DEFINITION

Action from BasicActions [17], which are allowed as base classes for the stereotype «SAction» [18].

The execution model in Section II allows a functional block to have a set of input or output ports. Such ports can manage tokens to transfer control, step by step, between functional blocks through the defined dataflow. In the UML Activity Diagram, the actions already provide this concept by means of the notion of *pins*. Since we have represented a functional block by means of an extension of the action of an Activity Diagram, hence the ports can be rightly represented using the pins.

External event. As usual, in the case of embedded systems there appears quite often the notion of some event, generated for example by a sensor, which triggers system execution. Activity Diagrams can model the starting point for executing an activity by means of their InitialNodes [17], but in its usual definition no real-time annotations are possible. The SPT-Profile, on the other side, provides the instrument to model events of this kind by means of the «SAttrigger» stereotype. However, this stereotype cannot be applied to an InitialNode, since neither the latter, nor any of its generalization, appears in the base class list of «SAttrigger» [18]. Therefore for this case we introduce a new stereotype («ExtEvent») defined as «SAttrigger», which can be applied to the InitialNode modeling concept. At the moment, the only tagged value that we associate to this new stereotype is the SAoccurrence one, which defines the pattern of inter-arrival times between consecutive occurrences of the trigger, as specified in [18]. Referring to the notation adopted in Section II, this tagged value corresponds to the characteristic of an external event denoted as T .

Path. In their classical formulation, Activity Diagrams allow to group flows of actions by means of *activity partitions* (these are also sometime referred to as *swimlanes*). The grouping is made with respect to the role of some entity or process who is responsible for carrying on those actions. In our

case we want to force somehow this concept and model the notion of paths as introduced in Section II. A path is an ordered sequence of steps that the system has to execute following up an input by an external event: we can then see an activity partition as a container of a set of related actions triggered by an external event and whose execution has to be performed in accord to some timing constraints. Figure 3 depicts the notion of path into the SPT-Profile domain viewpoint. The stereotype «Path», applicable to the ActivityPartition UML modeling element, defines its end-to-end constraint for the computation by means of a Deadline tagged value. In the execution model of Section II the latter represents the maximum reaction time expected for a computation denoted as Δ .

In some cases an action could belong to more paths; in these cases the usual graphical and textual notations of Activity Diagrams can be adopted. In particular, the classical UML Activity Diagram textual notation entails to annotate each node with the activity partition or partitions to which it belongs, enclosed within round brackets.

A resume of the definitions and uses of the previously described stereotypes is given in Table II. Each row specifies: the name of the stereotype; the classes of the UML metamodel to which it can be applied (**Base Class**); the name, if it exists, of the extended SPT-Profile stereotype (**Parent**), and the tagged values defined or required (**Used Tags**).

IV. SOFTWARE ARCHITECTURE

According to our methodology, a functional model must be *mapped* on a set of real-time tasks.

A task is denoted with τ_i and its j -th instance is $\tau_{i,j}$, activated at a time $a_{i,j}$. Relative task deadlines are denoted by $D_{i,j}$, while absolute ones are $d_{i,j}$ (referring to the j -th instance of task τ_i).

Task τ_i starts its execution for the j -th instance at time $s_{i,j}$, and the execution is terminated at time $f_{i,j}$. The deadline is respected if $f_{i,j} \leq d_{i,j}$.

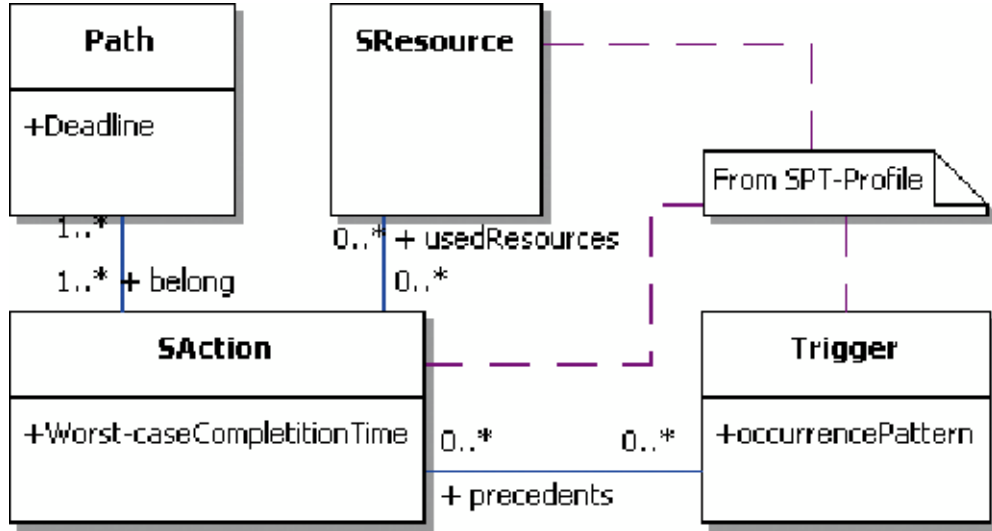


Fig. 3. The extension on SPT-Profile with the Path

The *response time* for the j -th instance of task τ_i is the time difference between its finishing time and its activation: $r_{i,j} = f_{i,j} - a_{i,j}$.

The *computation time* $c_{i,j}$ for the j -th instance of task τ_i is the time it would take executing if there were no other concurrent tasks ($c_{i,j} \leq r_{i,j}$). The worst case execution time of a task τ_i , denoted as C_i , is given by the sum of the WCETs γ_k of all functional blocks contained in the task. Since a task is strictly sequential, the only approximation introduced is neglecting the overhead resulting from signal latency on the bus, scheduling overheads, and so on.

The *base deadline* of a task instance $\tau_{k,l}$ is the time difference between its absolute deadline and the arrival of the external event which caused such instance to execute: $D_{k,l}^b = d_{k,l} - A_{i,j}$, where $e_{i,j}$ is responsible for activating $\tau_{k,l}$.

A task can use resources in mutual exclusion. The use of such resources derive from the functional blocks that the task executes. Therefore, if a functional block F_j is implemented by task τ_i and the functional block uses resource R_r , we say that task τ_i uses R_r with a critical section of duration equal to the WCET of F_j . We denote by $\xi_{i,r}$ the longest critical section among all critical sections of task τ_i on resource R_r .

We assume that the underlying scheduling policy implements a synchronization protocol like Priority Ceiling [21] or Stack Resource Policy [2]. Therefore, we define a preemption level π_i for task τ_i

TABLE III
SUMMARY OF NOTATIONS AND DEFINITIONS.

F	Functional block
l	Functional link
e	External event
o	Output
P	Functional chain or path
τ	Task
γ	Functional block WCET
C	Task WCET
c	Task computation time (actual)
A	Event activation time
a	Task activation time
s	Task start time
f	Task finishing time
$r = f - a$	Response time
$w = s - a$	Waiting time
Δ	Path relative deadline
δ	Path absolute deadline
D	Task relative deadline
d	Task absolute deadline
D^b	Base deadline
ξ	Duration of critical section
π	Preemption level
$ceil(R)$	Ceiling of resource R

inversely proportional to its base deadline:

$$\pi_i = \frac{1}{D_i^b}.$$

For each resource R , we define a *ceiling*:

$$ceil(R) = \max_i \{\pi_i \mid \tau_i \text{ uses } R\}.$$

Table III summarizes the symbols defined in our mathematical model.

A. Execution platform

We assume that the application is implemented on a single processor architecture, with a real-time operating system. We assume earliest deadline first (EDF) as the scheduling algorithm, but the task model is not the usual periodic/sporadic model and tasks are not assigned periods or minimum interarrival times. While a task activated by an event is considered to be periodic or sporadic (having the same period or minimum interarrival time as the event), a task that is activated by another task cannot be assigned a period (analysis based on the minimum interarrival time would be too pessimistic). A task is assigned an absolute deadline d upon activation.

V. GENERATION OF THE TASK SET

In this section we present the methodology for mapping functional blocks to real-time tasks, and to generate the task real-time scheduling parameters. The general problem is quite difficult because there is a high number of possible mappings. Rather than exhaustively searching among all possibilities, we propose two different algorithms that derive the task set with complexity linear in the number of functional blocks, according to their topology. We will prove in Section VIII that such algorithms are optimal for the single processor case and assuming EDF as scheduling policy.

The basic ideas underlying the two algorithms are the following:

- each functional block is mapped onto only one task (this assumption can be removed in future work);
- if two blocks belong to the same paths and one is the successor of the other, then the two blocks are candidates for placement in the same task;
- if there is no path between two given blocks, these cannot be assigned to the same task;
- once a task starts executing, it never blocks waiting the completion of another task; in other words, the only synchronization mechanisms used in our model are the activation of a task and mutual exclusion semaphores for shared resource;
- once a task starts executing, it can be preempted before completion only when an external event results in the activation of a higher

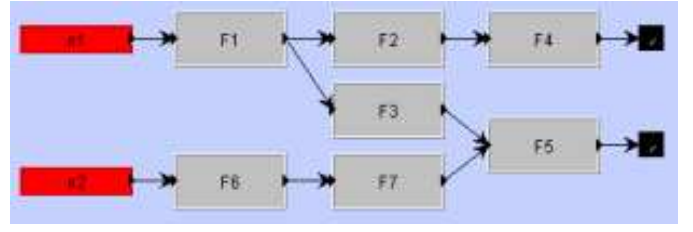


Fig. 4. Sample application graph.

priority task. In other words, a task never activates higher priority tasks. This provides an upper bound to the maximum number of preemptions;

- the generation of the task set must not depend on the computation times of the blocks;
- the parameters of the tasks (deadlines or priorities) depend on the end-to-end path deadlines;

The two algorithms presented in this research are called Late Activation (LA) and Joined Late Activation (JLA). To show the differences between them, the sample application shown in Figure 4 will be used. The WCETs of the functional blocks are shown below:

Block	F_1	F_2	F_3	F_4	F_5	F_6	F_7
γ	6	3	3	1	4	2	3

The DAG has three paths with the following characteristics:

- $P_1 = [e_1, F_1, F_2, F_4]$ has a relative deadline of $\Delta_1 = 18$;
- $P_2 = [e_1, F_1, F_3, F_5]$ has a relative deadline of $\Delta_2 = 22$;
- $P_3 = [e_2, F_6, F_7, F_5]$ has a relative deadline of $\Delta_3 = 25$.

A. Late Activation

The algorithm analyzes the DAG and creates the task set in two steps: in the first step, the tasks are created, and the functional blocks are assigned to the tasks; in the second step, it assigns the real-time parameters to the tasks.

The algorithm for creating the tasks is shown in Figure 5.

- Lines 1-3: every event activates at least a block which must be processed by the algorithm. After all successors of an event have been considered (and the queue Q emptied), the analysis proceeds to the next event.

```

Algorithm LA
Variables: a queue  $Q$  of blocks,
initially empty;
1: For every event  $e_i$  {
2:   insert the successor of  $e_i$  in the queue  $Q$ ;
3:   while (the queue is not empty) {
4:     extract block  $F_k$  from  $Q$ ;
5:     if ( $F_k$  is not an output) {
6:       create a new task  $\tau_j$ ;
7:       condition = true;
8:       while (condition) {
9:         insert  $F_k$  in  $\tau_j$ ;
10:        if ( $F_k$  has only one successor  $F_h$ 
11:         and  $F_h$  has only  $F_k$  as predecessor)
12:          insert  $F_h$  in  $\tau_j$ ;
13:          continue the cycle with  $F_k = F_h$ ;
14:        } else condition = false;
15:      }
16:      all successors of  $F_k$  that are not
17:      already part of a task are inserted
18:      in  $Q$ 
19:    }
20:  }
21}

```

Fig. 5. Pseudo-code for Algorithm LA.

- Lines 4-17: this cycle represents the processing of an event and all its successors.
- Line 5: if an output has been reached, then there is no other block in the path. The algorithm moves to the next element in the queue.
- Lines 6-9: a new task must be created, and blocks will be inserted into it. At least one block will be used; the presence of other blocks depends on the DAG topology.
- Lines 10-14: if F_k has only one successor F_h , and is its only predecessor, then all paths containing F_k also contain F_h . When this happens, the two blocks will belong to the same task. Otherwise, the current task is finished, and we leave the cycle.
- Lines 16-18: at this point, F_k will have some successors; these might have been processed earlier in the algorithm. If this is not the case, then they will be queued for later iterations.

Notice that the precedence relations over the DAG imply precedence relations over the tasks. As a consequence, we obtain a set of precedence related tasks, a model similar to the one by Chetto and Chetto [11]. All tasks will have the same following cyclic structure:

- wait for next activation;
- execute the assigned functional blocks;

- activate the successor tasks.

Notice that all successors are activated just before completion of the task instance. Also, the activations are buffered; therefore, it can be that while a task executes, it is activated again by some other external event or by some other task. In this case, the activation remains pending and is buffered until the task completes the currently executing instance.

Among all pending activations, the one corresponding to the path with the shortest deadline must be served first. Therefore, each activation carries information about the corresponding path deadline, and the incoming activations are inserted in a queue ordered by deadline.

Once the set of functional blocks has been partitioned into tasks, we must assign the scheduling parameters to the tasks. Each instance of an external event may result in multiple activations of internal tasks, but only one for the first task in the path. The rules for assigning the deadlines are listed below:

- every time a task is activated by an external event or by a functional link, it is assigned an absolute deadline equal to the minimum absolute deadline of all sub-paths originating from the activation event.
- a task activated by an external event, let it be

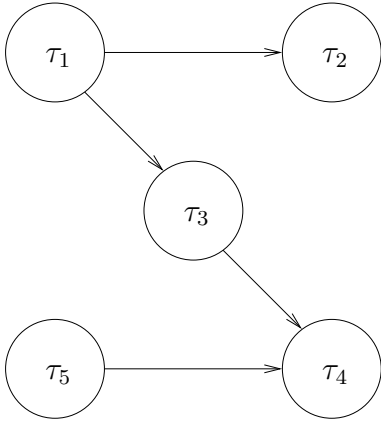


Fig. 6. The precedence relations among the tasks generated by LA.

τ_1 , has an absolute deadline of

$$d_1 = \min_i \{\delta_i \mid \tau_1 \in P_i\};$$

- the absolute deadline of a task $\tau_{i,l} \in P$, which is activated by $\tau_{i-1,h} \in P$ at time $f_{i-1,h}$, is

$$d_{i,l} = \min_j \{\delta_j \mid \tau_{i,l} \in P_j\}.$$

Notice that knowledge of the WCETs of the functional blocks is not required for computing the deadlines.

It is important to underline that, when activating a task, we must specify its current absolute deadline. Also, activations must carry information about the absolute deadline, computed according to the previous equations. Unfortunately, this is not usually available in commercial implementations of the EDF scheduler. We will discuss the implications of this characteristic of the algorithm in Section IX.

Finally, in the classical EDF scheduler, deadline ties can be broken arbitrarily. In this research, we assume that our EDF scheduler does not allow preemption between tasks with the same deadline.

B. Example with LA

To demonstrate how LA works, we show the results of the partitioning on the example of Figure 4. The task set and its parameters are listed below:

Task	τ_1	τ_2	τ_3	τ_4	τ_5
Components	F_1	F_2, F_4	F_3	F_5	F_6, F_7
C	6	4	3	4	5

The precedence relations among tasks are depicted in Figure 6.

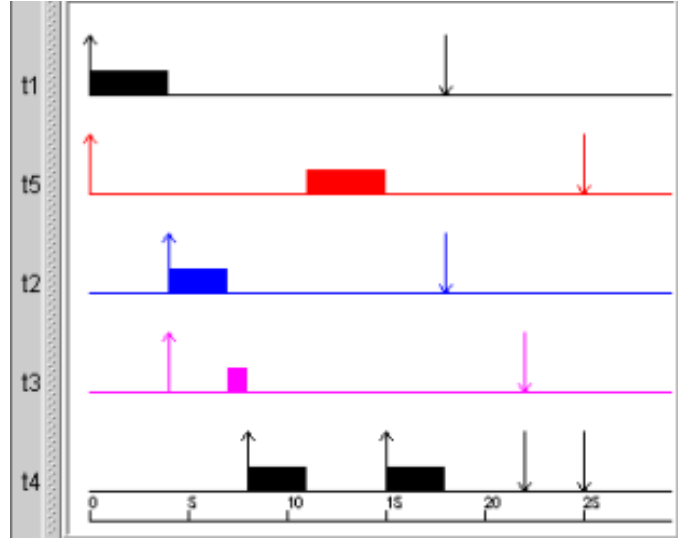


Fig. 7. Schedule produced by LA.

The first interval of the schedule obtained when all events are activated at time 0, is shown in Figure 7. One important property of this algorithm is that a task can only be preempted by the activation of an external event. In other words, the tasks in a path going from one external event to an output have increasing absolute deadlines: therefore, every task has always priority over its successors.

Another important observation regards the way the algorithm generates the tasks. Basically, every time one functional block has more than one successor, we create one new task for every successor. Also, every time a functional block has more than one incoming link, we create a new task that starts from this block. This second rule makes the schedulability analysis more difficult: in fact, such a task cannot be considered a periodic because it can be activated by two different external events that can have different periods or minimum interarrival times. Also, every time this task is activated, it can be assigned a different base deadline, depending on which event the activation comes from.

In the example, task τ_4 can be activated by task τ_3 and by task τ_5 : when activated by τ_3 , it is assigned a base deadline equal to the deadline of path $P_2 = [e_1, F_1, F_3, F_5]$, which is equal to $\Delta_2 = 22$; when it is activated by task τ_5 , it is assigned the base deadline of path $P_3 = [e_2, F_6, F_7, F_5]$, which is $\Delta_3 = 25$. Notice also, that τ_4 is not a periodic task, since it can be activated by event e_1 and by event e_2 .

As we will see in Section V-E, a schedulability analysis can still be carried out: one possible solution is to analyze each different activation separately.

C. Joined Late Activation

Algorithm JLA is an improvement over LA that tries to reduce the number of generated tasks. The pseudo-code for the algorithm is shown in Figure 8. The differences from the one shown in Figure 5 are displayed in **boldface**.

- Lines 10-12: if F_k is assigned to task τ_j , it is likely that one of its successors will be assigned to the same task. F_k belongs to a set of paths $\mathcal{P} = \{P_1, \dots, P_m\}$; each one of its successors belongs to a subset of \mathcal{P} . The minimum path deadline will be $\Delta = \min\{\Delta_l \mid P_l \in \mathcal{P}\}$. There will be at least one successor of P_k contained in a path with deadline Δ ; this will be the candidate successor. In case there is more than one they will all be candidates, and failing the first another one may be tested. If the candidate has only one input link, then it will be included in task τ_j . The task cannot contain a block which does not belong to the path with the minimum deadline, otherwise τ_j would activate a task which would immediately preempt it.
- Lines 14-15: all successors except the one which was included in the task, unless they have already been assigned to other tasks, must be queued for later processing.

The main difference is that, if a functional block has more than one successor, the one belonging to the path with the shortest base deadline is selected and inserted in the same task. Notice that, as before, the precedence relations in the DAG imply precedence relations among the generated tasks.

D. Example with JLA

When partitioned with JLA, the example shown in Figure 4 produces a set of four tasks, as shown in following table:

Task	τ_1	τ_2	τ_3	τ_4
Components	F_1, F_2, F_4	F_3	F_5	F_6, F_7
C	10	3	4	5

The precedence relations between the tasks are depicted in Figure 9.

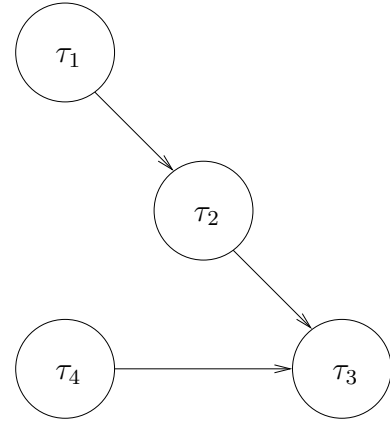


Fig. 9. The precedence relations among the tasks generated by JLA.

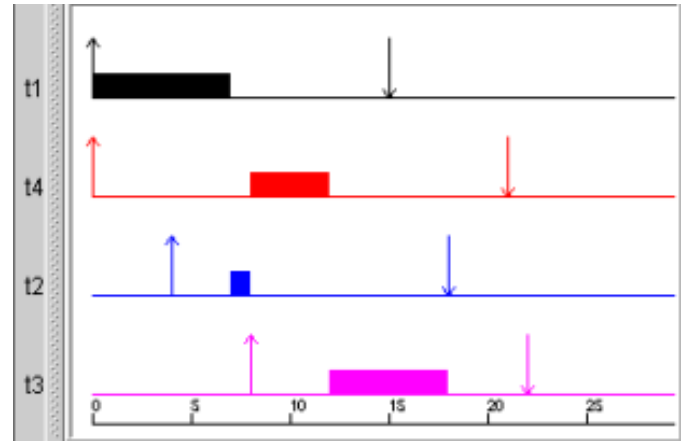


Fig. 10. Schedule produced by JLA.

It is important to point out that, since these tasks strictly follow the blocks' properties, τ_1 will execute for at most 6 (which will actually be 4 in our example) time units, then activates τ_2 before going on with its execution. The resulting schedule is charted in figure 10.

E. Schedulability test

In this section, we propose a schedulability analysis test for the generated task set. Let us start to analyze the schedulability of the task sets produced by Algorithm LA. We will then show that the same analysis can be applied to the task sets generated by JLA.

As anticipated in the previous section, the main problem is that when a task can be activated by more than one task, belonging to different events, it cannot be modeled as a periodic or sporadic task. In fact, each time this task is activated, it can be assigned a different task deadline. The basic idea

```

Algorithm JLA
Variables: a queue  $Q$  of blocks, initially empty;
1: For every event  $e_i$  {
2:   insert the successor of  $e_i$  in the queue  $Q$ ;
3:   while (the queue is not empty) {
4:     extract block  $F_k$  from  $Q$ ;
5:     if ( $F_k$  is not an output) {
6:       create a new task  $\tau_j$ ;
7:       condition = true;
8:       while (condition) {
9:         insert  $F_k$  in  $\tau_j$ ;
10:        select  $F_h$  as the successor of  $F_k$  that belongs to
11:        the path with the minimum relative deadline  $\Delta$ 
12:        if ( $F_h$  has only one predecessor) {
13:          insert  $F_h$  in  $\tau_j$ ;
14:          all successors of  $F_k$  not already assigned
15:          to a task, except  $F_h$ , are inserted in  $Q$ 
16:          continue the cycle with  $F_k = F_h$ ;
17:        } else condition = false;
18:      }
19:      all successors of  $F_k$  that are not already
20:      part of a task are inserted in  $Q$ ;
21:    }
22:  }
23:}

```

Fig. 8. Pseudo-code for Algorithm JLA.

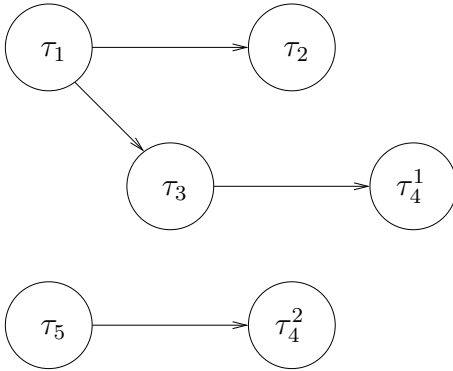


Fig. 11. Splitting task τ_4 .

underlying our analysis is to *split* such tasks (and all the successors) in many tasks, one for each different activation.

Let us clarify the idea with an example. Consider again the precedence graph of Figure 6. We split task τ_4 into two different tasks, τ_4^1 and τ_4^2 , activated by tasks τ_2 and τ_5 , respectively. The resulting graph is shown in Figure 11.

As you can see, in this case, the original precedence graph has been separated into a “forest” of trees. Each tree has an event as root. Now, task τ_4^1 can only be activated by task τ_2 and hence by event e_1 . Therefore, all tasks in the tree with e_1 as a

root can be associated the same period T_1 and each task is always assigned the same base deadline. The same thing happens with tasks τ_5 and τ_4^2 .

However, we also have to take into account an additional constraint: since τ_4^1 and τ_4^2 are actually the same task, τ_4^1 is not allowed to preempt τ_4^2 and vice versa. To account for this non-preemption constraint, we introduce a *pseudo-resource* for each split task. In the example, a resource R_{τ_4} is created, and every time τ_4^1 or τ_4^2 are activated, they must first lock resource R_{τ_4} before starting to execute.

Generalizing, we transform a precedence graph in a set of trees, according to the following rules:

- every external event becomes the root of a tree;
- the tree contains all the tasks activated by the corresponding root event;
- suppose that a task τ_i is activated by many events or tasks; we split the task into many identical “pseudo-tasks”, one for every different incoming activation: $\tau_i^1, \dots, \tau_i^m$, where m is the number of different incoming activations. All of them will have the same WCET; each one has a different base deadline, depending on the corresponding path. Moreover, since a task cannot interrupt itself, we create a “pseudo-resource” R_{τ_i} , shared between all the split tasks originated by τ_i , to prevent them from

preempting one another. Each split task τ_i^k is assigned a critical section ξ_{i,τ_i}^k with duration equal to the WCET of τ_i . In addition, when a task is split, all its successors must be split too with the same method.

At this point, each task (both normal and pseudo-task) can be associated a period T_i equal to the period (or minimum interarrival time) of the root event. We have then transformed our set of complex tasks into a set of sporadic tasks, for which schedulability analysis techniques are well-known.

To test the schedulability of the generated task set, we use the processor demand criterion, first proposed by Baruah et al. [4]. According to this criterion, the worst case condition is when all events are activated at the same time, that by convention we denote with time 0. Then, we must test the processor demand in all time intervals starting from time 0 until the first idle time. The two conditions to check are the following:

$$U = \sum_{i=1}^N \frac{C_i}{T_i} < 1$$

$$\forall L \leq L^*, \sum_{i=1}^N \left[\left(\left\lfloor \frac{L - D_i^b}{T_i} \right\rfloor + 1 \right) C_i \right] + B(L) \leq L, \quad (1)$$

where N is the number of tasks, including the pseudo-tasks generated by the transformation described above, and $U = \sum_{i=1}^N \frac{C_i}{T_i} < 1$ is the total system load. The term $B(L)$ is the blocking time in interval L and it is defined as follows [19]:

$$B(L) = \max_{k,r} \left\{ \xi_{k,r} \mid D_k^b > L \wedge \text{ceil}(R_r) \geq \frac{1}{L} \right\}.$$

Finally, L^* is an upper bound to the end of the busy period. It is defined by a recursive equation:

$$L^*(0) = \sum_{i=1}^N C_i$$

$$L^*(k) = \sum_{i=1}^N \left\lceil \frac{L^*(k-1)}{T_i} \right\rceil C_i$$

The recursion ends when $L^*(k) = L^*(k-1)$, and convergence is guaranteed when $U < 1$. In the special case in which all deadlines are less than the

period, an alternative formula for L^* is the following [4]:

$$L^* = \frac{U \max_i (T_i - D_i^b)}{1 - U}.$$

Theorem 1: Given a task set generated by algorithm LA, and the corresponding precedence graph, consider the task set obtained by transforming the precedence graph in a forest of trees according to the above methodology. The task set is schedulable if Equation (1) is verified.

Proof: The proof simply descends from the processor demand analysis. Proofs of correctness for Equation (1) can be found in [4], [10] and [19]. ■

Finally, it is straightforward that the above schedulability test is valid both for the task sets generated by LA and by JLA. In fact, the transformation method can be applied to both JLA and LA.

VI. EXAMPLE

This section exposes an illustrative example on the complete use of the proposed methodology, from the description of the system using the extended profile, to the application of the JLA algorithm for generating the task set from the functional blocks, to the execution of that task set on a scheduler. In particular this example shows a scenario where a system collects informations from the environment by means of a sensor, filters them and update the its control subsystem. At the same time the filtered informations capturer, can be transformed in a human readable form and stored in a log repository. Periodically a user can query such repository in order to monitor the whole system.

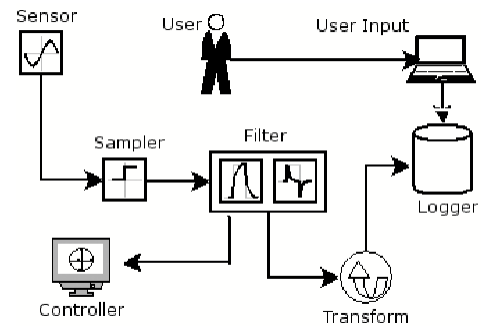


Fig. 12. The Scenario in the Example

Figure 13 depicts the use of the extended notions of UML Activity Diagrams in order to model the informal scenario in Figure 12.

The model is composed by three paths (DataControl, SignalRecord, UserLog). Each of them is represented by an Activity Partition stereotyped as $\ll\text{Path}\gg$ and their deadline is represented by the tagged value Deadline set to 18, 40, 200 respectively. DataControl and SignalRecord share the same external event (e1) and the first two functional blocks (Sampler, Filter). e1 produces the stimulus for the system with a constant interval of time and in particular each 40 msec. On the other hand, UserLog has its own periodic initial node (e2), but share its last action node with SignalRecord (Logger). At last, each functional block was modeled by an action stereotype as $\ll\text{SAction}\gg$ and the duration of the their execution is represented by the tagged value SAworstCase.

From this formalization of the system, by means of a transformation engine the UML scenario can be provided as input to the algorithms described in Section V in order to obtain an activity scheduling if it is possible. For the motivations argued in Section I-B, at the moment the engine is not already implemented, although theoretically such transformation is possible.

The table below shows in a compact form the path structure of the example:

Path	Functional blocks	Δ
DataControl	Sampler, Filter, Ctrl	18
SignalRecord	Samp., Filter, Transf., Log	40
UserLog	UserInput, Logger	200

We can now apply our partitioning methodology, as explained in Section V. We first use rule 1, which states that every task must be included in a path. One possible resulting task set is shown below:

Task	Functional blocks
τ_1	Sampler, Filter, Transform, Logger
τ_2	Ctrl
τ_3	UserInput

This partitioning is not in accordance with rule 2, because the Logger block has more than one input link. By applying the second rule, we obtain the following:

Task	Functional blocks
τ_1	Sampler, Filter, Transform
τ_2	Ctrl
τ_3	UserInput
τ_4	Logger

Last, we apply rule 3. It specifies that τ_1 should not include the Transform block, but rather the Ctrl block; this is because the DataControl path has a stricter deadline than the SignalRecord path. If the partitioning used were the one described above, the Transform function would be executed before the Ctrl block, resulting in a priority inversion. The final task set is as follows:

Task	Functional blocks	Deadline	Period
τ_1	Sampler, Filter, Ctrl	18	40
τ_2	Transform	40	-
τ_3	Logger	40/200	-
τ_4	UserInput	200	100

Deadlines have also been assigned according to the rules described in Section V. These are the absolute deadlines relative to the first instance of the tasks; subsequent instances will have greater deadlines. It is possible to assign relative values for deadline; these would be the base deadlines; however, they are not in the scope of this article.

τ_1 and τ_4 are activated by external events, therefore they can be assigned a period equal to that of the event. On the other hand, τ_2 and τ_3 do not have an explicit period, since they are activated at non-periodic intervals after the execution of their predecessors.

The reason why τ_3 is assigned two different deadlines is because it is activated twice, once by τ_2 and the other by τ_4 . When activated by τ_2 , its deadline is that of the SignalRecord path, i.e. 40; when activated by τ_4 , it has the deadline of the UserLog path, which is 200.

This task set is ready for execution on an EDF scheduler. The Gantt chart produced by a simulation of the schedule is shown in Figure 14. In the chart, up-pointed arrows represent task activation times, while a task deadline is depicted with an arrow pointing down. Rectangular blocks (independently of their color) show the task running on the CPU.

Starting from an XML representation of the DAG, the whole methodology, from the task partitioning and the deadline assignment to the simulation, has been executed with a tool we have developed.

VII. PREEMPTION BOUNDS

In this section, we provide worst-case bounds for estimating the number of preemptions in the system. The following theorems are based on the concept of

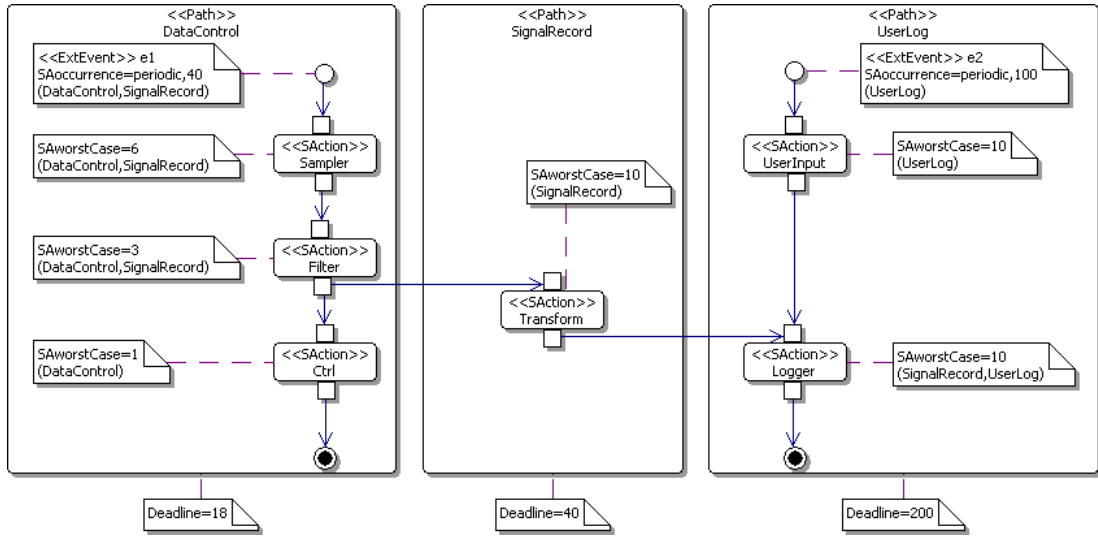


Fig. 13. A DAG representation by means of the UML Activity Diagram extension.

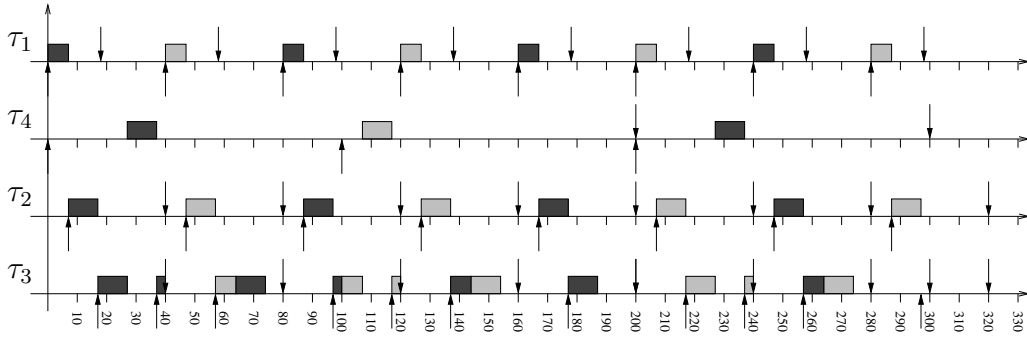


Fig. 14. Schedule chart example.

transaction. A *transaction* Γ is the complete, worst-case set of functional blocks executed upon arrival of an external event with a minimum interarrival time equal to that of its source event. The number of transactions in the system is clearly equal to the number of external events and each transaction is a tree of functional blocks (Section V-E). $\Gamma_i(k)$ refers to the k -th instance of transaction Γ_i . For practical reasons, we denote an event, its minimum interarrival time and its transaction with the same index.

A transaction contains a number of paths and the path sets of different transactions have a null intersection. Among all paths in Γ , the minimum and maximum deadlines are labeled, respectively as $\Delta^{\min} = \min\{\Delta_i \mid P_i \in \Gamma\}$ and $\Delta^{\max} = \max\{\Delta_i \mid P_i \in \Gamma\}$.

Theorem 2: *The execution of a task τ in the transaction instance $\Gamma_i(k)$ cannot be preempted by another task in the same transaction instance.*

Proof: In LA, a task can only activate other tasks in the same transaction instance at its completion time, hence no preemption can possibly occur. In JLA, it is possible for a task to activate another task in the same transaction before it completes, but the newly activated task always has a deadline higher than, or equal to the deadline of the activating task τ . This is because the set of paths to which τ belongs is a superset of the path sets of its successors, and the minimum path deadline for τ will always be less than or equal to the minimum path deadline for its successors. Since preemption among same priority tasks is disallowed, then proof follows. ■

Theorem 3: *If a transaction Γ has a minimum interarrival time T such that $T + \Delta^{\min} > \Delta^{\max}$, then a job in the k -th instance $\Gamma(k)$ will never be preempted by a job belonging to the following instances $\Gamma(k')$ with $k' > k$.*

Proof: Assume the triggering event of instance

$\Gamma(k)$ arrives at time A . Then, the least possible time for the activation of the next instance $\Gamma_i(k+1)$ will be $A+T$. The maximum deadline for a task instance τ_i in $\Gamma(k)$ is $d_i = A + \Delta^{max}$ and the minimum possible deadline for a task τ_j in $\Gamma(k+1)$ is $d_j = A+T+\Delta^{min}$. From our hypothesis on T it is clearly $d_j > d_i$. Therefore, every job deadline in $\Gamma(k)$ is lower than any job deadline in $\Gamma(k+1)$ and also in $\Gamma(k')$ with $k' > k$. ■

Corollary 1: In a system with n external events, if $\forall i, T_i + \Delta_i^{min} > \Delta_i^{max}$, then the maximum possible number of preemptions at any given time is $n - 1$.

Proof: Given Theorems 2 and 3, if the above property is true a task can never be interrupted by another task in the same transaction (regardless of the instance). The only possible preemptions are between different transactions, and since there are n in total, there can be a maximum of $n - 1$ preemptions and n active tasks at any time. ■

Please note, if deadlines are not assigned to paths, but to transactions (as in [20]), then $\Delta^{min} = \Delta^{max}$ and the limit on the number of preemptions holds regardless of the interarrival times of the transactions.

The above results can be generalized to find the maximum number of active tasks in the system when the condition on event interarrival times and transaction deadlines of Theorem 3 cannot be guaranteed. In this case, it is possible to bound the number of future instances of a transaction that can possibly preempt a transaction instance.

Theorem 4: An instance of a transaction, $\Gamma(i)$, can be preempted by a maximum of k^M future instances of the same transaction, i.e., the last instance that can preempt $\Gamma(i)$ is $\Gamma(i+k^M)$, where

$$k^M = \left\lfloor \frac{\Delta^{max} - \Delta^{min}}{T} \right\rfloor \quad (2)$$

Proof: If we assume periodic transactions, which is the worst-case assumption, then $\Gamma(i)$ will be activated at time iT . Its longest deadline will be $iT + \Delta^{max}$. The $i+k$ -th instance of the same transaction will be activated at time $(i+k)T$, and the shortest deadline of its tasks will be $(i+k)T + \Delta^{min}$. The preemption is possible if $(i+k)T + \Delta^{min} < iT + \Delta^{max}$, which is true if $k < \frac{\Delta^{max} - \Delta^{min}}{T}$. In addition, k must be an integer; hence Equation 2. ■

The bound on the number of preemptions occurring at any given time can be used to bound the

number of task frames that are active and need to be stored in the system stack at any given time. This bound can be used to provide a better estimate of the amount of RAM memory that is required for the execution of the system.

VIII. PROOF OF OPTIMALITY FOR LA AND JLA

In this section, we demonstrate that the task synthesis algorithms of both LA and JLA are optimal under EDF scheduling. The following theorems prove that task sets generated by LA and JLA are EDF schedulable if and only if the task set produced by one-to-one mapping of tasks to functions is schedulable under EDF. Given that EDF scheduling is optimal in the case of independent tasks, then both JLA and LA are also optimal scheduling algorithms in the case of independent function trees.

If scheduling overheads are considered, then JLA produces less context switches and it is clearly to be preferred over the other methods.

In the following, the term *Full Deadlines* refers to the deadline assignment policy giving a task a deadline equal to its minimum path deadline.

We first show that, in case of independent trees and functions, a one-to-one mapping (one task per function) using the *Full Deadlines* assignment is optimal. Then, we prove that the grouping of functions into tasks performed by the LA and the JLA algorithms do not compromise schedulability.

A. Full Deadlines optimality

Theorem 5: If a generic DAG system S is mapped into a task set T by assigning a unique task τ_i for each functional block F_i , and if every task τ_i is assigned a deadline d_i equal to the lowest path deadline δ among all paths to which block F_i belongs to, then a necessary and sufficient condition for feasibility of S is that T be EDF feasible.

Proof: S is feasible $\Rightarrow T$ is EDF feasible. If S is feasible, then a schedule σ exists such that every path P_j finishes its computation within its path deadline δ_j . Every functional block F_i , consequently, must have ended before the path deadline of all paths to which it belongs: $\forall j \mid F_i \in P_j, f_i \leq \delta_j$, where f_i is the finishing time for F_i . If $d_i = \min_j \{\delta_j \mid F_i \in P_j\}$ is the minimum path deadline for F_i , we have $f_i \leq d_i$. According to the *Full Deadlines* assignment, d_i is also the deadline for task τ_i (implementing block F_i). It follows, that the

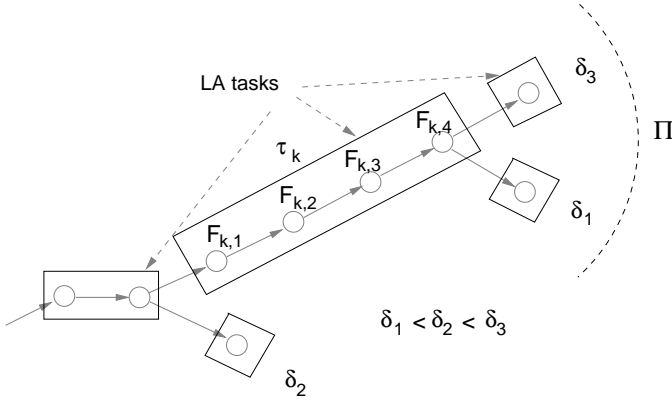


Fig. 15. LA grouping of functional blocks.

feasible schedule for S is also a feasible schedule for T . Since EDF is optimal and a feasible schedule for T exists, then the task set T is EDF schedulable.

T is EDF feasible $\Rightarrow S$ is feasible. If T is EDF feasible, then every task τ_i completes before its deadline and so does the corresponding functional block F_i : $f_i \leq d_i$. But $d_i = \min_l \{\delta_l \mid F_i \in P_l\}$, then $f_i \leq d_i \Rightarrow \forall j \mid F_i \in P_j, f_i \leq \delta_j$. This means that every functional block executes before the deadlines of all paths it belongs to. Then, S is feasible. ■

B. LA optimality

The second step is to prove that the LA grouping does not compromise the optimal schedulability of the *Full Deadlines* algorithm.

Theorem 6: *If a task set T is a one-to-one assignment of functional blocks from a feasible DAG system S using the Full Deadlines assignment, and T' is the task set created by applying the LA grouping to S , then T is EDF feasible $\Leftrightarrow T'$ is EDF feasible.*

Proof: T is EDF feasible $\Rightarrow T'$ is EDF feasible.

Let $\mathbf{F}_k = \{F_{k,1}, F_{k,2}, F_{k,n}\}$ be the set of functional blocks assigned to the same task τ_k in T' where $F_{k,j} \prec F_{k,j+1}$ (Figure 15). They belong to the same set of paths Π_k . This means that task $\tau_k \in T$ and all tasks implementing the functional blocks $\{F_{k,1}, F_{k,2}, F_{k,n}\}$ in T have the same deadline $d_{k,1} = d_{k,2} = \dots = d_{k,n} = d_k = \min_l \{\delta_l \mid P_l \in \Pi_k\}$. According to the EDF scheduling rules, tasks with the same deadline can be scheduled in any order. Hence, if T is EDF feasible, then there exists a feasible schedule where all the functional blocks sets belonging to the sets \mathbf{F}_k for all k , are executed consecutively, from $s_{k,1}$ to $f_{k,n}$. A feasible EDF

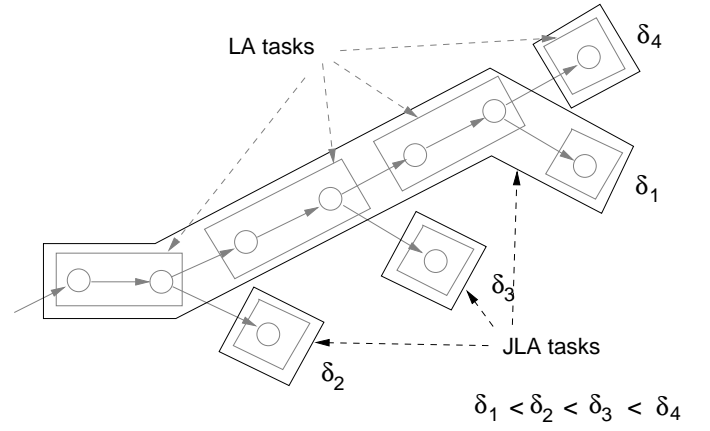


Fig. 16. JLA grouping of LA tasks.

schedule for T' can now simply be constructed by scheduling task τ_k on the CPU at time $s_{k,1}$. τ_k will finish in $f_{k,n}$, since it executes for the sum of the computation times of the functional blocks mapped into it. The task schedule for T' is guaranteed to be EDF because the task deadlines match the deadlines of the functional sets they are replacing.

T' is EDF feasible $\Rightarrow T$ is EDF feasible. Let τ_k be a task in T' which contains F_i and F_j , respectively assigned to τ_i and τ_j in T . If T' is schedulable, then $f_k \leq d_k$. According to the *Simple Partitioning* rules, F_i and F_j must necessarily belong to the same set of paths Π , so their minimum path deadlines d_i and d_j are the same, and they are equal to d_k . Therefore, independently of the order of the two blocks, since τ_k executes both, $f_i \leq d_i$ and $f_j \leq d_j$, making the task set T feasible. ■

C. JLA optimality

Finally, we assume that a DAG system is partitioned using the JLA algorithm, with the *Full Deadlines* assignment. We need to demonstrate that such a task set is schedulable if the task set generated with LA is schedulable, and vice versa.

Once again, the proof exploits equivalence of the deadlines of all the LA tasks that are merged into a single JLA task. In fact, when a functional block F_i has more than one output, the successor F_j , which is assigned to the same task by JLA is the one on the path with the shortest deadline (Figure 16).

Theorem 7: *If a task set T is created using the LA algorithm from a DAG system S , and T' is the reduced task set created with JLA, T is feasible $\Leftrightarrow T'$ is feasible.*

The proof is similar to the proof of LA optimality, considering that JLA tasks group LA tasks with identical deadline similarly to the way LA tasks group functional locks with the same deadlines. The interchangeability of tasks with the same deadline in an EDF schedule can be used to demonstrate equivalence with respect to EDF schedulability of the two sets.

IX. CONSIDERATIONS ON IMPLEMENTATION

As described in Section V-A, the way the task set is generated requires the operating system to support some special functionality that is not available from current operating systems. In particular, if a task can be activated by more than one task, and hence by more than one event, it must be assigned a different base deadline depending on which event the activation comes from. Therefore, each activation has to carry information on the event that originated the activation itself and on the deadline of the shortest path that can be originated from the task. Also, since pending activations must be buffered, they must be served in an earliest deadline order.

While the shortest relative deadline among all the possible outgoing paths can be statically computed, the activation time of the event and the external source that triggered the execution path must be dynamically communicated along the chain of computation by adding the appropriate information to the signal.

Implementing such mechanism can be hard, since it demands for additional run-time overheads, limiting its adoption in embedded systems with limited computational resources.

In Section V-E, we presented a mechanism to transform a precedence graph in a set of precedence trees (transactions), by splitting tasks that have more than one incoming activation. In this way, every task is assigned a fixed base deadline. Therefore, all tasks activated by one external event can be activated at the same time with their own fixed base deadline. This can be easily implemented also as an user space library, without any modification to the scheduling mechanism. However, the transformation method generates many more tasks and resources, and this in turn may require more overhead and more memory to implement the application.

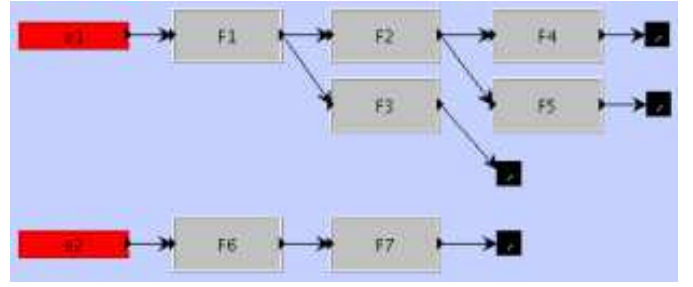


Fig. 17. An example of DAG not schedulable by fixed priority.

X. A SAMPLE COMPARISON

In this section, we show the advantage of using our methodology, based on EDF scheduling, over the methodology in [20] that is based on fixed priority scheduling. The authors of [20] mention a heuristic algorithm that tentatively assigns functional blocks and priorities to tasks until a schedulable solution is found. However, since the algorithm was not described in detail, we could not compare it against ours. Rather, we will try to use both rate monotonic and deadline monotonic priority assignments.

The example DAG is shown in Figure 17. The WCET of the blocks are the following:

Block	F_1	F_2	F_3	F_4	F_5	F_6	F_7
γ	30	10	30	20	50	40	35

The relative deadlines assigned to the paths are the following:

- $P_1 = [e_1, F_1, F_3]$ has a deadline of $\Delta_1 = 100$;
- $P_2 = [e_1, F_1, F_2, F_5]$ has a deadline of $\Delta_2 = 200$;
- $P_3 = [e_1, F_1, F_2, F_4]$ has a deadline of $\Delta_3 = 300$;
- $P_5 = [e_2, F_6, F_7]$ has a deadline of $\Delta_4 = 150$.

Finally, the period of e_1 is $T_1 = 300$ while period of e_2 is $T_2 = 150$.

By applying algorithm JLA to the DAG, the following tasks are derived:

Task	τ_1	τ_2	τ_3	τ_4
Components	F_1, F_3	F_2, F_5	F_4	F_6, F_7
C	40	50	50	75
D^b	100	200	300	150
T	300	300	300	150

Now we suppose that these tasks are assigned fixed priorities, using rate monotonic: tasks τ_1 , τ_2 and τ_3 have priority 1 (the lowest) while task τ_4 has priority 2 (the highest). By applying response time analysis, task τ_1 has a worst case response

time of 115 that is greater than its base deadline. Therefore, the system is not schedulable with this priority assignment.

Another possibility is to try a deadline monotonic priority assignment. In this case, τ_1 has priority 4 (the highest), τ_4 has priority 3, τ_2 has priority 2 and τ_3 has priority 1 (the lowest). Again, by applying response time analysis, we obtain a worst case response time for τ_2 of 240, greater than its deadline.

Instead, by applying Equation 1 the system results schedulable by EDF.

Please note that the example is not simply a pathological case, but it is representative of an entire class of software models where a single external event triggers more paths with different deadlines.

XI. EXPERIMENTS

We performed several experiments on randomly generated task graphs in order to verify the time and schedulability efficiency of the algorithms, when compared to state of the art solutions based on fixed priority scheduling. The following parameters were subject to evaluation:

- feasibility: to verify the efficiency in terms of the percentage of functional graphs that can be scheduled;
- grouping efficiency: to check the size of the generated task set.
- execution time: to give an indication upon the size of the problems (in terms of number of functional blocks) that can be effectively solved by our algorithms.

A. Experimental setup

We compared our algorithms against an implementation of [20] where fixed priorities are assigned according to the Rate Monotonic or the Deadline Monotonic rule. We tested the scheduling algorithms with randomly generated graphs of functional blocks, produced using the TGFF, *Task Graphs For Free* tool. The TGFF tool, now at version 3.0, is widely used in the research community on embedded systems and almost a de-facto benchmark when testing algorithms working on graphs of tasks or functional blocks. Unfortunately, when used for generating graphs with timing constraints, it only provides limited control over the generation of the

deadlines and the activation rates of tasks and events.

The TGFF tool generates task graphs (graphs of functional blocks, for the purpose of our study) based on the configuration parameters defined in the *tgffopt* input file and it produces a *tgff* file with a text description of the graph. The graph description produced by TGFF contains a table with the timing attributes that are assigned to each block: computation time, period, and also deadline (in the last version of the program). In our case, period and deadline are used in events (root nodes) and outputs (leaf nodes), respectively and ignored in other nodes. Unfortunately, TGFF assigns deadlines to nodes in a deterministic way, based only on their distance from the root node (in direct proportion). Furthermore, even if the current version of the program allows generating graphs with multiple root nodes, the graph generator assigns the same activation period to all of them.

In order to provide the additional flexibility and generality required for our real-time analysis, we wrote a feeder program (in C++), which creates the *tgffopt* option file, setting the parameters required by TGFF for graph generation and we also developed a post-processing program then transforms the *tgff* output file by modifying the deadlines and the activation rates of the leaf and root nodes. Uncertainty in the values of periods and deadlines is obtained by multiplying the TGFF values, by random values uniformly drawn from the interval $[1.0, 3.0]$. Activation periods are finally rounded up in quanta to lower the hyperperiod and the complexity of the EDF test. The final output of the post-processing phase is an XML file describing the functional graph. The XML file is processed by an analysis and simulation tool (written in Java) implementing the grouping and schedulability analysis algorithms. The whole process is invoked repeatedly by a bash script which assigns different parameters at every cycle.

There is a large number of parameters that can be used to define a functional graph:

- the number of (external) event nodes n_{ev}
- the number of functional nodes n_f
- the (average) in-degree n_{in} and out-degree n_{out} of the nodes
- the average utilization U for the entire functional set
- the average ratio between the Deadline of the leaf node with highest depth and the period of

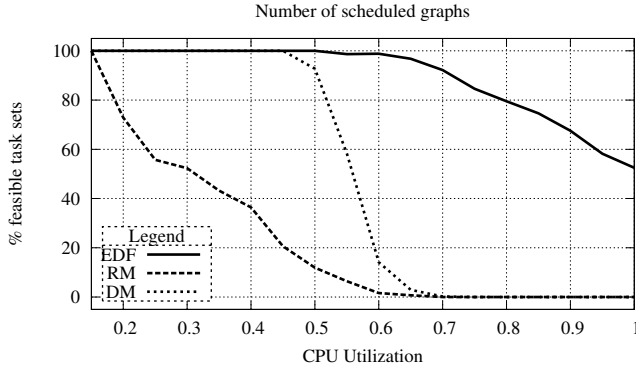


Fig. 18. Percentage of schedulable graphs when $D/T \approx 1$ ($n_f = 25 \pm 10$).

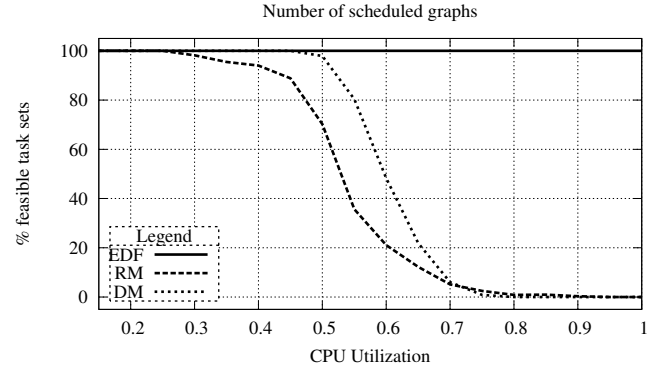


Fig. 19. Percentage of schedulable graphs when $D/T \approx 2$ ($n_f = 25 \pm 10$).

its triggering event (D/T)

B. Results

In the first series of tests we used JLA and LA for grouping functional blocks into tasks and then we compared our methods against an implementation of Saxena's schedulability analysis where fixed priorities are assigned according to the event rates (in Rate Monotonic fashion), or according to the smallest deadline among all the output nodes that can be triggered by the block (that is, Deadline Monotonic).

In this case, we selected $n_{ev} = 3 \pm 1$ external events, and $n_f = 25 \pm 10$ functional blocks. Initially, we ran the test with the maximum deadlines approximately equal to the periods $D/T \approx 1$, then the ratio was increased with steps of 0.1, until $D/T \approx 2$. For each set of values, we tried different utilization values, from $U = 0.1$ to $U = 1.0$.

In all cases, our algorithm performed much better than its fixed priority scheduling counterparts. Figure 18 shows the percentage of schedulable sets when the deadline approximately equals the activation period. When the ratio D/T is increased, an increasing number of task sets is found schedulable, but the gap between our EDF-based method and fixed-priority based algorithms widens even more (Figure 19).

We then went back to our initial test with a deadline/period approximate ratio of 1, and spanned the results with a variable number of average functional blocks, from 5 ± 1 (Figure 20) to 50 ± 10 (Figure 21). The percentage of schedulable sets did not change significantly with the number of

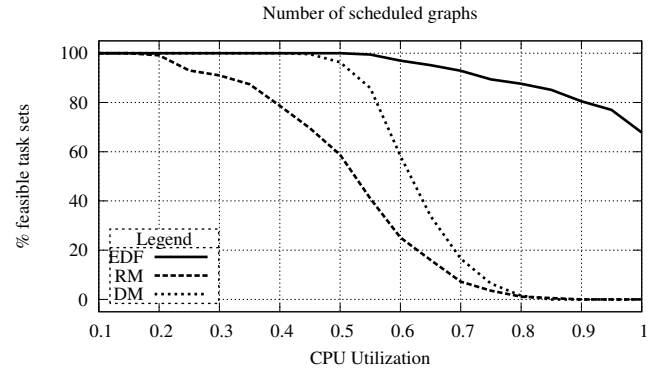


Fig. 20. Percentage of schedulable graphs when the number of functional blocks is 5 ± 1 ($D/T \approx 1$).

functional blocks. The slight increase in the performance of our method for larger sets and the corresponding decrease for fixed-priority scheduling should be considered within the range of experimental uncertainty.

To test for grouping efficiency, we generated graphs with increasing numbers of average functional blocks, and tested the grouping obtained with LA and JLA in terms of number of tasks. We ran batches of 500 seeds, with 5 to 50 average blocks for each seed. Every test batch had different values for the maximum in-degree and out-degree (number of input and output links for each block, respectively). We ran batches with maximum in-degrees of 1 and 2, and maximum out-degrees of 2 to 4.

Figures 22 and 23 show the result for the two extreme settings ((in-degree,out-degree) respectively equal to (1,2) and (2,4)). The graphs clearly show that, unless the graph is very sparsely connected, only JLA results in a number of tasks significantly smaller than the number of functional blocks and,

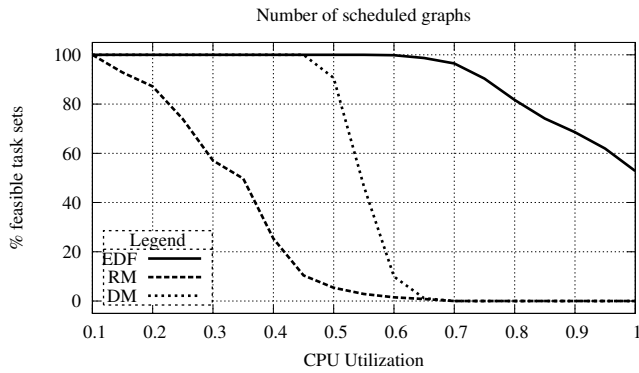


Fig. 21. Percentage of schedulable graphs when the number of functional blocks is 50 ± 10 ($D/T \approx 1$).

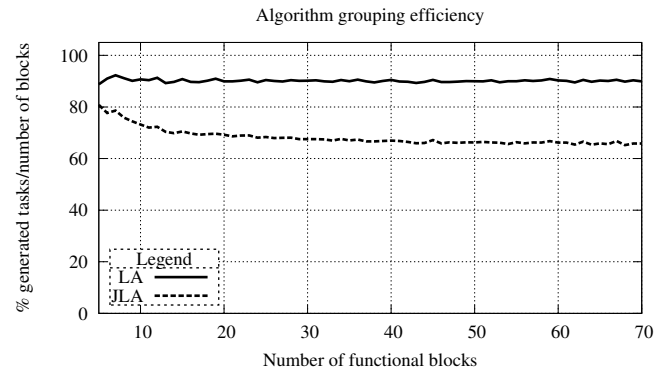


Fig. 23. Percentage of generated tasks with respect to number of blocks, assuming max in-degree = 2 and max out-degree = 4.

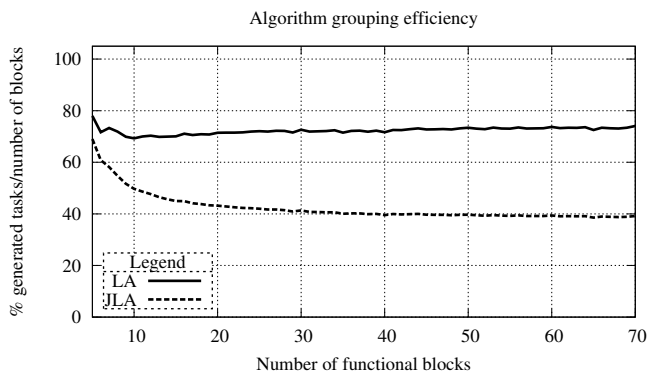


Fig. 22. Percentage of generated tasks with respect to number of blocks, assuming max in-degree = 1 and max out-degree = 2.

more importantly, it produces tasks of larger size and less subject to large context switch overheads.

All the experiments done so far have shown that our methodology is computationally tractable: on a 2Ghz AMD Athlon 64, running algorithm JLA and the corresponding scheduling analysis on a DAG with 100 functional blocks takes less than 4 seconds.

XII. CONCLUSIONS

We have presented a model for the description of the dataflow architecture of embedded systems and algorithms for the synthesis of the architecture-level design, the automated logical-to-architectural mapping and schedulability analysis of the resulting task set. Our proposal is based on runtime support from a real-time operating system capable of earliest deadline scheduling. The presented solution allows to reduce the overheads and excessive priority inversions of existing solutions that map all functional blocks (or reactions) into a single thread or assign a thread of execution to each action or possibly

to each active object. Sample cases show how our method can possibly improve the schedulability of the dataflow graph, implemented in a set of threads scheduled with dynamic priorities in comparison with existing solutions based on fixed priority.

Extensions of this research will be focused on further reducing the number of tasks and applying the methodology to multiprocessor and distributed systems.

REFERENCES

- [1] *UML Profile For Schedulability, Performance, And Time, Version 1.0*. <http://www.omg.org>, 2001.
- [2] T. Baker, "Stack-based scheduling of real-time processes," *Journal of Real-Time Systems*, vol. 3, 1991.
- [3] F. Balarin, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, and Y. Watanabe, "Metropolis: An integrated environment for electronic system design," *IEEE Computer*, vol. 36, April 2003.
- [4] S. Baruah, L. Rosier, and R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor," *The Journal of Real-Time Systems*, vol. 2, 1990.
- [5] T. Beck, "Current trends in the design of automotive electronic systems," in *Proceedings of the Design Automation and Test in Europe Conference*, 2001.
- [6] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, Jan. 2003.
- [7] J. Bézivin, "On the unification power of models," *Journal of Software and Systems Modeling*, vol. 4, no. 2, pp. 171–188, May 2005.
- [8] A. Brown, "Model driven architecture: Principles and practice," *Journal of Software and System Modeling*, vol. 3, no. 4, pp. 314–327, 2004.
- [9] A. Burns and A. Wellings, *HRT-HOOD: A Structured Design Method for Hard Real-Time Systems*. Elsevier Science, Amsterdam, NL, 1995.
- [10] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Boston: Kluwer Academic Publishers, 1997.

- [11] H. Chetto and M. Chetto, "An adaptive scheduling algorithm for fault-tolerant real-time systems," *Software Engineering Journal*, pp. 93–100, May 1991.
- [12] H. Eriksson, M. Penker, and D. Fado, *UML 2 Toolkit*. New York, NY, USA: John Wiley & Sons, Inc., 2003.
- [13] R. Gerber, S. Hong, and M. Saksena, "Guaranteeing end-to-end timing constraints by calibrating intermediate processes," in *Proceedings of Real-Time Systems Symposium*, December 1994.
- [14] E. Lee, "Overview of the ptolemy project," University of California, Berkeley, Tech. Rep. UCB/ERL-M01/11, 2001.
- [15] Mathworks, "The mathworks simulink and stateflow user manuals," available on Internet: <http://www.mathworks.com>.
- [16] A. K. Mok and C. Puchol, "Integrated design tools for hard real-time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, December 1998.
- [17] *UML 2.0 Superstructure Specification*, OMG Document – formal/05-07-04 ed., OMG, July 2005.
- [18] *UML Profile for Schedulability, Performance and Time Specification*, OMG Document – formal/05-01-02 ed., OMG, January 2005.
- [19] R. Pellizzoni and G. Lipari, "Feasibility analysis of real-time periodic tasks with offsets," *Real-Time Systems*, vol. 30, no. 1-2, pp. 105–128, May 2005.
- [20] M. Saksena, P. Karvelas, and Y. Wang, "Automatic synthesis of multi-tasking implementations from real-time object-oriented models," in *Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, March 2000.
- [21] L. Sha, R. Rajkumar, and John P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE transaction on computers*, vol. 39, no. 9, September 1990.
- [22] J. Stankovic *et al.*, "Vest: An aspect-based composition tool for real-time systems," in *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003)*, May 27-30, 2003, Toronto, Canada. IEEE Computer Society, 2003.
- [23] L. Tratt, "Model transformations and tool integration," *Journal of Software and Systems Modelling*, vol. 4, no. 2, pp. 112–122, May 2005.