High-Level Modeling of Communications in Real-Time Embedded Systems

Dinesh Ramanathan
Advanced Technology Group
Synopsys Inc.
Mountain View, CA 94043
dinesh@synopsys.com

Ali Dasdan
Dept. of Computer Sci.
University of Illinois
Urbana, IL 61801
dasdan@cs.uiuc.edu

Rajesh Gupta
Dept. of Info & Computer Sci.
University of California
Irvine, CA 92697
gupta@ics.uci.edu

Abstract
An embedded system continuously interacts with its environment under strict timing constraints. We model an embedded system using a generalized task graph (GTG). Nodes in a GTG represent individual tasks and edges represent communication between tasks. Tasks are usually concurrent and communicate among themselves to exchange data and perform the functionality of the system. In this paper, we model the communication between the tasks and analyze their impact on the system performance. An interesting and novel aspect of this work is that the communication is modeled using channels nodes in the task graph along with canonical tokens that are user and channel specific.

We explicitly expose the token delivery delay by using channel nodes in the task graph. We introduce the notion of a canonical token for an equivalence class of channels and describe a metric that allows designer to pick implementations for shared channels that satisfy the system's timing requirements. Further, we propose a refinement methodology that attempts to provide us with better bounds on the delays by using the notion of a process timing simulation of the entire system. In this simulation model, the timing behavior of the individual tasks is the main focus, and the functionality unimportant.

1 Introduction
An embedded system (the system for short) is typically reactive and real-time in nature: it continually reacts to the stimuli coming from its environment and performs this interaction under strict timing constraints. The timing constraints are called the system's external timing constraints.

In a typical embedded system design flow, the requirement specification phase describes what the system's external behavior must be without describing how the systems works internally. The latter is expressed in the architectural design phase by means of a task graph, which describes a decomposition of the system into manageable components or tasks. These tasks are usually concurrent.

From a requirement specification point of view, the system is temporally correct if it satisfies its external constraints. Unfortunately, the problem of designing a temporally correct system is a difficult one, and current practice of this problem is ad hoc; based on trial and error guided by previous engineering experience [7, 9]. Moreover, the emphasis is usually on designing a functionally correct system and the temporal correctness of the system is usually checked after the system's components are integrated and the combined system's functional correctness is ensured. This approach has a huge drawback: the system's integration stage comes very late in the design cycle, and correcting any timing problems detected at this stage becomes very expensive. The expense, in terms of time and cost, to correct problems is large since a lot of design commitment has already been made.

1.1 Our earlier work
In our earlier work, we have proposed a solution to the problem of designing a temporally correct system up front. The idea is to provide time budgets on the system's tasks very early in the design flow in such a way that any violation of individual time budgets lead to a violation of the system's external constraints. These time budgets are also called the system's internal timing constraints. We derive these time budgets from the system's task structure and its input rate constraints using an algorithm called "the rate derivation algorithm". We then use the derived time budgets to validate the system's external timing constraints and the design choices at every stage of the design flow. Our solution reduces the complexity of ensuring temporal correctness from system level to the task level. This enables the designers to have a simultaneous control on the system's functional and temporal correctness at the early stages in the design flow. Further, we can apply the same techniques on the design of each task itself.

Figure 1 illustrates the above discussion. The system's task structure is shown using a graph where each circle represents a task and each arrow represents a task interaction. The system's external constraints are: response time constraints on the system and the environment (labeled (1) in the figure), input jitter constraint (labeled (2) in the figure), output jitter constraints (labeled (3) in the figure), the input rate constraints (labeled (4) in the figure) and output rate constraints (labeled (5) in the figure).

The internal constraints or the time budgets on the
tasks are: latency bounds (labeled (8)), the input (labeled (7)) and output (labeled (9)) jitter bounds and the rate bound (labeled (10)). Here the jitter constraint or bound defines the time separation between two different events, whereas a rate constraint or bound defines the time separation between two successive events of the same kind. An event can be a stimulus arrival, a response generation, or a task starting its execution.

We model an embedded system using a very general task graph, briefly explained in Section 2, and then use the rate derivation algorithms [17, 4] RADHA and RATAN, to derive a rate bound for each task assuming that all the input rate constraints are known. We use these bounds to derive other internal constraints and validate external constraints.

1.2 Current contributions

In this paper, we model the communication between the tasks and analyze their impact on the system performance. An interesting and novel aspect of this work is that the communication is modeled using channel nodes in the task graph along with canonical tokens that are user and channel specific. In our previous work, we have generated time budgets on tasks with the assumption that the tokens are user specific and channel specific. This implies that different channels can have different kinds of tokens being passed between tasks. Channels transport the same token, but the semantics of tokens could differ across channels. We assumed that once the tokens were fixed, they were indistinguishable within and across channels. Furthermore, we also assumed that token delivery time was a part of the time budget for a particular task.

In this paper, we explicitly expose the token delivery delay; we use *channel nodes* in the generalized task graph to represent channels with delays. We show that our model is powerful enough to handle the sharing of these communication channels among various tasks. Our analysis techniques allow us to analyze the timing of channel nodes as well. However, since we know the functionality of the channel nodes, it is possible to tighten the timing bounds obtained by our analysis algorithms for channel nodes. We show how this can be done, both analytically and using simulation. Consequently, we introduce the notion of a process timing simulation of the entire system where the timing behavior of the individual tasks is the main focus, and the functionality unimportant. A process timing simulation of the entire system enables us to fine-tune the timing requirements of the system, especially that of the nodes representing channels, thereby obtaining better bounds on the timing of the system. In addition, we introduce the notion of a canonical token using which we can assist the system designer in picking a suitable implementation for a channel. This issue has great impact on the design of the system since tokens and channels bind the entire system together. Using our scheme, we can model complex communication schemes in embedded systems at a *very high-level*, where communication can range from bit transfers to transfer of extensive (and voluminous) data sets: these are modeled uniformly and incorporated in to the timing analysis. We illustrate these ideas using the dashboard controller [1] as an example.

1.3 Related Earlier Work

There has been a lot of excellent earlier work on modeling communications systems [6, 21, 23, 11]. There are several tools, like MISTREL [22], PTOLEMY [19], PHIDEO [24], PROPHEID [15, 16] that include communication specification and sharing of channels among various tasks. However, most of these tools do not address the problem at the level of abstraction that we do in this paper. Also, some of these tools, like PROPHEID and PHIDEO are specific to the digital signal processing domain, whereas our work is not. Though both MISTRAL and PHIDEO can handle large degrees of multiplexing in their channel representation, they need specific details of the channels in order to do so. In our channel representation, we do not specify the implementation of the channel till the user is ready to do so. We perform our analysis and simulation with no assumption on the implementation details. In addition, none of the previous works tie a simulation model to their analytic framework to generate a quick prototype of the system at a very high level.

1.4 Paper organization

Section 2 provides an overview of the generalized task graph model as well as discusses our earlier work on tools RADHA and RATAN. Section 3 explains how we model channel delays by using a channel node to represent the edges along which there are delays. It also explains the problems that arise when channels are shared among various tasks and provides solutions to these problems. Section 3 also introduces the notion of a canonical token for an equivalence class of channels. Section 4 introduces the dashboard controller example [1] and explains the concepts of channel nodes and canonical tokens for this example. Section 5 explains the notion of process timing simulation and uses the dashboard controller as an example. Finally, Section 6 has the conclusions.

2 Generalized Task Graph Model

During the early stages of an embedded design flow or at stages where higher-levels of abstraction are con-
sidered, the system tasks have coarser granularity. As a result, at higher-levels of abstraction, we have a task graph rather than an event graph. This section presents our task graph model. Notice that in this presentation, we expose the channel delay for each communicating task.

Our task graph model is called the generalized task graph (GTG) model. It builds upon many previous models in the literature such as [2, 8, 9, 10, 12, 13, 14, 18, 20, 25]. Our model is an abstraction and combination of these models for the purposes of timing analysis. For example, we borrow both AND and OR causality types from [10], both acyclic and cyclic task dependencies from [2, 13, 14, 18], modeling of both control and data from [9, 20, 25], and skipped and unskipped behaviors from [8]. We also kept our model as similar to these models as possible so that we can use the techniques developed for them as well as we can refer the readers to them for the concepts we borrowed from them.

We now define the generalized task graph model. A GTG $G = (V, E)$ is a directed graph in which each node in $V$ is a task and each arc in $E$ is a channel connecting a producer (task) to a consumer (task). We will refer to $G$ as the GTG in the sequel. Each task is either a data task that processes data or a control task that controls the data tasks and/or the other control tasks. The granularity of each task is large, e.g., a control task is an entire finite state machine. The set of data tasks can be thought of as the system’s data path that is controlled by its control path, the set of its control tasks. We assume that tasks are concurrent, and channels are asynchronous and bounded. Channels carry data and control items called tokens. Token granularity is channel specific, and once fixed, it is identical within a single channel and across different channels.

In the GTG, the tasks without any predecessors are called the input tasks, and those without any successors are called the output tasks. The input tasks and output tasks correspond to the sensors and actuators of the system, respectively. Tokens into the system are introduced by the input tasks.

Interactions within almost any embedded system can be modeled by using enables, disables, and triggers [9]. An enabled task runs until it is disabled. A triggered task runs and terminates by itself. We are concerned with the start time of each task; therefore, we use enables to mean both enables and triggers, and do not consider disables at all because we assume that only enabled tasks can be disabled. As a result, each arc in the GTG is for an enable. In this paper, we do not consider additional relations on the tasks in the GTG such as mutual exclusion or priority.

Task interactions in the GTG are causality based as in an event graph. Thus, corresponding to AND causality and OR causality, we have AND and OR tasks, respectively. However, these task types need further refinement because of the presence of tokens. A task is called an unskipped task if it has to consume every token produced for it by its producers; it is a skipped task otherwise. Both of these concepts apply to both AND and OR tasks such that we now have AND/unskipped (AND/u), AND/skipped (AND/s), OR/unskipped (OR/u), and OR/skipped (OR/s) tasks. In the GTG, skipped behavior is not obtained by making channels lossy; skipped behavior is intentional in that a skipped consumer knows that it will lose some tokens from its input channels because it cannot keep up with the rate of its every predecessor. A task is called a disjoint task if it ignores its input tokens for the current execution only. We apply this concept only to the OR/skipped task only to obtain and OR/skipped/disjoint (OR/s/d). The rate analysis for an OR/skipped-disjoint task when it reads in one token and outputs one token is one contribution of this paper.

The properties of tasks and channels are defined as follows. Consider Figure 2. In this figure, we have a channel $(p, c)$ from producer $p$ to consumer $c$. Each task, say $c$, has an integer period $T(c)$, a rate $r(c)$, and an integer start time $t(c)$. Producer $p$ puts $P(p, c)$ tokens per execution into channel $(p, c)$, and consumer $c$ gets $C(p, c)$ tokens per execution from that channel. These numbers are called the token numbers of the channel, and are integers. The token numbers need not be constants, e.g., when we fix the period of the consumer but not that of the producer, or known in advance because if they are unknown, we derive them too. Channel $(p, c)$ has an integer, bounded delay of $d(p, c) \in [d(p, c), D(p, c)]$ such that it takes $\delta(p, c)$ time units for the tokens of producer $p$ to enable consumer $c$.

Without loss of generality, we use intervals for the period, rate, and start time of each task. Let $T(c) = [T_1(c), T_2(c)]$ and $r(c) = [r_1(c), r_2(c)]$ be the period and rate intervals of task $c$, respectively. Then, the basic relation between the period and rate is $r(c) = 1/T(c)$, which translates to $r_1(c) = 1/T_2(c)$ and $r_2(c) = 1/T_1(c)$. Due to this basic relation, we will use the terms period and rate interchangeably. We also have $0 < T_1(c) \leq T_2(c) \leq \infty$ so that $0 \leq r_1(c) \leq r_2(c) < \infty$. Note that we abuse the notation when we write a bounded interval for the period even though the upper bound can be infinity. This notational convenience is used for every interval in the sequel.

With respect to channel $(p, c)$, there must be at least $C(p, c)$ tokens in that channel for consumer $c$ to get enabled. Suppose $p$ executes $i$ times till the time $c$ gets enabled for the $j$th time. Then, since we must have $i \times P - (j - 1) \times C \geq C$, we get

$$j \times \left[ \frac{C(p, c)}{P(p, c)} \right] \geq i \geq \left[ j \times \frac{C(p, c)}{P(p, c)} \right]$$

which implies that the start times of these two tasks are related to each other as

$$t_p(U(j, p, c)) + \delta(p, c) \geq t_p(L(j, p, c)) + \delta(p, c),$$

where $U(j, p, c) = j \times [C(p, c)/P(p, c)]$ and $L(j, p, c) = j \times [C(p, c)/P(p, c)]$.

Using Equation 1, the definitions of AND and OR causality types, and the bounds on the channel delay, we arrive at the following start time expressions for AND and OR tasks: The $j$th start time of an AND task $c$ is governed by

$$\max_{(p, c) \in E} \{ t_p(U(j, p, c)) + D(p, c) \} \geq t(j)$$

and the $j$th start time of an OR task $c$ is governed by

$$\min_{(p, c) \in E} \{ t_p(U(j, p, c)) + D(p, c) \} \geq t(j)$$
\[ \geq \max_{(p,c) \in E} \{ t_p(L(j,p,c)) + d(p,c) \}, \]

whereas that of an OR task \( c \) (except an OR/s/d task) is governed by

\[ \min_{(p,c) \in E} \{ t_p(U(j,p,c)) + D(p,c) \} \geq t_c(j) \]
\[ \geq \min_{(p,c) \in E} \{ t_p(L(j,p,c)) + d(p,c) \}, \]

whereas that of an OR/skipped/disjoint task \( c \) is governed by

\[ \min_{(p,c) \in E} \{ t_p(U(j,p,c)) + D(p,c) \} - \epsilon_u \geq t_c(j) \]
\[ \geq \min_{(p,c) \in E} \{ t_p(L(j,p,c)) + d(p,c) \} - \epsilon_l, \]

where tasks \( p \) are predecessors of task \( c \) in the GTG, and \( U(j,p,c) \) and \( L(j,p,c) \) as defined previously. \( \epsilon_u \) is a ratio of \( \prod_{(p,c) \in E} \min(p,c) \in E \{ t_p(U(j,p,c)) \} \) and \( \sum_{(p,c) \in E} t_p(U(j,p,c)) \cdot \epsilon_l \) can be obtained similarly.

The input behavior of the task types is as follows: An AND/u task \( c \) reads \( C(p,c) \) tokens from each one of its predecessor \( p \) whereas an AND/s task \( c \) reads \( C(p,c) \) tokens from its slowest predecessor \( p \). Any AND task \( c \) has to wait for their slowest predecessor \( p \) to accumulate at least \( C(p,c) \) tokens for them. An OR/u task \( c \) reads \( C(p,c) \) tokens from each one of its predecessor \( p \) whereas an OR/s task \( c \) reads \( C(p,c) \) tokens from at least one of its predecessors, usually the fastest one. Any OR task \( c \) can start executing as soon as one of their predecessors \( p \) has sent \( C(p,c) \) tokens for it. Note that an OR/u task is like an AND/u task in that both of them wait for all their predecessors to send enough tokens to them but an OR/u task can start executing with the arrival of enough tokens from only one of its predecessors whereas an AND/u task has to ensure that its every predecessor has sent enough tokens.

The output behavior of a task in the GTG is determined by its successors. Suppose a channel has \( P(p,c) \) as a token number. Then, task \( p \) has to send at most \( P(p,c) \) tokens to its successor \( c \). If \( c \) is an AND task, we expect \( p \) to do so every time it executes whereas if \( c \) is an OR task, \( p \) may not even send any tokens to \( c \) (but we expect at least one predecessor \( c \) will send it some tokens). In the above derivations of the expressions, we assumed that \( p \) sends exactly \( P(p,c) \) every execution. The reason is due to the possible behavior of \( p \). If it is possible that \( p \) can produce \( P(p,c) \) over channel \( (p,c) \), then we assume that it does so in order to validate the timing performance of the system conservatively. If the designers know that \( p \) produces \( P(p,c) \) every \( K \) executions, then we can easily take \( K \) into account in all our derivations.

The GTG model also has hierarchy. The hierarchy in a GTG consists of two levels: top level and bottom level. The bottom level contains all the strongly connected components (SCCs) of the GTG, including those with a single node. The top level corresponds to the component graph of the original GTG in that some nodes, called supernodes, represent the SCCs at the bottom level. Note that the bottom level contains cyclic portions of the GTG whereas the top level is acyclic. This is actually the reason for introducing hierarchy because the algorithms for cyclic and acyclic task structures are generally different.

Rate derivation refers to deriving the rate or rate bounds (in terms of rate intervals) of all the tasks in the task graph [3, 5, 4]. We use the task rates to derive the task periods since the task periods are the ones that are mostly used for further derivation and validation. Details of RADHA can be found in [4]. It suffices to say that we use RADHA for the upper level of the GTG (acyclic part) and another tool called RATAN for the cyclic part.

We assume that the task periods are real numbers, since if two predecessors simultaneously try to enable the task, only one will be successful, i.e., we cannot have a zero time separation. In this model, we have assumed that the token delivery delay for enabling tokens and data tokens are accounted for in the latency of the producer.

The main contribution of this paper is the introduction of a second refinement pass that works on the results of the previous iteration to explicitly expose the token delivery delays by creating new nodes in the GTG called channel nodes. We show that our model is powerful enough to model the sharing of channels among various tasks. We then introduce the notion of a canonical token for an equivalence class of channels. Canonical tokens allow us to describe a measure on the communication channel using which the designer can pick an implementation that fits the system’s timing requirements. We explain this second refinement phase with the help of the design of the dashboard controller as an example.

### 3 Modeling Channel Delays Using Nodes and Canonical Tokens

Consider a generalized task graph \( G \), for the design of an embedded system. The edges between tasks represent the communication channels between the tasks and an edge weight represents the number of user specified tokens required to enable one execution of the task at the sink. Previously, we had two attributes (weights) for a given channel: the number of tokens consumed, \( C \), and produced, \( P \), we add the channel delay as another attribute to the edge. In order to represent the channel delay, we add a node to the task graph called the channel node that represents the channel and its implementation. Now the channel delay of the edges is encapsulated in the channel node. In order to share
channel implementation we describe an equivalence relation \( \mathcal{R} \) on the channels (edges) in the system (task graph). We then introduce a canonical token for that specific class of channels. The designer has to specify the canonical token for an entire equivalence class of channels. Typically, the canonical token is an indivisible entity that can be transported by a channel node another. Once a canonical token is chosen, it is fixed and all other tokens for that particular class of channels are expressed as multiples of the canonical token. We then describe a metric called the canonical token rate interval that helps the designer pick an implementation for the channel that would meet the system timing constraints.

3.1 Channel Nodes

Let \( a \) and \( b \) be two tasks in \( G \). Let \( e(a,b) \) be an edge from task \( a \) to task \( b \). This edge represents a communication channel between tasks \( a \) and \( b \). Let task \( a \) produce \( P \) tokens and let task \( b \) consume \( C \) tokens. Let \( T_a \) and \( T_b \) be the latencies of tasks \( a \) and \( b \) respectively.

We model the channel delays in the generalized task graph model by using an intermediate node called the channel node in the task graph to represent the delay of the channel connecting two nodes. Using this approach, we make sure that edges have no delay associated with them; the delays being associated with the newly created channel node. This approach does nicely for modeling delays on single non-shared channels. However, it has two consequences: (1) The number of nodes in the graph increases and (2) sharing of channels is not simple; it introduces a level of hierarchy in the task graph.

The increase of the number of nodes in the graph makes the graph more complex and not easy to understand. In typical embedded system design, it is recommended [9] that there be no more than ten nodes in the graph. This is a good recommendation since the purpose of our endeavor is to keep the design simple and at a high level. However, since we are proceeding with a refinement of our original design, this increase in complexity in the task graph is acceptable.

The sharing of channels by various tasks introduces a level of hierarchy in the task graph model. Consider the graph in Figure 3(a1), if we wish to model the delays along edges \( e(a,b) \), we get the graph in Figure 3(a2). The newly created nodes accept one token from \( a \) and passes it to node \( b \), hence its rate is the same as the rate of \( a \), as shown in Figure 3(a2). We can apply RADHA to perform rate analysis on the system, with this transformation on the task graph.

Now, if we wish to share the implementation of channels \( e(a,b) \) and \( e(e,t) \), two arbitrary non-adjacent edges in the task graph of Figure 3(b1), we get the graph in Figure 3(b2). The newly create node \( q \) is an OR/skipped/disjoint node according to our node classification. Figure 3(b2) shows the transformed graph. In this case too, we can apply RADHA to perform rate analysis on the system.

Now, consider the case where we wish to share channels \( e(a,b) \) and \( e(e,c) \) of graph in Figure 3(c1), we introduce a new channel node \( q \) and as shown in Figure 3(c2), we get a loop between edges \( e(b,q) \) and \( e(q,b) \). In this state, we cannot directly apply RADHA to the graph to compute the rates of the tasks. We must transform \( G \) into strongly connected components (SCC) and represent each SCC as a super-node in the graph \( G \). Figure 3(c3) shows nodes \( g \) and \( q \) in one super-node represented at the higher level of hierarchy by the node \( gq \). Once we have created an additional level of hierarchy, we can now apply RADHA to provide us with the necessary timing validation at the upper level. We use RATAN on the SCC (lower level) to get the timing constraints that have to be met there. The constraints derived at the higher level drive the constraints obtained for the SCC. Figure 3(c3) also shows the rate analysis performed by RADHA after we have encapsulated the hierarchy into a super-node. Figure 3(c3) also shows how the rate analysis at the higher level of hierarchy imposes a rate constraint on the lower level of the hierarchy.

RADHA provides us with bounds on the latency of the channel node \( q \) that are a little pessimistic. Con-

---

**Figure 3: Rate Analysis for Channels.** (A) Simple case - no sharing. (B) Sharing non-adjacent channels. (C) Sharing of two adjacent channels create a hierarchy and the constraints on the lower level (SCC) in the hierarchy.
sider, the channel node of Figure 3(b2). Assuming that one token is produced by the sources of an OR-node, i.e., a and b, and both tokens are consumed by sinks b and t, RADHA will tell us that the period interval of the channel node q is bounded below by:

$$\min_{(p,q) \in E} \{ t_p(j + D(p,q)) \} - \epsilon_u \geq t_q(j)$$

$$\geq \min_{(p,q) \in E} \{ t_p(j) + d(p,q) \} - \epsilon_c,$$

where tasks p are the predecessors of task q in the edge set E of the GTG and \( \epsilon \) is on the minimum latency of all incoming edges of node q.

However, for the channel OR/disjoint this period interval is implementation dependent, since we know that q in this case has to process all the incoming tokens, i.e., deliver them to the appropriate output channels even if they all arrived one after the other. As a result, if the rates of the predecessors of q were all the same and their arrival time overlapped, we would have to store the incoming tokens in some buffer. As a result, a pipeline implementation of token delivery within the channel node will have a different rate equation than a non-pipelined one. It is for this reason that we propose the process timing simulation model, where we can explore the implementation design space for channel nodes.

The results from RADHA are not in error, they are merely pessimistic since RADHA does not know the functionality of the OR-tasks. However, in this case, the channel nodes are special and their functionality is clearly understood: they must pass incoming tokens from their source to their sink within the specified delays in order to meet the system's response time. As a result, we can get tighter bounds on the delays that can be tolerated by the system.

We will show in Section 5 that we need to perform a timing simulation to ascertain that the lower bound on the rate interval for a channel node. Each newly introduced channel node has its own internal latency when an implementation is chosen. By deriving the latency bounds for the channel nodes depending on its predecessors, we can validate the implementation choice.

We will show in Section 5 how we can explore the design space using the timing simulation model.

3.2 Canonical Tokens

In order to generalize the approach taken to identify channels that can be shared, we describe an equivalence relation \( \mathcal{R} \) on all the channels (edges) of \( G \). This equivalence relation is user specific. Two channels may lie in the same equivalence class because they transport a similar number of tokens or they share a similar implementation or for any other reason the user deems fit. In order to get a handle on the implementation aspects of the shared channels, we introduce the notion of a canonical token for each equivalence class. For each equivalence class \( \mathcal{R} \) induced by \( \mathcal{R} \), we define a canonical token \( t_R \). This canonical token is also user specific.

Further, we define the delay of a canonical token \( t_R \) in a channel \( \epsilon(a,b) \), as the time it takes for \( t_R \) to travel from task a to task b along channel \( \epsilon(a,b) \). Now, we represent both \( C_t \), the number of tokens consumed, and \( P_t \), the number of tokens produced, as integral multiples of the canonical tokens for the channel \( \epsilon(a,b) \). Since we model edges in the GTG using a channel node, the canonical token delay now applies to the channel nodes.

Consequently, channel delays are dependent on the delay of the canonical token for that specific class of channels and the number of canonical tokens that travel through the channel. We know that some multiple of the canonical tokens will be passed from task a to task b for one execution of task b. Each canonical token has a specific delay and this delay is dependent on the implementation of the channel. This delay is not an absolute delay: it is an amortized delay, in the sense, that we expect a certain number of tokens to be communicated within a specified deadline between the two tasks. Consequently, for a pipelined implementation of the channel between tasks a and b, the delay of the canonical token will be different from the non-pipelined implementation.

3.3 Applications

After having performed rate analysis (using the tools RADHA and RATAN) on the system, the designer has the time budgets for each task in the system, including the nodes that represent channels. At this stage, the designer can make certain implementation decisions using the canonical tokens and the period interval of nodes that represent channels. The designer knows the total number of canonical tokens that pass through a given channel node and the period interval of the channel node, and can therefore compute the period interval during which one canonical token must be processed by an implementation of the channel node. We call this interval the canonical token processing interval (CTPI). The designer can now classify various implementation of the channel node by their canonical token processing interval and therefore eliminate implementations that do not satisfy the CTPI for a given channel node.

Further, the designer can compute the critical path of the system. This path tells the designer the path through the task graph that controls the timing of the entire system. From this path, the designer can determine implementations for a particular equivalence class of channels that would not satisfy the timing constraints of the entire system. The designer can also use the slack times in the task graph to model a slower, and maybe cost effective, implementation of communication along the non-critical paths in order to reduce the net cost of the system. The notion of critical path computation [1] is not new, however, to the best of our knowledge, it has not been done at the task graph level.

An important fact to note here is that at this stage, we have no implementation in mind for any of the tasks in the task graph. We have only modeled the communication between tasks and allowed sharing of communication channels. Using this high-level representation and abstraction, we have been able to characterize communication channels and analyze their timing requirements. The next step is to develop specific implementations that meet the timing requirements of the system.

4 The dashboard controller example

In Figure 4, we present the task graph of the dashboard controller, which is taken from [1]. The functionality of these tasks dictates that task h (the LCD dis-
play driver) is OR/skipped/disjoint. The other tasks are all AND/unskipped tasks. The speedometer (task \(d\)) registers vehicle speed in the range of 0-260 kmph, where any speed value less than 5 kmph is regarded as zero. The odometers (tasks \(i\) and \(j\)) register distance traveled at increments of 0.1 km starting from 0 km. The trip odometer can sometimes be reset by the driver. The dashboard controller gets four pulses from them in the figure. Task \(a\) generates a pulse for every 1/4 rotation, i.e., a pulse when the vehicle travels 0.66/4 m. As the vehicle’s speed is between 5 kmph and 260 kmph, the smallest and the largest time separation between two consecutive pulses are (0.66/4)/260 and (0.66/4)/5, respectively. This corresponds to a period interval of \(T_a = [2.28, 118.80] \text{ ms}\) for task \(a\). Note that task \(a\) resides in the system’s environment i.e., it is an input task and it drives the entire system, so its rate constraint (an input rate constraint of the system) is known to hold by the rate derivation algorithm.

The rate constraints for the other tasks are computed as follows: Task \(d\) (the speedometer) consists of two coils and a magnetized needle. The speedometer should get its inputs at a rate of at least 100 Hz to drive these coils [1]. This means that \(T_d \leq 10 \text{ ms}\). Tasks \(i\) and \(j\) are the odometers that register the increments of 0.1 km. If the vehicle travels at a speed of \(r\) kmph, it takes \(0.1/r\) h for it to travel 0.1 km. So it takes at least \(0.1/260 = 1.38 \text{ s}\) and at most \(0.1/5 = 72 \text{ s}\). Naturally, it should not take much longer time for the odometers to register this increment. Consequently, both odometers have the same timing behavior, and their period interval is \(T_i = T_j = [1.38, 72] \text{ s}\).

This represents the first pass we mentioned earlier, using our rate analysis tool. In this analysis, we have not assumed any channel delays, incorporating them into the period of the task itself. In the next section, we present the second refinement pass for the dashboard controller design where we extract the channel delay from the tasks into intermediate channel nodes in the task graph and use canonical tokens to get CTPI for various channel nodes.

### 4.1 Modeling Channels for the Dashboard Controller

In this section, we will consider the case when we try to share the implementation of channels \((c, d)\) and \((e, f)\). A new node \(q\) is used to represent the two channels. Notice that by exposing the channel delays in intermediate channel nodes, we have removed the timing constraints of the particular task from the problem of sharing the standard bus among two different tasks.

We can attempt to analyze the impact of sharing some standard bus between tasks \(c\) and \(d\) and tasks \(f\) and \(h\) by using RADHA to analyze the timing of the system. We can also simulate a model of some standard bus by plugging it into our system simulation model or generate test-bench data for the bus model from our timing simulation model. The canonical tokens and CTPI allow us to analyze various implementations of channels that will meet the system requirements. The next section discusses some methods of modeling the timing of the channels using a hardware description language (HDL) by generating HDL code from our task graph representation.

### 5 Process Timing Simulation

In this section, we present an overall view of the dashboard controller by using channel nodes and defining canonical token processing interval (CTPI) for shared channel nodes.

The rate derivation algorithm derives a time budget for each task in the task graph. Hence, we know the timing constraints that each task should obey. Also, from the task graph, we know the input/output characteristics of each task in terms of the token usage. These two can be combined to define a high-level process timing model for the system. This timing model can be simulated to evaluate its performance characteristics, which we call process timing simulation. Timing simulation can be used to gain insight into the timing behavior of the system as well as validate many timing tradeoffs. It is a method to quickly develop a prototype of the system at the very early stages in the design cycle.

Process timing simulation is different from functional simulation: functional simulation assumes that the internals details of tasks are known, whereas timing simulation does not.

### 5.1 Transformation to an HDL

We now describe how we can transform a typical task into any hardware description language for timing simulation. Recall that a task needs enabling tokens to get enabled. These tokens are generated by the task’s predecessors. For example, only one token sent from task \(b\) to task \(c\) is enough to enable the latter task. However, we do not know how many tokens task \(b\) needs to get enabled, but we know that it should get enabled
every 250 ms. We can achieve this behavior by using an internal timer for task c (dock, in HDL).

Now, consider task c. We know that it gets enabled by one token from task b and generates 25 tokens for task d, one every 10 ms. However, we do not know how task c filters speed: this could be through some complicated algorithm, which we do not care about. We model this algorithm with no-ops. Our model merely waits for the specific time instance and generates the required communication tokens.

From this discussion, it is clear that we can perform a process timing simulation of the system since we have the time budgets and know the behavior of the communication between tasks. Due to space considerations, we will not go into the details of generating Verilog code from our task graph representations.

5.2 Scenarios for process timing simulation of channels

We can generate behavioral level description for a task without knowing its internal operations. In addition to the task, we can also generate behavioral descriptions for each channel node in the system. The behavior of each channel is quite simple: it receives a token from its predecessor, waits (simulating the channel delay) and delivers its token to its successor. If the channel is being shared by various tasks, it must make sure that it delivers the token to its rightful owner.

For the dashboard controller, we generated Verilog code for each task and decided to model the delay of channel c(b, c) using node q. We must validate that token delivery delay does not violate the timing constraints of the speedometer. Since the sets of tasks a, b, c, and d are pipelined, we must ensure that the delivery of tokens is done in a timely manner and that tokens are not lost. We choose various delay values for channel c(b, c): 1 µsecs, 10 µsecs, 100 µsecs, 1 ms and 11 ms. For each delay value, we found that the speedometer’s response time decreased. We lost no tokens, except in the case when the delivery delay was 11 ms. This was obvious since the rate of the speedometer is 10 ms and we are now delaying tokens in the channel for too long.

In another scenario, we decided to share the channels c(c, d) and d(f, h) using a node q. We choose various delivery delays for node q: 1 µsecs, 10 µsecs, 100 µsecs, 1 ms and 11 ms. For each delay value the response time of the speedometer decreases which was expected. When the delay was 11 ms, we experienced a loss of tokens to the speedometer which was expected since at low speeds the latency of node c is in the interval [4.56, 237.6] ms and it produces 24 tokens to ensure that task d’s rate is 10 ms. Also, the distance traveled by the trip odometer and the lifetime odometer began to differ by small increments: this was expected since we were taking different times to deliver tokens to h from nodes f and g. Analytically, the rate interval for node q is [0.189 + d, 8.989 + d] milliseconds when the channel delay is d ms. The process timing simulation tells us that we can tighten the delay bound for the channel to only 10 milliseconds. At tighter lower bounds, we begin to loose tokens at the speedometer.

Further, we identified a canonical token for node q to be a byte (we could have chose an integer, if we wanted to), as a result, we get a CTPI for node q to be [0.0474, 2.475] bytes per millisecond since we are transporting an integer token across channels c(c, d) and d(f, h). This is the worst case CTPI, since we have chosen the interval with the delay of node q to be 10 ms, the least it can be without token loss. Any implementation that fails within these timing requirements will satisfy the systems’ timing constraints.

Notice that we have reduced the problem of integrating the implementation of the shared channel node into a detailed simulation of the entire system to using the timing simulation to interact as the test bench of the implementation. In this way, we can hope to simulate typical scenarios for the system at the behavioral level and achieve good simulation performance.

6 Conclusions

We have shown how we can abstract the implementation of the communication delays using intermediate channel nodes and canonical tokens. We have also shown that we can categorize channels into an equivalence class in order to share their implementation. Channel delays in each equivalence class can be modeled using a channel node in the graph and delay of the canonical token. Our work attempts to characterize the scenarios that arise from sharing communication channels among one implementation model. We believe that from this work, we can generate behavioral models for communication protocols that can encapsulate communication in a complex embedded real-time system at the same time deliver the performance required by the system. The metric CTPI enables the designer to pick the implementation for channels that satisfy the system’s timing constraints. We believe that the designer can iterate using the analysis tools (RADHA and RATAN) and the process timing simulation for channels to converge to functional time budgets for all the tasks and channels in the system.

References


