A Timing-Driven Design and Validation Methodology for Embedded Real-Time Systems

ALI DASDAN
University of Illinois at Urbana-Champaign
DINESH RAMANATHAN
Synopsys, Inc.
and
RAJESH K. GUPTA
University of California, Irvine

We address the problem of timing constraint derivation and validation for reactive and real-time embedded systems. We assume that such a system is structured into its tasks, and the structure is modeled using a task graph. Our solution uses the timing behavior committed by the environment to the system first to derive the timing constraints on the system's internal behavior and then use them to derive and validate the timing constraints on the system's external behavior. Our solution consists of the following contributions: a generalized task graph model, a comprehensive classification of timing constraints, algorithms for derivation and validation of timing constraints of the system modeled in the generalized task graph model, a codesign methodology that combines the model and the algorithms, and the implementation of this methodology in a tool called RADHA-RATAN. The main advantages of our solution are that it simplifies the problem of ensuring timing correctness of the system by reducing the complexity of the problem from system level to task level, and that it makes the codesign methodology timing-driven in that our solution makes it possible to maintain a handle on the system's timing correctness from very early stages in the system's design flow.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*systems specification methodology*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*real-time and embedded systems*; C.4 [**Computer Systems Organization**]: Performance of Systems—*modeling techniques, performance attributes*; D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems and embedded systems*; D.4.8 [**Operating Systems**]: Performance—*modeling and prediction*; J.6 [**Computer Appli-**

This work is partially supported by National Science Foundation (NSF) CAREER award MIP 95-01615, NSF/DARPA ASC-96-34947, DARPA DABT63-98-C-0045, an Interstate Electronics Corporation Fellowship, and a DAC Design Automation Graduate Scholarship.

Authors' addresses: A. Dasdan, Department of Computer Science, University of Illinois at Urbana-Champaign, IL 61801; D. Ramanathan, Advanced Technology Group, Synopsys, Inc., Mountain View, CA 94043; R. K. Gupta, Department of Information and Computer Science, University of California, Irvine, CA 92697.

cations]: Computer-aided engineering—*computer-aided design (CAD)*

General Terms: Design, Performance, Verification

Additional Key Words and Phrases: Performance verification, period assignment, period derivation, rate assignment, rate derivation, requirements analysis, system-level design, timing analysis, timing-driven codesign

1. INTRODUCTION

An embedded system (the system) is typically reactive and real-time in nature because it continuously has to react to the stimuli coming from its environment and it also has to perform this interaction under strict timing constraints. These timing constraints are called the system's *external timing constraints* as they are imposed on the system's external behavior.

In a typical embedded system design flow, the requirements specification phase describes what the system's external behavior must be without describing how the system 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*. Since the system has many activities occurring in parallel, these tasks are usually concurrent. Viewing them as being potentially concurrent ensures the maximum flexibility for the designers, especially during the early phases of the design flow [Gomaa 1993].

The system's external timing constraints can be classified into two groups: those on the system and those on the environment. The former are called the *required timing constraints* or the *timing requirements* because the system is required to meet them, whereas the latter are called the *given timing constraints* because the environment rather than the system is required to meet them, that is, they are committed to the system.

From a requirements specification point of view, the system is temporally correct if it satisfies its required timing constraints. Unfortunately, the problem of designing a temporally correct system is a difficult one, and the current practice for this problem is ad hoc; it is based on trial and error, guided by engineering experience [Gerber et al. 1995]. Moreover, the emphasis is usually on designing a functionally correct system [Gomaa 1993]. The temporal correctness of the system is usually checked after the components of the system are integrated and the functional correctness of the resulting system is ensured. The biggest drawback of this approach is that as the system integration stage comes very late in the design flow, correcting any timing problems detected at this stage is very expensive in terms of time and cost because of the considerable amount of commitment that has already been made.

One important observation here is that the designers, on the one hand, know that the system's required timing constraints are imposed on the entire system but during the early stages of the design flow, the designers do not have a grasp of the entire system since the system is going to be designed task by task. On the other hand, they know that the temporal correctness must be taken into account from the early stages onwards in order to eliminate the aforementioned drawback of the current practice. Thus, the main issue is to ensure that the designers can maintain a handle on the system's temporal correctness from very early stages onwards even though they do not have the entire system ready. An effective solution to this issue is to provide time budgets on the system's tasks very early in the design flow in such a way that any violation of these individual time budgets leads to a violation of the system's timing requirements. Effectiveness of this solution comes from the fact that it not only helps in the process of ensuring the system's temporal correctness but it also helps reduce the complexity of this process from system level to task level. This complexity reduction in turn reduces the cost of the process as well as the time to market the system. We will refer to these time budgets as the system's *internal timing constraints*. This paper is about the details of our proposed solution.

We propose a solution to the problem of deriving and validating the timing constraints of an embedded system. Our solution contains the following three main contributions: (1) A general task graph model called the generalized task graph model and a comprehensive classification of timing constraints. Our task graph model builds upon earlier task graph models by combining and extending all those features of these earlier models that are necessary for timing analysis. Our model has both AND causality and OR causality, both acyclic and cyclic task dependences, and both skipped and unskipped task behaviors. (2) Algorithms to derive the system's internal timing constraints. Since any timing constraint is either a rate constraint or a separation constraint, we propose new algorithms as well as adaptation of the existing algorithms in the literature to derive both rate and separation constraints. The basis of our derivation is the rate constraints for the tasks, which is why we will refer to the derivation as the *rate derivation*. Note that a rate constraint bounds the rate of a task, the rate of a task being its frequency of execution. (3) Algorithms to validate the system's required timing constraints. We show how to use the derived timing constraints to validate the system's timing requirements.

Our solution is implemented in a tool called RADHA-RATAN, and is given within the framework of a codesign methodology. This tool combines RADHA [Dasdan et al. 1998] and RATAN [Mathur et al. 1998]. This methodology is illustrated using the design flow in Figure 1. The steps handled by RADHA-RATAN are identified in the figure by dashed rectangles. The other steps are similar to those in a traditional codesign methodology, for example, POLIS [Balarin et al. 1997]. The use of RADHA-RATAN makes this methodology timing-driven. Being timing-driven means taking the timing constraints of the system into consideration at every stage in the system's design flow but, more importantly, doing so during very early stages in the design flow such as the requirements specification stage.

The rest of the paper is organized as follows: We discuss the fundamental issues for the timing of events in the system in Section 2, and the system's

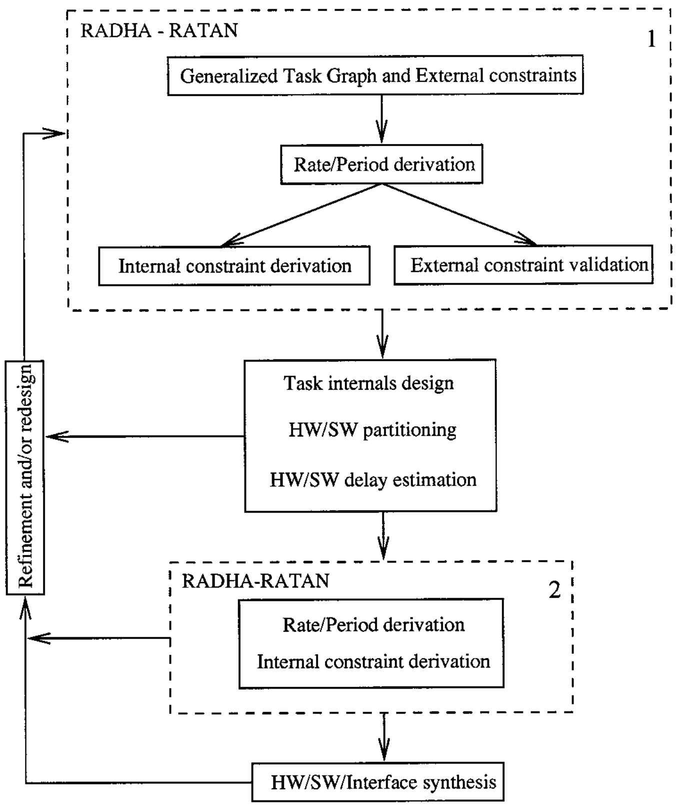
RADHA - RATAN
1
Generalized Task Graph and External constraints
Rate/Period derivation
Internal constraint derivation
External constraint validation
Task internals design
HW/SW partitioning
HW/SW delay estimation
Refinement and/or redesign
RADHA-RATAN
2
Rate/Period derivation
Internal constraint derivation
HW/SW/Interface synthesis

Fig. 1. The timing-driven design flow for embedded systems using RADHA-RATAN.

high-level timing constraints in Section 3. We then present the generalized graph model in Section 4 and define the derivation and validation problems in Section 5. We use the following two sections, Sections 6 and 7, to give our solution to these problems. Finally, we illustrate our algorithms on an example embedded system in Section 8 and conclude with Section 9.

2. FUNDAMENTAL TIMING ISSUES

This section describes the fundamental issues related to the timing of events within the system to establish the background for the following sections.

An embedded system interacts with its environment through its sensors and actuators. The sensors carry the stimuli (inputs) into the system from its environment, and the actuators carry the responses (outputs) out of the system to its environment. They are illustrated in Figure 2. These stimuli and responses are actually called *events*. Events can also result from the interaction within the system or simply from the passage of time [Kopetz 1997].

Events occur along the timeline, which extends from the past into the future. A mapping of events to their occurrence times along the timeline defines an event order called *temporal order*. Events can also be ordered causally in that one event can affect another event. This order is called *causal order*. Causal order helps determine which event happens before what others, and thus implies temporal order but temporal order does not necessarily imply causal order.

Temporal and causal orders between the events in the system can be used to model these events using a directed graph called the *event graph* in which the nodes correspond to the events and the arcs to the orders between the events [Amon 1993; Burns 1991]. An event in such a graph depends on its predecessors in temporal order or causal order or both.

When an event gets caused by an enough number of its predecessors in the event graph, these predecessors are said to *enable* that event. An event can need zero, one, some, or all of its predecessor(s) to get enabled. Then, the event is called an input event, OR event, AND/OR event, or AND event, respectively. An OR event uses *OR causality*, and an AND event uses *AND causality*. These two causality types are the fundamental ones for causal order because AND/OR events can easily be shown to use a combination of them [Gunawardena 1993]. Because of this, we assume that the event graph contains only AND and/or OR events.

There are *intervals* on the timeline between different event occurrences. Each interval is bounded by the occurrence of a *start event* and that of an *end event*. The *duration* of an interval is equal to the nonnegative difference between these occurrence times. Note that events are assumed to be instantaneous [Kopetz 1997] but we still use an interval to represent their occurrence times, as advocated in Allen [1983].

A *timing constraint* is an interval on the timeline; hence, it is between two event occurrences. If these occurrences belong to the same event, we refer to the timing constraint as a *rate constraint*, where the *rate* of an event is defined to be its frequency of occurrence. If these occurrences belong to different events, we refer to the timing constraint as a *separation constraint*. In both cases, a timing constraint is essentially a separation in time (or *time separation*) between two occurrence times. Without loss of generality, we use an interval of nonnegative integers for the duration of each timing constraint. The duration of a timing constraint can range from zero to infinity.

Since each timing constraint is between a pair of events, each timing constraint can be mapped to an arc of the event graph. The duration of the timing constraint becomes the delay of the arc that it is mapped to. Arcs between different events represent separation constraints, and arcs on the same event, i.e., self-loops in the event graph, represent rate constraints. It should be observed that although each arc defines a binary timing constraint in the event graph, temporal and causal orders can easily define a timing constraint between more than two events, e.g., between one event and its potentially many predecessors. In particular, temporal order between one event and its predecessors defines a *linear* (*timing*) *constraint*, and causal order between them defines a *max* (*timing*) *constraint* or a *min* (*timing*) *constraint* depending on whether the event is an AND event or an

OR event, respectively. These timing constraints are defined in Yen et al. [1994] for the event graph. In Section 4, we will define them more precisely for task graphs.

There are two fundamental problems of timing analysis on the event graphs: the *separation problem* and the *rate problem*. These problems are a natural byproduct of the fact that a timing constraint is either a separation constraint or a rate constraint.

Problem 1 (Separation Problem). Find the maximum and minimum time separations between particular occurrences of any two different events in the event graph.

Problem 2 (Rate Problem). Find the maximum and minimum time separation between particular occurrences (usually successive) of any one repetitive event in the event graph.

In the latter case, the time separation is sometimes called the *interoccurrence time* or simply the *period*. The *rate* of an event is then equal to the reciprocal of the period of the event. The derivation and separation problems that we will define in Section 5 are generalization of these problems.

3. HIGH-LEVEL TIMING CONSTRAINTS

This section gives a comprehensive classification of timing constraints of the system. Our classification is an extension of those in Chou et al. [1994] and Dasarathy [1985]. This section defines all the timing constraints that will be either derived and/or validated in the following sections.

For our purposes, events of interest in the system are the stimuli and responses between the system and its environment as well as between the tasks within the system. The initiation and termination of a task execution are also events of interest.

As the system is real-time, events are governed by timing constraints. All the timing constraints of the system can be cast into either a separation constraint or a rate constraint; however, at a high level, it is more useful to classify them further.

In Section 1, we mentioned the external and internal timing constraints of the system. The former constraints are defined on the system's external behavior, the one between the system and its environment, and the latter ones are defined on its internal behavior, the one between its tasks. Figure 2 illustrates these timing constraints. In this figure, the timing constraints labeled 1–6 are the external timing constraints, and those labeled 7–11 are the internal timing constraints.

Consider Figure 2(a) for the external timing constraints. These timing constraints can be classified further as follows. The labels in parenthesis below refer to the labels in Figure 2(a), and for each category of timing constraints below, we describe the place where they are defined and give the category name.

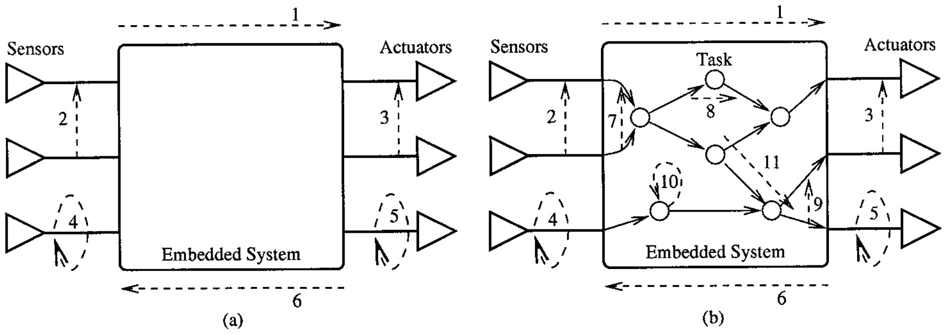

Fig. 2. (a) An embedded system with external constraints (labeled 1–6). (b) The same system with internal constraints (labeled 7–11) on tasks.

(1) On the same input or output: an *input rate constraint* (4) or *output rate constraint* (5), respectively.
(2) Between two different inputs or outputs: an *input correlation constraint* (2) or *output correlation constraint* (3), respectively.
(3) From an input to an output or vice versa: a *response time constraint* on the system (1) or on the environment (6). It is also possible to define these constraints between any input–output pair [Hatley and Pirbhai 1987].

In Figure 2(a), the external timing constraints labeled 2, 4, and 6 are the given timing constraints of the system whereas those labeled 1, 3, and 5 are the required timing constraints or the timing requirements of the system.

Similarly, consider Figure 2(b) for the internal timing constraints. Then, we classify them further as follows: They are defined on the tasks.

(1) On the same input to or output from a task: a *rate constraint* (10).
(2) Between two different inputs or outputs: an *input correlation constraint* (7) and an *output correlation constraint* (9).
(3) From an input of a task to its output: a *latency* constraint (8) or an arbitrary separation constraint (11), which is useful to define response time constraints on the subsystems of the system.

If any of these timing constraints on the tasks are required ones, then they are also referred to as the timing requirements (of the tasks).

4. GENERALIZED TASK GRAPH MODEL

During the early stages of an embedded design flow or at stages where higher-levels of abstraction are considered, 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.

Our task graph model is called the *generalized task graph* (GTG) model. It builds upon many previous models in the literature such as Buck and

Lee [1993], Gillies and Liu [1995], Gomaa [1993], Gunawardena [1993], Jeffay [1993], Karp and Miller [1966], Lee and Messerschmitt [1987], Mathur et al. [1998], Puchol and Mok [1994], and Yen [1996]. 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 Gunawardena [1993], both acyclic and cyclic task dependencies from Buck and Lee [1993], Karp and Miller [1966], Lee and Messerschmitt [1987], and Mathur et al. [1998], modeling of both control and data from Gomaa [1993], Puchol and Mok [1994], and Yen [1996], and skipped and unskipped behaviors from Gillies and Liu [1995]. 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.

Definition. 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, for example, 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.

Task Interactions. Interactions within almost any embedded system can be modeled by using enables, disables, and triggers [Gomaa 1993]. 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.

Task and Channel Properties. Consider Figure 3. 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 or known in advance, for example, when we fix $T(c)$ but not $T(p)$, $C(p, c)$ becomes variable. Channel (p, c) has an integer, bounded delay of $d(p, c)$ that ranges from a lower bound of $D_l(p, c)$ to an upper bound of $D_u(p, c)$, that is, $d(p, c) \in [D_l(p, c), D_u(p, c)]$. It takes $d(p, c)$ time units for the tokens of producer p to enable consumer c.

Let $T(c) = [T_l(c), T_u(c)]$ and $r(c) = [r_l(c), r_u(c)]$ 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 $r_l(c) = 1/T_u(c)$ and $r_u(c) = 1/T_l(c)$. Due to this basic relation, we will use the terms "period" and "rate" interchangeably. We also have $0 < T_l(c) \leq T_u(c) \leq \infty$ so that $0 \leq r_l(c) \leq r_u(c) < \infty$. 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 jth time. Then, since we must have $i * P(p, c) - (j - 1) * C(p, c) \geq C(p, c)$, we get

$$j * \lceil C(p, c)/P(p, c) \rceil \geq i \geq \lceil j * C(p, c)/P(p, c) \rceil \geq j * \lfloor C(p, c)/P(p, c) \rfloor,$$

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

$$t_p(U(j, p, c)) + d(p, c) \geq t_c(j) \geq t_p(L(j, p, c)) + d(p, c), \tag{1}$$

where $U(j, p, c) = j * \lceil C(p, c)/P(p, c) \rceil$ and $L(j, p, c) = j * \lfloor C(p, c)/P(p, c) \rfloor$.

Using Eq. 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 jth start time of an AND task c is governed by

$$\max_{(p,c)\in E} \{t_p(U(j, p, c)) + D_u(p, c)\} \geq t_c(j) \geq \max_{(p,c)\in E} \{t_p(L(j, p, c)) + D_l(p, c)\}, \tag{2}$$

whereas that of an OR task c is governed by

$$\min_{(p,c)\in E} \{t_p(U(j, p, c)) + D_u(p, c)\} \geq t_c(j) \geq \min_{(p,c)\in E} \{t_p(L(j, p, c)) + D_l(p, c)\}, \tag{3}$$

where tasks p are predecessors of task c in the arc set E of the GTG, and $U(j, p, c)$ and $L(j, p, c)$ as defined previously.

Task Input Behavior. An AND/u task c reads $C(p, c)$ tokens from each one of its predecessors 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 its 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 predecessors 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 its predecessors p has sent $C(p, c)$ tokens for it. Note that an OR/u task is like an AND/u task but an OR/u task can start executing with the arrival of enough tokens from only one of its predecessors.

Task Output Behavior. The output behavior of a task in the GTG is determined by its successors. Suppose a channel has $P(p, c)$ as one of its token numbers. 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 of c will send it some tokens). Then, task p has to send at most $P(p, c)$ tokens to its successor c. However, in the above start time expressions, we assumed that p sends exactly $P(p, c)$ tokens every execution. The reason is due to the possible behavior of p. If it is possible that p can produce $P(p, c)$ tokens 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 $P(p, c)$ tokens every K, $K > 1$, executions, then we can easily take K into account in all our derivations.

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.

Single-Rate Vs. Multi-Rate. For the sake of simplicity, we assume that the channels in the top level of the GTG do not have tokens in them when the system starts executing for the first time. The channels at the bottom level, however, must have tokens initially because they are needed to enable the tasks that are connected to each other under cyclic dependencies. We also assume that the cyclic portions of the GTG are *single-rate* whereas the acyclic portions can be *multi-rate*. In a multi-rate system, each task can have a different rate whereas in a single-rate system, each task has the same rate [Ito and Parhi 1995]. Since there are techniques to transform a multi-rate cyclic system into an equivalent single-rate one (e.g., see Ito and Parhi [1995]), we do not lose generality with our simplifying assumption.

5. DERIVATION AND VALIDATION PROBLEMS

In this paper, we propose a solution to the following problems.

Problem 3. (*Derivation Problem*). Derive every rate and separation constraint in the system modeled by the GTG.

Problem 4. (*Validation Problem*). Validate every rate and separation requirement in the system modeled by the GTG.

These problems are generalizations of rate and separation problems that are introduced in Section 2.

Our solution works as follows: We first use the given and sometimes required rate constraints of the system to derive its internal rate constraints, that is, we find the rate of every task or equivalently the period of every task in the system. This step is referred to as the *rate derivation* or *period derivation*. We then use these periods to derive the internal and external separation constraints of the system such as its response time and correlation constraints. This step is referred to as the *separation derivation*. We finally compare the derived timing constraints with the corresponding timing requirements of the system to validate its timing correctness as well as with the corresponding timing requirements on the tasks to validate their timing requirements. This step is referred to as the *rate and separation validation*. We always perform a conservative validation. In the sequel, we will discuss the rate derivation together with rate validation and also the separation derivation together with separation validation.

6. RATE DERIVATION AND VALIDATION

Consider the GTG G. We derive the rate and period of every task in G. We discuss the derivation in two parts. In the first part, we discuss the rate derivation for the top level of G, that is, for an acyclic GTG. In the second part, we discuss the rate derivation for the bottom level of G, that is, for a cyclic GTG. We finally discuss the rate validation and ways to combine the results of top and bottom levels.

Related work in rate derivation include Buck and Lee [1993], Lee and Messerschmitt [1987], Gerber et al. [1995], Goddard and Jeffay [1998], and Seto et al. [1996]. A common feature of these works is that they assume to know most of the properties of tasks and arcs: Gerber et al. [1995], Goddard

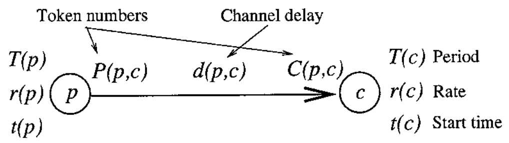

Fig. 3. Properties of two tasks, producer p and consumer c, and the channel between them.

and Jeffay [1998], and Seto et al. [1996] assume to know even the execution time of each task, which are almost impossible to know at the high level of abstraction that we discuss in this paper, and Buck and Lee [1993] and Lee and Messerschmitt [1987] assume to know all the variables in a rate equation except for the periods. Among these works, only Buck and Lee [1993] considers OR behavior; the others are for AND/u tasks only.

Consider the arc (p, c) from producer p to consumer c in Figure 3. Our rate derivation algorithm assumes that the rate of producer p, the predecessor of c, is known, and we are to derive the rate of consumer c. Thus, to ensure that channel (p, c) is bounded, we require that the rates of p and c match, that is, $r(p) = r(c)$. This equation is called a *rate equation*, which is similar to a balance equation [Buck and Lee 1993], but we will use the period rather than the number of iterations of these tasks in our derivation.

Now suppose that c has a required period interval $RT(c) = [RT_l(c), RT_u(c)]$. If we derive a period interval of $DT(c) = [DT_l(c), DT_u(c)]$ for c, then it must satisfy

$$RT_l(c) \leq DT_l(c) \leq T_l(c) \leq T_u(c) \leq DT_u(c) \leq RT_u(c), \tag{4}$$

where $T(c) = [T_l(c), T_u(c)]$ is the actual period interval for c, which we may not know until after the system's implementation. We use $DT(c)$ as the period of c in our algorithms.

We say that a property of a task or channel is *known* if it is a finite number that is specified in advance; it is *unknown* otherwise. The rate derivation algorithm is based on an enumeration of the cases as to how many of the attributes in the following equation

$$\frac{RT_l(c)}{DT_l(p)} \leq \frac{C(p, c)}{P(p, c)} = \frac{DT_l(c)}{DT_l(p)} = \frac{DT_u(c)}{DT_u(p)} \leq \frac{RT_u(c)}{DT_u(p)} \tag{5}$$

are known or infinite. This equation is obtained from both Eq. 4 and the rate equation.

We use the algorithm *DerivePeriod* in Figure 4 to derive period interval for c *with respect to* the arc (p, c). This algorithm invokes two other algorithms, *DeriveWithGiven* in Figure 5 and *DeriveWithRequired* in Figure 6, depending on whether or not the period of c is also known in advance. If the period of c is known, then *DeriveWithGiven* is invoked to derive the properties of channel (p, c) and those of c. If it is not known, *DeriveWithRequired* is invoked to do the job. These algorithms also check

```
$DerivePeriod(required, p, c, P, C, DT_l(c), DT_u(c))$
    Input: producer $p$ and consumer $c$.
    Output: properties of arc $(p,c)$ and derived period bounds of consumer $c$.
    /*** Check for violations ***/
1   if both $DT_l(c)$ and $RT_l(c)$ are known and $DT_l(c) < RT_l(c)$ then
2     return with violation.
3   if both $DT_u(c)$ and $RT_u(c)$ are known and $DT_u(c) > RT_u(c)$ then
4     return with violation.
    /*** Copy the arc properties in order not to modify them ***/
6   $P \leftarrow P(p,c)$; $C \leftarrow C(p,c)$
    /*** Derive the properties of consumer $c$ and arc $(p,c)$ ***/
8   if $c$ has known period bounds then
9     return $DeriveWithGiven(required, P, C, DT_l(p), DT_u(p), DT_l(c), DT_u(c))$
10  else
11    return $DeriveWithRequired(required, P, C, DT_l(p), DT_u(p), RT_l(c), RT_u(c), DT_l(c), DT_u(c))$
```

Fig. 4. Rate derivation algorithm for a single arc.

```
$DeriveWithGiven(required, P(p,c), C(p,c), DT_l(p), DT_u(p), DT_l(c), DT_u(c))$
        Input: properties of producer $p$, consumer $c$, and arc $(p,c)$.
        Output: properties of arc $(p,c)$ and derived period bounds of consumer $c$.
        /*** Check for violations ***/
1       if $DT_l(c) = \infty$ or $DT_l(p) = \infty$ then return with violation.
2       if $DT_u(p) = \infty$ then
3         if $DT_u(c)$ is unknown then $DT_u(c) = \infty$
4         else if $DT_u(c) \neq \infty$ then return with violation.
5       else if $DT_u(p) = \infty$ then return with violation.
        /*** Compute $P(p,c)$ and $C(p,c)$ if at least one is unknown ***/
6-23    Lines 3-20 from $DeriveWithRequired(\cdots)$ with $RT(c)$ replaced by $DT(c)$.
        /*** Check for inconsistency with the given bounds ***/
24      if $DT_l(c)$ is known then
25        if $P(p,c) * DT_l(c) \neq C(p,c) * DT_l(p)$ then output inconsistency.
26      if $DT_u(c)$ is known then
27        if $P(p,c) * DT_u(c) \neq C(p,c) * DT_u(p)$ then output inconsistency.
        /*** Check for violations ***/
28      if $C(p,c) = 0$ or $P(p,c) = 0$ then return with violation.
        /*** Compute the period bounds for the consumer if required ***/
29      if $required$ then
30        $DT_l(c) \leftarrow \lfloor C(p,c) * DT_l(p)/P(p,c) \rfloor$
31        if $DT_u(p) \neq \infty$ then $DT_u(c) \leftarrow \lceil C(p,c) * DT_u(p)/P(p,c) \rceil$
32      return with success.
```

Fig. 5. Rate derivation algorithm for a single arc with given period bounds.

for any potential violations or inconsistencies during derivation. Each of these algorithms runs in constant time.

Acyclic Rate Derivation. The algorithm *DerivePeriodsForAcyclicGTG* in Figure 7 is the main algorithm for the rate derivation for an acyclic GTG. It is used during the stage labeled 1 in Figure 1. It assumes that the rate of each input task is known. It traverses the input graph in topological order of its nodes from the inputs to the outputs, and derives the period of each task it visits. It performs a different action depending on the type of the task it visits. The algorithm *DerivePeriodsForAcyclicGTG* runs in time linear in the size of the input GTG.

```
$DeriveWithRequired(required, P(p,c), C(p,c), DT_l(p), DT_u(p), RT_l(c), RT_u(c), DT_l(p), DT_u(p))$
    Input: properties of producer $p$, consumer $c$, and arc $(p,c)$.
    Output: properties of arc $(p,c)$ and derived period bounds of consumer $c$.
    /*** Check for violations ***/
1   if $RT_l(c) = \infty$ or $DT_l(p) = \infty$ then return with violation.
2   if $RT_u(c) \neq \infty$ and $DT_u(p) = \infty$ then return with violation.
    /*** Compute $P(p,c)$ or $C(p,c)$ if none is known ***/
3   if both $P(p,c)$ and $C(p,c)$ are unknown then
4     if $RT_l(c)$ is unknown then
5       if $RT_u(c)$ is unknown or $RT_u(c) = \infty$ then $P(p,c) \leftarrow 1$
6       else if $DT_u(p) < RT_u(c)$ then $P(p,c) \leftarrow 1$
7       else $C(p,c) \leftarrow 1$
8     else if $RT_l(c)$ is known then
9       if $DT_l(p) < RT_l(c)$ then $P(p,c) \leftarrow 1$
10      else $C(p,c) \leftarrow 1$
    /*** Compute $P(p,c)$ or $C(p,c)$ if at least one is known ***/
11  if $P(p,c)$ is unknown but $C(p,c)$ is known then
12    if $RT_l(c)$ is unknown then
13      if $RT_u(c)$ is unknown or $RT_u(c) = \infty$ then $P(p,c) \leftarrow 1$
14      else $P(p,c) \leftarrow \lceil C(p,c) * DT_u(p)/RT_u(c) \rceil$
15    else if $RT_l(c)$ is known then $P(p,c) \leftarrow \lfloor C(p,c) * DT_l(p)/RT_l(c) \rfloor$
16  else if $P(p,c)$ is known but $C(p,c)$ is unknown then
17    if $RT_l(c)$ is unknown then
18      if $RT_u(c)$ is unknown or $RT_u(c) = \infty$ then $C(p,c) \leftarrow P(p,c)$
19      else $C(p,c) \leftarrow \lfloor P(p,c) * RT_u(c)/DT_u(p) \rfloor$
20    else if $RT_l(c)$ is known then $C(p,c) \leftarrow \lceil P(p,c) * RT_l(c)/DT_l(p) \rceil$
    /*** Check for violations ***/
21  if $C(p,c) = 0$ or $P(p,c) = 0$ then return with violation.
    /*** Compute the period bounds for the consumer if required ***/
22  if $required$ then
23    $DT_l(c) \leftarrow \lfloor C(p,c) * DT_l(p)/P(p,c) \rfloor$
24    if $DT_u(p) = \infty$ then $DT_u(c) = \infty$
25    else $DT_u(c) \leftarrow \lceil C(p,c) * DT_u(p)/P(p,c) \rceil$
    /*** Check for violations ***/
26  if $RT_l(c)$ is known and $DT_l(c) < RT_l(c)$ then return with violation.
27  if $RT_u(c)$ is known and $DT_u(c) > RT_u(c)$ then return with violation.
28  return with success.
```

Fig. 6. Rate derivation algorithm for a single arc with known required period bounds.

In line 16 of *DerivePeriodsForAcyclicGTG*, it is possible to use heuristics in order to bound the period of consumer c. For example, a very pessimistic bound is given by taking the least common multiple of the periods of the predecessors of c in Dasdan et al. [1998]. In this paper, we do not discuss any such heuristics.

Cyclic Rate Derivation. The algorithm *DerivePeriodsForCyclicGTG* in Figure 8 is the main algorithm for the rate derivation for a cyclic GTG. It is usually used during the stage labeled 2 in Figure 1. It assumes that the delay interval of each arc is known. It uses the fact that the period of each task in a SCC is the same and is equal to the *maximum or minimum cycle* mean of the SCC, depending on whether the SCC contains only AND tasks or OR tasks, respectively [Mathur et al. 1998]. The (cycle) *mean* of a cycle is its average arc delay, that is, the ratio of its total arc delay to its length, and the maximum or minimum cycle mean of a SCC is equal to the

```
    DerivePeriodsForAcyclicGTG($G$)
       Input: an acyclic GTG $G$.
       Output: properties of every arc and task in $G$.
1      Find a topological sorting of the tasks in $G$.
2      for each task $c$ in topologically sorted order from inputs to outputs do
3        if $c$ has no predecessors then
4          return with an error of insufficient data.
5        else if $c$ has one predecessor then
6          Let $p$ be the only predecessor of $c$.
7          $DerivePeriod(true, p, c, P(p,c), C(p,c), DT_l(c), DT_u(c))$
8        else if $c$ has more than one predecessor then
9          if $c$ is an unskipped task then
10           Look for an arc $(p,c)$ for which both $P(p,c)$ and $C(p,c)$ are known.
11           if such an arc $(p,c)$ is found then
12             $DerivePeriod(true, p, c, P(p,c), C(p,c), DT_l(c), DT_u(c))$
13           else if $RT_l(c)$ or $RT_u(c)$ is known then
14             $DerivePeriod(true, p, c, P(p,c), C(p,c), DT_l(c), DT_u(c))$
15           else /*** Heuristic solutions are possible here ***/
16             return with an error of insufficient data.
17         else if $c$ is a skipped task then
18           for each predecessor $p$ of $c$ do
               /*** Do not update any properties yet ***/
19             $DerivePeriod(true, p, c, \_, \_, DT_l, DT_u)$
               /*** Compute the period bounds ***/
20             if $c$ is an AND/s task then
21               $T_l \leftarrow \max\{T_l, DT_l\}$
22             else $c$ is an OR/s task then
23               $T_l \leftarrow \min\{T_l, DT_l\}$
24             $T_u \leftarrow \max\{T_u, DT_u\}$
             /*** Copy the periods bounds ***/
25           $DT_l(c) \leftarrow T_l$; $DT_u(c) \leftarrow T_u$
         /*** Update the properties of the predecessor arcs ***/
26       for each predecessor $p$ of $c$ do
27         $DerivePeriod(false, p, c, P(p,c), C(p,c), DT_l(c), DT_u(c))$
           /*** Ignore any error, violation, and inconsistency messages here ***/
28     return with success.
```

Fig. 7. Rate derivation algorithm for acyclic task graphs.

maximum or minimum of the cycle means over all the cycles in the SCC, respectively. This fact provably holds when the SCC contains only AND tasks or only OR tasks [Mathur et al. 1998]. Some promising results for the analysis of the case where a SCC with both causality orders are given in Gunawardena [1993]. The algorithm *DerivePeriodsForCyclicGTG* runs in polynomial time, cubic in the number of tasks in the input graph, since the maximum or minimum cycle mean of a graph can be computed in polynomial time [Dasdan and Gupta 1998].

Note that if we do not know the arc or channel delays in a SCC, we cannot compute its maximum or minimum cycle mean. In that case, we use the fact that any period derived for the supernode of the SCC, being its maximum mean, has to be larger than the mean of any cycle in the SCC, assuming that the SCC has all AND tasks. The result for the SCC with all OR tasks is analogous. This fact can be especially useful to determine bounds on the arc delays within short cycles.

```
   $DerivePeriodsForCyclicGTG(G)$
   Input: a strongly connected GTG $G$.
   Output: properties of every task in $G$.
1  if $G$ has only AND tasks then
2    Find the maximum cycle mean $\lambda_l$ of $G$ using $d_{pc}$ for each arc $(p, c)$.
3    Find the maximum cycle mean $\lambda_u$ of $G$ using $D_{pc}$ for each arc $(p, c)$.
4    for each task $c$ do
5      $DT_l(c) \leftarrow \lambda_l$; $DT_u(c) \leftarrow \lambda_u$
6  else if $G$ has only OR tasks then
7    Lines 2-5 with "maximum" replaced by "minimum".
8  else
9    output no known analytical solution.
   /*** Check for violations ***/
10 Find the intersection $IT = [IT_l, IT_u]$ of the required period intervals of the tasks in $G$.
11 Find the period interval $T = [T_l, T_u]$ derived for the supernode for $G$.
12 if $IT_l \leq T_l = \lambda_l \leq \lambda_u = T_u \leq IT_u$ does not hold then
13   return with violation.
14 return with success.
```

Fig. 8. Rate derivation algorithm for cyclic task graphs.

Rate Validation. As the rate requirements on the system can be transferred to the corresponding output tasks, we have to devise ways to validate the rate requirement on a task. Since after the rate derivation, we obtain the rate or period interval of every task in the system, we compare each derived interval with the corresponding required interval. If the required one subsumes the derived one, the required one is satisfied; otherwise, it is violated. If we have not obtained the period interval of every task in the SCCs of the system due to lack of the knowledge of the arc delays, we can still validate the rate requirements of each SCC. Since we know that each task in a given SCC has the same period interval, then the intersection of the required period intervals of the tasks in the SCC has to subsume the derived period interval for the corresponding supernode. If not, there is a violation stemming from the top level. These results are used in the algorithm *DerivePeriodsForCyclicGTG*.

Deriving Latency Bounds. The derived period intervals can be used to derive bounds on the latency of each task. An upper bound on the latency of a task is the lower bound of the period interval of that task. This simple result can be used during hardware/software partitioning and synthesis to determine if there is an implementation of the task and to eliminate those different implementations that will disrupt the system's timing correctness. The bound on the latency of a task can be tightened by using additional constraints involving the task latencies. For instance, if the task is in a cycle in the GTG of all AND tasks, then the latency of the task is a delay contributing to the delay of the cycle, which must be less than the maximum cycle mean of the GTG.

7. SEPARATION DERIVATION AND VALIDATION

In order to derive separations, we will reduce the GTG for the system to an event graph. The resulting event graph is identical to the GTG but its channel delays are different. Consider an arc (p, c) in the GTG and the same arc in its event graph. The GTG has $[D_l(p, c), D_u(p, c)]$ as the delay of arc (p, c). However, (p, c) in the event graph will have $T_l(p) * \max\{0, \lfloor C(p, c)/P(p, c) \rfloor - 1\} + D_l(p, c)$ as its lower delay bound and $T_u(p) * (\lceil C(p, c)/P(p, c) \rceil - 1) + D_u(p, c)$ as its upper delay bound. These equations are obtained from Eqs. 2 and 3, and follow from the fact that the top level is multi-rate. In these equations, if we do not know $D_l(p, c)$ and $D_u(p, c)$, then we can respectively use $T_l(p)$ and $T_u(p)$ in place of them to ensure the worst-case of the fastest implementation of the system.

If we assign zero as the start time of the input tasks, it is possible to determine bounds on the start time of any node in the event graph. We define the maximum and minimum time separations, S_{ij} and s_{ij}, between the start times of any two nodes i and j in the event graph as $s_{ij} \le \min\{t(j) - t(i)\} \le t(j) - t(i) \le \max\{t(j) - t(i)\} \le S_{ij}$. We note that due to the conservative delay assignment to the arcs, the maximum and minimum time separations may not be realizable but are still valid upper and lower bounds. We also note that since $\min\{t(j) - t(i)\} = -\max\{t(i) - t(j)\}$, we can set $s_{ij} = -S_{ji}$, which means that we only need to compute the maximum separations.

Computing the time separation of any pair of nodes in the event graph is the separation problem, as we defined in Section 2. There are many algorithms to solve the separation problem; for example, see Yen et al. [1994] for a list of them. The separation problem is NP-complete when the event graph has both AND causality and OR causality [McMillan and Dill 1992], but there are polynomial-time approximation algorithms, for example, the one in Chakraborty and Dill [1997], that seems to work well in practice. For the general case, we use that approximation algorithm.

The solution of the separation problem derives the separations in the event graph. We then validate any external or internal separation requirements by comparing them with their corresponding derived separations. As in the case of rate constraints, if the required separation subsumes the derived one, we say that the required one is satisfied; otherwise, it is violated. Note that the response time constraint and correlation constraints can be cast as separation constraints. For example, the response time constraint is the maximum separation between any pair of input and output tasks. The correlation constraints of the system can be translated to those on the tasks. The output correlation constraint on a task can be translated into a separation between the successors of the task. As for the input correlation constraint, we perform the translation in Figure 9 (from (a) to (b)). In this figure, the input correlation constraint with label 1 on task k is handled by translating it to the separation between two dummy predecessors of task k.

8. IMPLEMENTATION

The tool RADHA-RATAN was implemented in approximately 10,000 lines of C++ code. It reads its input GTG, which is specified using a simple

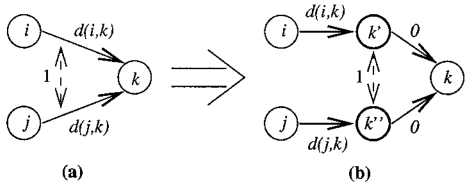

Fig. 9. Validating the input correlation requirement on a task.

specification language. The external timing constraints are also specified on the input GTG as linear constraints on its nodes. The tool outputs another GTG in which all the properties of the tasks and channels are derived using the algorithms discussed in this paper. It also outputs the validation results, any violations and the reasons for them.

We now discuss an application of the tool on an example embedded system, the dashboard controller of a car, taken from Balarin et al. [1997]. The task graph for the controller is depicted in Figure 10. The name of each task describes its functionality. More detailed explanation is in Balarin et al. [1997]. In this controller, the speedometer (task d) registers car speed in the range of 0–260 km/h where any speed value less than 5 km/h 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 controller gets four pulses from task a for every rotation of the tire such that every pulse represents 1/4 of a rotation. The tire travels 0.66 m per rotation.

The only input task of the controller is task a. Our algorithms need to know its rate. Its rate is derived by calculating the smallest and largest separation between two successive pulse generation times. The smallest time separation occurs when the car is the fastest, and the largest one occurs when the car is the slowest. Hence, the smallest one is (0.66/4 m)/(260 km/h) = 228 ms, and the largest one is (0.66/4 m)/(5 km/h) = 11880 ms, that is, $T(a) = [228, 11880]$ ms.

There are only four known token numbers: $C(a, b)$, $C(a, e)$, $P(f, h)$, and $C(f, h)$. Since task b needs at least two tokens to compute the distance and thus the speed of the car, the number of tokens that task b consumes is at least $C(a, b) = 2$. Since the odometers are required to display increments of 0.1 km, task e must consume $C(a, e) = \lfloor 0.1$ km/(0.66/4 m$\rfloor$ = 606 tokens. We set each token numbers of the channel (f, h) to one, that is, $P(f, h) = C(f, h) = 1$, in order to reduce the number of unknowns for the algorithm *DerivePeriodsForAcyclicGTG*.

The timing requirements on the controller are all in the form of rate requirements. In Figure 10, the rate requirements are expressed in the form of period requirements, which are $RT(d)$, $RT(i)$, and $RT(j)$. The first requirement is because the speedometer outputs must be produced at a rate of 100 Hz to drive the gauge coils in the speedometer. The other requirements are because the odometer outputs for 0.1 km must be produced faster than the time it takes for the car to travel 0.1 km.

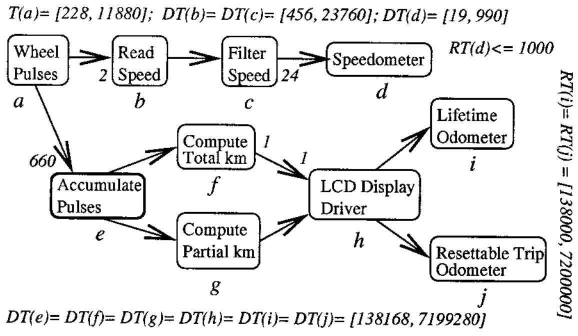

Fig. 10. The dashboard controller with its given (T), required (RT), and derived (DT) timing constraints.

The functionality of the tasks dictates that task h is an OR/s task and the other tasks are AND/u tasks. Since tasks f and g have the same output behavior, we consider task h as an AND/u task. The derived period intervals are shown in the same figure. All the times are in milliseconds. It is easy to see that all the rate requirements are satisfied. Our algorithms also derive the token numbers in the task graph. All the derived token numbers except for $P(c, d)$ are equal to one; $P(c, d) = 24$, making the speedometer display the speed changes more smoothly. For instance, if the speed changes from 30 km/h to 54 km/h, the speedometer will display 31, . . . , and 54 in order after 30 rather than displaying 54 right away. Note that $P(c, d)$ is derived to be 24 because of the rate requirement on the speedometer.

All the derived information can now be used to perform a trade-off analysis and validate the design choices. For example, in Balarin et al. [1997], the period of task b is chosen to be 250 ms. We see that the choice is in agreement with the derived period of task b; however, this choice makes $C(a, b)$ variable and task b depend on an enable signal from a timer instead of task a. It also modifies the properties of the tasks c and d and the channels (b, c) and (c, d), which can be derived by another run of our algorithms.

9. CONCLUSIONS

We have addressed the problems of timing constraint derivation and validation for reactive and real-time embedded systems. Such systems continuously interact with their environment under strict external timing constraints, and we have shown how these external constraints translate to internal timing constraints on the tasks of these systems. To handle these problems, we have developed the generalized task graph model, algorithms that operate on this model, a codesign methodology that extends a traditional codesign methodology with this model and algorithms, and finally, an implementation of these algorithms in the tool RADHA-RATAN. Our contributions in this paper allow a systematic analysis of timing constraints on an embedded system.

REFERENCES

ALLEN, J. F. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM 26*, 11 (Nov.), 832–843.

AMON, T. 1993. Specification, simulation, and verification of timing behavior. Ph.D. dissertation. Univ. Washington.

BALARIN, F., CHIODO, M., GIUSTO, P., HSIEH, H., JURECSKA, A., LAVAGNO, L., PASSERONE, C., SANGIOVANNI-VINCENTELLI, A., SENTOVICH, E., SUZUKI, K., AND TABBARA, B. 1997. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic, Boston, MA.

BUCK, J. T., AND LEE, E. A. 1993. The token flow model. In *Advanced Topics in Dataflow Computing and Multithreading*. L. Bic, G. Gao, and J. Gaudiot, eds. IEEE, New York.

BURNS, S. M. 1991. Performance analysis and optimization of asynchronous circuits. Ph.D. dissertation. California Institute of Technology.

CHAKRABORTY, S., AND DILL, D. L. 1997. Approximate algorithms for time separation of events. In *Proceedings of the International Conference on Computer-Aided Design* (Nov.). IEEE/ACM, New York, 190–194.

CHOU, P., WALKUP, E. A., AND BORRIELLO, G. 1994. Scheduling for reactive real-time systems. *IEEE Micro* (Aug.), 37–47.

DASARATHY, B. 1985. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *IEEE Trans. Softw. Eng. 11*, 1 (Jan.), 80–86.

DASDAN, A., AND GUPTA, R. K. 1998. Faster maximum and minimum mean cycle algorithms for system performance analysis. *IEEE Trans. Computer-Aided Design*, to appear.

DASDAN, A., RAMANATHAN, D., AND GUPTA, R. K. 1998. Rate derivation and its applications to reactive, real-time embedded systems. In *Proceedings of the ACM 35th Design Automation Conference*. 263–268.

GERBER, R., HONG, S., AND SAKSENA, M. 1995. Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Trans. Softw. Eng. 21*, 7 (July), 579–592.

GILLIES, D. W., AND LIU, J. W.-S. 1995. Scheduling tasks with AND/OR precedence constraints. *SIAM J. Comput. 24*, 4 (Aug.), 797–810.

GODDARD, S., AND JEFFAY, K. 1998. A software synthesis method for building real-time systems from processing graphs. In *Proceedings of the Real-Time Technology and Applications Symposium* (June). IEEE, New York.

GOMAA, H. 1993. *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley, Reading, MA.

GUNAWARDENA, J. 1993. Periodic behavior in timed systems with AND,OR causality. Tech. Rep. STAN-CS-93-1462. Stanford Univ., Stanford, CA.

HATLEY, D. J., AND PIRBHAI, I. A. 1987. *Strategies for Real-Time System Specification*. Dorset House, New York.

ITO, K., AND PARHI, K. K. 1995. Determining the minimum iteration period of an algorithm. *J. VLSI Signal Proc. 11*, 3 (Dec.), 229–244.

JEFFAY, K. 1993. The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *Proceedings of the ACM/SIGAPP Symposium on Applied Computing* (Indianapolis, Ind., Feb. 14–16). ACM, New York, 796–804.

KARP, R. M., AND MILLER, R. E. 1966. Properties of a model for parallel computations: Determinacy, termination, and queueing. *SIAM J. Appl. Math. 14*, 6 (Nov.), 1390–1411.

KOPETZ, H. 1997. *Real-Time Systems*. Kluwer Academic, Boston, MA.

LEE, E. A., AND MESSERSCHMITT, D. G. 1987. Synchronous data flow. *Proc. IEEE 75*, 9 (Sept.), 1235–1245.

MATHUR, A., DASDAN, A., AND GUPTA, R. K. 1998. Rate analysis of embedded systems. *ACM Trans. Design Automat. Electron. Syst. 3*, 3 (July).

MCMILLAN, K. L., AND DILL, D. L. 1992. Algorithms for interface timing verification. In *Proceedings of the International Conference on Computer Design*. IEEE, New York, 48–51.

PUCHOL, C., AND MOK, A. K. 1994. The integration of control and dataflow structures in distributed hard real-time systems. In *Proceedings of the 2nd Workshop on Parallel and Distributed Real-Time Systems* (Apr.). IEEE, New York, 104–107.

SETO, D., LEHOCZKY, J. P., SHA, L., AND SHIN, K. G. 1996. On task schedulability in real-time computer-controlled systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium* (Dec.). IEEE, New York, 13–21.

YEN, T.-Y. 1996. Hardware-software co-synthesis of distributed embedded systems. Ph.D. dissertation. Princeton Univ., Princeton, NJ.

YEN, T.-Y., ISHII, A., CASAVANT, A., AND WOLF, W. 1994. Efficient algorithms for interface timing verification. In *Proceedings of the European Design Automation Conference*. 34–39.

Received February 1998; revised August 1998; accepted September 1998