Rate Derivation and Its Applications to Reactive, Real-time Embedded Systems

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

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

Rajesh K. Gupta
Dept. of Info. & Comp. Sci.
University of California
Irvine, CA 92697
rgupta@ics.uci.edu

Abstract

An embedded system (the system) continuously interacts with its environment under strict timing constraints, called the external constraints, and it is important to know how these external constraints translate to time budgets, called the internal constraints, on the tasks of the system. Knowing these time budgets reduces the complexity of the system's design and validation problem and helps the designers have a simultaneous control on the system's functional as well as temporal correctness from the beginning of the design flow. The translation is carried out by first deriving the rate of each task in the system, hence the term "rate derivation", using the system's task structure and the rates of the input stimuli coming into the system from its environment. The derived task rates are later used to derive and validate the rest of the internal as well as external constraints. This paper proposes a general task graph model to represent the system's task structure, techniques for deriving and validating the system's timing constraints, and a hardware/software codesign methodology that puts everything together.[1]

1 Introduction

An embedded 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 do this interaction under strict timing constraints. As these timing constraints define the timing characteristics of the system's interaction with its environment, i.e., those of the system's external behavior, they are called *external (timing) constraints*.

A typical embedded design flow starts with the requirements specification phase, which describes what the system's external behavior must be without describing how the system works internally. The latter is for the first time expressed at a high level in the architectural design phase by means of a task structure diagram, 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 tasks.

The problem of designing a functionally and temporally correct system is a difficult one. The current practice for this problem is based on trial and error guided by engineering experience [7]. Despite the importance of the temporal correctness, most design methods pay significantly more attention to structural and behavioral aspects of embedded systems than to timing aspects [9]. The main concern in a typical design flow is first to design a functionally correct system and then to focus on its temporal correctness. For example, a typical design flow proceeds in the following four stages: (i) The system is structured into its tasks. (The issues related to task structuring are explained in [9].) (ii) Each task is designed one by one. (iii) The tasks are integrated together to make up the system. (iv) The system is first checked for functional correctness. If it is functionally correct, its temporal correctness is checked. In case of any timing violations, the designers will go back to fine tune or modify some tasks and/or the task structure of the system until all the violations are eliminated or a compromise between the designers and the customer is reached in case some violations are too difficult to eliminate.

However, the above design flow has many problems such as: (i) The designers design the tasks without knowing their time budgets. This seriously hinders the exploration of the design space for better task implementations in terms of performance, cost, and power. (ii) The integration stage occurs very late in the design flow; Any problems detected at this stage are very expensive to correct in terms of design time and cost.

A solution to the above problems should provide the time budgets on the system's tasks very early in the design flow and help the designers keep the system's correctness under control throughout the design flow. This paper offers such a solution. The main idea is to derive the time budgets on the system's tasks, called the *internal (timing) constraints*, from the system's task structure and its external constraints, and to use them to validate the design choices at every stage of the design flow. This reduces the complexity of ensuring temporal correctness from system level to task level and enables the designers to have a simultaneous control on the system's functional and temporal correctness from very early stages in the design flow onwards.

Figure 1 illustrates the above discussion. In this figure, (a) shows the system and its environment. The system gets the stimuli from the environment through its sensors and responds back to the environment through its actuators. This interaction is constrained by the external constraints labeled

[1] This work is supported by an NSF Career Award MIP 95-01615.

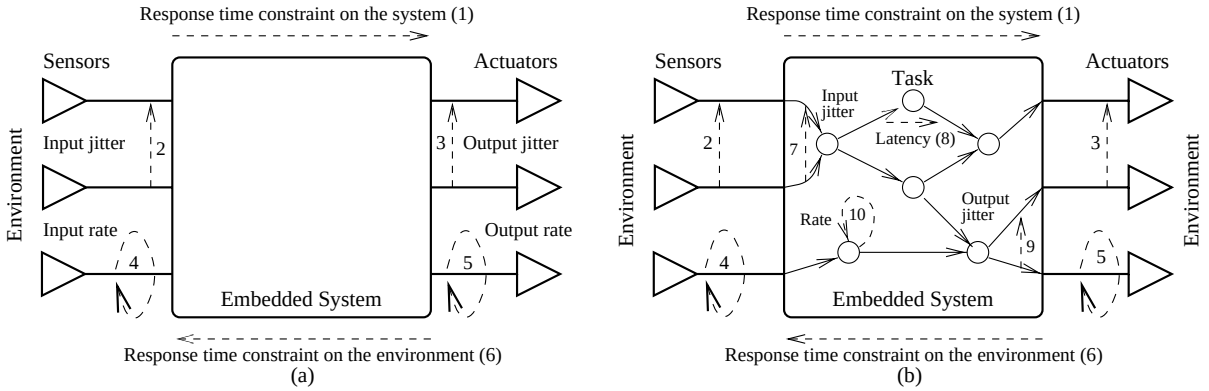

Figure 1: (a) An embedded system with its external constraints (labeled 1-6). (b) The same system with its task structure and internal constraints (labeled 7-10).

1-6, which are the (worst-case) *response time constraints* on the system (1) and on the environment (6), the *input* (2) and *output* (3) *jitter constraints*, and the *input* (4) and *output* (5) *rate constraints*. The jitter constraints define the time separation between two different stimuli or two different responses, whereas rate constraints define the time separation between two successive arrivals of the same stimulus or two successive departure of the same response. Note that the constraints 2, 4, and 6 are imposed on the environment, which we assume are already satisfied, and the constraints 1, 3, and 5 are imposed on the system, which we will validate.

In Figure 1, (b) shows the same system but this time with its task structure and internal constraints. The task structure is described using a task graph of tasks (circles) and their interactions (arcs). The system's internal constraints, called *bounds* for easy reference, are labeled 7-10, which are the *latency bound* (8), the *input* (7) and *output* (9) *jitter bounds*, and *rate bound* (10). Note that these bounds correspond to the time budgets on the tasks and are external to the tasks.

In this paper, we derive the system's internal constraints from its task structure, which is assumed to be given, and its external constraints. However, we do not use all of the external constraints for derivation; we use only the input rate constraints. This means that we assume only that the environment satisfies its timing constraints. We use the input rate constraints to first derive the rate bound of each task, hence the term "rate derivation", and then use them to derive and/or validate the rest of the internal and external constraints.

We perform the derivation under very general task interaction semantics, those that can be found in real-life embedded systems. For this, we propose a task graph model, called the *generalized task graph* model. This model is causality-based and can handle all of the existing causality relations as well as cyclic task dependencies.

The complexity of the internal constraint derivation problem is very high because (i) the external constraints are usually a few in number, (ii) they are defined on sets of tasks rather than for individual tasks, and (iii) tasks can interact in a rather complex manner. To reduce this complexity, this paper assumes that we are given *all* of the system's input rate constraints. Once we know all of them, we do not need the rest of the system's external constraints for derivation; instead, we will validate them. However, if we do not know all of the input rate constraints, we can use some of the system's external constraints as if they are already satisfied.

We perform rate derivation in a hierarchical manner. We first partition the system's task graph into its cyclic components and represent each component using a "super task" in the resulting component graph. We then perform rate derivation in this acyclic component graph using a method called RADHA (RAte Derivation and High-level Analysis). After that, we perform rate derivation in each cyclic component using a method called RATAN (RATe ANalysis). Finally, the results of these two are combined. RADHA and RATAN are also the names of the corresponding software tools [5].

We will present the above methods in a hardware/software codesign methodology. It starts with the system's task structure expressed in a generalized task graph and its external constraints, and ends with its hardware, software, and interface synthesis. This methodology is obtained by incorporating RADHA and RATAN into a standard hardware/software codesign methodology, and gives an overall view of our contributions and also of the organization of this paper. In the rest of this paper, we will first present this methodology (Section 2) and introduce our task graph model (Section 3). After that, we explain RADHA (Section 4.1) and RATAN (Section 4.2) and finally give their applications (Section 5).

2 Proposed Hardware/Software Codesign Methodology

The proposed codesign methodology is given using the flow diagram in Figure 2. The steps in the methodology are labeled by 1-5. The steps 2, 4, and 5 also exist in many other codesign methodologies, e.g., POLIS [2]. The steps 1 and 4 correspond to RADHA and RATAN. RATAN can also be used in step 1 [6]. As RADHA and RATAN will be explained later, we will briefly discuss the other steps now. Step 2 has three substeps: (i) The tasks are written in a hardware description language (HDL) or a programming language at behavioral level. This is actually the first step in the existing codesign methodologies. (ii) These tasks are then partitioned into hardware and software tasks to determine their implementation platforms, e.g., see [11]. (iii) The latency of each task and the task communication delays are estimated [16, 14]. Step 4 synthesizes all the tasks on their respective platforms together with the interface between these platforms [2, 4, 14]. Step 5 performs modifications in the design, e.g., in case the final design does not satisfy its functional and/or temporal correctness requirements. This step shows that the methodology is an iterative one as it should be.

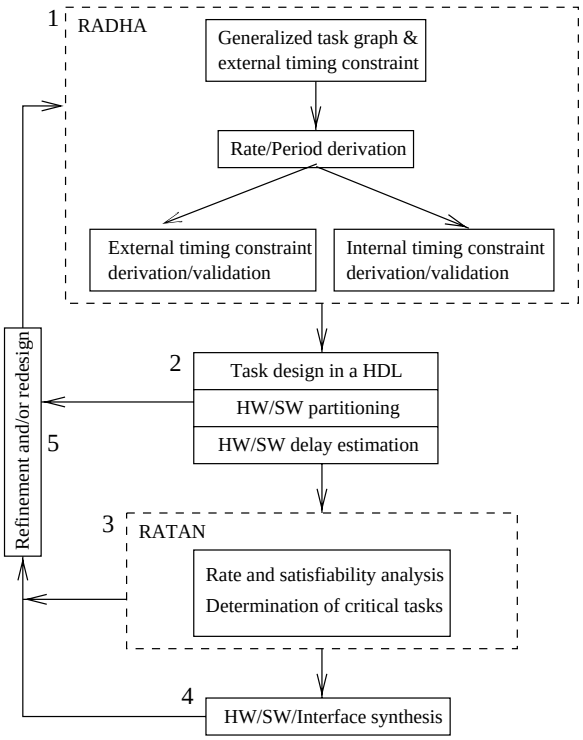

Figure 2: Hardware/software codesign methodology using RADHA and RATAN.

3 Generalized Task Graph Model

We model an embedded system using a hierarchical directed graph called a *generalized task graph* (the task graph for short). This model builds upon the models in [3, 8, 10, 12, 13, 15, 17, 18, 20, 21], as discussed in [6]. Each node corresponds to a task, and each arc corresponds to an asynchronous, unidirectional communication channel between its *producer* and *consumer*. This model actually represents the system's data/control flow diagram where each node is either a data or a control transformation, and each arc is either a data or a control flow, as explained in [9]. The sensors and actuators of the system are also included in its task graph, in which they are referred to as *input* and *output tasks*, respectively.

Each arc prescribes a dependence relation between its end nodes. The hierarchy in the task graph has two levels, the *top* and *bottom levels*, as depicted in Fig. 3. The bottom level consists of strongly connected components (SCCs) of the actual system graph (or the initial task graph) that have at least one arc. The top level is the component graph of the initial task graph such that every node with a self-loop, called a *super task*, corresponds to a SCC at the bottom level, and every node without a self-loop corresponds to the same node in the initial task graph. The hierarchy is mandated by the fact that the rate derivation techniques for acyclic and cyclic components are different. In the sequel, we will refer to each graph or SCC at any level of the hierarchy as "the task graph" within its context. Note that any directed graph can be represented using the above hierarchy.

We assume that each node represents a periodic or sporadic task, each of which has a rate interval bounded from above and below. Recall that a task can have a latency bound, a rate bound, and jitter bounds. The rate of a task gives the number of its executions per unit time. Its reciprocal is equal to the task's *period*. We assume integer values for periods.

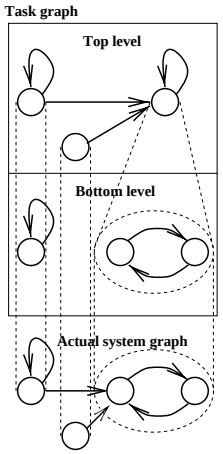

Figure 3: Two-level task graph model.

Both data and control flows through a channel is modeled using token flows. The granularity of every token is *channel-specific*, and once the designers determine the tokens in each channel, they are indistinguishable within and across channels for timing analysis purposes. The task behavior is causality-based such that it uses either AND-, OR-, or AND/OR-causality. Note that each causality relation also implies a timing relation. We say that a task is *enabled* when its predecessors have sent all the tokens that are needed to make the task ready to run. We now classify the tasks by asking three questions:

(i) How many predecessors are needed to enable a task?

1. an AND task needs all of its predecessors,
2. an OR task needs one of its predecessors, and
3. an AND/OR task needs some (more than one) of its predecessors.

(ii) Does a task allow any loss of tokens coming from its predecessors?

1. an AND/unskipped or OR/unskipped task does not allow any loss of tokens.
2. an AND/skipped or OR/skipped task may allow an intentional loss of tokens.

(iii) What happens to the tokens coming from the rest of the predecessors of an OR task?

1. an OR/unskipped task uses all the tokens before it completes.
2. an OR/skipped/joint task ignores the rest completely for its current execution.
3. an OR/skipped/disjoint task uses every token to start a new execution.

The above task types define the input behavior of the tasks. Their output behavior can be different based on what is required of them by their consumers. Moreover, no order is imposed for reading the input channels although a particular order can be imposed using task partitioning. If inputs arrive simultaneously for an OR task, only one of them enables the task.

4 Rate Derivation

Rate derivation refers to deriving the task rates or rate bounds (in terms of rate intervals) at the top and bottom levels. We next use the task rates to derive the task periods because the task periods are actually the ones used for further derivation and validation. As the rate of a task is equal to the reciprocal of its period or vice versa, it is easy to derive one from the other, and so we will use the terms "rate derivation" and "period derivation" interchangeably. We now discuss the rate derivation at the top level followed by the one at the bottom level. We assume that every task in the task graph is concurrent.

4.1 Rate Derivation for Acyclic Graphs

We now present the rate derivation algorithm to derive the rate of each task at the top level. This algorithm is part of RADHA. The algorithm assumes that a task graph and the rate of every input task are given. The algorithm proceeds in topological order of the nodes from the input tasks towards the output tasks. The derivation is done using a rate equation for each task type. This equation is then used to derive period intervals. The running time of the algorithm is polynomial in the size of the task graph; however, the algorithm may use expensive operations such as computing the least common multiple of two integer intervals.

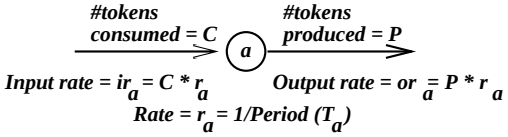

Figure 4: Basic rate relationships.

Figure 4 depicts the definitions of the *input* and *output rates* of a task in relation to its rate as well as the number of tokens produced and consumed. If the period is in seconds, the rate is in executions/seconds, and the input and output rates are in tokens/seconds. We will use these in the algorithm without further definition. The derivations are always done to find the smallest rate and integer period for the task in question and to ensure bounded channel capacities. In the following algorithm, knowing the number of tokens produced/consumed for each channel improves the accuracy of the derivation. Since the granularity of a token is channel-specific, these numbers are relatively easy to obtain from the data/control flow diagram of the system. In case these numbers cannot be obtained, using value 1 for each of them is a safe choice. More examples on how to specify and use these numbers are given in [6].

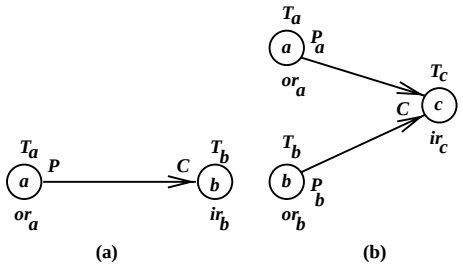

Figure 5: (a) One predecessor case. (b) Two predecessor case.

The rate derivation algorithms is as follows. The rate equations below directly follow from the definition of the corresponding task.

One predecessor: Consider the task graph in Figure 5(a). We want to derive the rate of task b from that of task a. The rate equation is $or_a = ir_b$, which implies that $P/[T_a(l), T_a(u)] = C/[T_b(l), T_b(u)]$, where l and u correspond to the lower and upper bound, respectively.

Two predecessors: Consider the task graph in Figure 5(b). We want to derive the rate of task c from those of tasks a and b. The derivation is different for each task type, as explained below:

1. **AND/skipped.** The rate equation is $r_c = \min\{r_a, r_b\}$, which leads to $T_c(l) = \max\{T_a(l), T_b(l)\}$ and $T_c(u) = \max\{T_a(u), T_b(u)\}$.

2. **AND/ and OR/skipped.** The rate equation is $ir_c = or_a + or_b$, which leads to $T_c(l) = \min\{LCM(i,j)\}$ and $T_c(u) = \max\{LCM(i,j)\}$, where $LCM(i,j)$ denotes the least common multiple of i and j such that $i \in [T_a(l), T_a(u)]$ and $j \in [T_b(l), T_b(u)]$. Here, $LCM(i,j)$ is computed by first enumerating the period intervals for tasks a and b and then using the Euclid's algorithm for the greatest common divisor (GCD), as $GCD(i,j) * LCM(i,j) = i * j$.

3. **OR/skipped/joint.** At least one predecessor of an OR/skipped/joint task generates tokens for it but it is not required that every predecessor of the task will do so. Thus, the rate equation is $\min\{r_a, r_b\} \leq r_c \leq \max\{r_a, r_b\}$, which leads to $T_c(l) = \min\{T_a(l), T_b(l)\}$ and $T_c(u) = \max\{T_a(u), T_b(u)\}$.

4. **OR/skipped/disjoint.** It is difficult to write a rate equation for this case. Instead, we will find the smallest and the largest time separations between token arrivals. A conservative period interval for task c is then $[1, \min\{T_a, T_b\}]$.

Three or more predecessors:

1. **AND or OR tasks.** This is a straightforward generalization of the above derivations.

2. **AND/OR tasks.** This is similar to the AND/skipped case where the "AND part" and the "OR part" behave like tasks a and b. Hence, we derive the period equation directly, which is $T_c = \max\{T_{ANDpart}, T_{ORpart}\}$, where AND and OR parts should be replaced by the actual predecessors.

Note that as the rate equations imply, the above derivations still hold for all the task types except for the case (iv) even if we assume different initial start times for tasks a and b. For that case, the time separation between the initial start times should also be taken into account. Also note that the above derivations also hold when periods are real numbers except for the AND/unskipped case, for which $LCM(i,j)$ should be replaced by $i * j$.

Related Work. Most of the previous works assume that the internal constraints of the system are known in advance [19], and there are surprisingly few works on their derivation [1, 7, 19]. Amongst them, [7] is the most comprehensive one. This work introduces the problem for real-time systems; however, its task semantics is restrictive. Furthermore, due to its assumptions, it is not useful at the level where RADHA is used; [7] can instead be used before scheduling analysis within software synthesis (in step 4) of the codesign methodology in Figure 2.

4.2 Rate Derivation for Cyclic Graphs

We now briefly describe the rate derivation algorithm to derive the rate of each task at the bottom level. This algorithm is part of RATAN, and is explained in detail in [17]. This algorithm assumes that the tasks in a given each SCC at the bottom level are either all AND tasks or all OR/skipped/joint tasks; the case with a combination of tasks is still an open problem [10]. It also assumes that each task starts its very first execution independently. Consider a SCC with only AND tasks. The algorithm uses the fact that the rate of each task in the SCC is the same and is equal to reciprocal of the *maximum cycle mean* of the SCC [17]. The maximum cycle mean of a graph is equal to the maximum of the *cycle means* over all the cycles. The mean of a cycle is defined as the average weight of any arc in the cycle. The weight of an arc is in turn equal to the sum of the latency of its producer and its channel delay. Arc weights are intervals, so are the task rates and periods. A cycle that determines the maximum cycle mean of the graph is called a *critical cycle*. Similarly, the tasks on this cycle are *critical tasks*. Since there are efficient algorithms to find the maximum cycle mean of a SCC [17], we can derive the rate of every task in the SCC. The above definitions are analogous for the OR/skipped/joint case.

5 Applications of Rate Derivation

We now consider the use of the task periods to derive latency bounds and validate response time constraints at the top level, and validate rate constraints at both top and bottom levels. More about the derivation and validation of the other internal and external constraints can be found in [6].

5.1 Deriving Latency Bounds

As a task executes *once* in its period, the lower bound of its period imposes an upper bound on its latency. Since the period is now derived, the designers know an upper bound on the time budget of the task. We need to derive a lower bound for OR/unskipped tasks because such tasks have to wait until they read all the tokens generated for their current execution. Consider Figure 5(b), if task c is an OR/unskipped task, its latency should be greater than $(T_c(u) - \min\{T_a(l), T_b(l)\})$. Note that a self-loop on a task can be used to ensure that the task waits on itself.

5.2 Validating Response Time Constraints

The worst-case response time of the system is at least as large as the longest path delay through the system. Here, the longest path delay is denoted by D, and is obtained using the derived task periods. If the response time constraint is specified a priori, it must be greater than D. This comparison amounts to validating the specified response time constraint.

The algorithm to find L is a modification of the standard longest path algorithm. The modification arises because of the following reason: The delay of a path of tasks in the task graph is not always equal to the sum of the upper bounds of their periods. For example, consider task c in Figure 5(b). Suppose $T_a = 2$ and $T_b = 3$, and that task c is an AND/unskipped task. Then, $T_c = LCM(2, 3) = 6$. The delay of the path with tasks a and c seems to be equal to $T_a + T_c = 2 + 6 = 8$; however, task a has to execute three times for one execution of task c. making the path delay equal to $3 * T_a + T_c = 3 * 2 + 6 = 12$. This reasoning also holds for the other task types.

5.3 Validating Rate Constraints

Recall that the rate derivation algorithms at the top and bottom levels derive the rate of each task (including output tasks). The validation of a rate constraint that is specified a priori on a task is performed by comparing it with the derived rate of the task. If the specified constraint subsumes the derived one, the specified one is said to be satisfied. If not, it is said to be violated. Violations can be eliminated by speeding up or slowing down the tasks involved. Note that for the bottom level, violations can only occur for the critical tasks.

So far, we have not mentioned the relationship between the results of the rate derivation at the top level and those at the bottom level. Recall that the top level has a super task for each SCC at the bottom level. As explained in Section 4.2, a SCC at the bottom level behaves like a single task, justifying the use of super tasks. Consider a super task and its corresponding SCC. The rate derivation algorithm for the top level derives a period interval for the super task. At the same time, the derivation algorithm for the bottom level derives a period interval for the SCC. Now, these two intervals should be the same; otherwise there is a violation. Note that the rate derivation algorithm for the bottom level assumes that the arc weights are known, which is possible after step 2 in the codesign methodology. If they are not known, we cannot derive a period interval for the SCC. However, we can still use the relationship between the super task and its SCC as follows: By the definition of the maximum (minimum) cycle mean of a SCC with all AND (OR/skipped/joint) tasks, the upper (lower) bound of the period interval of the super task defines an upper (lower) bound on the mean of *every* cycle in the SCC. This result can be used to derive additional latency bounds on the tasks at the bottom level.

6 Experimental Results

We have applied the rate derivation algorithms to two examples: the task graph in [7] and the dashboard controller in [2]. The application to the task graph in [7] is discussed in [6]. We now discuss the application to the dashboard controller. The derivation for the dashboard controller will be done using real numbers.

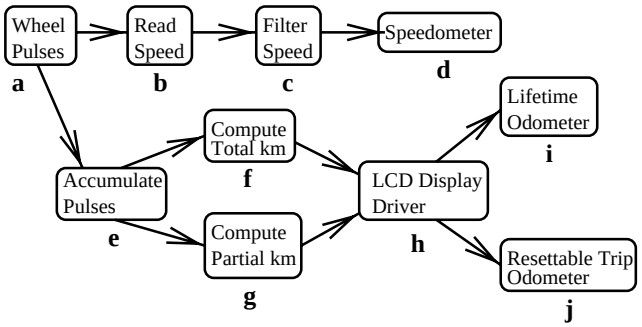

Figure 6: The task graph for the dashboard controller. It consists of the tasks for the speedometer and two odometers.

Figure 6 gives the task graph for the dashboard controller. The speedometer (task d) registers vehicle 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 dashboard controller gets four pulses from task a for every rotation of the tire. The tire travels 0.66 m per rotation.

In the above task graph, task h is an OR/skipped/disjoint task, and the other tasks are all AND/unskipped tasks. As we need the rate of the input coming into the system, we will first derive the period interval for the input task, task a. Task a generates one pulse or one token per one quarter rotation of the tire. The vehicle travels this distance in $(0/66/4\text{ m})/v$ seconds when it travels at a speed of v km/h. As $v \in [5, 260]$ km/h, task a will have the following period interval $T_a = [(0.66m/4)/(260km/h), (0.66m/4)/(5km/h)] = [2.28, 118.80]$ ms. Task b measures the instantaneous speed by counting the tokens coming from task a. To compute a speed value, task b needs at least two tokens. We know this fact because of the functionality of task b although we do not know how this task will be implemented. Hence, the period interval for task b is $T_b = 2 * T_a = 2 * [2.28, 118.80] = [4.56, 237.60]$ ms. Tasks c and d take one speed value from their respective predecessors, process it, and produce another speed value. This means that they have the same period interval as T_b.

The derivation for the odometers proceeds as follows: From the functionality of task e, we know that it needs to accumulate enough tokens to compute an increment of 0.1 km. Now, let us derive the number of tokens that task e needs. As task a produces one token for every 1/4 rotation (0.66 m/4), task e needs to collect $\lfloor 0.1\,km/(0.66\,m/4) \rfloor = 606$ tokens. This amounts to a period interval of $T_e = 606 * T_a = 606 * [2.28, 118.80]$ ms $= [1.38, 71.99]$ s. Tasks f and g have the same period as task e because they need to use one token per execution, which we know from their functionalities. Task h passes every token it gets from task f to task i and every token from task g to task j. Also, it does this every time it gets a token, regardless of the identity of its predecessors. Hence, it is an OR/skipped/disjoint task. We do not know the exact time separation between token arrivals from tasks f and g, but from their period intervals, we know that the smallest time separation can be some $\epsilon > 0$, and that the largest time separation can be 71.99 s, which is the upper bound of the period interval for tasks f and g; thus, $T_h = [\epsilon, 71.99]$ s. Finally, we know that tasks i and j consume one token per execution, so they have the same period interval as task h.

We can validate the above derivations by the implementation parameters given in [2]. One parameter is the period value selected for task b. This value is chosen to be 250 ms in [2]. As the derived period interval for T_b is $[4.56, 237.60]$ ms, we validate this choice. Another parameter is the response time of the odometer. This value is measured to be at most 363 s. The response time that we derive is $5 * 71.99 = 359.95$ s, which is satisfied by the implementation. The advantage of the rate derivation here is that it derives the time budgets far before the implementation, aiding in the design exploration.

7 Conclusions

This paper addresses the problem of deriving the system's internal constraints from its task structure and external constraints so that the designers can keep the system's functional as well as temporal correctness under control from very early stages in the design flow onwards. This paper proposes a generalized task graph model suitable for modeling real-life embedded systems, timing constraint derivation and validation techniques, and a timing-driven hardware/software codesign methodology. This methodology provides all the time budgets before the implementation of the system.

References

[1] Audsley, N. C., et al. Data consistency in hard real-time systems. Tech. Rep. YCS 203, Univ. of York, England, June 1993.

[2] Balarin, F., et al. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach.* Kluwer Academic Publ., Boston, MA, USA, 1997.

[3] Buck, J. T., and Lee, E. A. The token flow model. In *Advanced Topics in Dataflow Computing and Multithreading,* (1993), L. Bic, G. Gao, and J. Gaudiot, Eds., IEEE.

[4] Chou, P., Walkup, E. A., and Borriello, G. Scheduling for reactive real-time systems. *IEEE Micro* (Aug. 1994), 37–47.

[5] Dasdan, A., Mathur, A., and Gupta, R. K. RATAN: A tool for rate analysis and rate constraint debugging for embedded systems. In *Proc. Euro. Design and Test Conf.* (1997), IEEE, pp. 2–6.

[6] Dasdan, A., Ramanathan, D., and Gupta, R. K. A methodology for interactive design and validation of embedded real-time systems. Submitted to the acm trans. on design automation of electronic systems., 1998.

[7] Gerber, R., Hong, S., and Saksena, M. Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Trans. Software Eng. 21*, 7 (July 1995), 579–92.

[8] Gillies, D. W., and Liu, J. W.-S. Scheduling tasks with AND/OR precedence constraints. *SIAM J. Comput. 24*, 4 (Aug. 1995), 797–810.

[9] Gomaa, H. *Software Design Methods for Concurrent and Real-Time Systems.* Addison-Wesley, Reading, MA, USA, 1993.

[10] Gunawardena, J. Periodic behavior in timed systems with AND,OR causality. Tech. Rep. STAN-CS-93-1462, Stanford Univ., 1993.

[11] Gupta, R. K. Special issue on partitioning methods for embedded systems. *Design Automation for Embedded Systems 2*, 2 (Mar. 1997), 123–261.

[12] Jeffay, K. The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *Proc. ACM/SIGAPP Symp. on Applied Computing* (Feb. 1993), pp. 796–804.

[13] Karp, R. M., and Miller, R. E. Properties of a model for parallel computations: Determinacy, termination, and queueing. *SIAM J. Appl. Math. 14*, 6 (Nov. 1966), 1390–1411.

[14] Ku, D., and Micheli, G. D. *High Level Synthesis of ASICs Under Timing and Synchronization Constraints.* Kluwer Academic Publ., Boston, MA, USA, 1992.

[15] Lee, E. A., and Messerschmitt, D. G. Synchronous data flow. *Proc. IEEE 75*, 9 (Sept. 1987), 1235–45.

[16] Malik, S., Martonosi, M., and Li, Y.-T. S. Static timing analysis of embedded softare. In *Proc. 34st Design Automation Conf.* (1997), ACM/IEEE, pp. 147–52.

[17] Mathur, A., Dasdan, A., and Gupta, R. K. Rate analysis of embedded systems. *ACM Trans. on Design Automation of Electronic Systems 4*, 2 (Apr. 1999).

[18] Puchol, C., and Mok, A. K. The integration of control and dataflow structures in distributed hard real-time systems. In *Proc. 2nd Wrkshp on Parallel and Distributed Real-Time Syst.* (Apr. 1994), IEEE, pp. 104–7.

[19] Seto, D., et al. On task schedulability in real-time computer-controlled systems. In *Proc. 17th IEEE Real-Time Systems Symp.* (Dec. 1996), pp. 13–21.

[20] Yakovlev, A., et al. On the models for asynchronous circuit behavior with OR causality. *Formal methods in System Design 9*, 3 (Nov. 1996), 189–233.

[21] Yen, T.-Y. Hardware-software co-synthesis of distributed embedded systems. PhD thesis, Princeton Univ., 1996.