Abstract

We present our view of the high-level timing issues in the design and validation of embedded real-time systems. We first define the derivation problem: the problem of deriving internal timing constraints from external timing constraints in an embedded real-time system. We then give a comprehensive classification of timing constraints, discuss the state of the art on high-level system modeling and on the timing constraint derivation techniques. We finally give some pointers for future research.

1 Introduction

Most of the embedded systems are real-time in nature in the sense that their correctness depends on their functional as well as timing correctness. As a result, timing is the most precious resource to manage in such systems [28]. The design of such systems usually follows a life-cycle model [11]. Irrespective of the particular life-cycle model used, the following phases are usually the first three phases in the life cycle of an embedded system (the system for short): the requirements specification phase, the architectural design phase, and the detailed design phase [11]. In the requirements specification phase, the users' requirements from the system are identified and analyzed. In the architectural design phase, the system is structured into its components, each of which we refer to as a task. Since a real-time system has many activities occurring in parallel, these tasks are typically concurrent tasks. Finally, in the detailed design phase, the algorithmic details of each task are defined.

The requirements specification phase describes what the system's external behavior is without describing how the system works internally. The latter is described in the architectural design phase in terms of tasks. The detailed design phase in turn describes how each task works internally. Let us focus only on the timing aspects of embedded systems. Then, from the requirements point of view, the users are only concerned with external timing in the sense that they are concerned only that the system will respond to a certain stimulus with certain time constraints, called the external (timing) constraints or end-to-end constraints, but whether that response was achieved through a background or foreground task, how it was scheduled relative to other tasks, what the internal port-to-port timing was, and so on are issues that do not concern them [15]. These are the issues that the designers have to deal with. Of course, the designers are also concerned with external timing from the requirements point of view because this, after all, is the timing that the system will be evaluated upon; however, from the design point of view, the designers have to find out how to ensure that the system satisfies its external timing constraints when it is designed task by task. The issue then becomes a task-based timing issue or an internal timing issue, and the constraints become the internal (timing) constraints or the intermediate timing constraints. An illustration of these timing constraints is given in Fig. 1.

The problem that the designers face is called the derivation problem. We define this problem more precisely as follows: Given the system's external timing constraints and task structure, derive its internal timing constraints such that the satisfaction of the internal timing constraints implies the satisfaction of the external timing constraints from which they have been derived, and validate the remaining external constraints. A solution to this problem reduces the complexity of the problem of satisfying the system's timing requirements, which we will call the timing problem, from system-level to task-level. The designers will not have to wait until all the tasks are designed and integrated into the system to test the system's timing performance; they will be able to test it at every phase of the design life-cycle from the architectural design phase on. Note that the derivation problem is our definition of the timing problem although we think that it is a very reasonable definition.
The current practice for the timing problem is based on trial and error guided by engineering experience [9]. Typically, the designers will first focus on designing a functionally correct system. They will later test it for its timing performance. If the system does not satisfy its timing requirements, then there will be many fine tunings and rewrites, making the system increasingly more difficult to understand, maintain, and extend. Apart from the fact that the complexity of the timing analysis problem is enormous, the reason for the lack of systematic timing analysis approaches seems to be the fact that neither design methods nor real-time programming languages for the design of real-time systems provide sufficient support for timing constraints [11, 14]. For example, Gomaa [11] states “despite the importance of timing constraints, it is a characteristic (and limitation) of most software design methods for real-time systems that the methods tend to emphasize structural and behavioral aspects of real-time systems and generally pay significantly less attention to timing constraints”. Moreover, almost all of the previous works about real-time systems assume that the characteristics of tasks, i.e., their timing constraints such as their periods and execution times, are known [24, 27].

This paper addresses the derivation problem from three facets: the possible timing constraints, system modeling, and derivation techniques. We give a comprehensive classification of all the possible timing constraints in § 2. In § 3, we discuss a task graph model that generalizes the existing embedded system models in various ways. We present the state of the art on the derivation problem in § 4, followed by a brief outline of our methodology in § 5. Our methodology is based on two tools, called RADHA [8] and RATAN [7]. RADHA stands for RAt e Deriv ation and High-level Analysis, and RATAN stands for RAt e ANalysis. We conclude this paper in § 6.

## 2 Timing Constraints

A 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. A timing constraint can be either a rate constraint or a separation constraint, each of which can be a maximum or minimum constraint [5], implying the use of an interval to represent each. A rate constraint imposes bounds on the rates at which an input is sampled, an output is generated, or a task is executed; hence, it is a time constraint on the delay between two successive occurrence times of the same event. A separation constraint, on the other hand, imposes bounds on the delay between occurrence times of two different events.

Let S denote a sensor and R an actuator, as in [6]. Four combinations of these symbols are possible: S-S, S-R, R-S, and R-R, each of which corresponds to one or more timing constraints as explained below. Then, we have the following classification, built upon but more comprehensive than [5, 6]. These timing constraints are identified by a number in Fig. 1, to which we will refer in the sequel.

1. External timing constraints: These are imposed between the external inputs and outputs of the system.
   (a) Timing constraints on the system
      i. S-R is a (worst-case) response time constraint on the system (1).
ii. R-R is either a rate constraint (5) or a jitter constraint (3).

(b) Timing constraints on the environment
i. S-S is either a rate constraint (4) or a jitter constraint (2).
ii. R-S is a (worst-case) response time constraint on the environment (6).

2. Internal timing constraints: These are imposed on the system's tasks. They are still external to the tasks. These constraints are sometimes referred to as the task characteristics. They are: a rate bound (10), an input (7) and output (9) jitter bounds, a latency or execution time bound (8), and an arbitrary separation bound (11).

Note that we have used the term “bound” to refer to the internal timing constraints. Tasks can have other characteristics such as start time, finish time, deadline, and period, e.g., see [29] for definitions. Most common tasks in an embedded system are periodic tasks, i.e., those that execute at regular intervals, and the other tasks can be modeled using periodic tasks [29]. Hence, we will assume that all the tasks are periodic in the sequel. The rate and period of a (periodic) task is related so that their product is equal to one; its rate gives the number of its executions per unit time whereas its period gives the length of the regular interval during which it executes once.

3 System Models

As explained in [11], during the requirements specification phase in the life cycle, the system is represented using a system context diagram. This diagram, as in Fig. 1(a), defines the boundary between the system to be developed and the external environment. In the architectural design phase, the system is decomposed into its concurrent tasks, and the interfaces between them are defined in the form of data and control flows. The system is structured as a hierarchical set of data flow/control flow diagrams (graphs), as in Fig. 1(b). These graphs can also be used to model task internals. Data flows may be either discrete (more common) or continuous. Control flows are discrete signals that have no data value. They are used to signal that some action has happened or to initiate a command. We will say that a data or control flow enables the task that it is an input to. From a performance analysis point of view, both data and control flow can be modeled as token flows [26], and the task interactions can be modeled in such a way that the behavior of tasks depends upon the timing of token arrivals [3]. Furthermore, tasks can be combined using the Boolean operators AND, OR, and NOT in order to model a real-life embedded system. We now discuss our model called the generalized task graph. A more formal discussion is available in [8].

The task structure of the system is represented by a directed graph, corresponding to the generalized task graph, whose nodes and arcs correspond to tasks and their interactions through asynchronous, unidirectional communication channels, respectively. An arc represents a precedence constraint. If one task does not precede another in any directed path, then they are concurrent. Concurrent tasks can be serialized using exclusion constraints [29] and priority constraints. Exclusion and priority constraints are used to model mutual exclusion (or NOT) and task criticality, respectively. Note also that precedence constraints can model synchronization. Priority constraints impose a particular order between their tasks whereas exclusion constraints do not; hence, priority constraints can be modeled using precedence constraints.

We associate the Boolean operators AND and OR with the tasks. We have a total of three task types: AND, OR, and AND/OR. These operators model task interactions, which can typically be quite complex in real-life embedded systems [15]. The semantics of the task types is determined based on how many of their inputs are enough to enable them. They can use one (OR task), all (AND task), or some (AND/OR). An AND task needs to wait to get inputs from all of its predecessors before running whereas an OR task can start running as soon as it gets one input. An AND/OR task is a combination of both. They can further be refined by asking the following questions: Do AND and OR tasks skip any tokens? What happens to the remaining inputs of an OR task after it is enabled? The first question leads us to define “skipped” and “unskipped” task types in the sense that the former may skip some tokens whereas the latter never does. These task types apply to both AND and OR, so we have AND/skipped, AND/unskipped, OR/skipped, and OR/unskipped task types. The second question leads us to define “joint” and “disjoint” task types, which applies only to OR/skipped tasks. An OR/skipped/joint task ignores the remaining inputs for the current and any future executions whereas an OR/skipped/disjoint task ignores them for the current execution but each of these inputs is used to enable one future execution of the task.

In previous works, the task structure of an embedded system is typically represented an AND model (in which every task is an AND task), e.g., see [16, 18, 19, 22, 31]. Among these, only [18, 19, 22] allow cyclic dependencies between tasks. Since AND
models do not model real-life embedded systems properly [26], OR behavior is next introduced into the task graphs in [3, 8, 10, 12, 21, 25, 26, 30]. The skipped and unskipped concepts, and the joint and disjoint concepts are introduced in [10] and [30], respectively. Petri nets [23] can model both AND and OR behaviors but have some difficulty in handling all the different OR behaviors [30]. Our earlier work [8] is the first generalization that combines all of the above concepts with AND and OR. It also discusses other important issues such as task and arc properties as well as the use of hierarchies in the task graph to handle cyclic dependencies.

4 Deriving Timing Constraints

The search space for the problem of deriving timing constraints, i.e., the solution to the derivaton problem, is very large [9]. For example, assuming that the path with tasks \( a, b, \) and \( c \) determines the response time of 1000 ms of the system in Fig. 1(b), there are over half a million solutions to the equation \( a + b + c = 1000 \) where \( a, b, \) and \( c \) are the delays through each of the tasks, respectively. We now give an overview of techniques to solve such problems. Despite the magnitude and importance of the problem, there are surprisingly few works on it, and all of them are very recent: [2, 4, 9, 17, 27] and our work in [8]. Moreover, only [8, 9] discuss the derivation problem as defined in this paper.

Consider Fig. 1(b). The derivation problem has many dimensions. As [9] and [8] cover all of these dimensions, we will evaluate them in terms of the approaches in these two works. The first dimension is related to which timing constraint(s) of a system to use as the base of the derivation. Both [9] and [8] derive all the internal timing constraints out of task periods. This seems to be an effective approach.

The second dimension is related to how much to assume about the system and its environment. We assume only that the environment meets its timing requirements in the sense that we assume only that the input rate constraints (4) and the response time constraint on the environment (6) are known. We use input rate constraints to derive the system's all internal constraints, and then use the derived constraints in turn to validate the rest of the external constraints because we claim that the input rate constraints and the task structure dictate the timing performance of the system. [9], on the other hand, assumes that all the external constraints except the input rate constraints (4) are known, [9] uses all of them to derive the system's internal constraint as well as its input rate constraints. However, this approach in [9] makes the search space unnecessarily large. To reduce the size of the search space, this approach relies on additional assumptions such that the task execution times are known and that the task periods over a directed path in the task graph have common divisors. Unfortunately, these assumptions restrict the applicability of this approach because, for instance, the task execution times cannot be known at such a high level in the life-cycle.

The third dimension is related to what happens if some of the external constraints are left unspecified, i.e., those specified using “on demand”, “as fast as possible”, “reasonably fast”, etc. In that case, they can be represented using symbolic variables, and symbolic analysis techniques, e.g., see [1] for the application symbolic analysis to a related problem, can be used for the derivation to determine how fast is “as fast as possible”. We did not use symbolic analysis techniques in [8] because we didn’t need it as we assume that all the input rates are known. However, it turns out that even if all of the external timing constraints are known, the approach in [9] still needs to use symbolic analysis techniques.

The fourth dimension is related to how many of the task types to use in the derivation. Including both AND and OR task types and having cycles in the task graph complicates the derivation problem, which is why [4] uses independent tasks, and [9, 22, 27] use only AND tasks. We use all the task types in the generalized task graph for the derivation of timing constraints for acyclic task graphs. For cyclic components in the task graph, we assume only AND or only OR task types. Both task types are considered together only in [12, 20]; however, these works do not discuss the derivation problem but the performance analysis problem of cyclic systems, which is a related problem as identified in [8].

5 Methodology

Our methodology is illustrated in Fig. 2. This methodology covers the entire life-cycle of an embedded system. Note that we have identified the places where our tools, RADHA [8] and RATAN [7], can be used. In our task graph model, each cyclic portion (strongly connected component) is collapsed into one big task such that the task graph becomes acyclic. Then, RADHA derives the task periods for the acyclic portions whereas RATAN does it for cyclic portions. These tools have other uses: RADHA derives all the other internal constraints and validates all the external constraints; RATAN verifies the task periods after estimations on delays in the task graph are determined, and determines the critical tasks in the cyclic portions, which are those that determine the perfor-
Figure 2: The proposed design methodology for embedded systems using RADHA and RATAN.

6 Conclusions and Future Work

Despite the importance of the derivation problem, the state of the art unfortunately contains only a few works. To give the researchers a starting point in this paper, we have presented an overview of the problem and all the related issues: timing constraints, system modeling, and solution techniques. Finally, we identify the following for future research: improving the derivation techniques with simulation and designer interaction, handling all of the above dimensions of the derivation problem in a unified framework, and utilizing the derivation techniques in a hardware/software co-design environment.

References


[3] BUCK, J. T., AND LEE, E. A. The token flow model. In Advanced Topics in Dataflow Computing and Mul-
tithreading, (1993), L. Bic, G. Gao, and J. Gaudiot, Eds., IEEE.


