Independent Study

Schedulability Analyzer for AcmeStudio

Yong Hoon Choi
<yhchoi@cs.cmu.edu>

Institute for Software Research International
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

November 5, 2003
# Contents

1 Acronyms 3

2 Introduction 4

3 Introduction to Real-Time Systems 5

3.1 What are Real-time Systems .................................................. 5

3.2 Real-Time System Design Paradigms [Saksena 99] .......................... 6

3.2.1 Time-Driven Style ......................................................... 6

3.2.2 Event-Driven Style ......................................................... 6

3.3 Challenges in Real-Time System Designs [Isovic] ........................... 6

3.3.1 Concurrency .............................................................. 7

3.3.2 Dynamic Behavior ......................................................... 7

3.3.3 Load ................................................................. 7

3.3.4 Memory and Processing Capacities ...................................... 7

3.4 Real-Time Scheduling Algorithms ........................................... 7

3.4.1 Characteristics of Scheduling Algorithms .............................. 7

3.4.2 Examples of Scheduling Algorithms .................................... 8

4 Software Architecture for Real-Time Systems 10

4.1 Target System ............................................................... 10

4.2 Constraints on the Real-Time Software Architecture ........................ 11

4.3 Architectural Style for Real-Time Systems .................................. 11

4.3.1 Architectural Elements .................................................. 11

5 Schedulability Analysis for Real-Time System Style 17

5.1 Schedulability Analysis on Centralized Real-Time Systems ............... 17

5.1.1 Overview of Centralized Real-Time Systems .............................. 17

5.1.2 Independent Periodic Tasks ............................................. 17

5.1.3 Periodic Tasks Sharing Multiple Data Resources ........................ 19

5.2 Schedulability Analysis on Distributed Real-Time Systems ............... 22

5.2.1 Overview of Distributed Real-Time Systems .............................. 22

5.2.2 Single Processor Schedulability Analysis ................................ 22

5.2.3 Communication Network Schedulability Analysis ........................ 22

5.2.4 End-To-End Schedulability Analysis ..................................... 24
6 Example Application

6.1 Description of System ................................................. 25
6.2 Description of Software Architecture ................................. 25
6.3 Schedulability Analysis .................................................. 27
6.3.1 CPU/Network Utilization ........................................... 27
6.3.2 Task/Message Schedulability ....................................... 27
6.3.3 End-to-End Schedulability .......................................... 27

7 Future Work

7.1 Analyzing Schedulability for Aperiodic Tasks ......................... 29
7.2 Adopting Fiber Distributed Data Interface for Distributed Computing ............................... 29
7.3 Modifying Properties of Architectural Elements ....................... 29
7.3.1 Adding the Special Prefix "rma-" ................................ 29
7.3.2 Adding Schedulability Result Properties ........................ 30
7.4 Enhancing Usability of the Analysis Plug-In Tool ...................... 30
7.4.1 Allowing Easier Specification of Execution Paths ................. 30
7.4.2 Marking Unschedulable Tasks/Messages .......................... 30
Chapter 1

Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADL</td>
<td>Architecture Description Language</td>
</tr>
<tr>
<td>ABLE</td>
<td>Architecture Based Languages and Environments</td>
</tr>
<tr>
<td>RMA</td>
<td>Rate Monotonic Analysis</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>PI</td>
<td>Priority Inversion</td>
</tr>
<tr>
<td>PCP</td>
<td>Priority Ceiling Protocol</td>
</tr>
<tr>
<td>EDF</td>
<td>Earliest Deadline First</td>
</tr>
<tr>
<td>RM</td>
<td>Rate Monotonic</td>
</tr>
<tr>
<td>MUF</td>
<td>Maximum-Urgency-First Algorithm</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>FDDI</td>
<td>Fiber Distributed Data Interface</td>
</tr>
</tbody>
</table>
Chapter 2

Introduction

As embedded real-time systems are commonly used in everyday life, real-time computing are becoming more and more critical to the information technology industry. Their applications range from simple domestic devices, such as microwaves, washing machines, and electronic gatekeepers, to highly critical systems, such as nuclear plant control, air traffic control, and medical control systems. Since these real-time systems are spreading to more and more new domains, the scope, complexity and criticality of the systems are increasing rapidly.

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. [Bass 97] Developing a software architectural model is seen as a crucial step for producing high quality software. In addition to allowing designers to focus on high-level abstractions such as computational components and their interactions, an architectural model is suitable for analyses that can prevent errors from propagating to later phases of software development. [Schmerl 03] As the application domain of real-time systems grows rapidly, there have been several approaches to apply software architecture modeling to the design of real-time systems. [Vestal 94] [Muller 01] [Wall 00] [Dave 98] [Urting] [Stewart 96] [Zalewski 01] [Locke 99] [Tesanovic 03] [Dionisio 03] [MetaH 98]

The AcmeStudio is an editing environment and visualization tool for software architectural designs being developed by the ABLE group at CMU. Its software architectural designs are based on the Acme architectural description language (ADL), which is a simple, generic software architecture description language that can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools.

This project has two goals. One goal is to introduce a new software architectural style for real-time systems, as a family in the Acme ADL. The architectural style will define the vocabulary of elements that can be used in the domain, rules specifying how these elements may be assembled, and how they should be depicted in architectural diagrams. [Schmerl 03] The other goal is to provide a schedulability analysis tool that the real-time architectural style is coupled with as a plug-in extension for the AcmeStudio.
Chapter 3

Introduction to Real-Time Systems

3.1 What are Real-time Systems

A canonical definition of a real-time system from Donald Gillies is the following:

A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred.

These systems respond to a series of external inputs, which usually arrive in an unpredictable manner. The real-time systems process these inputs, take appropriate actions and generate output necessary to control the peripherals connected to them. It is essential that the timing constraints of the system are guaranteed to be met.

Based on the type of timing constraints, real-time systems can be categorized into the following: [Kopetz 00]

- **Soft real-time systems**: these are systems where a failure to meet a specified deadline reduces the utility of the result, but does not lead to a significant financial loss. An example of such a system is a letter sorting machine: If a letter is dropped into the wrong box because of a timing failure of the computer, no serious consequences will result.

- **Hard real-time systems**: these are systems where a failure to meet a specified deadline can lead to catastrophic consequences. An example of a hard real-time application is a computer system controlling a railway-shunting yard: If a wagon is released to the wrong track because of a timing failure of the computer, a serious accident may happen.

- **Firm real-time systems**: It has a combination of both hard and soft timeliness requirements. The scheduling result has no utility outside deadline window, but the system can withstand a few missed results. The computation has a shorter software requirement and a longer hard requirement

Most real-time systems interface with control hardware directly. The software for such systems is mostly custom-developed. Real-time applications can be either embedded applications or non-embedded applications. Real-time systems usually have a customized version of peripherals instead of traditional ones such as keyboards, mouse, and monitors.
3.2 Real-Time System Design Paradigms [Saksena 99]

There are two main paradigms for real-time system designs based on the way to represent a style of developing a real-time application and especially to model concurrent activities in the systems.

3.2.1 Time-Driven Style

The time-driven design style has been developed to model primary functional scenarios of a real-time system, and to make these designs easily analyzable for timeliness. It is based on a simple concurrency model, where a task is activated by an event trigger. Upon arrival of event trigger, it performs some computation and then awaits the arrival of the next event-trigger. The event trigger may be connected to an external interrupt source, a periodic timer, signal from another task, etc. Often timers are the main event sources to ensure predictable workload. The communication between these tasks is based on the shared memory model through shared protected data objects.

The time-driven style is naturally suitable for dealing with regular and recurring input signals, where the processing of such signals may flow through a number of processing steps (represented as tasks) before generating output signals. The main advantage of the time-driven style is that it maps directly to real-time scheduling models and thus such designs are highly amenable to schedulability analysis. The tasking model, however, is relatively so simple that complex task behaviors cannot be represented.

3.2.2 Event-Driven Style

The event-driven design style has been developed mainly to deal with unpredictable asynchronous events. The event-driven software is structured around event-handling code. An event triggers the appropriate event-handling code, and when the action is complete, the software enters dormant state awaiting the next event. Since event may arrive while a previous event is being processed, the software also allows for events to be queued for future processing. The system must respond to asynchronous events in the external world, and the reaction must depend on the system state. An event-driven system is often modeled as finite state machine, where the arrival of an event triggers transitions in the state machine, and includes the event handling code.

The main advantage of this style over the time-driven style is that it has mechanisms to model complex system behavior arising often from control aspects. This is manifested in a more general behavioral specification of concurrent activities. This generality, however, results in a less clear understanding of the data-flows in a system, making it more difficult to perform schedulability analysis.

3.3 Challenges in Real-Time System Designs [Isovic]

Designing reusable real-time systems is more complex than designing software systems in general case. This complexity arises from several aspects of real-time systems that do not appear in non-real-time systems. Embedded real-time targets are slower and have less memory, yet must still perform within tight deadlines. They must often run continuously for long periods of time, for instance cardiac pacemakers which last up to 10 years, or unmanned space probes which must function properly on their own for decades.
3.3.1 Concurrency

Single processor systems can do only one single thing at a time and therefore they must implement a scheduling policy that controls when the different tasks execute. Distributed real-time systems usually implement true concurrency, which means that they must support simultaneous execution of several high-level tasks. Concurrent real-time systems are extremely complex to specify/develop/validate because many independent operations occur at the same time.

3.3.2 Dynamic Behavior

The correct operation of a real-time system depends not only on the logical result of computation, but also on the time at which those results are produced. Hard real-time systems must produce functional results by a specific deadline. Otherwise, there may be catastrophic consequences for both the system and the environment it operates in. It is often impossible or astonishingly difficult to predict with certainty when particular events will occur, what their order or occurrence will be, and how long they will last. Yet real-time systems must respond to events within a specified, bounded time.

3.3.3 Load

The load on a real-time system that comes from its external environment is another source of complexity. For example, a telephone switch that must be designed to handle the stress on the Christmas’s Eve and yet be cost-effective to operate. This often implies developing a priority-driven system that drops less essential tasks under big load to the system. To be able to address this issue, real-time components must be designed to handle exceptional situations.

3.3.4 Memory and Processing Capacities

Embedded real-time systems are usually constructed with the last powerful computers that can meet the functional and performance requirements. Providing smaller CPU with less memory lowers the manufacturing costs. Embedded targets are slower and have less memory, yet must still perform within tight deadlines. Clearly, real-time developers need to ensure they are using available resources as efficient as possible.

3.4 Real-Time Scheduling Algorithms

In this section, concepts and examples of real-time scheduling algorithms will be introduced.

3.4.1 Characteristics of Scheduling Algorithms

A scheduling algorithm is a set of rules that determine the task to be executed at a particular moment. [Liu 73] More specifically, real-time scheduling is the process of creating start and finish times for sets of tasks such that all timing, precedence, and resource constraints are met.

CPU (Central Processing Unit) is the most common shared resource for software systems. The scheduling policy for a CPU is generally implemented by an operating system. Traditionally, a scheduling policy is a set of rules that determines which of all the actions that are ready to execute will be allocated the CPU. In distributed environments, however, scheduling problems deal with
CPU as well as communication network.

There are various approaches to real-time scheduling such as table-driven, priority-driven, planning-based, and best-effort approaches. [Ramamritham 94] In the rest of this paper, only priority-driven approaches will be considered. This means that whenever there is a request for a task that is of higher priority than the one currently being executed, the running task is immediately interrupted and the newly requested task is started. Thus the specification of such algorithms amounts to the specification of the method of assigning priorities to tasks.

**Static versus Dynamic**

A scheduling algorithm is said to be static if priorities are assigned to tasks once and for all. A static scheduling algorithm is also called a fixed priority scheduling algorithm. A scheduling algorithm is said to be dynamic if priorities of tasks might change from request to request. A scheduling is said to be a mixed scheduling algorithm if the priorities of some of the tasks are fixed yet the priorities of the remaining tasks vary from request to request. [Liu 73]

**Offline versus Online**

A scheduling algorithm is offline if all scheduling decisions are made prior to the running of the system. A table is generated that contains all scheduling decisions for use during run-time. This relies completely upon a priori knowledge of process behavior. A scheduling algorithm is said to be online if scheduling decisions are made during the run-time of the system. It can be either static or dynamic. The decisions are based on both process characteristics and the current state of the system. [Audsley 90]

### 3.4.2 Examples of Scheduling Algorithms

**Earliest-Deadline-First Scheduling Algorithm (EDF)**

The earliest-deadline-first (EDF), which is a preemptive and dynamic algorithm interprets the deadline of a task as its priority. [Liu 73] The task with the earliest deadline has the highest priority, while the task with the latest deadline has the lowest priority. The advantage of this algorithm is that 100% processor utilization is possible. The main disadvantage is no way to guarantee which tasks will fail in a transient overload. It is possible that a critical task will fail at the expense of a lesser important task.

**Minimum-Laxity-First Algorithm (MLF)**

The minimum-laxity-first algorithm assigns a laxity to each task in a system, then selects the task with the minimum laxity to execute next. Laxity is defined as follows:

\[
Laxity = \text{deadline time} - \text{current time} - \text{CPU time needed}
\]

Laxity is measure of the flexibility available for scheduling a task. A laxity tl means that even if the task is delayed by tl time units, it will still meet its deadline. A laxity of zero means that the task must begin to execute now or it will risk failing to meet its deadline.
The main difference between MLF and EDF is that MLF takes into consideration the execution time of a task, which EDF does not do. Like EDF, MLF has a 100% schedulable bound and there is no way to control which tasks are guaranteed to execute during a transient overload. [Stewart 91]

**Rate Monotonic Scheduling Algorithm (RM)**

Rate monotonic scheduling (RMS) was presented in the paper of Liu and Layland [Liu 73] that provided mathematical conditions for assessing the schedulability of a set of tasks. These results could be used only if all tasks are periodic, with deadline at the end of each period, perfectly preemptible, independent, and scheduled using the rate monotonic scheduling algorithm. It assigns the highest priority to the task with the highest rate (or with the shortest period) and assigned the priorities of the remaining tasks monotonically in the order of their rates.

One problem with the rate monotonic algorithm is that the schedulable bound is less than 100%. The schedulable bound of a task set is defined as the maximum CPU utilization for which the set of tasks can be guaranteed to meet their deadlines. Since the schedulable bound for the critical set is as low as 69%, a scheduler might not make as efficient use of limited CPU resources for a particular task set as can dynamic priority scheduling algorithms, which can have a schedulable bound of 100%. [Stewart 94]
Chapter 4
Software Architecture for Real-Time Systems

4.1 Target System

The target system is characterized by the following properties:

- **Hard Real-Time**
  Missing an important deadline in the system can cause catastrophic consequences. [Stankovic 96]

- **Rate Monotonic Scheduling Algorithm**
  The rate monotonic scheduling (RMS), the best known preemptive fixed-priority scheduling algorithm, is used for scheduling periodic tasks, potentially, as well as aperiodic tasks when they are encapsulated into periodic tasks. The scheduler of the system assigns highest priority to the task with the highest rate (or with the shortest period) and assigns the priorities of the remaining tasks monotonically in the order of their rates. [Klein 93]

- **Priority Ceiling Protocol for Preventing Priority Inversion**
  Priority inversion (PI) occurs when a high priority task is blocked by a low priority task, such as a shared semaphore or a protected operation. The blocking time of a task, which is the duration of priority inversion, is an important factor used for calculating the worst-case response time of the task. Some situations, such as chained resource locking, may make the blocking time unbounded. The priority ceiling protocol (PCP), which is an effective technique for minimizing priority inversions in real-time scheduling, is used for task synchronization to shared common data. [Yue 1996]

- **Centralized or Distributed Environment**
  The target system ranges from a centralized uni-processor real-time system to a distributed multi-processor real-time system.

- **Controller Area Network (CAN) for Distributed Computing**
  Controller area network (CAN) is extensively used in small scale distributed systems, such as automobile, medical and industrial applications. [CAN] It contains embedded networking bus that arbitrates between messages on the bus by using priorities. Each message is tagged with a unique priority which serves to identify the message. There are analysis approaches proposed
4.2 Constraints on the Real-Time Software Architecture

There are constraints on the software architecture to be designed for the real-time systems as follows:

- The real-time software architectures should be able to be designed by using the AcmeStudio. That means that the software architectures should be represented in the Acme architectural description language (ADL). [Garlan 00]

- Schedulability analysis by using the Rate Monotonic Analysis (RMA), should be applicable to the software architecture.

4.3 Architectural Style for Real-Time Systems

An architectural style defines a family of software systems in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of components and connector types, and a set of constraints on how they can be combined. [Shaw 96]

4.3.1 Architectural Elements

This section will present the description of architectural elements for real-time software systems including their name and topological/logical rules. The features for architectural elements in architectural diagrams are as shown in Figure 4.1.

Components

Components represent the primary computational elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of software architectures. Typical examples of components include clients, servers, filters, objects, blackboards, and databases. [Monroe 98]

The traditional component models, such as filters in pipe-and-filter architectures and tiers in tiered architectures, are not appropriate for real-time systems since they cannot provide the timing behaviors in real-time scheduling. The real-time system can be constructed out of sequential programs,
but most typically they are built of concurrent programs, called tasks. Since a task, which could be either a process or a thread, is the smallest element in scheduling or concurrency, a real-time component should not be smaller than one task. For this reason, a real-time component, as the most fine-grained reusable unit, is at least a task in many real-time system design approaches. [Iovic] [Vestal 94] [Alvarez 03] [Locke 99]

The only component the current project deals with is the Periodic Task component.

- (Periodic) Task
  The (Periodic) Task \(^1\) component implements the response to a periodic event and becomes ready to execute at fixed intervals while a periodic tasks implements the response to an aperiodic event and becomes ready to execute irregularly.

This component may have the Write/Read ports as interfaces for communication via shared data within a CPU node and the Send/Receive ports as interfaces for communication via CAN among different CPU nodes.

It has the following properties, used to calculate the worst-case response time of each task for task schedulability analysis:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival Period</td>
<td>The arrival period of an event that triggers the task to execute</td>
</tr>
<tr>
<td>Execution Time</td>
<td>The amount of time that the task is executed by a periodic event</td>
</tr>
<tr>
<td>Priority</td>
<td>The fixed-priority given to the task</td>
</tr>
<tr>
<td>Deadline</td>
<td>The maximum response time that can be allowed for the task</td>
</tr>
<tr>
<td>CPU</td>
<td>The CPU that where the task places</td>
</tr>
</tbody>
</table>

- Meta Task
  The Meta Task component is a composite component, which may contain one or more components and connectors inside as shown in Figure 4.2. It may have all kinds of ports that the Task component can have as interfaces. Those ports may be bounded to the ports of Task components, which reside inside the Meta Task component. Unlike the Task component, however, the Meta Task component have no real-time task properties since it doesn’t play as a concurrency unit in scheduling.

Ports
A set of ports define components’ interfaces. Each port identifies a point of interaction between the component and its environment. A component may provide multiple interfaces by using multiple ports. [Monroe 98]

There are two ports used for sharing data between Task components within a CPU node:

- Write
  The Write port represents writing data on shared resource. It may be attached to the Writer role of the Shared Data connector.

\(^1\)In this document, every task will be regarded as a periodic task since the current software architectural style doesn’t support any aperiodic task.
It has the following properties, used to calculate the blocking delays, caused by task synchronization, for task schedulability analysis:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used Time</td>
<td>The amount of time that the task will use to write data</td>
</tr>
</tbody>
</table>

- **Read**

  The Read port represents reading data on shared resource. It may be attached to the Reader role of the Shared Data connector.

  It has the following properties, used to calculate the blocking delays, caused by task synchronization, for task schedulability analysis:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used Time</td>
<td>The amount of time that the task will use to write data</td>
</tr>
</tbody>
</table>

There are two ports used for sending/receiving CAN messages between two Task components, each of which places on a different CPU node:

- **Receive**

  The Receive port represents receiving message via CAN. It may be attached to the Receiver role of the CAN connector.

  It has the following properties, used to identify the message received via CAN:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Name</td>
<td>The name of a message received via CAN</td>
</tr>
</tbody>
</table>

- **Read**
Connectors

Connectors represent interactions among components. Computationally speaking, connectors mediate the communication and coordination activities among components. Informally they provide the "glue" for architectural designs and correspond to the lines in box-and-line descriptions. [Monroe 98] Since the target systems are assumed to supports data sharing for local communications within a CPU node and message passing for global communications across different CPU nodes, this real-time software architectural style has two types of connectors. In addition, this style allows one or more connectors to be connected to one Task/Meta-Task component. That is, tasks in the system can have multiple data sharing and message passing.

There are two kinds of interactions between Task components:

- **Shared Data**
  It represents communication between Task components within the same CPU node. The Reader/Writer roles may be used to represent interfaces of the Shared Data connector.
  To synchronizing multiple data sharing between tasks without problems, such as priority inversion and deadlock, the target systems are assumed to use Priority Ceiling Protocol (PCP) as a synchronization protocol. Details about PCP will be discussed in Section 6.1.3.

- **CAN**
  It represents communication between Task components between two different CPU nodes. CAN is an advanced serial communications protocol for distributed real-time control systems. It is a contention-based multi-master network whose timeliness properties come from its collision resolution algorithm which gives a high schedulable utilization and guaranteed bus access latency of about 150µs for the highest priority message on a 1Mbit/s bus. [Henderson 00] Using CAN for inter-processor communications provides predictable schedulability, enabled by the prioritized bus access, and high reliability, enable by its Cyclic Redundancy Check (CRC) and bit-stuffing mechnism. [Nolte 03]

It has the following property, used to calculate the worst-case response time of each message for CAN-message schedulability analysis:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>The throughput amount of the CAN in bit-per-second</td>
</tr>
</tbody>
</table>

Roles

A set of roles define connectors’ interfaces. Each role of a connector defines a participant of the interaction represented by the connector. [Monroe 98]
There are two ports used for sending/receiving CAN messages between two Task components, each of which places on a different CPU node:

- **Reader**
  
  The Reader role is an interface of the Shared Data connector representing reading from shared data. It may be attached to the Read port of the Task or Meta Task component. It has no extra property used for schedulability analysis.

- **Writer**
  
  The Writer role is an interface of the Shared Data connector representing writing to shared data. It may be attached to the Write port of the Task or Meta Task component. It has no extra property used for schedulability analysis.

There are two roles used for sharing data between Task components within a CPU node:

- **Sender**
  
  The Sender role is an interface of the CAN connector representing send message via CAN. It has be attached to the Send port of the Task or Meta Task component. It has the following property, used to identify the message received via CAN:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Name</td>
<td>The name of a message received via CAN</td>
</tr>
</tbody>
</table>

- **Receiver**
  
  The Receiver role is an interface of the CAN connector representing receiving message via CAN. It has be attached to the Receive port of the Task or Meta Task component. It has the following property, used to identify the message received via CAN:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Name</td>
<td>The name of a message received via CAN</td>
</tr>
</tbody>
</table>

**System**

In the Acme ADL, systems are defined as graphs in which the nodes represent components and the arcs represent connectors. [Garlan 00] For the end-to-end schedulability analysis, a set of execution paths should be specified in the system level. Thus, the execution paths should be defined as properties of Acme systems.

Unlike properties of other architectural elements, each execution path specification has the same prefix "end-to-end-" but a different property name. The value of property specifying for an execution path is a sequence of strings.

Here is an example of execution path specifications.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>end-to-end-path1</td>
<td>&lt;&quot;path1&quot;, 2000, &quot;task1&quot;, &quot;task2&quot;, &quot;network0 message0&quot;, &quot;task3&quot; &gt;</td>
</tr>
<tr>
<td>end-to-end-path2</td>
<td>&lt;&quot;path2&quot;, 800, &quot;task4&quot;, &quot;network0 message1&quot;, &quot;task5&quot; &gt;</td>
</tr>
<tr>
<td>end-to-end-path3</td>
<td>&lt;&quot;path3&quot;, 1300, &quot;task1&quot;, &quot;task2&quot;, &quot;network1 message0&quot;, &quot;task7&quot; &gt;</td>
</tr>
</tbody>
</table>
The first item of the path sequence is the name of the path. The second item is the deadline of the path. A sequence of execution units, each of which is either a task in a processor or a message in a network, to be executed in order will place from the third item. If the execution unit is a task in a processor, it will be represented by the task’s name, such as "task1", "task2", and "task3". If the execution unit is a message in a network (CAN), it will be represented by the network’s name and the message’s name, such as "network0 message0" and "network0 message1".
Chapter 5

Schedulability Analysis for Real-Time System Style

5.1 Schedulability Analysis on Centralized Real-Time Systems

5.1.1 Overview of Centralized Real-Time Systems

The typically centralized real-time system contains one processor (CPU). The system’s software consists of concurrent tasks that may exchange messages with other tasks in the CPU. Tasks may also share data or resources via the usual synchronization mechanisms such as semaphores, critical regions, monitors. Each task in the system is activated by the arrival of a triggering event that may be generated by the external environment, a timer, or by the arrival of a message from another task.

5.1.2 Independent Periodic Tasks

Rate monotonic analysis is basically developed for analyzing schedulability of independent periodic tasks in a centralized environment. Analyzing systems consisting of only independent periodic tasks might be unrealistic but become a good basis that can be extended to a realistic schedulability analysis for more complex systems.

A periodic task is characterized by a worst-case computation time $C$ and a period $T$. Usually a periodic task finishes its computation by the end of its period. Tasks are said to be independent if they do not need to synchronize with each other. In the real world, a real-time system typically consists of both periodic and aperiodic tasks.

There are two approaches to analyze if a set of independent periodic tasks is schedulable.

- Using the total CPU utilization

  A set of $n$ independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all tasks phasings, if

  $$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq n \left(2^{\frac{1}{n}} - 1\right)$$
where

\[ C_i \] is the execution time,

\[ T_i \] is the period of task \( t_i \), and

\[ \frac{C_i}{T_i} \] is the utilization of the resource by task \( t_i \). [Klein 93]

The bound on the CPU utilization \( n(2^{\frac{1}{n}} - 1) \) rapidly converges to \( \ln 2 = 0.69 \) as \( n \) becomes large. The utilization bound is very pessimistic because the worst case task set is contrived and is unlikely to be encountered in practice. [Sha 94] The average scheduling utilization is 88%. [Lehoczky 89] Therefore, it is hard to ensure whether a system is schedulable by using this approach.

• Calculating response time for each task

Another approach is not to use utilization bounds but to calculate the response time of each task. Then, it would be checked whether the response time of each task is within its deadline.

The following steps can be used to calculate the worst case response time for a task \( \tau \). [Klein 93]

**Step 1** Compute the first approximation of response time \( (a_0) \) by summing the execution time associated with the task \( \tau \) and all higher priority tasks.

\[ a_0 = \sum_{\delta \in HE(\tau)} C_\delta \]

where

\( HE(\tau) \) is the set of tasks whose priority is higher than or equal to the priority of the task \( \tau \), and

\( C_\delta \) is the execution time of the task \( \delta \).

**Step 2** Compute the next approximation \( (a_{n+1}) \) of response time by using the previous approximation \( (a_n) \) of the response time

\[ a_{n+1} = C_\tau + \sum_{\delta \in H(\tau)} \left\lceil \frac{a_n}{T_\delta} \right\rceil C_\delta \]

where

\( H(\tau) \) is the set of tasks whose priority is higher than the priority of the task \( \tau \),

\( T_\delta \) is the period of the task \( \delta \), and

\( C_\delta \) is the execution time of the task \( \delta \).

**Step 3** Determine if the approximation is the answer

Repeat Step 2 if

\( (a_{n+1} \leq D_\tau) \land (a_{n+1} \neq a_n) \),

the task \( \tau \) misses its deadline if

\[ a_{n+1} > D_\tau, \text{ and} \]

the task \( \tau \) is schedulable and its response time is \( a_n \) if

\[ a_{n+1} = a_n, \]

18
where 

\[ a_{n+1} \] is the current approximation of the response time of the task \( \tau \),
\[ a_n \] is the previous approximation of the response time of the task \( \tau \), and
\[ D_{\tau} \] is the deadline of the task \( \tau \).

### 5.1.3 Periodic Tasks Sharing Multiple Data Resources

To provide a reasonably comprehensive theoretical framework, task synchronization had to be treated. The most common problem caused by task synchronization is Priority inversion (PI) that occurs when a high priority task is blocked by a low priority task, such as a shared semaphore or a protected operation. The blocking time of a task, which we don’t care in the system consisting of independent tasks, becomes a primary concern in the system of interacting tasks.

Many synchronization protocols have been proposed for reducing priority inversion and preventing deadlocks. Mutually exclusive access to shared resources from the critical sections of concurrent tasks may be protected by different synchronization techniques, such as semaphores, critical regions, and monitors. Most proposed protocols, however, use semaphores to protect critical sections. [Yue 1996] For this reason, in this project, every synchronization technique to shared data will be regarded as a semaphore.

As mentioned earlier, for minimizing the effects of priority inversion, the target real-time system uses Priority Ceiling Protocol (PCP), which has the following properties:

- Each resource has its own ceiling priority.
- An operating system variable, so called "the current system ceiling", is maintained as the highest ceiling of all locked semaphores.
- Unless a task holds the semaphore that set the current system ceiling, it can only lock a semaphore if its priority is higher than the current system ceiling.
- If a task is prevented from locking a resource, the task holding the lock inherits the priority of the blocked task.

The protocol works as follows. The priority ceiling of a binary semaphore \( S \) to be the highest priority of all tasks that may lock \( S \). When a task attempts to execute one of its critical sections, it will be suspended unless its priority is higher than the priority ceilings of all semaphores currently locked by tasks other than the task. If the task is unable to enter its critical section for this reason, the task that holds the lock on the semaphore with the highest priority ceiling is said to be blocking the task and hence inherits the priority of the task. As long as the task is not attempting to enter one of its critical sections, it will preempt every task that has a lower priority. [Sha 94]

- Calculate blocking time for each task [Klein 93]

This approach exploits the property of the Priority Ceiling Protocol (PCP) that a response of a task can be blocked by at most a single critical section. Therefore, the blocking term us derived from the longest critical section of a lower priority response. The element \( b_r \) is an element of a one-dimensional array that indicates the contribution to a blocking delay from resource \( r \in R \), where \( R \) is the set of shared resources.
We follow these steps to compute the value of the blocking delay for each task that shares the resource:

**Step 1** Order all the tasks by priority, highest first.

Then, index each task from 1 to \( n \), where \( n \) is the number of tasks.

**Step 2** Initialize the blocking array \( b_r = 0 \);

**Step 3** Begin with the lowest priority task \( \tau \), which is indexed by \( n \)

**Step 4** Compute the blocking value \((B_\tau)\) of the task \( \tau \)

\[
B_\tau = \max_{\phi \in R}(b_\phi)
\]

where \( R \) is the set of shared resources.

**Step 5** Update the blocking array

For each resource \( r \in R \), that the task \( \tau \) shares,

\[
\text{if } RU_{\tau,r} > b_r, \text{ then } b_r = RU_{\tau,r}
\]

where \( RU_{i,r} \) is the amount of time that task \( \tau \) will use resource \( r \).

**Step 6** Check ceilings

If the task \( \tau \) is the highest priority, i.e. its index is 1, terminate the calculation. Otherwise,

\[
\text{if } P_{\tau'} > PC_r, \text{ then } b_r = 0
\]

where

- \( \tau' \) is the next higher priority task than the task \( \tau \)
- \( i.e. \) the index of \( \tau \) - the index of \( \tau' = 1 \)
- \( P_{\tau'} \) is the priority of the task \( \tau' \), and
- \( PC_r \) is the priority ceiling of the resource \( r \).

**Step 7** Iterate with the next higher priority task \( \tau' \). Go to Step 4

\[
\text{set } \tau = \tau'
\]

where

- \( \tau' \) is the next higher priority task than the task \( \tau \)
- \( i.e. \) the index of \( \tau \) - the index of \( \tau' = 1 \).

**Calculate response time for each task with arbitrary deadline**

This calculation is a generalization of the response time calculation for independent periodic tasks as shown in 6.1.2.

The following steps can be used to calculate the worst case response time for a task \( \tau \).

[Klein 93]

**Step 1** Compute the first approximation of response time \((a_0)\) by summing the execution time associated with the task \( \tau \) and all higher priority tasks.

\[
a_0 = B_\tau + \sum_{\delta \in HE(\tau)} C_\delta
\]
where

- $B_\tau$ is the blocking time of the task $\tau$,
- $HE(\tau)$ is the set of tasks whose priority is higher than or equal to the priority of the task $\tau$, and
- $C_\delta$ is the execution time of the task $\delta$.

**Step 2** Initialize a counter, $k$, to 1.

The counter $k$ varies from 1 to the number of jobs in the busy period. The counter $k$ is incremented as we take into consideration the execution time of the next job.

**Step 3** Compute the next approximation ($a_{n+1}$) of response time by using the previous approximation ($a_n$) of the response time

$$a_{n+1} = B_\tau + kC_\tau + \sum_{\delta \in H(\tau)} \left\lceil \frac{a_n}{T_\delta} \right\rceil C_\delta$$

where

- $B_\tau$ is the blocking time of the task $\tau$,
- $H(\tau)$ is the set of tasks whose priority is higher than the priority of the task $\tau$,
- $T_\delta$ is the period of the task $\delta$, and
- $C_\delta$ is the execution time of the task $\delta$.

**Step 4** Determine if the approximation is the completion time of the $k$th job

Repeat Step 3 if

$$a_{n+1} \neq a_n, \text{and}$$

the task $\tau$ misses its deadline if

$$a_{n+1} > ((k - 1) \times T_\tau + D_\tau)$$

where

- $a_{n+1}$ is the current approximation of the response time of the task $\tau$,
- $a_n$ is the previous approximation of the response time of the task $\tau$,
- $T_\tau$ is the period of the task $\tau$, and
- $D_\tau$ is the deadline of the task $\tau$.

Otherwise, go to Step 5.

**Step 5** Determine the response time ($E_{\tau,k}$) of the $k$th job of the task $\tau$

$$E_{\tau,k} = a_n - T_\tau(k - 1)$$

where

- $T_\tau$ is the period of the task $\tau$, and
- $a_n$ is the completion time of the $k$th job of the task $\tau$.

**Step 6** Decide if the busy

Proceed to Step 7 if

$$E_{\tau,k} \leq T_\tau$$

where

- $T_\tau$ is the period of the task $\tau$, and
- $E_{\tau,k}$ is the response time of the $k$th job of the task $\tau$.

Otherwise, add 1 to $k$, set $a_n = a_n + C_\tau$, where $C_\tau$ is the execution time of the task $\tau$, and go back to Step 3.
**Step 7** Determine the worst-case response time \( (R_t) \) of the task \( \tau \)

\[
R_\tau = \text{MAX}_{0 \leq i \leq k} E_{\tau,i}
\]

where

\( E_{\tau,i} \) is the response time of the \( i \)th job of the task \( \tau \).

### 5.2 Schedulability Analysis on Distributed Real-Time Systems

#### 5.2.1 Overview of Distributed Real-Time Systems

The features of distributed real-time systems designed for the schedulability analysis purpose in [Gutierres 97] are adopted as guidelines for the target systems. The typical architecture in a distributed real-time system consists of several processor nodes interconnected through one or more interconnection networks. The system’s software consists of concurrent tasks that are often statically allocated to processing nodes, and may exchange messages with other tasks in the same node or in different nodes. Tasks allocated in the same node may also share data or resources via the usual synchronization mechanisms used in shared memory systems. Each task in the system is activated by the arrival of a triggering event that may be generated by the external environment, a timer, or by the arrival of a message from another task.

#### 5.2.2 Single Processor Schedulability Analysis

Each processor node has the same features as the centralized real-time system, described in Section 6.1.1, except the communication networks. The software on each process node consists of concurrent tasks that may exchange message with other tasks in the CPU. For generic cases, the analysis technique for periodic tasks sharing multiple data resources, as shown in the Section 6.1.2, can be applied to this analysis.

#### 5.2.3 Communication Network Schedulability Analysis

Scheduling in a network is different from scheduling in a centralized environment. In a centralized system, the centralized scheduler immediately knows of all resource requests. In some networks, distributed scheduling decisions must be made with incomplete information. From the perspective of any particular node, some requests could be delayed and some may never be seen, depending on the relative position of the node in the network.

In [Sha 90], transmission schedulability is defined as follows:

A set of message is said to be transmission schedulable if each message can be transmitted before its deadline.

Satisfaction of the end-to-end deadline of the message can be found using the relation:

\[
\text{End-to-End Deadline} \geq \text{Transmission Deadline} + \text{Propagation Delay}
\]

Schedulability Analysis for CAN Frames can be done by the following steps: [Henderson 00] [Tindell 94]
Step 1 Calculate the worst-case transmission time ($\theta_m$) for each message

$$\theta_m = \left\lceil \frac{34 + 8L_m}{4} \right\rceil + 47 + 8L_m \tau_{\text{bit}}$$

where

- $L_m$ is the length of message $m$ in bytes,
- $\tau_{\text{bit}}$ is the time to transmit a single bit.

The first part of the sum is to account for the possibility of stuff bits. In the standard CAN format, 34 bits beside the data part are exposed to the bit stuffing mechanism. [Nolte 03] 47 is the maximum bit overhead of a message in CAN. [Brooks 99]

Step 2 Calculate the Blocking Delay Time ($B_m$) for Message $m$

$$B_m = \text{MAX}_{\delta \in \text{lp}(m)} \theta_m$$

where

- $\theta_m$ is the worst-case transmission time for message $m$,
- $\text{lp}(m)$ is the set of messages with lower priority than message $m$.

Step 3 Calculate the Completion Time ($C_m$) for Each Message

The completion time of a message is the sum of the blocking time and the time that a message waits for the transmission of higher priority messages.

Step 3.1 Calculate the First Approximation of Completion Time ($C_m^0$) for Message $m$

$$C_m^0 = B_m + \sum_{\phi \in \text{hp}(m)} \left\lceil \frac{J_\phi + \tau_{\text{bit}}}{T_\phi} \right\rceil \theta_\phi$$

where

- $B_m$ is the blocking time for message $m$,
- $J_m$ is the jitter time for message $m$,
- $\text{hp}(m)$ is the set of messages with higher priority than message $m$,
- $\theta_m$ is the worst-case transmission time for message $m$,
- $\tau_{\text{bit}}$ is the time to transmit a single bit.

Step 3.2 Calculate the Next Approximation of Completion Time ($C_m^{n+1}$) for Message $m$

$$C_m^{n+1} = B_m + \sum_{\phi \in \text{hp}(m)} \left\lceil \frac{C_m + J_\phi + \tau_{\text{bit}}}{T_\phi} \right\rceil \theta_\phi$$

where

- $B_m$ is the blocking time for message $m$,
- $J_m$ is the jitter time for message $m$,
- $\text{hp}(m)$ is the set of messages with higher priority than message $m$,
- $\theta_m$ is the worst-case transmission time for message $m$,
- $\tau_{\text{bit}}$ is the time to transmit a single bit,
- $C_m^n$ is the current approximation of completion time for message $m$. 
**Step 3.2** Determine if the approximation is the answer
The completion time for message $m$ is $C_m^n$ if
\[ C_m^n = C_m^{n+1} \]
where
- $C_m^n$ is the previous approximation of completion time for message $m$, and
- $C_m^{n+1}$ is the current approximation of completion time for message $m$.
Otherwise, repeat Step 3.2.

**Step 4** Calculate the Response Time ($R_m$) for Each Message
\[ R_m = J_m + C_m + \theta_m \]
where
- $J_m$ is the jitter time for message $m$,
- $C_m$ is the completion time for message $m$,
- $\theta_m$ is the worst-case transmission time for message $m$.

**5.2.4 End-To-End Schedulability Analysis**
Through the schedulability analysis for tasks and messages, described in the previous section, we can now get the worst case response time of each task/message, which includes the execution time, the jitter time and all kinds of synchronization delay time. The worst-case response time for an execution path can be obtained by summing up the worst-case response times of all tasks and messages.
Chapter 6

Example Application

In this chapter, a software architecture for a valve control system will be modeled using the real-time software architectural style proposed described in 5.3. Also, the schedulability analysis results will be generated by the schedulability analysis tool plugged in AcmeStudio.

6.1 Description of System

The example system is a typical distributed embedded control system. The system gets pressure and temperature information from sensors, analyzes them, and then set the valve. The controlled valve may affect the pressure and the temperature again. In addition, the control system will display the status of the valve.

The system is assumed to use Rate Monotonic Scheduling (RMS) for scheduling algorithm and Priority Ceiling Protocol (PCP) for task synchronization. It consists of two CPU (Central Processing Unit) nodes, which communicate each other via Controller Area Network (CAN). One CPU node is dedicated to the sensor/actuator functions. The other is dedicated to the data analysis function.

6.2 Description of Software Architecture

As shown in Figure 6.1, The example system consists of seven task components: pressure_sensor, temperature_sensor, tracker, analyzer, controller, valve_actuator, and display. The pressure_sensor, the temperature_sensor, the controller, the valve_actuator, and the display components will be deployed in a process (CPU1). The tracker and the analyzer components will be deployed in the other process (CPU2).

The pressure_sensor/temperature_sensor task components get and process signals from the pressure/temperature sensors. Those two sensor components will send the processed signals as separate messages (pressure and temperature) to the tracker component via Controller Area Network (CAN). Then, the tracker component will gather two signal information and send them into the analyzer component via shared data. The analyzer component will produce appropriate valve control commands and send them into the controller component via CAN. The controller component send valve control signals to the valve_actuator component and get feedback (valve status) from it. The display component will read/display the valve status information.
The task components have the following temporal properties:

<table>
<thead>
<tr>
<th>CPU</th>
<th>Task ID</th>
<th>Period</th>
<th>Execution</th>
<th>Priority</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU1</td>
<td>controller</td>
<td>5000</td>
<td>200</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>CPU1</td>
<td>temperature_sensor</td>
<td>5000</td>
<td>180</td>
<td>100</td>
<td>5000</td>
</tr>
<tr>
<td>CPU1</td>
<td>pressure_sensor</td>
<td>5000</td>
<td>180</td>
<td>100</td>
<td>5000</td>
</tr>
<tr>
<td>CPU1</td>
<td>valve_actuator</td>
<td>5000</td>
<td>150</td>
<td>100</td>
<td>5000</td>
</tr>
<tr>
<td>CPU1</td>
<td>display</td>
<td>5000</td>
<td>150</td>
<td>100</td>
<td>5000</td>
</tr>
<tr>
<td>CPU2</td>
<td>tracker</td>
<td>5000</td>
<td>250</td>
<td>100</td>
<td>1200</td>
</tr>
<tr>
<td>CPU2</td>
<td>analyzer</td>
<td>5000</td>
<td>450</td>
<td>100</td>
<td>2000</td>
</tr>
</tbody>
</table>

The messages in the network connector (CAN0) have the following temporal properties:

<table>
<thead>
<tr>
<th>Network</th>
<th>Message Name</th>
<th>Period</th>
<th>Jitter</th>
<th>Length</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN0</td>
<td>valve</td>
<td>5000</td>
<td>0</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>CAN0</td>
<td>temperature</td>
<td>5000</td>
<td>0</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>CAN0</td>
<td>pressure</td>
<td>5000</td>
<td>0</td>
<td>4</td>
<td>20</td>
</tr>
</tbody>
</table>

The system has the following execution paths with their temporal contraints:

<table>
<thead>
<tr>
<th>Path Name</th>
<th>Deadline</th>
<th>List of Execution Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>path1</td>
<td>5000</td>
<td>pressure_sensor, CAN0 pressure, tracker, analyzer, CAN0 valve, controller, valve_actuator, display</td>
</tr>
<tr>
<td>path2</td>
<td>5000</td>
<td>temperature, CAN0 pressure, tracker, analyzer, controller, display</td>
</tr>
</tbody>
</table>
6.3 Schedulability Analysis

This section will present the result of schedulability analysis for the example software architecture using the analysis plug-in tool for AcmeStudio.

6.3.1 CPU/Network Utilization

Each CPU or network has its utilization, which is less than 100% and even less than the pessimistic utilization bound ($\approx 0.69$) mentioned in 6.1.2.

<table>
<thead>
<tr>
<th>CPU/Network Name</th>
<th>Number of Tasks/Messages</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU1</td>
<td>5</td>
<td>17 %</td>
</tr>
<tr>
<td>CPU2</td>
<td>2</td>
<td>14 %</td>
</tr>
<tr>
<td>CAN0</td>
<td>3</td>
<td>9 %</td>
</tr>
</tbody>
</table>

6.3.2 Task/Message Schedulability

Since the worst-case response time of every task or message is less than the deadline of the task, every task or message can meet its deadline.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Task ID</th>
<th>Period</th>
<th>Execution</th>
<th>Priority</th>
<th>Deadline</th>
<th>Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU1</td>
<td>controller</td>
<td>5000</td>
<td>200</td>
<td>100</td>
<td>1000</td>
<td>220</td>
</tr>
<tr>
<td>CPU1</td>
<td>temperature_sensor</td>
<td>5000</td>
<td>180</td>
<td>100</td>
<td>5000</td>
<td>400</td>
</tr>
<tr>
<td>CPU1</td>
<td>pressure_sensor</td>
<td>5000</td>
<td>180</td>
<td>100</td>
<td>5000</td>
<td>580</td>
</tr>
<tr>
<td>CPU1</td>
<td>valve_actuator</td>
<td>5000</td>
<td>150</td>
<td>100</td>
<td>5000</td>
<td>720</td>
</tr>
<tr>
<td>CPU1</td>
<td>display</td>
<td>5000</td>
<td>150</td>
<td>100</td>
<td>5000</td>
<td>860</td>
</tr>
<tr>
<td>CPU2</td>
<td>tracker</td>
<td>5000</td>
<td>250</td>
<td>100</td>
<td>1200</td>
<td>270</td>
</tr>
<tr>
<td>CPU2</td>
<td>analyzer</td>
<td>5000</td>
<td>450</td>
<td>100</td>
<td>2000</td>
<td>700</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Network</th>
<th>Message Name</th>
<th>Period</th>
<th>Jitter</th>
<th>Length</th>
<th>Priority</th>
<th>Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN0</td>
<td>valve</td>
<td>5000</td>
<td>0</td>
<td>1</td>
<td>30</td>
<td>310</td>
</tr>
<tr>
<td>CAN0</td>
<td>temperature</td>
<td>5000</td>
<td>0</td>
<td>4</td>
<td>20</td>
<td>494</td>
</tr>
<tr>
<td>CAN0</td>
<td>pressure</td>
<td>5000</td>
<td>0</td>
<td>4</td>
<td>20</td>
<td>494</td>
</tr>
</tbody>
</table>

6.3.3 End-to-End Schedulability

Since the worst-case response time of every execution path is less than the deadline of the path, every execution path can meet its deadline. Figure 6.2 and Figure 6.3, generated by the schedulability analysis tool, shows more details about the end-to-end schedulability analysis result, including the worst-case delay time and the actual execution/transmission time for each task/message.

<table>
<thead>
<tr>
<th>Path Name</th>
<th>Deadline</th>
<th>List of Execution Units</th>
<th>Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>path1</td>
<td>5000</td>
<td>pressure_sensor, CAN0 pressure, tracker, analyzer, CAN0 valve, controller, valve_actuator, display</td>
<td>4154</td>
</tr>
<tr>
<td>path2</td>
<td>5000</td>
<td>temperature, CAN0 pressure, tracker, analyzer, controller, display</td>
<td>2543</td>
</tr>
</tbody>
</table>
Figure 6.2: Schedulability Analysis Graph for Path1

Figure 6.3: Schedulability Analysis Graph for Path2
Chapter 7

Future Work

7.1 Analyzing Schedulability for Aperiodic Tasks

In the real world, the real-time systems consists of periodic tasks as well as aperiodic tasks. The pure Rate Monotonic Analysis, proposed by Liu and Layland, is targeted to the unrealistic systems comprised of only independent periodic tasks. As more work was done in applying RMA to practical systems, several approaches have been developed to incorporate aperiodic tasks into the Liu and Layland model. [Klein 93]

The basic strategy for handling aperiodic tasks is to cast such tasks into a periodic framework. Typical examples are using polling tasks and using sporadic servers. Those kinds of tasks and schedulability analysis techniques applied to them [Sha 91] [Sha 93] [Klein 93] could be supported as components in the future real-time software architectural style.

7.2 Adotning Fiber Distributed Data Interface for Distributed Computing

There are approaches to design distributed real-time systems using networks other than Controller Area Network (CAN). Lui and Shirish [Sha 93] used Fiber Distributed Data interface (FDDI), which is a token ring protocol that uses a timed-token access method, and proposed related analysis techniques. [Sha 95]

7.3 Modifying Properties of Architectural Elements

7.3.1 Adding the Special Prefix ”rma-”

The timing properties, such as arrival period, execution time, and deadlines, may be useful when software architects want to edit or to analyze those properties. By adding ”rma-” prefix to every timing properties, used for schedulability analysis, the future AcmeStudio will be able to filter properties of architectural elements depending on the users’ preference. It would require some change in user interfaces to choose whether to make the timing-specific properties visible or not to users.
7.3.2 Adding Schedulability Result Properties

The current schedulability analysis tool does show its analysis result on the screen but not write any analysis result, such as completion time, response time, and schedulability, on Acme source files. Additional properties for recording the schedulability analysis result could be added to Task components (for task schedulability), CAN connectors (for message schedulability), and Acme systems (for end-to-end schedulability).

7.4 Enhancing Usability of the Analysis Plug-In Tool

7.4.1 Allowing Easier Specification of Execution Paths

In the current schedulability analyzer, users need to enter the sequence of execution paths by typing. It would be better to provide an interactive way to do this, like defining the sequence of execution path by clicking on elements in the Acme diagram.

7.4.2 Marking Unschedulable Tasks/Messages

By adding schedulability result properties as described in Section 8.3.2, it is possible to modify AcmeStudio to mark unschedulable tasks/messages with specific colours or features in the Acme diagram.
Bibliography


[CAN] Controller Area Network (CAN), an overview, http://www.can-cia.de/can/


[Henderson 00] Henderson, Xrma - A User Guide, University of Northumbria at Newcastle, Jan 2000

[Isovic] Damir Isovic, Markus Lindgren, Real-Time Components, Malardalen University


