DISSERTATION

Real-Time Monitoring for the Time-Triggered Architecture

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

Univ.-Prof. Dr. Peter Puschner

Institut für Technische Informatik 182-1

eingereicht an der Technischen Universität Wien,
Fakultät für Informatik

von

Dipl.-Ing. Idriz Smaili

Matr.-Nr. 9626459

Thomas-Morus Gasse 10/2/4, A-1130 Wien

Wien, im November 2004 .....

Abstract

The application of real-time systems is vitally necessary in many industrial areas, such as automotive and aerospace industry. The development of these systems is more time consuming than the development of non real-time systems, due to testing and debugging of these systems being much more difficult. In the literature, the time spent for testing and debugging of a new developed real-time system is estimated to take about 70% of the total development time. Therefore, the application of real-time monitoring systems during development of real-time systems is crucial.

The correctness of real-time systems depends not only on the results they deliver, but also on the point in time at which the results are delivered. Thus, the basic requirement during monitoring of real-time systems is the determinism of the monitoring system. Therefore, the key issue during monitoring of a real-time system is to keep the interference that is caused by a monitoring system on the real-time system deterministic, if it cannot be completely removed. This interference depends on the way how monitoring data are collected from the target system, and on the amount and the observation rate of monitoring data being collected within an observation interval. In order to predict this interference and to keep it deterministic, we define monitoring data types and present different gathering methods in this thesis. The appropriate gathering method and the amount (even in the worst case) of each selected data type within an observation interval can be discovered in advance, i.e., before the monitoring process is started. Another objective of the thesis is the deterministic real-time monitoring for the Time-Triggered Architecture (TTA). We define and classify different abstraction levels at which TTA target systems can be monitored: i) cluster, ii) node, and iii) transducer abstraction level. The monitoring of target systems that consist of multiple clusters is also analyzed. Furthermore, the real-time triggers that can be used to search significant events in real-time during monitoring of target systems are discussed. The real-time trigger system (RTTS) records system operation of the target real-time system only in time windows of interest, around significant events. It does so by buffering and observing of significant events in real-time. The RTTS can be successfully applied during long-term monitoring for saving disk storage. In conjunction with the monitoring at node abstraction level the RTTS also can be applied for long-term diagnosing and correctness checking of real-time target systems. The concepts presented in this thesis have been implemented either in commercial software product TTPview or in the case study also presented in this thesis. TTPview is successfully applied in industry for monitoring of distributed safety-critical real-time systems built on the TTA.

Kurzfassung

Die Anwendung von Echtzeitsystemen ist in vielen Industriegebieten notwendig, wie z.B. in der Automobil- und Flugzeugindustrie. Die Entwicklung dieser Systeme ist zeitintensiver als die Entwicklung von nicht Echtzeitsystemen, weil das Testen und Debuggen von Echtzeitsystemen viel aufwändiger ist. In der Literatur wird die Zeit, die für das Testen und Debuggen eines neu entwickelten Echtzeitsystems benötigt wird, mit ungefähr 70% der vollen Entwicklungszeit abgeschätzt. Daher ist die Anwendung von Monitoringsystemen während der Entwicklung von Echtzeitsystemen notwendig.

Die Korrektheit der Echtzeitsysteme hängt nicht nur von gelieferten Resultaten ab, sondern auch von den Zeitpunkten, an denen die Resultate geliefert werden. Daraus wird ersichtlich, dass die Grundanforderung während der Überwachung von Echtzeitsystemen der Determinismus des Monitoringsystems ist. Daher muss während der Überwachung dieser Systeme die vom Monitoringsystem auf das Zielsystem erzeugte Beeinflussung deterministisch gehalten werden, wenn man sie nicht vollständig beseitigen kann. Diese Beeinflussung hängt sowohl von der Art ab, wie Monitoringdaten vom Zielsystem erfasst werden, als auch von deren Menge und Überwachungsrate, die innerhalb eines Beobachtungsintervals gesammelt werden. Um diese Beeinflussung vorhersagen und sie deterministisch halten zu können, definieren wir in dieser Arbeit Monitoringdatentypen und stellen Erfassungsmethoden vor. Die passende Erfassungsmethode und die maximale Datenmenge jedes selektierten Datentyps kann im Voraus, d.h., bevor das Monitoringsystem gestartet ist, herausgefunden werden. Eine andere Zielsetzung dieser Arbeit ist die deterministische Echtzeitüberwachung von zeitgesteuerten Systemen. Wir definieren und klassifizieren verschiedene Abstraktionsniveaus, auf dem die zeitgesteuerten Systeme überwacht werden können, wie z.B.: Clusterniveau, Knotenniveau, und Transducerniveau. Die Überwachung von Zielsystemen, die sich aus mehreren Cluster bestehen, wird ebenfalls analysiert. Außerdem werden in dieser Arbeit Echtzeittrigger eingeführt, die für das Suchen der signifikanten Events während der Überwachung von Echtzeitsystemen eingesetzt werden können. Das Echtzeittriggersystem (RTTS) speichert die Systemabläufe nur in interessanten Zeitfenstern ab, d.h., rund um die signifikanten Events. Das wird vom RTTS mittels Pufferung und Beobachtung von signifikanten Events in Echtzeit erreicht. Kombiniert mit der Überwachung auf dem Knotenniveau kann das RTTS als Langzeitdiagnosesystem oder als Korrektheitsüberprüfer eingesetzt werden. Die in dieser Arbeit präsentierten Konzepte sind in zwei Systemen implementiert, in dem kommerziellen Softwareprodukt TTPview und in der Fallstudie. TTPview wird erfolgreich in der Industrie für die Überwachung von zeitgesteuerten Systemen eingesetzt.

Abstrakt (in Albanian)

Aplikimi i sistemeve të kohës-reale është me rëndësi jetike në shumë sfera industriale siç është p.sh., industria automobilistike dhe ajo ajrore. Zhvillimi i këtyre sistemeve kërkon nje kohë shumë më të gjatë se sa zhvillimi I sistemeve non-real time, pasi testimi dhe debugimi i tyre është shumë më i vështirë. Në literaturë vlerësohet se rreth 70 % (perqind) e kohës së përgjithshme e cila nevojitet për zhvillimin e një sistemi të ri të kohës-reale, shpenzohet (perdoret) për testimin dhe kontrollin e tij. Për këtë arsye, aplikimi i sistemeve monitoruese është i nevojshëm gjatë zhvillimit të këtyre sistemeve.

Saktësia e sistemeve të kohës-reale varet jo vetëm nga rezultatet e tyre, por edhe nga koha në të cilën shfaqen këto rezultate. Kështu që kërkesë kryesore gjatë monitorimit të këtyre sistemeve është determinizmi i sistemit monitorues. Njëkohësisht vlenë për tu përmedur se interferenca (ndërhyrja) që shkaktohet nga sistemi monitorues në sistemin e monitoruar (synuar) duhet të jetë deterministike, në rast se ajo nuk mund të eleminohet tërësisht. Kjo ndërhyrje varet nga mënyra se si mblidhen të dhënat monitoruese nga sistemi që monitorohet, si dhe nga sasia dhe përshpejtimi i obzervimit të tyre brenda intervalit obzervues. Per të parashikuar dhe mbajtur nën kontroll këtë ndërhyrje, punimi ynë definon dhe klasifikon lloje të ndryshme të të dhënave monitoruese si dhe të metodave për grumbullimin e tyre. Metoda përkatëse grumbulluese si dhe sasia e të dhënave për secilin lloj (ashtu sikurse dhe në rastin e jashtzakonshem - *"worst-case"*) mund të përcaktohet paraprakisht, d.m.th., para se të vihet në përdorim sistemi monitorues. Një qëllim tjetër i këtij punimi është monitorimi deterministik i kohës-reale për Arkitekturën Time-Triggered (TTA). Ndër të tjera ne definojmë dhe klasifikojmë disa nivele abstrahuese nëpërmjet të cilave sistemet TTA mund të monitorohen, si p.sh.: cluster niveli, node niveli, si dhe transducer niveli. Monitorimi i sistemeve që përbëhen nga shumë cluster-a përbën një tjetër analizë në këtë punim. Përveç kësaj, në këtë punim trajtohen edhe trigerët, të cilët në kohë-reale kërkojnë *ngjarjet me rëndësi* (*"significant events"*) gjatë monitorimit të sistemve të synuara. Sistemi i trigerëve në kohë-reale (RTTS) i ruan operacionet sistemore të sistemeve që monitorohen vetëm gjatë intervaleve kohore interesante, d.m.th. për rreth ngjarjeve te rëndësishme. Kjo arrihet përmes baferimit dhe obzervimit të ngjarjeve të rëndësishme të kohës-reale nga ana e RTTS-it. RTTS-i mund të aplikohet gjatë monitorimit afatëgjatë, për ruajtje të sukssesshme të hapësirës në disk. RTTS-i i kombinuar me nivelin abstrahues *node* mund të aplikohet jo vetem si diagnostikues afatëgjatë por dhe si kontrollues i saktësisë së sistemeve që monitorohen. Konceptet e paraqitura në këtë punim janë implementuar në produktin komercial siç eshte software TTPview ose në *case study* të ilustruar gjithashtu edhe në këtë punim. TTPview është aplikuar me sukses në industri për monitorimin e sistemeve të shpërndara kritike kompjuterike të bazuar në TTA.

Danksagung

Dieser Dissertation entstand als Abschluss meiner Forschungs- und Arbeitstätigkeit sowohl am Institut für Technische Informatik, Abteilung für Echtzeitsysteme, als auch bei TTTech Computertechik AG. Mein besonderer Dank gilt Herrn Prof. Dr. Peter Puschner, der meine Arbeit betreute, und mir dabei immer wieder wertvolle Vorschläge gab. Außerdem, möchte ich mich bei dem Leiter des Instituts, Herrn Prof. Dr. Hermann Kopetz, bedanken.

Mein Dank gilt auch meinen langjährigen Kollegen bei TTTech Computertechik AG, besonders Harald Angelow, der direkt durch seine Implementierungen an dem TTPview Projekt mitgewirkt hat und mit dem ich mich ausgezeichnet verstanden habe. Außerdem, möchte ich mich bei folgenden Kollegen des Instituts bedanken: Leo Mayerhofer, Raimund Kirner, Wilfried Elmenreich, Wilfried Steiner, Roman Obermaisser, und Thomas Losert.

Meinen Freunde möchte ich herzlich danken: Astrit Ademaj, Nysret Musliu, Ylber Ramadani, Yll Haxhimusa, Suela Lezaj, Arian Shala, Bedri Dragusha, Driton Statovci und Ilirjana Gashi. Ich bedanke mich bei Astrit, Nysret, Ylber, Ylli, Ilirjana, Bedri, Suela und Arian für Korrekturlesen von Teilen dieser Dissertation. Ein besonderer Dank gilt Astrit, mit dem ich viele wertvolle Diskussionen (nicht nur im unseren Fachbereich) führte.

Und nun zu meinen Liebsten - einen speziellen und herzlichen Dank haben meine Eltern verdient, die mir immer zur Seite standen und mich sowohl in guten als auch in schlechten Zeiten unterstützten. Gleichzeitig möchte ich meinen Geschwistern danken, dafür, daß sie mich geliebt haben und dadurch einen wichtigen Teil zu meinem Erfolg beigetragen haben. Im speziellen danke ich meiner Frau und Partnerin, Merita, für die viele entgegengebrachte Geduld, die ständige Motivation, und liebevollen Aufmunterungen während der Zeit, in der diese Dissertation entstand. Ohne ihre liebevolle Unterstützung wäre diese Dissertation nie entstanden. Und nun zum Schluß, möchte ich mich bei meiner vor kurzem geborenen Tochter, Hana, bedanken, dafür, daß sie mich immer wieder anlächelt und dadurch meiner Motivation einen kräftigen Schwung immer wieder gibt.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The development of *real-time systems* (RTS) has grown so much recently that they can be found almost *everywhere* in our everyday life. Such a development has been enforced by the rapid growth of the information technology. There are many industry branches in which the use of real-time systems is irreplaceable. For example, command and control systems, flight control systems, space shuttle control systems, aircraft avionics control systems, robotics, patient monitoring systems, nuclear power plant control systems, etc. [Kop97].

The development of real-time systems enables the substitution of old hydraulic and pneumatic non-safety and non fault-tolerant controlling systems through high safety time-critical fault tolerant real-time systems, e.g., *brake-* and *steer-by-wire*. The main advantage of this substitution is the increased *safety* of this equipment and on the other hand the cost reduction for the development and maintenance of these types of equipment. This replacement may reduce overall system costs, reduce weight, improve reliability, and introduce additional quality of service [Bau00]. In the near future, especially in the automotive industry, this trend will find a widespread application.

A considerable part of costs of a newly developed computer system is spent for testing and debugging. Statistical evidence indicates that testing and debugging represents approximately 50% of the cost of developing a new system [TBYS96]. These costs are higher in case of real-time systems, because the origins of error in these systems can be found both in the time and value domain. In [Sev87] Seviora found out that 70% of the development time in case of real-time systems is spent for testing and debugging. These costs are caused due to the fact that the debugging of real-time systems cannot be done with the traditional debugging techniques. The most popular traditional debugging technique is the *breakpoint* technique, which uses breakpoints to stop the execution of the application at predefined points in the code. This technique cannot be applied for debugging of real-time systems, because if the program

execution is interrupted, the physical time cannot be stopped and the system is no longer consistent in the time domain [Sma02].

Recently, different techniques were developed related to debugging of real-time systems. These techniques can be grouped into *static* and *dynamic* debugging techniques. The static debugging techniques are based on the static analysis of the source code. They try to detect parts of the code that probably contain anomalies and (or) bugs. In [AAC+94] this technique is defined as *static verification*, i.e., the verifying of the system without actual execution. These techniques are not suitable for debugging of real-time systems, because they do not deliver any information over the run-time behavior of target systems. Therefore, for successful debugging of real-time systems, monitoring systems must be used.

Monitoring systems are used for collecting run-time data that represent the run-time behavior of a target system during the monitoring (observation) process at the intended abstraction level. In [HS90] Haban et. al. define monitoring as the extraction of data about the activities of a computer system, and the authors refer to monitoring under timing constraints as *real-time monitoring*. The collected run-time information can be used during debugging of real-time systems for searching of the suspected faults that could not be detected by merely studying the source code.

In this thesis the collected monitoring data are used in context of debugging of target systems that are based on the Time-Triggered Architecture (TTA). The correctness of real-time systems depends not only on the results they deliver, but also on the point in time they are delivered. Therefore, the run-time behavior of the target systems during monitoring must be changed neither in the time nor in the value domain. Thus, the basic objective of this thesis is the *deterministic monitoring* of these systems.

1.1 Objectives

Monitoring data represent the run-time behavior of a target system at an intended abstraction level. The goal of a monitoring system is to collect all monitoring data from which the run-time behavior of the target system at the intended abstraction level can be reproduced. The key issue during monitoring of real-time systems is to keep the *interference* of the monitoring system on the real-time system *deterministic*, if it cannot be completely removed. This interference depends on the *way* how monitoring data are *collected* from the target system, and on the *amount* and the *rate* of monitoring data being collected within an observation interval. The first objective of this thesis is to define monitoring data types and to present different gathering methods that

enables the monitoring system to predict this interference in advance[1] and to keep it deterministic.

The second objective of this thesis is the design and implementation of a *deterministic* monitoring system that can be used for monitoring of safety-critical distributed real-time systems, e.g., time-triggered systems. Different abstraction levels are supported, such as : i) *cluster*, ii) *node*, and iii) *transducer* abstraction level. Furthermore, the monitoring of target systems that consist of *multiple clusters* is part of this objective. However, as mentioned above, the basic requirement that must be taken into consideration during design and implementation of monitoring systems is their *influence*[2] on the target systems, which must be *limited* and *deterministic*. Thus, such an influence must not change the behavior of the target real-time system, neither in the time nor in the value domain.

A real-time monitoring system must be capable of collecting all monitoring data that are relevant to reproduce phenomena of interest. On the other hand, the amount of collected monitoring data must be kept small, for disk space and bandwidth reasons. An alternative to storing enormous amounts of monitoring data (by storing all monitoring data during a monitoring process) is the use of the *real-time trigger system* (RTTS) that stores data of interest selectively. The design and development of the RTTS is the *third objective* of this thesis. The RTTS records the system operation of the target RTS only in time windows of interest, around significant events. A *significant event* is an event of interest, which has to be defined by the user of the monitoring system. Examples of such events are: the *temperature* of a controlled physical system or the *velocity* of a (controlled) vehicle exceed their allowed limits.

1.2 Contribution of this Thesis

The contributions of this thesis are:

- The definition of different monitoring data types and gathering methods, which make it possible to predict monitoring resource requirements before run-time, i.e., during the monitoring setup phase before the monitoring process is started, and to keep the interference of the monitoring

[1] Before the monitoring process has been started.

[2] The presented monitoring system is based on a software monitoring approach in which the monitoring system uses resources of the target system during the monitoring process. This means that the target system is influenced by the monitoring system, or the monitoring and the target system are interfering during the monitoring process. Therefore, the terms monitoring *influence* and *interference* will be used alternatively throughout this thesis.

system on the target system deterministic. The practical applicability of these concepts is demonstrated in a monitoring application presented in Chapter 8, where they are used for calculation of the expected amount of monitoring data within a cluster cycle. Currently, the selection of entities, the calculation of the expected amount within an observation interval and the instrumentation process is done manually. The intention is to incorporate the concepts presented in this thesis in the design tools to calculate the needed resources (i.e., the calculation of the expected amount of monitoring data within an observation interval) for monitoring process and to automate the instrumentation process. This approach can be applied during design of time-triggered systems, e.g., TTA, FlexRay, TTCan. It can be also applied during design of event-triggered systems, e.g., CAN and LIN.

- The design and development of a deterministic real-time monitoring system that can be used for monitoring of time-triggered systems at different abstraction levels: i) cluster, ii) node, and iii) transducer abstraction level. The presented concepts are implemented both in the software tool TTPview (see Chapter 7), and in the monitoring application presented in Chapter 8. TTPview is successfully applied in industry (Honeywell, AUDI, VW, etc.) for monitoring of TTA systems.

- The design and implementation of RTTS (see Chapters 6 and 7). The RTTS can be used during long-term monitoring of target system saving disk storage. They also can be applied during automatic testing of new developed target system. Another important application field of RTTS in conjunction with the monitoring at the node abstraction level is the long-term diagnosing and correctness checking of target systems. In the context of long-term on-line diagnosing the RTTS can store the selected monitoring data (i.e., around the significant events) on a *dedicated non-volatile storage*. In the automotive industry this application of the RTTS enables the usage of *black-boxes* similar to the black-boxes that are used in the aircraft industry. In case a car has a problem, the evaluation of the monitoring data stored in the black-box helps the user to find the reason of the suspected problem.

1.3 Structure of the Thesis

This thesis is organized as follows: Chapter 2 presents the terms and notions that are used throughout this thesis. It starts with the short presentation of real-time systems with special view on real-time data, and their *temporal*

consistency. Then we continue with the presentation of the terminology that is used by the real-time monitoring research community, and the presentation of the survey of present and past real-time monitoring system architectures. At the end, this chapter presents some case-studies in which different real-time monitoring systems are shown.

Chapter 3 introduces the system model of target systems that can be monitored by the presented monitoring system. After presentation of the used terminology, the system structure is introduced. The next sections of this chapter present transmission medium, timing characteristics, and communication services of target systems that can be monitored by the proposed monitoring system.

The objective of Chapter 4 is to define and classify different groups of monitoring data, and for each group the monitoring system must be able to either exactly calculate or estimate their amount in advance. Furthermore, different gathering methods used for gathering of classified monitoring data types are defined and specified in this chapter. This classification of monitoring data and gathering methods contributes to make the interference that is caused by monitoring systems on target systems deterministic.

In Chapter 5 the proposed real-time monitoring system (RTMS) for monitoring of time-triggered systems is presented. It offers to the user the capability of monitoring the target systems at multiple abstraction levels: i) *cluster*, ii) *node*, and iii) *transducer* abstraction level. In addition, special attention will be paid on simultaneously monitoring of target systems that consist of multiple clusters.

In Chapter 6 the real-time triggers that are used for searching of significant events are presented. After the presentation of *trigger definition language* (TDL), which is used (by the user) for description of significant events, the triggered actions are introduced. The triggered actions are executed each time when the evaluation of trigger conditions yields `true`. Special attention will be paid in this chapter on the triggered recording actions, which have to record monitoring data collected before and after points in time at which significant events are found.

In Chapter 7 the implementation of the proposed monitoring system is presented. This monitoring system is implemented in a commercial software product called TTPview. TTPview has been shown to work very well and is successfully applied in industry.

Chapter 8 deals with the description of the design and implementation of a real-time monitoring system for monitoring of TTA target real-time systems at the *node* abstraction levels, i.e., at the *operating system* (*OS*) and the *application* abstraction levels. This serves as a *case study* for the presented concepts on

monitoring of TTA target systems at the above mentioned abstraction levels.

Finally, the thesis ends with a conclusion presented in Chapter 9, in which the key results of the presented work are summarized. This chapter also gives an overview on future research in this area.

Chapter 2

Real-Time Monitoring Systems

The objective of this chapter is to present the terms and notions that are used throughout this thesis. The next section begins with a short presentation of real-time systems with a special view on real-time data. Special attention will be paid to the notion *temporal consistency* of real-time data, because these data are *observed* and collected by monitoring systems.

The monitoring of real-time systems is the objective of the rest of this chapter. After presentation of the basic concepts and terminology that are used by the real-time monitoring research community, this chapter continues with presentation of a survey of present and past real-time monitoring system architectures. The goal of the survey is to present the state of the art of monitoring of real-time systems. At the end, this chapter presents some case-studies in which different real-time monitoring systems are illustrated. A brief summary closes this chapter.

2.1 Real-Time Systems

The correctness of real-time systems depends not only on the correctness of results they deliver, but also on points in time when these results are delivered. In [Kop97] Kopetz defines a real-time system as a computer system in which the correctness of the system behavior does not only depend on the logical results of the computations, but also on the physical instant of time at which these results are produced. In [Ram95, Ram96] Ramamritham describes the difference between real-time systems and non real-time systems as the presence of data that becomes invalid with the passage of time, the presence of events that must occur in a timely fashion, and the presence of actions whose timely completion is as important as the results produced.

2.1.1 Timing Constraints

The key-characteristic of real-time systems is the existence of timing constraints which such a system has to fulfil. These timing constraints are referred to as *deadlines*. Schütz in [Sch94b] defines a real-time system as a (computer) system which is required by its specification to adhere not only to functional requirements, but also to temporal requirements, often also called *"timing constraints"* or *"deadlines"*. In [Kop97] Kopetz defines a deadline as the instant at which the real-time system must produce its result.

Real-time systems can have *soft*, *firm* and *hard* deadlines, depending on the fact, how the system is affected in case these deadlines are violated. If a real-time system can use the result produced after the deadline has been missed, then this deadline is called soft. The deadline is considered to be firm, if the result produced after deadline cannot be used by the real-time system. A special kind of firm deadlines are hard deadlines, which must never be violated. Missing a hard deadline can have catastrophic consequences and must be avoided at all costs [vdSvdWA+97]. In [Kop97] Kopetz determines that a real-time computer system that must meet at least one hard deadline is called a *hard real-time* computer system or a *safety-critical* real-time computer system.

2.1.2 Real-Time Data

The type of data that lose their validity with the passage of real-time are called *real-time data*. The goal of real-time systems is to process real-time data in real-time, before they become invalid. Real-time data model the real-word processes that are controlled by real-time systems. These data are valid only within a finite *short* time duration. The time point after which real-time data lose their validity is called *data deadline* [KSS02]. Since the validity of real-time data is lost with the passage of real-time they must be updated after a certain time, in order to be able to reflect the actual state of the environment. Some real-time systems are used for controlling of physical devices, and therefore they must store real-time data which represent the condition of these devices. Such information includes input data from devices as well as system and machine state [Kim95, Son95].

Real-time data can be produced from different sources. Some examples are: i) real-time data that represent the position (3 dimensional) and the velocity of an aircraft, etc, ii) data that represent the temperature in a power plant, iii) data found in the navigation system of a spacecraft.

Real-time data can be classified into two different groups: *basic* and *derived* data. Basic real-time data are gathered directly from sensors from the environ-

ment physical processes, which are controlled by real-time computer systems. Derived data are derived from different basic data.

Temporal Consistency

Real-time applications typically consist of the *controlling system* (CS) and the *controlled system*. The controlling system uses a (finite) set of real-time data, which reflect the state of the controlled system. The controlled system can be viewed as the *environment* with which the controlling system interacts [Ram93]. The designer of the CS must make sure that the state that is reflected by the contents of its real-time data and the actual state of the environment are consistent otherwise the CS could take false decisions, which could have catastrophal consequences. In [Ram96] Ramamritham defines the notion of *temporal consistency* in real-time databases, which represents the consistency between the actual state of the environment and the state reflected by the contents of the database.

The temporal consistency of real-time data consists of two components [Ram93, Ram96]:

Absolute Consistency (AC): AC represents the imperative need to keep the controlling system's view of the state of the environment consistent with the actual state of the environment. The absolute consistency can be understood as the imperative constraint for the controlling system to periodically refresh its real-time data, which represent the actual state of the environment.

A real-time data item, e.g., d, contains the following characteristics:

$$d : (value, avi, timestamp) \tag{2.1}$$

where d_{value} denotes the current state of d, and $d_{timestamp}$ denotes the time when the observation related to d was made. d_{avi} denotes d's *absolute validity interval*, i.e., the length of the time interval following $d_{timestamp}$ during which d is considered to have absolute validity [Ram96, Ram95].

Relative Consistency (RC): RC represents the need for (basic) real-time data which are used for derivation of other (derived) real-time data to be temporally close to each other. A set of data items used to derive a new data item form a *relative consistency set*. Each such set R is associated with a *relative validity interval* denoted by R_{rvi} [Ram96].

Ramamritham in [Ram96] determines that a real-time data item ($d \in R$) is considered to be temporal consistent if and only if:

- Absolute Consistency:

$$(current\_time - d_{timestampt}) \leq d_{avi} \tag{2.2}$$

- Relative Consistency:

$$\forall d^{'} \in R, |d_{timestamp} - d^{'}_{timestampt}| \leq R_{rvi} \tag{2.3}$$

The correctness of the decisions that are made by the controlling system depends directly on the temporal consistency of the used real-time data. Gerber et al. in [GHS95] define the *freshness* and *correlation constraint* of real-time data. The freshness constraint represents the time after which the data may not be used anymore. The correlation constraint represents the time window within which the correlated real-time data[1] must be generated. The absolute consistency can be seen as freshness constraint whereas relative consistency can be seen as correlation constraint [Ram96].

Monitoring data that are gathered from real-time target systems must be relative consistent. These data are correlated by monitoring systems for representation of the run-time behavior of the target systems at the intended abstraction level. Therefore, the relative consistency is of utmost interests for monitoring of real-time systems.

2.2 Monitoring Systems

During the monitoring process run-time information are gathered from the target system which represent the behavior of the target system at the intended abstraction level. Monitoring systems are used in run-time for extraction, analyzing, presentation and (in some cases) for acting on gathered information from systems that are monitored [Pla84, JLSU87, Sch95, HS90, WLS99, MSS92].

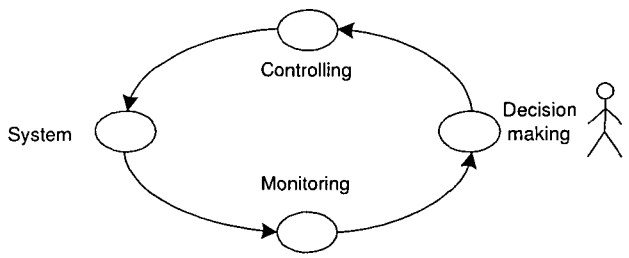


Figure 2.1: Relationship between Management and Monitoring [Hof94]

[1] These data are derived from the basic real-time data.

In literature we find different definitions of monitoring. In [Sch94a], Schmid defines monitoring as the process of extracting and gathering information regarding the behavior of a particular system. In [JLSU87], Joyce et al. define monitoring as the process of dynamic collection, interpretation and presentation of information concerning objects or software processes under scrutiny. In [Sch95], Schroeder defines the *on-line* monitoring as a system, which is a process or set of possibly distributed processes whose function is the dynamic gathering, interpreting, and acting on information concerning an application as that application executes. In [TBYS96], Tsai et al. define monitoring as collection of run-time information on the target system that cannot be obtained by merely studying the program text. Haban and Shin in [HS90] define monitoring as the extraction of data about the activities of a computer system. Another definition of monitoring is given by Kaelbling and Ogle in [KO90]. They define monitoring as the collection of information from *targets* (either programs or underlying systems) and presentation of analysis based on that information to users or programs. Dodd and Ravishankar in [DR92], define monitoring as the measurement, collection, and processing of information about the execution of tasks in a computer system. In [UB02], Urting and Berbers define monitoring as an activity of observing the (target) system's properties and activities, analyzing the information and eventually controlling (part of) the (target) system based on the collected information. Hoffner in [Hof94], determines that monitoring is carried out in order to obtain information about a system, and in general monitoring is part of the process of management (see Figure 2.1).

However, the monitoring process is defined in this thesis as a process during which the run-time behaviour of the target system is observed at an intended abstraction level, i.e., its run-time information is gathered by the monitoring system. In addition the collected run-time information is processed by the monitoring system either at run-time (i.e., on-line) or off-line, and the processed results are presented the user.

2.2.1 Terms and Notations

It is the objective of this section to present the terms and notations that are used in the context of real-time monitoring and which are found throughout this thesis.

Target System

The system that is monitored by the monitoring system is called *target system* [Pla84, PP00]. The application that is executed on the target system is

called *target application*. The *monitoring system* is a system that is used for monitoring of *monitored* systems (i.e., target systems) [TBYS96].

Sensors and Probes

In [UB02], Urting and Berbers define a *sensor* as an entity (software or hardware) that observes a small part of the system, and which is responsible for gathering of monitoring information. In [Sch95], Schroeder defines a sensor as an entity that observes the behavior of a small part of the application system state space. Sensors can include additional functionality. For example, the user can define conditions that must be satisfied before sensors start their operation for gathering monitoring information.

A sensor collects (gathers) monitoring information when it is triggered. In [Sch95] Schroeder denotes that the triggering process can be done either by the change to the entity the sensor observes or by a request from the monitoring system. Depending on how the sensor is triggered, there are two different types of operations that can be done by the sensors: *sampling* and *tracing*. In [KO90], Kaelbling and Ogle define sampling as an asynchronous operation, which is started on-demand by a monitoring system for gathering of monitoring information. If a monitoring system decides to collect monitoring information from a set of sensors, it sends a message to them, and they return the current values from entities they are observing. On the other side, tracing is a synchronous operation, during which the sensor reports the new value of the entity it observes, when the state of the entity is changed. In [DR02], Dias and Richardson state that by sampling, information about the execution *state* is collected synchronously (in a specific time rate), or asynchronously (through the direct request of the monitoring system), while by tracing, information is collected when an *event* of interests occurs in the system [OSS93]. Sampling sensors can be applied in applications where there is no need for collecting of all events. However, in cases where collection of all events must be guaranteed, then tracing sensors must be used. In this case the sampling rate of the sensor must be higher than the fastest expected change rate of the entity the sensor observes [UB02].

Sensors are widely used by software monitoring systems. An example is given in [Mah01] by Mahrenholz, where the use of two different types of sampling sensors is presented: *block* and *expression level* sensors. He uses block sensors for collecting information about entries into and/or exits from basic code blocks (e.g., functions, etc), while expression level sensors are used for collecting information on passing of a single code position or the state of a variable at this point.

In [OSS93], Ogle et al. differentiate *sensors* from *probes*. Whereas sensors are small pieces of code residing within the program being monitored, the *probes* are code fragments residing within the *resident monitor* (rather than application), i.e., the resident monitors are responsible for gathering of run-time information from target systems. If the monitoring system uses probes then the code of the target application being monitored does not need to be changed.

Events, Actions and Event Histories

Jahanian [Jah95], Chodrow et al. [CJD91], and Raju et al. [RRJ92] determine that a computation of a real-time system can be viewed as a sequence of event occurrences that represents things that happen in a system.

In [Sch95] Schroeder describes an event as an activity, which usually involves just a small part of the application state space. In [Jah95] Jahanian determines that events denote state changes in a system as seen by the monitoring system. In [Kop97] Kopetz defines an event as an occurrence (a state change) that happens at a cut of a directed time line that extends from the past into the future, and from which the flow of real-time can be modelled. In [Kla92] Klar defines an event as an atomic instantaneous action. In [UB02] Urting and Berbers define an event as a *unit of observation* in an event based monitoring system. In [MSS93] Mansouri-Samani and Sloman define an event as an atomic entity that reflects a change in the status of an object. In [LM97] Liu and Mok state that events represent *state changes of interests* that may occur in a system. "End of transaction T"is an example presented by Liu and Mok illustrating events that may occur in a system. In [Sch94a] Schmid determines that an event characterizes the occurrence of a specific change of the system state - expressible via a certain *predicate* - of the target. Furthermore, he differentiates between the *occurrence time* and the *recognition time* of events. The recognition time represents the point in time at which the monitoring system becomes aware of the occurrence of the event. In [RRJ92] Raju et al. determine that events represent things that happen in a system, e.g., an event may denote the start/completion of a program segment, reading a new sensor value into a program variable, receiving a message from another task, etc.

Events have been differently classified by different researchers. Urting and Berbers in [UB02] distinguish between *non-monitored* events (these are events that occur in the target system but are not reported to the monitoring system), and *monitored* events that are collected and processed by the monitoring system. Mansouri-Samani and Sloman in [MSS93] classify events according to their level of abstraction into:

- events are considered *primitive* if they signify simple changes in the state of an object, and
- *combined* events are a combination or grouping of primitive or combined events.

In [Sch95], Schroeder classifies events into three primary categories: i) *hardware-level*, ii) *process-level*, and iii) *application-dependent* events. This kind of distinction presented by Schroeder corresponds to the abstraction level of the monitoring system. Another classification has been presented in [Jah95] by Jahanian, where events are grouped into two categories: *task* and *run-time system* events. *Task* events denote the state changes of an existing task. Examples of these events include assignment of a value into a variable or receipt of a message. *Run-time* system events are a set of predefined state changes in the run-time system. Examples of these events include preemption of a task or blocking a task for a resource.

Depending on the monitoring approach (see Section 2.2.6) events can be generated from different sources. In hardware monitoring approaches such events can be generated by low-level parts (i.e., hardware "probes") of the system that snoop the busses of the target system. In software monitoring systems, events are generated either by probes or sensors that are inserted into target systems.

In [Sch95], Schroeder determines that monitoring systems collect monitoring information in form of events. The relevant events occurring in a target system are time stamped, collected, and analyzed by a monitoring system at run-time. Furthermore, the collected events are stored into *event histories* for analyzing of the past history and the output of the analyzing process can be used for feedback to the target system or correctness checking of the target system (see Section 2.2.2) [CJD91, RRJ92, Pet97]. In [HKM+94] Hofmann et al. determine that whenever the monitor device recognizes an event, it stores a data record (a so called *event record*), which contains the information *what* happened *when* and *where*. The sequence of events is stored as an *event trace*.

Schroeder [Sch95] determines that an *action* is the response of the monitoring system to an event or set of events. Possible actions are: logging or recording a particular event or set of events, starting a particular process or activity to deal with the event(s), starting user or administration interactions, and so on [UB02].

Jahanian in [Jah95] observes that monitoring systems manage collected monitoring data (i.e., events) into *event histories*, which are used for analyzing the past history. If the monitoring system is used for feedback monitoring then the produced results are sent back as feedback into the target system. The size

of the event history depends on the application that is being monitored, on the monitoring approach and monitoring intention, and on the size of the available memory in the system [Sch95, UB02].

Intrusive vs. Non-intrusive Monitoring

A monitoring system is called *intrusive*, if it uses resources of the target system during gathering of run-time information. This is in case a small piece of code (called *sensor*, see Section 2.2.1) is inserted into the target system for gathering of monitoring data. On the other hand, a monitoring system is called *non-intrusive* if it does not use any resources of the target system for information gathering. This can be achieved by use of a dedicated hardware for information gathering from the target system. Examples of non-intrusive monitoring approaches can be found in [Pla84, TFC90, TFCB90, SL01].

Schroeder in [Sch95] defines a monitoring system as *intrusive* if it requires the use of application resources (i.e., CPU, I/O devices, or shared communication channels, etc.). Thane in [Tha00a] determines that any intrusive observation, or probing, of the distributed real-time system affects the timing and consequently the outcome of the races. In literature the effect of intrusive monitoring on target system is called *monitoring perturbation*, which is explained in more detail in Section 2.2.3.

Time- vs. Event-Driven Monitoring

Depending on how the monitoring information is *collected*, monitoring systems are classified into *time-driven* (or *time-based*) and *event-driven* (or *event-based*) monitoring. Time-driven monitoring systems periodically gather monitoring information from target systems (i.e., the state of target systems is periodically *sampled* by the monitoring system), while event-driven monitoring systems collect occurrences of events of interests from target systems at the times when these events occur.

In the research community we find researchers who have analyzed advantages and disadvantages of time-driven and event-driven monitoring systems. In [MSS92, MSS93], Mansouri-Samani and Sloman define the time-driven monitoring as a monitoring process which is based on acquiring periodic status information to provide an instantaneous view of the behavior of the target system, while the event-driven monitoring is defined as a process which is based on obtaining information about the occurrence of events of interests which provide a dynamic view of system activity. In [Sch95], Schroeder defines an event-based monitoring system as a monitoring system in which the gathered information arrives in the form of events. In [HKM+94], Hofmann et al. denote

that event-driven monitoring represents the dynamic behavior of a program by a sequence of events, while time-driven monitoring (sampling) provides only summary statical information about program execution. Furthermore, they state that the event-driven monitoring is much more suitable in comparison to the time-driven monitoring for *gaining insight* into the dynamic behavior of a target system. In [Hof96], Hofmann points out the difference between time-driven and event-driven monitoring. In the time-driven monitoring approach the monitoring system periodically has to *sample* the *state* of the target system at a fixed rate. Hofmman observes that *for getting insight* into the dynamic behavior of target systems, the time-driven monitoring approach requires a *very high* time resolution, and a fairly long interval for recording. Therefore, he states that the use of a time-driven monitoring approach would result in a *high* data rate with a *huge* amount of data, while this amount could be reduced by several orders of magnitude if an event-driven monitoring approach is applied.

The advantages and disadvantages between time-driven and event-driven monitoring *strongly* depend on the architecture of the target system. If the target system is based on the event-triggered architecture, then event-triggered monitoring approach will be used. In this case the target system can be viewed as a finite set of partially ordered event occurrences, which represent its system behavior and the monitoring system has only to collect this finite set of event occurrences. On the other hand, time-driven monitoring systems can be much more suitable for target systems based on the time-triggered architecture.

Synchronous vs. Asynchronous Monitoring

Depending on how collected monitoring data are *processed* the monitoring approaches are divided into *synchronous* and *asynchronous*. A monitoring system is called synchronous if collected monitoring data are processed by a monitoring system running on the target system. Therefore, the synchronous monitoring can be seen as intrusive monitoring, because it uses the resources of the target system for processing of collected monitoring data.

A monitoring approach is called asynchronous if collected monitoring data are processed by a monitoring system running on a separate task or machine. For processing of collected monitoring data these monitoring systems do not occupy any resources of target systems. Therefore, in general the influence of asynchronous monitoring systems on target systems is smaller[2] in comparison with the influence produced by synchronous monitoring systems.

[2]Only during gathering of run-time information the target system can be influenced by the monitoring system.

Run-Time Monitoring

Chodrow et al. [CJD91] denote that the run-time monitoring of a real-time target system is achieved by examining the observable events at run-time. Raju et al. in [RRJ92] denote that run-time monitoring of a system requires time-stamping and recording of the relevant event occurrences, analyzing the past history as other events are recorded and provide feedback to the rest of the system.

In general, we can define the run-time monitoring as a process that has to collect *relevant* events (i.e., monitoring-data at the intended abstraction level) from the target system and to process these collected events at run-time. In the literature we can find examples, in which run-time monitoring systems are presented [JG90, CJD91, JRR94, Jah95, BJHL96, ML97, LBAK+98, KVBA+99, WLS99] (see Section 2.2.2). In most cases these monitors use *Real-Time Logic* (RTL) [JG90] for the definition of timing constraints which are checked by run-time monitors. The run-time monitor presented by Jahanian et al. in [JRR94] can be used as an example. In this model a system computation is viewed as a sequence of event occurrences. The design assumptions and system properties that must be maintained are expressed as invariant relationships between various events which are monitored at run-time. In this model, if a violation of an invariant is detected the target system is notified so that suitable recovery actions can be taken.

In [UB02], Urting and Berbers state that run-time monitoring systems consist of three important parts:

- *Annotation Formalism*: an RTL like formalism for annotating timing constraints,
- *Probes*: are responsible for intercepting events and time-stamping them, and
- *Satisfiability Checker*: is responsible for checking of the RTL constraints making use of the events generated by the probes.

Deterministic and Reproducible Monitoring

For the purpose of debugging of the target systems *deterministic and reproducible monitoring* is very important, because it enables the user to *deterministically* reproduce the behavior of the target system at an intended abstraction level. This is very helpful in case of erroneous behavior, because it helps the user to locate the faults on the target system. In order to achieve that the monitoring system must be able to collect *all needed* information from which the

run-time behavior of a target system at the intended abstraction level can be represented. However, the exact reproduction of the target behavior strongly depends on the architecture (and nature) of the target system. Deterministic and reproducible monitoring of a target system that can be regarded as deterministic is *easier* than the monitoring of a target system that cannot be regarded as deterministic. In [Tha00a, Tha00b], Thane defines a system as deterministic if an observed behavior, $\mathbb{P}$, is uniquely defined by an observed set of necessary and sufficient parameters/conditions, $\mathbb{O}$.

Obermaisser et al. in [OPEL01] determine that deterministic monitoring means that all conditions are observed which cause a specific system behavior. Therefore, to achieve deterministic monitoring, all entities must be observed (see definition of entities in [Kop97]) with respect to contents, order and timing. The reproducibility of conventional sequential programs can be achieved, if we start them with the same initial state and provide the same inputs. In a real-time system, there is the additional requirement for reproducible timing behavior. Therefore, the inputs have to be reproduced with respect to contents, order and timing.

Sequential programs, for example, are usually regarded as having deterministic behavior, i.e., given the same initial state and inputs, the sequential program will consistently produce the same output on repeated executions, even in the presence of systematic errors [Tha00a, Tha00b]. As opposed to the sequential programs, (most of) parallel programs (systems) are regarded as being in general nondeterministic. Therefore, a major problem in debugging parallel systems is the impossibility to guarantee the identical behavior of multiple system's executions, even if identical input data are provided. This is especially problematical with real-time systems, since there are external events which usually cannot be reproduced identically [Sch94a].

In the literature we can find solutions which are based on some special architectures that offer the possibility for re-execution of the target system by using the collected monitoring data. In these architectures the subsystems of interest are *disconnected* from the rest of the system and they are redirected to the monitoring (debugging) system, which manages the collected monitoring information [TFCB90, DR92]. This means that parts of system that *directly* communicate with system's environment are redirected to the monitoring system, which makes sure that the system's inputs are exactly reproduced as they were during the gathering phase. This issue will be dealt with in more detail in Section 2.3.3.

2.2.2 Application Domains

Run-time information that is gathered by monitoring systems from target systems represents their run-time behavior at the intended abstraction level, and it can be used for different purposes. Schroeder [Sch95] and Schmid [Sch94a] have presented different application domains, where monitoring systems can be applied. In this section we will categorize the application domains of monitoring systems in following categories: *testing* and *debugging*, *feedback monitoring*, *correctness checking*, and *performance evaluation.*

Testing and Debugging

Using monitoring systems to assist testing and debugging of real-time systems has drawn high attention of the monitoring research community. The debugging of real-time systems is much more complicated than the debugging of non real-time systems, because the correctness of real-time systems depends both on the temporal and value domain. The most popular traditional debugging technique is the *breakpoint* technique, which uses breakpoints to stop the system execution at predefined points in code. This technique is not *suitable* for debugging of real-time systems because, if the program execution is interrupted, the physical time cannot be stopped and the system is no longer consistent in the temporal domain. Therefore, the debugging of real-time systems cannot be done with this traditional debugging technique.

In the last few years different debugging techniques for debugging of real-time systems were developed. These techniques can be classified into *static* and *dynamic* debugging techniques. The static debugging techniques are based on the static analysis of the source code. They try to detect parts of code that probably contain anomalies and (or) bugs. The *dynamic debugging* techniques are based on the monitoring techniques. The monitoring system collects run time information from the target system, and the developer can analyze the collected run-time data off-line in order to find the bugs in the target system. Some examples of using monitoring systems for debugging purposes can be found in [TFCB90, TFB90, Gor91, DR92, Bor92, Sch94a, MW94, Sch94b, Maj95, TB96, Tha00a, Tha00b, TH00, SL01, AKMM01]. The debugging of real-time systems is presented in more detail in Section 2.3.

Feedback Monitoring

There are some cases in which the output produced by the monitoring system is used as a feedback by the target system to make decisions for further processing. This kind of monitoring application has been applied especially to assist the

operating systems's scheduler aiding it to schedule task sets. Some examples of feedback monitoring can be found in [HS90, Jah95, DMW98, WLS99, KLS+02]. Feedback monitors are usually run-time monitors that can be seen as a part of real-time target systems.

Correctness Checking

Monitoring systems can be used to ensure the consistency of the target system with its formal specification [Sch95]. These monitoring systems can be considered as feedback systems, too, because they can be used to trigger the target system to start some recovery actions if the monitoring system detects an invariant violation [RRJ92]. However, their primary goal is to observe and make sure that target systems are running in conformance to their formal specifications. Some examples of monitoring approaches of this category can be found in [JG90, CJD91, RRJ92, Lut92, JRR94, BJHL96, ML97, BJWL97, Pet97, SS97, LBAK+98, KVBA+99, PP00, PP02, UB02, KLS+02].

Performance Evaluation

Monitoring systems gather run-time information for assessing system performance. Some examples in which monitoring systems are used for performance evaluation of target systems are found in [HW90, CP98, VW02].

2.2.3 Monitoring Perturbation

A monitoring process can be defined as an observer that observes the execution of the target system. Fidge [Fid96] defines an *observer* as any entity - a person or a network process - that attempts to examine a computation. An observer may watch the system while the computation is in progress or examine a post-mortem event log or trace. The ideal monitor (or the ideal observer) would not at all disturb the target system. However, it is very difficult in practice to design and develop a monitoring system that does not impact the target system. In [Blo01], Blom points out that one of the most important issues during a monitoring process is that the original target system which is being monitored must not be disturbed in any way. If it is impossible not to disturb the target system, then it is of *greatest importance* to have full control over the impact caused by the monitoring system.

During the information gathering, the monitoring system can perturb or influence the target system. This perturbation is called *monitoring perturbation*, which is a quantitative representation of the perturbation that the monitoring

system causes on the target system. Tsai [TFC90] states that any interference in the monitoring activity in the real-time distributed environment is *intolerable*. This is very important because the interference caused by the monitoring system can change the temporal behavior of the target system. In this case, the gathered information would represent the run-time behavior of the target system at the intended abstraction level, which is not compatible with the state of the target system, if it would not be observed by the monitoring system.

Monitoring perturbation changes the ordering and timing of events on the target system, on which the system's correctness depends. Tsai et al. [TBYS96] conclude that the ordering of events refers to the sequence of events and the timing of events refers to the time when an event occurs. The ordering of events can be classified into *partial* and *total* ordering. A partial ordering is a *local* sequence of events occurring within a processor, while the total ordering is a global sequence of all events occurring in a (distributed) system [TBYS96]. In [Kra00], Kranzlmüller determines that ordering of events in a single processor system is trivial, because a correct causal order is established by adopting the event occurrence time. However, the ordering of events in distributed systems is difficult through the drifts of local clocks of the distributed system. Therefore, the basic prerequisite for a successful total ordering of events in a distributed system is the use of global time [Kop92, Kop97].

Thane [Tha00b, Tha00a] and Blom [Blo01] present some examples in which the influence and consequences of monitoring perturbation on the target system are illustrated. In both examples a task set is presented. The task execution order of the task set is changed through the monitoring perturbation, which change the temporal behavior of the target system. The disturbances caused by the observation activities of the monitoring system on target system are indeterministic. They can bring the target system to suffer from incorrectness or, in the worst case, to miss its deadlines. These indeterministic disturbances can be found in the literature as *probe-effect* [Gai86, MH89, Tha00b] or as *Heisenberg uncertainty* [LP85]. *Probe effect* is the effect caused by the interference that occurs when a program's execution is monitored [TCO91]. Thane [Tha00b] indicates that it is of great importance to ensure that the actual act of observation does not disturb or intrude on system behavior. If the observations are intrusive then it is imperative that their effect can be calculated and compensated for. If we cannot guarantee this, there is no guarantee that the observations are accurate or reproducible [Tha00b]. Furthermore, Kirschbaum et al. [KBG98] point out that the special constraint of monitoring real-time systems is to keep the interference with the target system *so small* that the change in system performance due to the hardware-monitor will neither affect the order nor the timing of events.

Related Work

In literature the monitoring perturbation has drawn the attention of the research community. There are some proposals and solutions that try:

- *Not to* **impact** *the target system.* Most hardware monitoring approaches can be grouped in this category. Hardware monitoring approaches use a dedicated monitor hardware device which guarantees that the monitoring system does not use any resources of the target system. Some examples of these approaches can be found in [Pla84, TFC90, TFCB90, HS90].
- *To keep the perturbation caused by the monitoring system* **"small"**. Samples of monitoring approaches that belong to this category can be divided into:
  - *Hybrid Monitoring*: It consists of a combination of hardware and software approaches. These systems use a specialized (dedicated) hardware and therefore the aim of these systems is to keep the interference with the target system small. Some examples of such approaches can be found in [HW90, Gor91, CP98, SL01].
  - *Perturbation Analysis*: It is used to predict the effect of the monitoring influence on the target system. Tsai [TBYS96] denotes that perturbation analysis examines event ordering and timing in an attempt to find ways to reduce the effects of monitoring interference by adjusting the event ordering and event timing. Further information can be found in [TBYS96, MRW92, SG94, WSG96, WSG98].

2.2.4 Monitoring Levels

Monitoring systems must provide a presentation of gathered monitoring information at different abstraction levels in order to assist users for better understanding of the run-time behavior of target systems. Tsai et al. [TFB90] point out that abstract views of the system are an essential method to manage complexity. For example, higher-level information refers to events such as interprocess communication and synchronization, while lower-level information refers to events such as the step-by-step execution trace of a process [TBYS96].

In the literature we can find related work on different monitoring abstraction levels. For example, Gorlick in [Gor91] presents three different monitoring abstraction levels:

Kernel-level monitoring: In the kernel-level monitoring the process dispatching is an important form of monitoring, which can be characterized by four events:

- *assigning* a ready process to a processor,
- *relinquishing* a processor and returning a process to the ready queue,
- *blocking* a running process, and
- *unblocking* a waiting process.

Language-Level Monitoring: Monitoring at the language-level is quite the same as the monitoring at the *function-level* presented further below.

Application-Level Monitoring: At the application-level the source-level execution of concurrent programs can be reconstructed.

In [Tha00b, Tha00a], Thane categorizes the gathered monitoring information into three groups: *data-flow*, *control-flow* and *resources*. In [UB02], Urting and Berbers categorizes the collected monitoring information into three groups: *hardware-level*, *process-level* and *application-level*. Monitoring information is categorized by Mansouri-Samani [MS95] into three groups: *control-flow*, *data-flow*, and *process-level*. The classification used here will be presented in more details in Chapter 4, where the monitoring data are discussed in detail.

In [TFCB90, TFC90, TFB90, TBYS96], two different monitoring abstraction levels are presented: *process-level* and *function-level*. In this section we will discuss only monitoring at the *process* and *function* abstraction levels.

Process-Level Monitoring

For monitoring program execution at the process-level, Tsai et al. [TBYS96] consider a process as a black-box which can be in one of the three states: *running*, *ready*, or *waiting*. Furthermore, for process-level monitoring, Tsai et al. distinguish the events that affect program execution (e.g., interprocess communication and synchronization events) from those events that affect execution at lower levels (e.g., assigning a value to a variable, procedure calls, etc.).

Tsai et al. [TBYS96] present two reasons for the necessity of monitoring of target systems at the process abstraction level:

Nondeterminism: Processes are the minimum program unit that can exhibit nondeterministic behavior. Furthermore, the interactions between different processes on one hand and the interactions between processes (of real-time applications) and their environment, on the other hand, are frequently sources of possible faults. If the user can isolate anomalies on an individual process then he can use monitoring at lower-levels of abstraction for successive fault isolation [TBYS96].

Reconstruction: The execution behavior for interprocess communication and synchronization and the interaction between the software processes and the environment using the collected process-level events can be reconstructed [TBYS96].

In [TFCB90, TFC90, TBYS96], some events are presented that can be collected during monitoring of target systems at the process abstraction level: *creating process*, *terminating process*, *process synchronization*, *process state change*, *interprocess communication*, etc. These events can contain several subevents.

Function-Level Monitoring

The user can monitor the target system at lower abstraction levels for localization of possible faults, after they have been identified in a process. One of this monitoring levels is the monitoring at the *function abstraction level*, at which the user can try to localize the faulty modules or functions. Tsai et al. [TBYS96] point out that functions (or modules) at the function level of abstraction are the basic units of the program model and each function can be viewed as a black box that interacts with others by calling them, or being called by them, with a set of parameters as arguments.

At the function-level monitoring there are only two events of interests: *function call* and *function return*. Both of these events have their corresponding key values (e.g., *function call* has following key values: *calling function identification*, *called function identification*, *passed-in parameters*, and *time*) [TFCB90].

2.2.5 Monitoring Targets

The goal of this section is the presentation of requirements and difficulties for monitoring (or debugging) systems. The target systems that are presented below are considered to be real-time systems, i.e., they have to fullfil their timing constraints. Dodd et al. in [DR92] point out that a real-time system requires from the monitor itself to operate under strict reliability and performance constraints. The reliability constraints require that the monitored system and the monitor continue to operate in the presence of faults, while performance constraints require that the interference caused to the system by the monitor's presence must be predictable, minimal, and bounded [DR92].

Single-Task (Sequential) Target Systems: In [Tha00b, TH00], Thane indicates that the reason, why traditional debuggers cannot be used for

monitoring of sequential real-time applications, is the impossibility to directly reproduce the inputs to the system that depends on the time, when the program is executed. Such examples are the reading of sensors and the local real-time clock.

To successfully debug these target applications Thane [Tha00b, TH00] proposes a monitoring and debugging system which collects all significant events at run-time (e.g., reading values from an external process, accesses to a local clock, etc.). During the debugging process, the monitoring system has to *short-circuit* and redirect all system's inputs to the recorded events. An alternative to the approach presented by Thane is the use of a simulator of the external processes, and to synchronize the time of the simulator with the debugged system.

Multitasking Target Systems: Thane in [Tha00b, TH00] determines that during debugging (or monitoring) of multitasking applications, in contrast to the debugging of sequential applications, the mechanisms for reproducing of task *interleaving* must be supported by the monitoring (or debugging) system.

Distributed Target Systems: Tsai et al. in [TFC90] define a real-time distributed computing system as a system that consists of a collection of communicating and cooperating processors or computers (nodes) that work toward a common goal, and on which critical timing constraints are imposed. Kirschbaum et al. in [KBG98] determine that the special constraint of monitoring distributed real-time systems is to keep the interference with the target system *so small* that the change in the system performance due to the monitor will neither affect the order nor the timing of events. Although in the most distributed systems there is no total order defined over events that occur on different nodes, monitored data must be collected from several sites and integrated to obtain a coherent view of the system [DR92].

In literature we can find different related work in which real-time monitoring of distributed real-time systems are presented, e.g., are [TFC90, For90, HW90, DR92, Tha00a, Tha00b, TKM88, TFCB90, RRJ92, JRR94, Sch94a, LM97]. In these works the difficulties (among other things) are indicated, which one has to overcome during monitoring (or debugging) of distributed real-time systems.

Urting and Berbers in [UB02] for example point out that there are two important problems during monitoring of distributed systems: *clock synchronization* and *delay*. The problem of clock synchronization is a *well known* problem in distributed systems which consist of multiple nodes, and each of them contains a local clock that can drift from the clocks of

the other nodes with the progression of physical time. The delay problem is caused by the potentially conflicts between different nodes during sending of their messages.

Raju et al. in [RRJ92, JRR94] present four fundamental issues that need to be addressed when monitoring distributed real-time target systems. These issues are listed briefly here:

- *Time of Detection of Violation:* Detecting violations as early as possible is a desirable property because it allows the system to take corrective actions before the violation actually happens.
- *Number of Messages:* Messages are used for communicating event occurrences. Minimizing the number of these messages is crucial for reducing overhead.
- *Clocks and Timer Granularity:* When an event occurs it must be timestamped, which is done by reading the clock on the local processor. For this reason the clocks of a distributed system must be synchronized.
- *Resource Management:* Another fundamental aspect of distributed monitoring involves the need to quantify the timing intrusiveness of the monitoring activities on the timing behavior of the real-time application.

Moreover, Tsai et al. in [TFC90] note that the monitoring of distributed real-time computing systems is much more complicated than the monitoring of centralized, sequential computing systems because of:

- *Multiple Asynchronous Processes:* Since real-time distributed computing systems feature asynchronous parallel processes, their computation shows nondeterministic and irreproducible behavior. This behavior, caused by race conditions and unpredictable synchronization among processes, makes the results observed in a distributed computing system harder to understand, and in many cases, hard to reproduce.
- *Critical Timing Constraints:* Since, for a real-time distributed system, the correctness of the system depends on its behavior with respect to time, any interference of the monitoring system with it is intolerable.
- *Significant Communication Delays:* The nodes of a real-time distributed computing system can be geographically dispersed, which may introduce a significant communication delay. This delay can

cause improper synchronization among the nodes and make it difficult to determine the precise global time and accurate global state.

2.2.6 Monitoring Approaches

Monitoring systems use different approaches for gathering of monitoring data, on which the influence of the monitoring system caused on the target system directly depends. For the purpose of gathering of monitoring data the target system may need to be instrumented, and depending on this fact, the monitoring approaches can be classified into: *hardware*, *software*, and *hybrid* approaches. The hardware monitoring system separates the monitoring overhead from the target system's workload, whereas the software monitoring adds this overhead to the target system's workload. This is the main difference between these two monitoring approaches. Hybrid monitoring is a combination of hardware and software monitoring.

Hardware Monitoring

Hardware monitoring systems use dedicated hardware for gathering of *significant events* from target systems. The dedicated hardware snoops the busses of a target system and tries to match the bus signals. After successfully matching of significant events, they are stored for post-processing. Hardware monitoring systems are proposed in [Pla84, TFB90, TFCB90, TFC90, KBG98, SL01].

The advantages of the hardware monitoring approach are:

Low Interference: Hardware monitoring systems cause no or only *minimal* intrusion on the execution of target systems, because the event detection is done within the monitoring hardware and its control module, which does not share any computational resources with the target system [KBG98].

No Instrumentation: Hardware monitoring systems do not need to make any modification (instrumentation) of the target application or operating system [KBG98, TBYS96].

The disadvantages of hardware monitoring systems are their costs and the lack of portability, which is because the monitoring is carried out at the electronic signal level and the dedicated hardware depends on the target system, or at least on the target's processor [TBYS96]. Another drawback of hardware monitoring systems are the inability to deal with cache memories, which are used by all modern CPUs, and thereby they make impossible the use of bus

snooping, because the modern CPUs do not have to access memory after each program instruction. The write-through of an on-chip cache can arbitrarily delay the appearance of data store on the bus [Gor91].

Software Monitoring

In contrast to hardware monitoring, software monitoring systems use only software for gathering of monitoring information and therefore, these monitoring systems share the computational resources with their target systems. The target systems must be instrumented with sensors or probes presented in Section 2.2.1. These sensors or probes are executed on target systems and they are used for detection of significant events, which are stored in the memory of target systems. Examples of software monitoring systems can be found in [TKM88, KO90, For90, CJD91, DR92, RRJ92, JRR94, BOSS95, Maj95, LBAK+98, NGM98, KVBA+99, Tha00a, Mah01, MSSP02, DR02].

The advantages of the software monitoring approach are:

Flexibility: Software monitoring systems are flexible, i.e., one can use software monitoring systems for monitoring of different target systems that consist of different hardware (e.g., different processor, etc), because they are independent from the low-level hardware details.

High-Level Events: Software monitoring systems are popular because they allow users to view the monitored system at various levels of complexity or abstraction [DR92].

The main disadvantage of software monitoring systems is the *high* level of perturbation that they cause on target systems, because they share the computational resources with target systems. Kirschbaum et al. [KBG98] indicate that there is always the dilemma of finding a good balance between system disturbance and retaining of exact event information.

Hybrid Monitoring

Hybrid monitoring systems try to combine the advantages of both above presented monitoring approaches. Hybrid monitoring systems typically use software probes for detection of significant events, however the collection of these events is the job of a specialized hardware, which is not directly dependent on the target system. Software monitoring systems are proposed in [HW90, Gor91].

Al-Shaer in [AS98] presents the advantages of the hybrid monitoring approach:

Intrusiveness: Hybrid monitoring systems cause less intrusiveness than pure software monitoring systems.

Efficiency: Hybrid monitoring systems are more efficient than pure software monitoring systems, since events are processed in hardware.

Flexibility: Hybrid monitoring systems are more flexible and less expensive than pure hardware monitoring systems.

2.3 Debugging Systems

Debugging is defined by the ANSI/IEEE glossary as "the process of locating, analyzing and correcting suspected faults", where a fault is defined as an accidental condition that causes a program to fail to perform its required function [BJRW94]. Kranzlmüller and Volkert [KV00] note that the goal of debugging is the location and correction of bugs in arbitrary programs to improve their reliability. Myers [Mye76] defines debugging as the activity of diagnosing the precise nature of a known error and then correcting the error. Shih in [Shi96] defines debugging as a process of locating, analyzing, and correcting suspected faults that cause a program to fail to perform its required function. Debugging usually involves transforming an incorrect program into a correct program, and for that reason sometimes it is also called *"correctness debugging"* [SG97].

Thane in [Tha00b] presents the difference between testing, which is the process of revealing failures by exploring the run-time behavior of the target system for violation of the specification, and debugging, which is concerned with revealing the errors that cause the failures. Watson in [Wat00] determines that the following steps are a commonly used systematic approach for dealing with each error:

Gather Information: This step is used to establish the nature of the error, the behaviour that is being observed, data structures likely to be involved, and a broad indication of the error location, such as the relevant module or subsystem.

Analyse and Locate: Once sufficient information about the error has been obtained, it must be analysed to determine the cause of the error.

Correct the Error: After the cause of the error has been identified, a solution for removing the error must be proposed.

In contrast to the debugging of sequential programs the debugging of parallel programs is quite more complicated, because parallel programs have nondeterministic behavior. As presented in 2.2.2 debugging techniques can be classified into *static* and *dynamic* techniques, depending on the fact if the target system needs to be executed during the debugging process or not.

2.3.1 Static Debugging

Static debugging techniques are based on the *static analysis* of the source code. They try to detect parts of code that probably contain anomalies and (or) bugs. In [AAC+94], this technique is defined as *static verification*, which is defined as verifying the system without actual execution. Static debugging techniques are not adequate for debugging of real-time systems, because they cannot deliver any information about the timing constrains and the behavior of the target system in the time domain.

Watson [Wat00] determines that static analysis usually consist of:

Control Flow Analysis: Analyzing the flow of control through the program in order to identify unexpected control flows. An example is presented by Gustafsson in [Gus02].

Data Flow Analysis: Examining the use of variables within a program to detect errors such as references to uninitialized variables.

Inter-Procedural Flow Analysis: Analyzing control and data flow across procedure boundaries in order to examine the procedural structure, and identify data usage both within and across procedure calls.

2.3.2 Dynamic Debugging

In contrast to static techniques, dynamic debugging techniques are applied at run-time during the execution of the target system. Watson [Wat00] presents three different classes of dynamic debugging techniques:

Cyclic Debugging: One of the widely used dynamic debugging techniques is *cyclic debugging*. Stewart and Gentlman [SG97] state that cyclic debugging refers to repeatedly stopping the execution of a program to examine the program state and then either continuing the execution or restarting it in order to stop at some earlier point [SG97].

Tsai et al. [TBYS96] point out that there are three techniques that are used to perform cyclic debugging:

Memory Dumps: This technique collects the program status, including program object code, register and memory contents and dumps it into a file when the program execution is terminated abnormally or by user's request. However, this technique requires from the user to have a strong background in machine-level language to examine the dumped code [TBYS96].

Tracing: The tracing technique utilizes special tracing facilities in the compiler or operating system to continuously track and display every step of the program's execution, including control flow, data flow, variable contents, and function calling sequences. Tracing gives programmers a sense of the step-by-step flow of the program's execution [TBYS96].

Breakpoints: The breakpoint technique utilizes predicates that are inserted by the programmer or by the system to suspend program execution. Once a predicate is satisfied, the program execution is suspended and the relevant run-time information, such as variable values, stack contents and register values, can be displayed [TBYS96].

Event-Based and Post-Mortem Debugging: Event-based debuggers are used for collecting of event sequences into (large) event histories during execution of parallel programs. These event histories can later (*post-mortem*) be browsed, analyzed or even replayed [SG97].

Relative Debugging: A relative debugger is a system that allows the user to compare the state of two executing programs [AW97]. Relative debugging is not applicable for debugging of real-time systems, because real-time systems are usually not implemented for different platforms.

Most of the above presented debugging techniques are not suitable for monitoring of real-time and especially distributed real-time systems. Especially the *breakpoint* technique, which is the most often used cyclic debugging technique for debugging of non real-time systems, cannot be applied for debugging of real-time and specifically distributed real-time systems for two reasons:

Consistency Problem: Real-time systems are used for controlling of physical processes (their environment), whose state is changed with the progression of the physical time. An alternative to this problem is the use of environment simulators.

Global Time: Another problem is the absence of the *perfectly* synchronized global time by many distributed real-time systems, which makes the simultaneous stop of execution of the distributed real-time system impossible.

Recently, Smaili and Ademaj [SA02] proposed a cyclic debugging approach, which can be used for debugging of distributed real-time systems that are based on the Time-Triggered Architecture (TTA). The prerequisite for using this technique is the *sparse time base* [Kop92] by the target system as a model for its global time (like the TTA). The global physical time is substituted with a synchronized virtual global time that enables the debugging system to simultaneously stop and resume the execution of the target system.

2.3.3 Debugging with Monitoring Support

As depicted above traditional debugging techniques are not suited for debugging of real-time and especially distributed real-time systems. Therefore, debugging approaches with monitoring support must be used for debugging of real-time systems. Tsai [TBYS96] presents the following classification of debugging approaches with monitoring support:

Real-time Display: The execution of a program is monitored and displayed in a continuous real-time mode. In this approach the target program cannot be suspended or repeated, and no modification to the target program is allowed [TBYS96].

Real-time Debugging: The execution of a target program is monitored, analyzed, and debugged in a continuous real-time mode. The execution of the target program cannot be suspended or repeated, but it can be modified automatically in real-time during execution [TBYS96]. This debugging technique is the same as the feedback monitoring mentioned in 2.2.2.

Interactive Debugging: The execution of a target program is monitored and will be suspended if a predefined situation is encountered. In this approach the target program can be suspended and modified, but the execution is not repeatable [TBYS96]. This approach is the same as the cyclic debugging presented above.

Deterministic Replay: A trace of the program execution is recorded during execution and can be replayed deterministically after the execution [TBYS96]. This approach will be presented in more detail in the next section.

Dynamic Simulation: A trace or log is recorded during a target program's execution.The log used for dynamic simulation to evaluate the runtime behavior of a target program in a different run with different test data [TBYS96].

Deterministic Replay

Deterministic replay belongs to the dynamic debugging techniques that uses the support of monitoring systems. Harris in [Har02] notes that deterministic replay schemes have been designed to allow consistent re-execution of multi-threaded processes. Tsai in [TBYS96] states that the deterministic replay can be used to replay the event sequence that led to a deadlock so that the cause of the deadlock can be determined, or it can be used to replay the precedence and timing relationship between relevant processes and events so that a data dependency analysis can be performed.

2.4 Case Studies

Tis section presents some case studies of real-time monitoring systems.

2.4.1 Peters and Parnas Monitor

Peters and Parnas in [PP02] present a monitoring system that can be used for *correctness checking* (see Section 2.2.2) of real-time target systems. The authors note that before a safety-critical real-time system is designed, a specification of the required behavior of the system should be produced and reviewed by domain experts, and after such a system has been implemented, it should be thoroughly tested to ensure that it behaves correctly. Furthermore, the authors state that this is best done using a monitor, a system that observes the run-time behavior of a target system and reports wether that behavior is consistent with the requirements [PP02].

2.4.2 Wedde, Lind and Segbert Monitor

Wedde et al. in [WLS99] present a feedback distributed task monitoring system, which is integrated within the MELODY distributed operating system (see Figure 2.2). The monitoring system finds out task instances that are not going to meet their deadlines. This information is then fed back to the scheduler of the target operating system (to which the monitoring system belongs), aiding it to timely abort task instances which would otherwise miss their deadlines.

The authors briefly discuss an automated landing system (ALS) for bad weather conditions. Furthermore, they state that the response time of mechanical control systems are considerably long (more than 4 seconds for big aircrafts), which could be a problem in case the environmental situation has

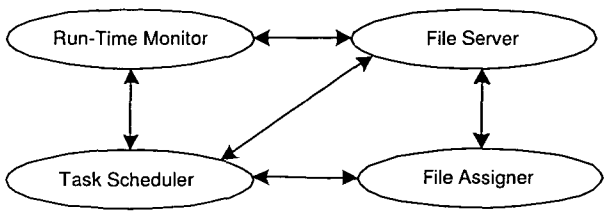


Figure 2.2: Management Function at each Node [WLS99]

changed drastically, in which the initiated actions can be obsolete or even dangerous for the aircraft. Therefore, the authors propose that for these systems the correction actions must be designed to cause considerable *small-scale changes*, which, in the negative case mentioned, are unlikely to have disastrous consequences [WLS99]. This is the reason why the authors state that aborting task instances as early as possible is a key issue for *system's survivability* in distributed safety-critical systems, because the subsequent task instances will take corrective actions. However, they note that if a task instance is aborted then the criticality of the subsequent task instances will be automatically increased, which means that these tasks then have to absolutely meet their deadlines.

The authors derive sub-deadlines for the location, allocation, and locking phases of task instances, in order to be able to abort these task instances which cannot meet their deadlines as early as possible. The key idea is to determine, for every task instance, sub-deadlines for characteristic computation phases prior to the execution phase, such as resource location, allocation and locking, and to abort an instance, whenever, according to the monitoring model, a particular sub-deadline is about to be missed [WLS99].

2.4.3 Akgul, Kuacharoen, Mooney and Madisetti's Debugger

Akgul et al. in [AKMM01] present a monitor (debugger) that uses the software monitoring technique for gathering of run-time information. The proposed debugger is a loadable module that can be added into the operating system of the target system. However, the instrumentation code is added into the kernel of the operating system, which is the only intrusion into the target system. The instrumentation code is always present in the kernel and it may not be removed from the kernel, because its timing behavior can be changed. During normal operation of the target system the debugging module is not loaded. However, in case an error condition occurrs (either a hardware exception or an algorithmic error), the debugger module is dynamically loaded to the operating system.

The advantage of this monitoring system is the flexibility, because it is based on a software approach. However, since the debug module must be involved in debugging mode, the timing behavior of the target system cannot be the same in the case when this module has not been loaded.

2.4.4 TTA Monitoring

The core of the Time-Triggered Architecture (TTA) is the Time-Triggered Protocol (TTP), which uses two different derived protocols TTP/C and TTP/A [Kop97]. It is the goal of this section to present the related work, which was done on the monitoring of these systems.

Sikula's Monitor

Sikula in [Sik98, KS98] presents a software monitoring system for monitoring of time-triggered distributed real-time systems. This monitoring system adds a dedicated *monitoring node* (*analyzer tool*) to the TTA network that collects and stores the monitoring data. Data are gained by the monitored nodes at run-time and transferred to the analyzer tool via dedicated *monitoring messages*. The monitoring system consists of an *off-line part*, in which the system is prepared for monitoring, and an *on-line part*, which denotes the execution of the monitored application [KS98].

In the off-line part, which runs during design phase, the target application is *modified* (i.e., instrumented) for monitoring. In this phase the monitoring messages (dedicated TTA messages that are used for carrying of the monitoring data) are defined, and the application code is instrumented. Since the monitoring messages consume transmission bandwidth and the instrumented code consumes computational resources, they must receive appropriate attention by the offline scheduler, which produces a schedule only if the intrusion process does not hinder the application to meet its real-time requirements.

In the online part the run-time monitoring data are collected by the analyzer tool, and they are visualized or stored on the disk, depending on the user's request. The monitoring data are generated by the instrumentation that has been code inserted into the application during the off-line part.

Glavan's Monitor

Glavan in [Gla00] presents an approach for monitoring of time-triggered real-time operating systems. The solution presented in this master's thesis is used

for monitoring of TTPos, which is designed and implemented by TTTech Computertechnik AG [TTT98]. This monitoring approach collects monitoring data that enables the monitoring of target systems at the operating system abstraction level, and transmits these data via a serial interface to the data collector running on a PC. The amount of the monitoring data that can be collected by this approach is limited by data throughput that can be achieved via a serial interface. Furthremore, this approach does not support the simultaneous monitoring of multiple nodes at the operating system abstraction level.

Mayer's Monitor

Mayer in [May00] presents a monitoring system that can be used for monitoring of TTP/C systems. This monitoring system was implemented under Linux, a freely available UNIX operating system, as an easy to use file-system based monitoring interface. As a pure software monitoring approach it does not require any specialized monitoring hardware. Moreover, there is no need to insert special instrumentation code in the application which is being monitored [May00].

The monitoring system was implemented as an operating system's device driver, which consists of two modules: i) *generic*, and ii) *application dependent* part. The generic part does not need to be adapted, while the application dependent part must be adapted for monitoring of a new application. The adaptation and reconfiguration of the dependent part can be made using the *description language*, which is used by the preprocessor for generating of C-code. This monitoring approach is limited for monitoring of only one cluster.

Obermaisser's Monitor

Obermaisser et al. [OPEL01] present a monitoring system that can be used for monitoring of TTP/A target systems. Such a monitoring system provides monitoring of TTP/A applications using the the *interface file system* (IFS), which enables the system to monitor TTP/A applications, without modification of run-time behavior of the target application in the temporal or in the value domain. Furthermore, this monitoring system offers a CORBA interface, which enables the user to remotely monitor the TTP/A target system.

2.5 Chapter Summary

In this chapter we presented the basic concepts for monitoring of real-time systems. We started with presentation of real-time systems with special view on

real-time data. Special attention was paid to the notion *temporal consistency* of real-time data. Especially, the relative consistency is very important for monitoring systems, because during correlation of collected monitoring data, these systems must be certain that these data are relative consistent. Otherwise, the gathered monitoring information would not correctly represent the behavior of the monitored system at the intended abstraction level. Another goal of this chapter was the presentation of a survey of the real-time monitoring research area. Finally different monitoring systems are presented, including the monitoring approaches used for monitoring of time-triggered systems that are based on the TTA.

Chapter 3

System Model

This chapter introduces the system model of target systems to be monitored. After presentation of the used terminology, the system structure is presented. In the next sections, transmission medium, timing characteristics, and communication services are presented. The chapter ends with a summary.

3.1 Terminology

In [Sch94a] Schmid denotes that a typical real-time system consists of a computer system (the *controlling system*) commanding and controlling a critical environment (the *controlled system*). In [Kop97] Kopetz and in [Nos97] Nossal make a distinction between the *real-time system* and the *real-time computer system*. In [Gal99] Galla notes that a real-time system is defined as a "closed system"consisting of a controlled object (e.g., a physical process) imposing the timing constraints and a controlling computer system is defined as an actual real-time computer system which must fulfill these timing constraints.

Real-time systems are designed to react to the stimuli generated by their environments. Depending on the way how these systems react to these stimuli they can be partitioned into: i) *event-triggered* and ii) *time-triggered*. A real-time system is called event-triggered if it reacts immediately after the detection of the stimulus generated by the environment. On the other hand, a real-time system is called time-triggered, if the system's reactions to environment's stimuli are triggered by the progression of the real-time. In the rest of this thesis only time-triggered real-time systems are considered.

In the rest of this thesis we will use the notion *real-time system* to denote both the controlling and the controlled system. Furthermore, the notion *environment* will be used alternatively to the notion *controlled system*, which in fact is the environment of the real-time computer system.

3.2 System Structure

Real-time systems that can be monitored by the presented monitoring system are time-triggered distributed real-time systems. A distributed real-time system consists of one or more *clusters* (see Section 3.2.1) that are interconnected and communicate with each other to achieve the intended computational goal of the distributed application. In Figure 3.1 the system structure is illustrated. If a distributed real-time system consists of more than one cluster, then the

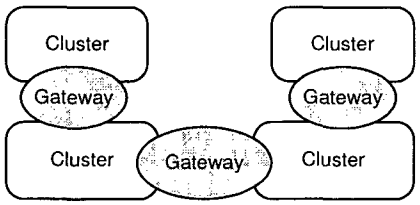


Figure 3.1: System Structure

clusters must be interconnected and synchronized by *gateway nodes*. Gateway nodes are dedicated nodes (see Section 3.2.2) that provide:

- synchronization of interconnected clusters in the temporal domain, and
- exchange of *inter-cluster* messages among interconnected clusters. The inter-cluster messages are defined by Galla [Gal99] as those messages that are exchanged among interconnected clusters.

3.2.1 Cluster Structure

Clusters are "building blocks"of distributed real-time systems. The part or the whole real-time distributed application is executed on a cluster depending on the fact, whether a system consists of one or more clusters. Clusters consist of a limited set of interconnected nodes (see Section 3.2.2), on which the local part of the distributed application is executed. The interconnected nodes share the same *transmission medium* and the same *global time* (see Section 3.4). The messages that are exchanged among the nodes of a cluster are called *intra-cluster messages*. In Figure 3.2 the cluster structure is illustrated.

The design of a cluster can be done using the *cluster design tools*, e.g., TTP-Plan [TTP02b]. Such a tool stores the design information into the *cluster database* (CDB). Examples of such information are: messages and frames that are sent by each node in the cluster (see Section 3.3). During monitoring of these systems the monitoring system must use the CDB to retrieve information needed for processing of collected monitoring data (see Section 5.3.1).

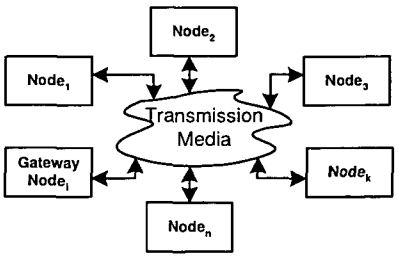


Figure 3.2: Cluster Structure

3.2.2 Node Structure

As depicted in the previous section, clusters consist of a limited set of nodes, which are the *real executors* of the local part of a distributed real-time application, and which *directly* interact with the real-time processes, i.e., the environment of the real-time computer system. In [Kop97] Kopetz defines a node as self-contained computer with its own hardware (processor, memory, communication interface, interface to the controlled object) and software (application programs, operating system), which performs a set of well-defined functions within the distributed computer system. In many applications, a node of a distributed computer system is the *smallest replaceable unit* (SRU) that can be replaced in case of a fault [Kop97].

The design of a node can be done using the *node design tools*, e.g., TTP-Build [TTP02a]. Such a tool stores the design information into the *node database* (NDB). Examples of such information are: tasks that are executed in each operational mode, messages that are sent/received by each node, for each task the list of input/output messages, etc. This information is used by the monitoring system, too.

Figure 3.3 illustrates the node structure. A node consists of four main parts:

- communication network interface (CNI),
- communication subsystem,
- host subsystem, and
- controlled object interface.

Communication Network Interface

The interface between the communication subsystem and the host subsystem is called *communication network interface* (CNI). In [Krü97] Krüger points out that the CNI can be regarded both as:

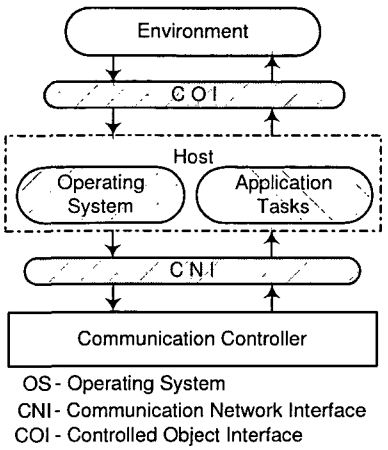


Figure 3.3: Node Structure

Data Sharing Interface: The primary purpose of the CNI is the exchange of messages that are generated and received by the computational entities constituted by the host subsystem, and that are transmitted by the communication subsystem to the other computational enties involved in the messages exchange.

Control Interface: In addition, the CNI facilitates the necessary exchange of control information between the subsystems.

This interface is well-defined both in the value and in the temporal domain. The CNI both on the structural and on the functional level can be divided into the following three distinct areas [Gal99]:

Status Area: In the status area the communication subsystem provides information about the current status of the communication protocol to the host subsystem. The information flow in this part is directed from the communication subsystem to the host subsystem.

Control Area: The control area can be used by the host subsystem to modify the behavior of the communication subsystem (e.g., to send a request to the communication subsystem for *mode changes*) [TTT02]. The information flow in the control area is from the host to the communication subsystem.

Message Area: The message area contains the messages that are exchanged between the nodes of the cluster. The information flow in the message area is bidirectional, i.e., the incoming messages are written into the CNI by the communication subsystem, and the outgoing messages are written by the host subsystem.

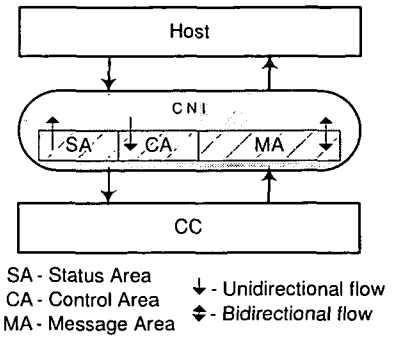


Figure 3.4: CNI Structure

Figure 3.4 depicts the CNI structure.

Communication Subsystem

The communication subsystem is responsible for communication with other nodes in the cluster. In time-triggered (TT) systems the send and receive actions are started independently from the host subsystem. As depicted in Section 3.1, the points in time at which these actions are triggered depend on the progression of real-time. The communication controller consists of the

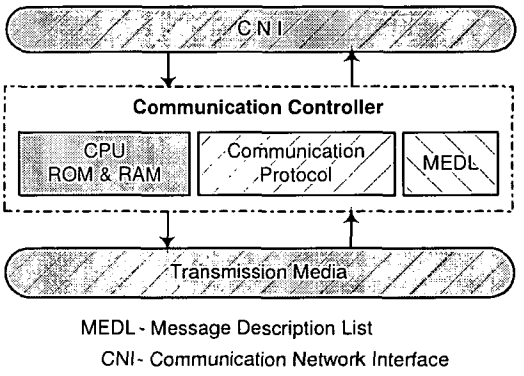


Figure 3.5: Communication Controller Structure

hardware (CPU, ROM and RAM, timers), communication protocol and the data structure that is used for scheduling of actions for sending and receiving of messages (see Figure 3.5). The latter data structure is called *message description list* (MEDL). The MEDL is *generated* during the design phase of the distributed real-time system. Among other things the MEDL contains the points in time when messages must be sent and received by the node.

Another integral part of the communication subsystem is the *bus guardian* (BG). The BG prevents faulty nodes from disturbing the shared transmission medium. The BG permits the communication controller to access the transmission medium only during predefined time windows, i.e., *bus slots* (see Section 3.4).

Host Subsystem

The host subsystem is responsible for executing the local part of a real-time distributed application. Figure 3.6 depicts the structure of a host. It consists of the hardware, operating system, fault-tolerance layer, and the application.

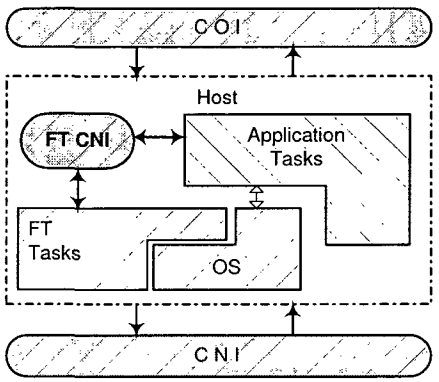


Figure 3.6: Host Structure

Operating System (OS): In TT systems the scheduling process is done off-line, before the execution process is started. The role of the OS in these systems is to dispatch tasks on the basis of the *dispatch table* generated by the static scheduler.

Fault-Tolerance Layer (FTL): In safety-critical applications the resources (i.e., nodes, transmission medium, etc.) are replicated in order to be able to tolerate singe failures. Replicated messages (i.e., messages that are either sent by replicated nodes or are transmitted via replicated channels or both of them) must be processed by this layer using *agreement algorithms* [KBP01] before they can be used by the application [TTP02a, Ose01].

Application: Application tasks, on the one hand, communicate via the *controlled object interface* (COI) with the environment. On the other hand

they communicate with other tasks running on different nodes via the CNI.

3.3 Transmission Medium

As depicted in Section 3.2.1, the nodes of a given cluster are interconnected by the shared transmission medium, which consists of at least two replicated communication channels. The exchange of data between the interconnected nodes is via broadcast, i.e., every node can receive all messages sent over the shared transmission medium. There are two topologies of the transmission medium, over which the nodes are interconnected to each other: *star* and *bus* topology. ¿From the monitoring system's point of view there are no differences between those systems that use star and those that use bus topology.

Messages vs. Frames

In time-triggered systems the messages are packed into *frames*, before they are transmitted by the communication subsystem via the replicated channels. The relation between frames and channels is 1:1, i.e., each node sends on each replicated channel exactly one frame. However, the relation between messages and frames is n:n, i.e., one message can be replicated in multiple frames, and one frame consists of one or more messages. The decision about the replication levels of messages must be taken by the system designer, because such a decision is application specific. E.g., messages that contain time-critical information must be replicated to avoid the loss of messages in case one of the channels gets disturbed.

State vs. Event Channels

Depending on the nature of the messages that are sent over the transmission medium, the communication channels can be partitioned into logical *state* or *event* channels, respectively (see Figure 3.7).

State Channels: State channels are logical channels used for transmission of *state messages* [Pol96]. As depicted in Section 3.2.2, the points in time when these messages are sent are defined in the MEDL.

Event Channels: Event channels are used for transmission of *event messages* [Pol96]. From the monitoring system's point of view, the support of event channels is of utmost importance, because they are used

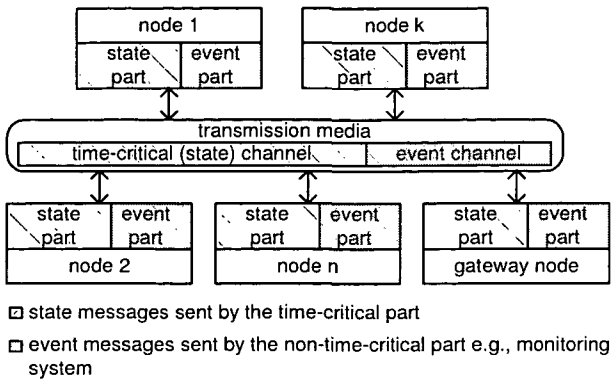


Figure 3.7: Transmission Medium

for transmitting of collected monitoring data from different nodes during monitoring of target systems at *node* or *transducer abstraction levels* (see Sections 5.3.2 and 5.3.3).

3.4 Timing Characteristics

Global Time

The synchronized global time is one of the basic time characteristics of time-triggered systems. The synchronized global time facilitates the monitoring process of these systems, because the monitoring data collected from different nodes can be easy temporally-ordered based on their time-stamps. The collected monitoring data must be time-stamped with the synchronized global time.

In time-triggered systems the concept of *sparse time base* [Kop92] is used as basis for the global time. In the sparse-time model the continuum of time is partitioned into an infinite sequence of alternating time intervals *activity* and *silence*. From the point of view of temporal ordering all events that occur within an interval of activity are considered to happen at the *same time* [Kop92, KB01].

TDMA Aproach

As depicted in Section 3.1, in time-triggered systems all activities are triggered by the progression of physical time. In particular communication activities on the transmission medium are triggered based on *a priori* known time schedule, which is based on the time-division multiple access (TDMA) approach. In the

TDMA approach every node receives a unique *time window* called *slot* (or bus slot), during which the node can utilize the full capacity of the transmission medium. The sequence of a finite number of slots is called *round*[1]. Furthermore, a finite sequence of rounds forms the *cluster cycle*. In time-triggered systems, a cluster cycle can be regarded as an *atomic repetitive unit* (ARU), because all application's activities[2] within a current cluster cycle are repeated in the following cluster cycle as long as the system's mode is not changed (see Section 3.5). Figure 3.8 depicts the schedule of messages that are processed

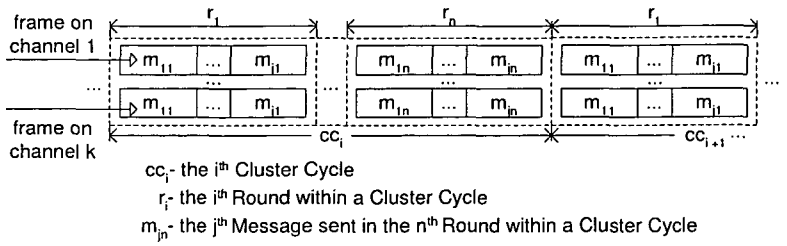


Figure 3.8: Cluster Cycle

and sent by a given node in different rounds within different cluster cycles.

3.5 Communication Services

In this section only the communication services will be presented that are important from the monitoring system's point of view.

Membership Service

The *membership service* [TTT02] is used to establish a *global knowledge* about the *node activity status*. The membership service ensures that at each point in (global) time each node knows the activity status of other nodes. The membership service is important for monitoring systems, because the membership information can be used, for example, during the off-line deterministic replay of collected monitoring data for representing the status of a particular node.

[1]Each node in a given cluster receives a unique slot, which makes sure that the access to the shared transmission medium is collision free. Furthermore, this approach makes sure that every node can access the transmission medium within a given round.

[2]In different cluster cycles the node actions are repeated, i.e., the same tasks set is executed, which sends the same messages.

Multiple Operational Modes

Many real-time systems exhibit mutually exclusive phases of operation and control. For example, an aircraft can be on the ground, in take-off, or landing [TTT02]. These mutually exclusive phases are called *modes*. From the node's point of view in each mode the set of tasks that are processed and the set of messages that are sent or received is different. This information is stored in the cluster and node database during the design phase (see Sections 3.2.1 and 3.2.2).

Nossal [Nos97] and Galla [Gal99] distinguish between three different types of modes in time-triggered systems:

Local Mode: A local mode belongs to one host and is characterized by a certain task set.

Cluster Mode: A cluster mode affects all nodes of a cluster, which means that another set of messages is exchanged among the interconnected nodes.

Global Mode Global modes are defined for the whole system.

From the monitoring system's point of view multiple modes are important, because the monitoring activities must be planned and analyzed for each target system's operational mode.

3.6 Chapter Summary

In this chapter the system model was presented. We presented the structure of the system at different abstraction levels (system's, clusters's and node's structure). Especially, attention was paid to the definition of logical state and event channels. The support of event channels is of vital interest for providing of monitoring, because collected monitoring data are transmitted via this channel to the central monitor for further processing.

Chapter 4

Monitoring Data

In this chapter monitoring data (MD) types are defined that enable the predicting of the influence of monitoring systems on the target systems. This chapter is organized as follows. After the presentation of the motivation and objectives in the next section, the terms and notions that are used throughout this chapter are explained in Section 4.2. The definition and classification of monitoring data are presented in Section 4.3. Section 4.4 deals with the methods for gathering of classified monitoring data. The types of monitoring data that can be collected during monitoring of time-triggered systems is the focus of Section 4.5. A summary closes this chapter.

4.1 Motivation and Objectives

Monitoring data represent the run-time behavior of the target system at the intended abstraction level. The goal of monitoring systems is to collect all MD, from which the run-time behavior of the target system at the intended abstraction level can be reproduced. The correctness of real-time systems (RTS) depends not only on the results they deliver, but also on the point in time on which these results are delivered. Therefore, the key issue during monitoring of RTS is to keep the interference that is caused by a monitoring system on a RTS deterministic, if it cannot be completely removed.

The interference of the monitoring system depends on the *way* how monitoring data are *collected* from the target system, and on the *amount* and the observation *rate* of monitoring data being collected within an observation interval (see Section 4.3.1). Thus, the user of the monitoring system has to answer the following key questions before starting to monitor the target system [SP04]:

- *Which types of monitoring data must be collected at the intended abstraction level within an observation interval?*

- *What is the amount of these monitoring data within the observation interval?*
- *How can these monitoring data be gathered?*

The classification of collected MD is influenced by abstraction levels at which target systems are monitored. In [Sch95] the MD are classified into: i) *hardware-*, ii) *process-*, and iii) *application-level* events. A similar classification has been also presented by Tsai et.al. [TFCB90]. Schütz [Sch94b] presented an interesting classification in context of testing. He denotes that the tester may wish to observe the *input(s)*, *intermediate* variables (*auxiliary* output), the *output(s)*, or all of them. Another detailed classification was presented by Thane in [Tha00b], in which the MD are categorized into three main groups: i) *data flow* - information on the data flow, ii) *control flow* - information on the control flow, and iii) *resources* - information on the resources of the target system.

To our best knowledge, no approach exists in literature that can help the monitoring system to predict its influence on the target system in advance. To be able to predict this influence and to keep this influence deterministic we define MD types and present different gathering methods.

4.2 Terms and Notations

In [Kop97] Kopetz determines that a controlled object, e.g., a car, changes its state as a function of time, and the dynamics of a real-time application are modeled by a set of relevant state variables. A significant state variable is called a *real-time (RT) entity*, and an observation of an RT entity is represented by a *real-time (RT) image*. An *observation* is defined in as information about the state of an RT entity at a particular instant of time. Furthermore, Kopetz states that an observation is an *atomic data structure* [Kop97]

$$Obs =< Name, t_{obs}, Value > \tag{4.1}$$

consisting of the name of the RT entity ($Name$), the instant when this observation was made (t_{obs}, i.e., the time stamp that is made using the global time), and the observed value of the RT entity ($Value$).

An RT image is a mirror of an RT entity within the real-time controlling system (Figure 4.1) during the *accuracy interval* [Kop97], after which the RT image becomes invalid. We conclude that the *state* of the physical controlled system at a particular instant t can be described by the set of values of its RT entities at instant t. The *state* of the controlled system at a particular instant t

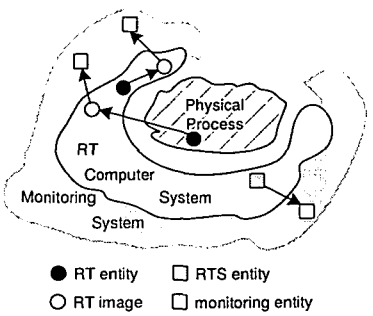


Figure 4.1: Monitoring Entity

as seen by the controlling system (i.e., the real-time computer system (RTCS)) can be described by the set of *temporally accurate* RT images. These states can be formally presented as:

$$P_S(t) = \{RTentity_{i,t} \mid i \in [1, N]\} \tag{4.2}$$
$$C_{P_S}(t) = \{RTimage_{i,t} \mid i \in [1, N]\} \tag{4.3}$$

where, P_S denotes the *state* of the physical (controlled) system and C_{P_S} denotes the *state* of the controlled system at the particular instant t as seen by the RTCS. $RTentity_{i,t}$ and $RTimage_{i,t}$ denote the i^{th} RT entity and its image at the instant t, and N denotes the number of RT entities belonging to the system.

Analog to the notions presented above we found that the state of the target RTCS (i.e., the controlling system) that is monitored by the monitoring system can be modeled (from the monitoring system's point of view) by the set of values of *significant state variables*, i.e., variables that are relevant for the monitoring system. Similar to the RT entities the values of these *significant variables* change with the progression of physical time.

Analog to the RT entity, we introduce a new notion called *real-time system (RTS) entity*, which is a significant "state variable" of the target real-time computer system and which is relevant only for the monitoring system. We say that the dynamics of the target RTCS from the monitoring system's point of view can be modeled by a set of RTS entities. Examples of RTS entities are: *number of tasks* that are waiting for a semaphore, *function ID* that is generated by an instrumentation code during monitoring process, when a correlated function is called, etc. The difference between RT and RTS entities is:

- *RT entities are used for modeling of the dynamics of the controlling and controlled system and they are used by the RTCS (i.e., target system) during its operation.*
- *RTS entities are used* only *for monitoring purposes. RTS entities together with RT entities are used for modeling of the dynamics of the target RTCS from the monitoring system's point of view.*

For deterministic reproducibility of the run-time behavior of the target system, (i.e., the real-time computer controlling and physical controlled system) the monitoring system has to collect both RT and RTS entities[1]. Therefore, for monitoring purposes we introduce the notion *monitoring entity* (Figure 4.1), which contains either an RTS entity or an RT object (i.e., RT entity or RT image [Kop97]). Thus, every entity that must be observed by the monitoring system for representing the run-time behavior of the target system (i.e., the RTCS and its controlled physical environment) at the intended abstraction level, is called monitoring entity. In the rest of this thesis the notions *entity* and *monitoring entity* will be used alternatively. From the monitoring system's point of view the *state* of the target RTS at the intended abstraction level at a particular instant t can be modeled by the set of values of monitoring entities:

$$T_S(t) = \{entity_{i,t} \mid i \in [1, N]\} \tag{4.4}$$

where, T_S denotes the state of the target system (as seen from the monitoring system's point of view) at the instant t.

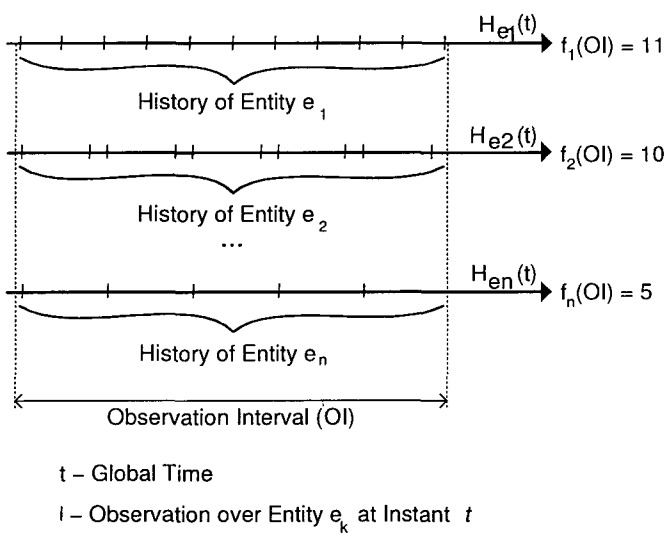


Figure 4.2: Observation History

Definition 4.1 - (Observation Interval): *The* **observation interval** *is the periodic time interval, which in fact is the smallest interval of the monitoring*

[1]The monitoring system must be able to collect the same values of the target system's *input/output variables* as they are used by the target system in order to be able to deterministically reproduce the run-time behavior of the target system (i.e., controlling and controlled system) at the intended abstraction level.

duration. The start point and the duration of the observation interval[2] *are the same as the start point and the duration of the ARU (see Section 3.4) of the target system.*

The whole time interval during which the target system is monitored consists of an integer number (m) of observation intervals[3] (OI). m is the number of OI, on which might depends the disk space if the collected monitoring data shall be stored on the disk (see Figure4.3).

Definition 4.2 - (Observation History): *The sequence of timely ordered observations that are made over a particular (single) entity within an observation interval is called* observation history*:*

$$H_i(OI) = \{Obs_j^i \mid i \in OI \wedge j \in [1, K] \wedge K = f(OI)\} \tag{4.5}$$

where, Obs_j^i is the j^{th} *observation* over the i^{th} entity observed at a particular instant t, K denotes the number of observations in the history, while $f(OI)$ denotes number of observations over the i^{th} entity made within an observation interval (OI).

The size of the *observation history* ($|H_i(OI)|$) of the particular entity i within the observation interval (OI) can be calculated as follows:

$$\mid H_i(OI) \mid = \sum_{j=1}^{K} \mid Obs_j^i \mid \tag{4.6}$$

where, $|Obs_j^i|$ is the size in bytes of the j^{th} observation over the i^{th} entity observed at a particular instant t. This size belongs to the static characteristics[Kop97] of the observation that is not changed during the runtime.

In Figure 4.2 different observation histories ($H_{e_1,e_2,...,e_n}$) over different entities ($e_1, e_2, ..., e_n$) are presented.

4.2.1 Amount of Monitoring Data

Definition 4.3 - (Monitoring Data): *The set of observation histories that contain observations of an observation interval and which represent the runtime behavior of the monitored target system at the intended abstraction level are called* **monitoring data**.

[2]In case of a TTA system the start point and the duration of the observation interval are equal to the start point and the duration of the cluster cycle. The duration of a cluster cycle is constant as long as the operational mode of the TTA system is not changed (see Chapter 3).

[3]In the rest of this thesis we will consider only the observation interval.

$$MD(OI) = \{H_i(OI) \mid i \in [1, N]\} \tag{4.7}$$

Definition 4.4 - (Amount of Monitoring Data): *The sum of the sizes of observation history of each entity within an observation interval is called the* **amount of monitoring data** *within the observation interval:*

$$A_M(OI) = \sum_{i=1}^{N} \mid H_i(OI) \mid \tag{4.8}$$

The amount of monitoring data within an observation interval depends on:

- the *number* of observed entities from which observations the run-time behavior of the target system at the intended abstraction level can be reproduced,
- the *rate* at which the entities must be observed,
- the *length* of the observation interval.

We call *monitoring capacity* the (rest of the) bandwidth of the shared transmission medium that is not utilized by the target application within an observation interval. The bandwidth of the shared transmission medium that is actually used for transmitting of monitoring data collected within an observation interval is called *bandwidth occupation*. The bandwidth occupation is usually smaller than the monitoring capacity. The monitoring capacity and the bandwidth occupation are formally described as:

$$C_m(OI) = C - C_{ap}(OI) \tag{4.9}$$

$$BO(OI) = h * A_M(OI) \wedge BO(OI) \leq C_m(OI) \tag{4.10}$$

C denotes the capacity of the shared transmission medium and $C_{ap}(OI)$ denotes the capacity used by the local part of the target application running on the monitored node within the observation interval. The bandwidth occupation within an observation interval is proportional to the amount of monitoring data within the observation interval. h is denoted as *data processing ratio*[4] per observation interval. h is equal to $1/\Delta t$ in case the monitoring data are not processed at the target node before they are transmitted. Δt is the time duration (in seconds) of the observation interval.

The capability for calculation in advance (i.e., before the monitoring process has been started) of the amount of monitoring data within an observation

[4]In case the monitoring data are compressed before being transmitted the *data processing ratio* is equal to the *compress ratio* of the applied compressing algorithm.

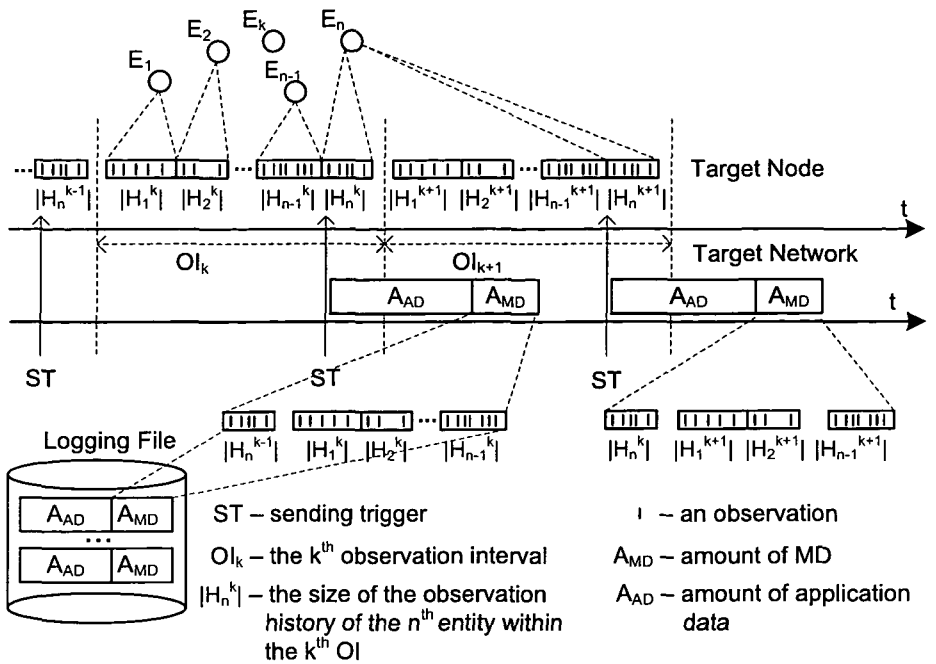


Figure 4.3: Amount of Monitoring Data

interval being collected from a specified target node is of utmost importance for the determinism of the monitoring process used to monitor the specified node. If one knows the amount of monitoring data being collected within an observation interval for a target node, then one can calculate in advance (see Figure 4.3):

- the computational resources (e.g., CPU, memory, etc.) on the particular target node that are needed for gathering, processing and transmitting of collected monitoring data within an observation interval,
- bandwidth occupation of the shared transmission medium, and
- disk space needed for storing of monitoring data at the central monitor (see Section 5.2.1).

The *amount* of monitoring data within an observation interval is *proportional* to the *rate* at which the corresponding entities are observed. In the rest of this thesis we will use only *"the* **amount** *of monitoring data being collected within an* **observation interval***"*, but with it we mean *"the* **amount** *and*

the **observation rate** *of monitoring data being collected within an* observation interval".

4.3 Monitoring Data Types

The *interference* caused by the monitoring system on the target system during monitoring at the intended abstraction level depends:

- on the *amount* of MD that must be collected within an observation interval, and
- on the *way* how these MD are gathered.

4.3.1 Regular vs. Non-regular Monitoring Data

In order to make the interference of the monitoring system *deterministic*, the monitoring system must provide support to either exactly calculate or estimate the expected amount of MD per time interval before the monitoring process of the target system starts. To achieve this, we define the *regular* and *non-regular* MD, depending on the rate at which the respective entities must be observed.

Regular Monitoring Data.

We call MD *regular*, if there is a regular pattern of observations made over the respective entities. Examples of regular MD are: observations over an entity that represents the temperature of the engine and which is observed regularly, or observations over a message that is sent regularly over the shared transmission medium to other interconnected nodes. Another example of regular MD are observations over entities[5] that are introduced within a particular periodic task in form of instrumentation code for monitoring of this task.

Regular MD consist of a set of regular *state observations*, which are defined in [Kop02] as observations that record the state of *state variables* at particular instants, the *point of observations*. A basic characteristic of regular MD is their *constant rate*, at which they are generated, i.e., the respective entities are observed. Therefore, for regular MD the respective amount within an observation interval is constant and it can be exactly calculated in advance.

[5]These are typical examples of RTS entities, which are used only for monitoring purposes (see Section 4.2).

Non-regular Monitoring Data.

We call MD *non-regular*, if there is no regular pattern of observations made over the respective entities. Typical examples of non-regular MD are observations over entities (e.g., RTS entities) that are introduced within interrupt service routines in form of instrumentation code for their monitoring.

Non-regular MD consists of a set of *event observations* which are defined in [Kop02] as observations that contain the difference between the state *before* and *after* the event. A basic characteristic of non-regular MD is their *non-constant* rate at which they are generated. Thus, the amount of these MD within an observation interval cannot be exactly calculated in advance, but it can only be estimated. In order to make the monitoring process deterministic, we must find out the amount of non-regular MD within an observation interval in the worst case, i.e., the *worst-case amount* (WCA). In contrast to the regular MD the amount of non-regular MD is not constant within an observation interval, but it is bounded with the WCA ($A_{WCA}(OI)$ in Equation 4.11).

$$A_M(OI) \leq A_{WCA}(OI) \tag{4.11}$$

Depending on the fact, whether the A_{WCA} can be calculated in advance or only estimated (aproximated), non-regular MD can be classified into:

Sporadic Monitoring Data: We call non-regular MD *sporadic* if they contain observations over entities which are not observed regularly but a *minimum time interval* between successive observations exists and is known.

A_{WCA} of these MD can be calculated in advance, if we suppose that the sporadic entities would be observed regularly with the period equal to the minimum time interval between two successive changes.

Aperiodic Monitoring Data: We call non-regular MD *aperiodic*, if they contain observations over entities, which are observed non-regularly, and no *minimum time interval* between successive observations exists. The calculation of A_{WCA} of these MD is *impossible*, because theoretically they can be observed *arbitrarily often*, i.e., at each instant of time. However, in target systems that use the *sparse time base*[6] as model for their global time, A_{WCA} of aperiodic MD can be calculated, if we suppose that the respective entities are observed only one time between two successive *action lattices*. In this case the aperiodic MD can be regarded as sporadic MD, which contain observations over entities that are observed with the

[6] In the *sparse-time* model the continuum of time is partitioned into an infinite sequence of alternating durations of activity and silence. From the point of view of temporal ordering, all events that occur within an interval of activity are considered to happen at *the same time* [Kop97].

minimum time interval between successive observations equal to the time interval between two *action lattices* [Kop97].

MD	Observations	Time Interval#	Amount$
Regular	State	Constant	Exact Calculation
NR§/Sporadic	Event	Minimum	Calculation of A_{WCA}*
NR/Aperiodic		No Minimum	No Calculation of A_{WCA}

#) Time interval between successive observations.
$) The ability in advance to either calculate or estimate the expected amount.
*) Worst-Case Amount of MD within an observation interval.
§) Non-regular monitoring data.

Table 4.1: Regular vs. Non-regular Monitoring Data

Table 4.1 presents the classification of MD based on the amount, i.e., on the ability of the monitoring system to either calculate or estimate the expected amount of MD within an observation interval in advance.

4.3.2 Monitored Application vs. Pure Monitoring Data

As mentioned at the beginning of this section, the way how MD are collected from target systems is the second factor on which the monitoring system's interference caused on target systems depends. To make this interference deterministic, we define the following MD types:

Pure Monitoring Data: We call MD *pure monitoring data* if they contain observations over entities that do not provide any contribution for achieving the computational goal of the target application. This type of entities, i.e., RTS entities (see Section 4.2) are observed by the instrumentation code inserted into the target operating system or target application. Observations over these entities are used only for monitoring purposes. The visibility of these entities is limited within only a particular node.

Monitored Application Data: We call MD *monitored application* data if they contain observations over entities that contribute to achieving the computational goal of the target application. Examples of monitored application data are: observations over an entity that represents the velocity of a car, or the operational mode of the real-time controlling computer system. The monitored application data can be classified into *network* and *node* MD depending on the scopes in which these observed entities are visible.

Network Monitoring Data. *Network MD* contain observations over entities, the observations of which are used by tasks running on different nodes, i.e., they are exchanged between interconnected nodes via the shared transmission medium. Examples of such MD are messages exchanged between nodes in a distributed RTS.

Node Monitoring Data. We call MD *node MD* if they contain observations over entities that are only visible within a particular node. Node MD are classified into *local* and *global* MD, depending on the scopes in which these entities are visible.

- **Global Monitoring Data:** We call the node MD *global*, if they contain observations over entities that are *globally* visible within a particular node. Examples of global data are: observations over inter-task (i.e., interprocess) messages, input values to the tasks (i.e., environment inputs that are processed, before they are correlated with other local data), output values of the tasks, which must be further processed before they are sent to another nodes, etc.
- **Local Monitoring Data:** Node monitoring data, which contain observations over entities that are not *globally* visible within a particular node are called *local monitoring data*. Examples of local data are: intermediate values or variables (auxiliary outputs), etc.

The difference between global and local MD is manifested during the gathering process, because different gathering methods must be used for gathering data of these two groups (see Section 4.4).

4.4 Gathering Methods

We suppose (according to the system model presented in Chapter 3) that a target system is a distributed embedded real-time computer system, and the monitoring approach used for gathering of monitoring information is either software or hybrid monitoring approach (see Section 2.2.6). Furthermore, the collection of MD, i.e., the gathering of observations, is done either by sensors or probes[7].

The classification of the gathering methods is influenced by the following three questions:

[7]In this thesis we will use the notions presented by Ogle et al. in [OSS93]. Thus, sensors reside within the target application inserted during instrumentation process, while probes reside within the resident monitors that are parts of the monitoring system, and both of them are responsible for gathering of MD.

- Is the target system *influenced* by the monitoring system during the monitoring process?
- Is the gathering process of the MD *transparent* to the target application[8]?
- Does the target application have to be *instrumented* for successfully gathering of MD?

In order to answer these questions we have classified the gathering methods into three groups:

- *Monitoring-Node* (MN),
- *Operating-System* (OS), and
- *In-Line* Gathering Method.

4.4.1 MN Gathering Method

The *monitoring-node* gathering method uses a dedicated *monitoring node* for gathering of MD. A monitoring node *snoops* on the transmission medium used for interconnection of nodes (Figure 4.4). With this method one can collect

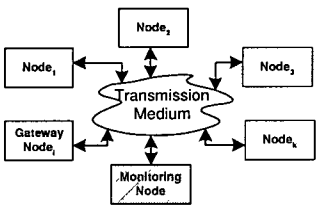


Figure 4.4: Monitoring-Node Gathering Method

only the *network* MD, because they contain observations over entities whose values are exchanged over the shared transmission medium.

4.4.2 OS Gathering Method

The *OS* gathering method can be used for gathering of MD that contain observations over entities that are not transmitted over the shared communication medium but are visible within the global scope of the given node. These MD have been presented in the previous section. They are called *global node* or *pure* MD.

[8]The target application is the software application that is running on the target system.

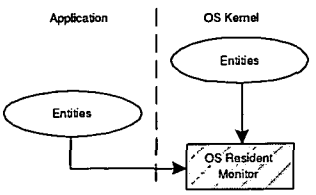


Figure 4.5: OS Gathering Method

For collecting these MD the operating system must be instrumented (Figure 4.5) with instrumentation code, i.e., *resident monitor* that is responsible for gathering the MD. Thus, in this case the monitoring process is transparent to the application programmer.

4.4.3 In-line Gathering Method

The *in-line* gathering method is used for gathering of MD that contain observations over entities that are not visible from the operating system. For gathering of these MD sensors must be inserted into the target application. This gathering method can be used for gathering of *local* MD.

4.5 Monitoring Data in Time-Triggered Systems

As depicted in Section 3.2.2, a node consists of a communication controller, the host, and the CNI (see Figure 4.6). On the host, the application tasks and FT tasks are running. The CNI is used as temporal firewall [Kop97, Nos97] between the communication controller and the host. During monitoring of time-triggered (TT) systems the following monitoring data can be collected (see Figure 4.6):

Network Monitoring Data: These MD contain observations over entities the observations of which are exchanged between different nodes of TT systems in form of messages. These messages can be either *state* or *event messages* as depicted by Poledna in [Pol96]. The monitoring data that contain observations over state messages are *regular*, while monitoring data that contain observations over event messages are *non-regular* monitoring data. For regular network data we can calculate the exact amount of monitoring data in advance, i.e., before the monitoring process is started, because the transmission of state messages in TT systems are scheduled statically, i.e., during the design phase.

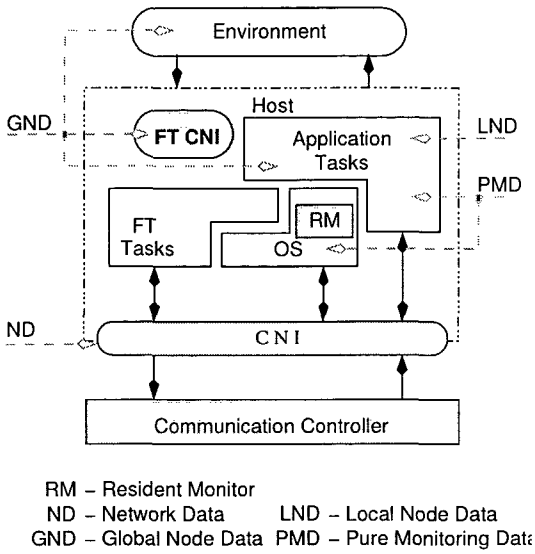


Figure 4.6: Monitoring Data in Time-Triggered Systems

On the other hand we can calculate the A_{WCA} for the non-regular network data. For gathering of these monitoring data the monitoring-node gathering method is used (see Section 4.4.1).

Global Node Monitoring Data: The *sources* from which these data can be *gathered* (see Figure 4.6) are:

- *FT CNI* - The fault-tolerant (FT) messages are processed by FT tasks [Bau00], the results of which are stored into FT CNI.
- *Environment* - While controlling their environment TT applications have to *read* values from sensors or to *write* values to actuators.
- *Application* - Messages that are exchanged between different tasks, or global variables could be relevant for monitoring systems.

The node global monitoring data can be gathered using the OS gathering method presented in Section 4.4.2, because the locations where these entities can be found and the points in time when these entities must be observed are known in advance. The above presented sources, from which these monitoring data are collected, will be dealt with in more detail in Section 5.3.2.

Local Node and Pure Monitoring Data: The last group of monitoring data that may need to be gathered during monitoring of TT systems are *local*

node (e.g., intermediate values) and *pure monitoring* (e.g., function calls or the start of a task) monitoring data (see Figure 4.6). For gathering of these monitoring data the target operating system or application must be instrumented, i.e., the in-line gathering method is used. These monitoring data can be either *regular* or *non-regular*, depending on the fact how the functions or methods are called, where the instrumentation code is inserted. This must be *analyzed* by the monitoring system during the instrumentation process, which must be supported by the monitoring system.

4.6 Discussion

Figure 4.7 presents the classification of MD based on two different criteria: i) on the ability of the monitoring system to calculate the expected *amount* of MD within an observation interval in advance, and ii) on the *gathering methods*. The interference of the monitoring system on the target system depends on these two factors.

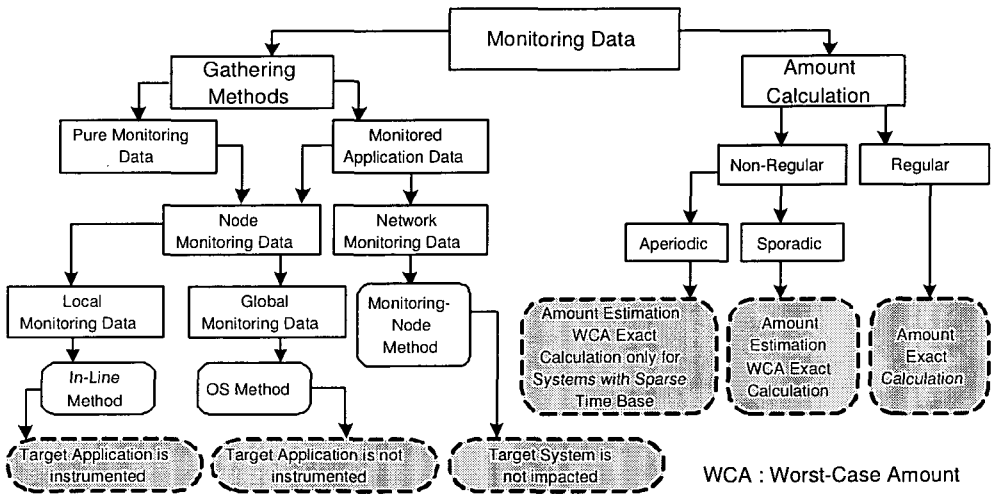


Figure 4.7: Classification of Monitoring Data

During monitoring of a target system at the user's intended abstraction level the prediction of the expected amount (and the observation rate) of MD within an observation interval is of utmost importance. The amount of the MD that must be collected determines the resource needs of the monitoring system on the target system. These resources are used for collecting, processing, and transmitting the collected MD. They relate to: CPU, memory, communication bandwidth, etc. On the basis of this information one can quantify the influence of the monitoring system on the target system.

On the other hand, the need for instrumentation of target systems depends on the used (and needed) gathering methods (Figure 4.7). The applied gathering method is another factor on which the influence caused by the monitoring system on target systems depends. For example, the MN method for gathering of MD does not cause any influence on target systems. However, target systems are influenced when OS or in-line methods are used.

4.7 Chapter Summary

In this chapter we defined different types of MD and gathering methods. These make possible to predict in advance and to keep the interference of the monitoring system caused on the target system deterministic. Furthermore, in Chapter 8 a case study is presented, in which these concepts are successfully applied in monitoring of TTA systems for calculation of the expected amount of the MD within an observation interval.

Currently, the selection of entities, the calculation of the expected amount within an observation interval and the instrumentation process are done manually. The intention is to incorporate the classification of the MD presented in this thesis into the design tools to calculate the needed resources (i.e., MD amount calculation) for monitoring process and to automate the instrumentation process. Examples of design tools are DECOMSYS::Designer[9] or TTP-Plan[10] used for designing of FlexRay or TTA systems. This approach could also be incorporated into design tools for other distributed RTS, e.g., CAN[11], LIN[12], etc.

[9] www.decomsys.com/flyer/DESIGNER.pdf
[10] www.ttagroup.org/ttp/pdf/TTTech-TTP-Plan-Flyer.pdf
[11] http://www.can.bosch.com
[12] http://www.lin-subbus.org

Chapter 5

Monitoring of Time-Triggered Systems

This chapter focuses on real-time monitoring for the time-triggered systems. The following abstraction levels will be defined: i) *cluster*, ii) *node*, and iii) *transducer* abstraction level. Furthermore, the monitoring of target systems that consist of multiple clusters will be dealt with.

The chapter is organized as follows. First, the objectives and goals of this chapter are presented. After presentation of the monitoring system in Section 5.2, the different abstraction levels supported by the proposed monitoring system are described in Section 5.3. Section 5.4 deals with the monitoring of target systems that consist of multiple clusters. Section 5.5 deals with the debugging support. The chapter ends with a summary.

5.1 Objectives and Goals

The influence of the monitoring system must not change the behavior of the target real-time system, neither in the time nor in the value domain. Another requirement that must be considered during design of monitoring systems is that the monitoring system must use the *operational network* (i.e., the network that is used by the distributed target system) for transmission of collected monitoring data to the *central monitor*, where they are further processed (see Section 5.2.2). This requirement can be regarded on the one hand as an economic requirement, because of high costs that would arise for adding a dedicated network for monitoring. On the other hand, this requirement has a technical reason as well: in many cases the user of the monitoring system does not have the possibility to add a second network to the already assembled target system.

The main objective is to design a real-time monitoring system that both offers to the user the capability for deterministic monitoring of time-triggered systems at different abstraction levels and provides monitoring of target systems that consist of multiple clusters. The presented monitoring system is tailored for monitoring of distributed real-time systems that are based on the *Time-Triggered Architecture* (TTA). However, such a monitoring system can be used for monitoring of other time-triggered systems that fulfill requirements given in Section 5.2.1.

In Section 4.1 we noted that one of the three key questions the user has to answer before monitoring the target system is the selection of monitoring entities. Therefore, another objective of this chapter will be the selection of those entities whose observations must be collected at each abstraction level. From these observations the run-time behavior of the target system at each abstraction level must be reproducible.

5.2 Monitoring System

5.2.1 Requirements and Assumptions

In this section the requirements and assumptions are presented that must be fulfilled by the monitoring system and target systems.

Requirements: The monitoring system must fulfil the following requirements:

- Target systems must support a *sparse time base* [Kop92, KB01].
- Target systems must support *event channels* (see Section 3.3), which will be used for transmitting of collected monitoring data.
- If a target system consists of more than one cluster, these clusters must be interconnected and synchronized by *gateway* nodes as depicted in Section 3.2. Furthermore, depending on the interconnection topology (see Section 5.4.1), gateway nodes have to support event channels that are used for transmitting of collected monitoring data to the monitoring node (see Figure 5.1).
- If a target system consists of more than one cluster then the monitoring system must support the *simultaneous* monitoring of these multiple clusters.
- Monitoring data that are collected from different interconnected clusters must be *temporally correlated* by the monitoring system, so that the monitoring system is able to compare these data in the temporal domain.

Assumptions: For target systems that can be monitored by the presented monitoring system we assume that:

- Clusters are part of the whole complex target system and they are interconnected and synchronized in the temporal domain by gateway nodes.

5.2.2 System Structure

Figure 5.1 presents the structure of the proposed monitoring system, which consists of three main parts:

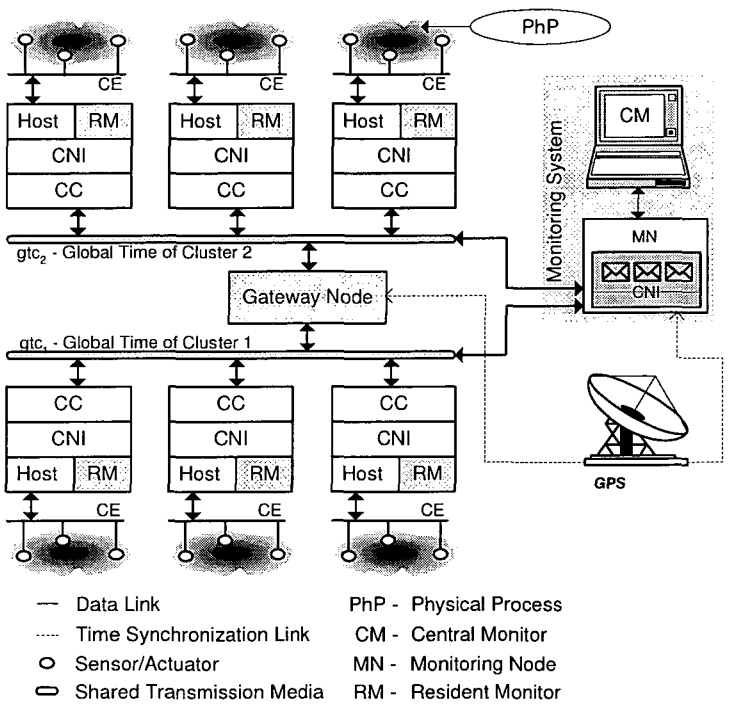


Figure 5.1: Monitoring System's Structure

Central Monitor (CM): The *central monitor* is the main part of the monitoring system. This part is responsible for analysis and setup of the monitoring process (see Section 5.2.3). Furthermore, the monitoring data collected from the target system are managed and processed by the CM. The CM both communicates with the user and with the monitoring node.

The CM uses the *monitoring clients* that perform different jobs for the monitoring system. Examples of such jobs are visualization of collected

monitoring data in real-time. For performing of these jobs GUI monitoring clients are used (see Chapter 7.2.2). Another important monitoring client is the *real-time trigger system* (RTTS), which uses triggers to search for significant events within the collected monitoring data. The RTTS is discussed in detail in Chapter 6.

The CM is implemented on a *commercial off-the-shelf* (COTS) system (see Chapter 7.2).

Monitoring Node (MN): The gateway between the CM and the distributed real-time target system is the *monitoring node*. The monitoring node both collects the monitoring data from target systems and receives monitoring commands from the CM. It sends the monitoring commands to the resident monitors running on the target system[1]. The monitoring system must contain at least one monitoring node.

Resident Monitor (RM): *Resident monitors* are executed on nodes that are monitored at *node* or *transducer abstraction level* (see Sections 5.3 and 5.4). They are responsible for gathering monitoring data from nodes and transmitting them to the monitoring node. Furthermore, gateway nodes must contain resident monitors that are responsible for gathering of monitoring data from target systems that contain more than one cluster (see Section 5.4), even in case these systems are monitored only at the cluster abstraction level (see Sections 5.3.1 and 5.4).

5.2.3 Operational Modes

In general, the monitoring process can be partitioned into two phases: i) *setup* and ii) *monitoring* phase. The operational modes of the proposed monitoring system depend on these two phases. There are three operational modes: i) *setup* mode, ii) *run-time* mode, and iii) *off-line* mode.

Setup Phase

The monitoring system has to assist the user during *setup* of the monitoring process. During this assistance the monitoring system is in the *setup* operational mode. There are three key issues that must be dealt with by the monitoring system during this operational mode:

[1]Such commands can be for example *start* and *stop* triggers for collection of monitoring data at node or transducer abstraction level (see Sections 5.3 and 5.4).

- **Entity Selection:** For each supported abstraction level the monitoring system must assist the user to select entities of a target system that *need* to be observed by the monitoring system. The run-time behavior of the target system at the intended abstraction level within an observation interval must be reproducible from these observed entities.
- **Expected Amount:** The monitoring system must be able either to calculate or estimate the expected amount of monitoring data within the observation interval (see Section 4.3). The monitoring data contain observations over the selected entities.
- **Gathering Methods:** The method for gathering of observations over each selected entity must be selected and analyzed during this phase (see Section 4.4).

The first issue is the objective of the next section, which deals with the monitoring of time-triggered systems at different abstraction levels. The last two issues were covered in Chapter 4.

The monitoring process can be planned either during the design phase of the target system, or it can be planned for an already assembled target system:

Design of a Target System: The monitoring system must support the system designer during design of the target system by listing the additional needed computational resources (i.e., CPU time, memory, and communication bandwidth) that are necessary for monitoring of them at each supported abstraction level. This list of requested resources must be taken into account by the system designer during design of the target system.

Already Assembled Target System: The monitoring system must also assist the user in monitoring an already assembled target system. However, in an already assembled target system one cannot add additional resources that could be needed by the monitoring system. Thus, in this case the monitoring system can use only those resources of the target system which are not used by the target application. Therefore, during setup of the monitoring process for an already assembled target system the monitoring system must inform the user when computational resources of the target system that can be used for monitoring purposes are exhausted.

An important issue, which must be done by the monitoring system in the setup mode in case the target system consists of multiple clusters is the search for the *shortest monitoring route* that has the sufficient bandwidth for transmitting of monitoring data collected from a particular cluster. This search is

done by the *monitoring router* that is presented in Section 5.4.1. Furthermore, *real-time triggers* presented in Chapter 6 are also defined in this mode.

Monitoring Phase

During the monitoring phase the monitoring system collects run-time information from the target system. The collected monitoring data are processed by the monitoring system according to the monitoring intention. There are two different operational modes during this phase in which the monitoring system can be:

Run-Time Mode: In the *run-time* operational mode the monitoring system has to collect the monitoring data.

Off-line Mode: During the off-line analysis (i.e., off-line operational mode), collected monitoring data are analyzed by the monitoring system. Activities that are executed in this operational mode are: *off-line analyzer* (see Section 6.4.2), or *deterministic replay* (see Section 5.5.2).

5.3 Monitoring Abstraction Levels

The user usually starts monitoring of a target system at a higher abstraction level in which the high-level run-time behavior of the target system can be observed. In case there are some suspected faults in a subsystem, the user needs the capability to get a view insight the run-time behavior of the subsystem. Therefore, the monitoring system must support different abstraction levels in order to be able to allow the user to take a look at different depth of the run-time behavior of the target system. This capability allows the user to deal only with the monitoring data that represent the run-time behavior of the target system at the intended abstraction level during localization of the suspected faults.

In this section the following abstraction levels are defined:

- Cluster Abstraction Level,
- Node Abstraction Level, and
- Transducer Abstraction Level.

As depicted in Section 3.4, in time-triggered systems all activities of the target application are repeated between successive *cluster cycles* as long as the operational mode of the target system is not changed. Therefore, the expected amount of monitoring data collected by the monitoring system at the above presented abstraction levels within a cluster cycle must be calculated or estimated. If the expected amount of monitoring data within a cluster cycle is known, then the expected amount within any observation interval can be found. In these systems each observation interval consists of one or more cluster cycles.

5.3.1 Monitoring at Cluster Abstraction Level

Definition 5.1 - (Monitoring at Cluster Level): *We define the monitoring of target systems at the* **cluster abstraction level** *as a monitoring process during which the monitoring system observes entities whose observations are visible within the* **cluster scope**, *i.e., these observations are exchanged among the interconnected nodes within a given cluster via the shared transmission medium.*

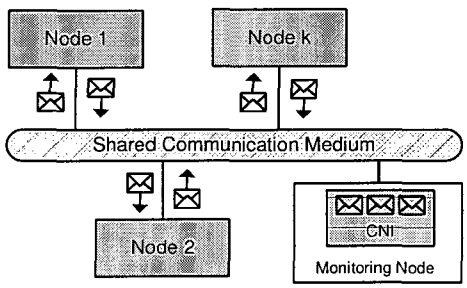


Figure 5.2: Monitoring at Cluster Abstraction Level

At this abstraction level monitoring data are collected by *"snooping"* of the shared transmission medium by the monitoring system. This means that the *monitoring-node* gathering method presented in Section 4.4.1 is used for gathering of monitoring data. In Figure 5.2 a monitoring approach at the cluster abstraction level is presented. At this abstraction level the monitoring system cannot *gain insight* into the dynamic behavior of interconnected nodes. Therefore, the interconnected nodes for the user of the monitoring system at the cluster abstraction level remain as black boxes, which send and receive predefined messages at predefined points in time. However, the monitoring at the cluster abstraction level is well suited for *continuous monitoring*[2] of target

[2]In this case the *continuous monitoring* is defined as a long-term monitoring process, which could use real-time triggers for example that are presented in Chapter 6 to search for

systems, because the monitoring system at this abstraction level does neither instrument nor impact the target system.

The *cluster design database* (CDB) (see Section 3.2.1) must be used by the monitoring system for successful monitoring of target systems at the cluster abstraction level. The CDB contains all information needed by the monitoring system for collection and interpretation of monitoring data collected at this abstraction level (see Section 3.2.1).

Monitoring at the cluster abstraction level does not have an impact on the operation of target systems, because the monitoring node does neither occupy the observed system's computational resources nor its communication bandwidth for collecting and transmitting of monitoring data. However, this is not true if a target system consists of more than one cluster, and the monitoring system is not directly interconnected to each cluster (see Section 5.4.1). In this case the gateway nodes have to collect the monitoring data from the monitored target clusters and send them via the shared transmission medium to the central monitor (see Section 5.2.2). During this transmission, the gateway nodes *occupy* communication bandwidth for transmission of the collected monitoring data. Despite this influence the behavior of target systems remains unchanged in both the time and the value domain, because this influence is deterministic and most notably known pre run-time. It is therefore of utmost importance that this influence is pre-planned during the setup phase (see Section 5.2.3) of the monitoring process.

Monitoring Data. The monitoring system at the cluster abstraction level has to collect all messages that are transmitted via the shared transmission medium. In the system model presented in this thesis, these messages are received and written into the CNI of the monitoring node by the communication controller (see Section 3.2.2), and they can be read out by the monitoring software running on the host of the monitoring node (see Section 7.1.1). Monitoring data at this abstraction level are collected from the following sources:

Message Area: The monitoring data collected from the CNI message area contain all messages that are received and written by the communication controller into the CNI of the monitoring node [TTT02].

Status/Control Area: The monitoring data that are collected from the CNI's status/control area contain information about the status of the communication subsystem. This information is also written into the

significant events, e.g., faults. In case some faults are found, the user can perform monitoring at another abstraction level, e.g., node abstraction level, in order to locate, analyze and correct the detected faults.

CNI [TTT02] of the monitoring node by the communication controller. The most important information from this area are: i) *membership vector*, ii) *cluster mode*, and iii) *cluster time field* [TTT02]. The membership vector contains the information about the activity or silence of each interconnected node in the cluster, while the cluster mode contains the information about the operational mode in which the cluster is currently running. The cluster time field represents the globally synchronized time with a granularity of one macrotick [TTT02].

Amount of Monitoring Data. At the cluster abstraction level the expected amount of monitoring data within one *cluster cycle* (A_C) can be exactly calculated during the setup phase:

$$A_C = A_{SCA} + \sum_{i=1}^{N} A_{Ni} \tag{5.1}$$

where A_{SCA} denotes the amount of monitoring data collected from the status/control area of the monitoring node within a cluster cycle, N denotes the number of interconnected nodes in a given cluster, and A_{Ni} denotes the amount of data sent by the i^{th} node[3] within a cluster cycle. In case a target system consists of multiple clusters, the amount of collected monitoring data within a cluster cycle is equal to:

$$A_M = \sum_{j=1}^{K} A_{Cj} \tag{5.2}$$

where K denotes the number of interconnected clusters, and A_{Cj} denotes the amount of monitoring data collected from the j^{th} cluster within a cluster cycle (see Equation 5.1).

Characteristics. During monitoring of target systems at the cluster abstraction levels:

- no instrumentation of the target systems is needed,
- the target systems are not influenced at all (in case the target system does not consist of multiple clusters see Section 5.4),

[3]In this case, the interconnected nodes can send both state and event messages [Sto01, Mai02]. However, the bandwidth for transmitting of event messages in time-triggered systems is limited and known in advance, and therefore we can calculate the A_{WCA} of monitoring data that contain observations over event messages.

- the expected amount of monitoring data is deterministic, and it can be calculated during the setup phase.

The main restriction of monitoring target systems at the cluster abstraction level is that only entities whose observations are exchanged between interconnected nodes can be observed.

5.3.2 Monitoring at Node Abstraction Level

Definition 5.2 - (Monitoring at Node Level): *We define the monitoring of target systems at the* **node abstraction level** *as a monitoring process during which the monitoring system observes entities whose observations are visible only within the* **node scope**, *i.e., these observations are not exchanged between the interconnected nodes within a given cluster.*

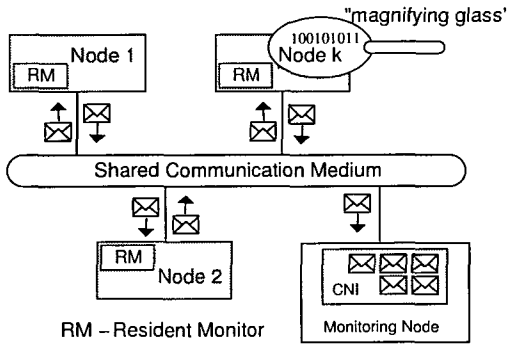


Figure 5.3: Monitoring at Node Abstraction Level

In Figure 5.3 a monitoring approach at the node abstraction level is presented. At this abstraction level the monitoring system can observe the behavior within single nodes, i.e., nodes are considered as white boxes.

The execution of a *resident monitor* on the node is the prerequisite for its monitoring at the node abstraction level (see Section 5.2.2 and Figure 5.3). Another requirement is the availability of sufficient bandwidth of the shared transmission medium to transfere the collected monitoring data, i.e., sufficient *monitoring capacity* (see Section 4.2.1). The amount of monitoring data within an observation interval that can be collected at the node abstraction level depends among other things (e.g., the rest of computational resources) on this capacity[4]. In time-triggered systems the monitoring capacity is known in advance. Therefore, during the setup phase the monitoring system must inform

[4]This restriction appears in case the operational network (see Section 5.1) is used for transmitting of collected monitoring data to the central monitor.

the user about the remaining monitoring bandwidth during selection of entities that need to be observed at this abstraction level.

At the node abstraction level the monitoring system offers the user the capability for monitoring nodes at different abstraction levels:

- Operating System (OS) Abstraction Level, and
- Application Abstraction Level

The monitoring at these abstraction levels is performed on the host of a particular node.

Monitoring at OS Abstraction Level

The main requirement of real-time operating systems is the guarantee of timing constrains of real-time applications, which are under their control. The role of the operating system in *time-triggered* systems is limited to *dispatching* tasks from a *static dispatch time-table*. Such a table is produced during design phase of a node by the off-line node design tool.

Definition 5.3 - (Monitoring at OS Level): *We define the monitoring at the OS abstraction level as a monitoring process during which the monitoring system observes entities that cannot be seen outside of the OS, and the run-time behavior of the OS of a particular node must be represented from these observations*[5].

In order to be able to support the monitoring at the OS abstraction level, the operating system of a particular node must be instrumented. The instrumentation code inserted into the OS is responsible for gathering of observations over entities that are not visible outside the OS. The introduced overhead into the OS caused by the instrumentation code must be taken into account by the off-line scheduler during the scheduling process. However, the monitoring at the OS abstraction level is transparent to the target application.

[5] At the OS abstraction level the run-time behavior of the target application is excluded, i.e., only the behavior of the operating system is regarded here. However, since the operating system has to control target applications, then only their tasks are considered here, which in fact are regarded as black boxes, which have their *worst-case execution times* (WCET) and deadlines.

Monitoring Data. During monitoring of time-triggered systems at the OS abstraction level the collected observations can represent the:

- activity of the OS's dispatcher,
- state of the stack and its consumption by different tasks, and
- state of the internal data structures.

The activity of the OS's dispatcher is one of the important goals[6] during monitoring of time-triggered target systems at the OS abstraction level. Thus, the activities of the dispatcher in time-triggered systems can be represented by observations over the following entities[7]:

S - denotes the start of a particular task instance,

E - denotes the end of a particular task instance,

PS - denotes the start of the preemption of the currently executed task instance,

PE - denotes the end of the preemption, i.e., the preempted task instance is resumed.

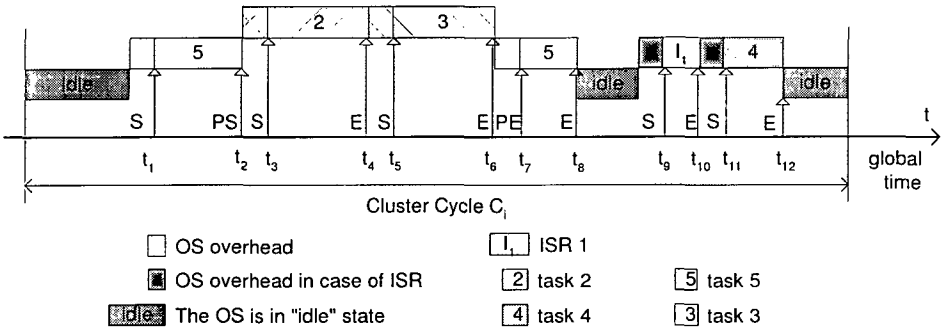


Figure 5.4: Monitoring at OS Abstraction Level

Therefore, the observations that are collected by the monitoring system at the OS abstraction level can be formally presented as:

$$OS_{obs} =< E_n, t_{obs}, ID >;\ E_n \in \{S, E, PS, PE\} \tag{5.3}$$

[6]The focus of this thesis during monitoring of the time-triggered systems at the OS abstraction level is the dispatcher's activity.

[7]These entities are typical examples of *real-time system* (RTS) *entities* presented in Section 4.2.

E_n denotes the name of the observed entity, t_{obs} denotes the timestamp, i.e., the time at which this observation was made. ID denotes the unique ID of the task instance or ISR.

Figure 5.4 presents a scenario, in which four different tasks and one interrupt service routine are dispatched by the OS's dispatcher. In this scenario the preemption of the task with ID 5 is presented. The task is preempted at t_2 and is resumed at t_7. The sequence of observations generated by the scenario presented in Figure 5.4 is depicted below:

- $Obs_1 =< S, t_1, 5 >$, start of the task instance with ID 5 at t_1,
- $Obs_2 =< PS, t_2, 5 >$, start of the preemption of the currently executed task instance with ID 5 at t_2,
- $Obs_3 =< S, t_3, 2 >$, start of the task instance with ID 2 at t_3,
- $Obs_4 =< E, t_4, 2 >$, end of task instance with ID 2 at t_4,
  ...
- $Obs_7 =< PE, t_7, 5 >$, the preemption of the task instance with ID 5 at t_9 is ended, i.e., the task 5 is resumed,
- $Obs_8 =< E, t_8, 5 >$, end of (preempted) task instance with ID 8 at t_8.

Based on these observations at the OS abstraction level the monitoring system can perform the following activities:

Visualization: The monitoring system performs the visualization of the dispatcher's activity at a particular node. This helps the user to get an overview over the dispatching of tasks at the particular node. In the case of errors in the temporal domain this visualization would help the user to find the reasons.

Measurement of Execution-Times: The monitoring system can measure the execution times of tasks, even in case these tasks are preempted[8].

Amount of Monitoring Data. The amount of monitoring data that can be collected at the OS abstraction level is limited by the monitoring capacity of a given node (see Equation 4.9). As mentioned before, the observed entities at this abstraction level can describe either tasks or ISRs, and therefore the amount of the collected monitoring data consists of:

$$A_{os} = A_{tasks} + A_{ISR};\ where\ A_{os} \leq C_m \tag{5.4}$$

[8]This will be presented in more details in Chapter 8.

A_{tasks} and A_{ISR} denote the amount within a cluster cycle of collected monitoring data that contain observations over entities that describe *tasks* or *ISRs*, respectively.

Tasks: The amount of monitoring data within a cluster cycle that contain observations that describe the dispatcher's activity during dispatching of *tasks* consists of:

$$A_{tasks} = A_{TD} + A_{PD} \tag{5.5}$$

A_{TD} contains monitoring data that describe the *start* and the *end* of each *dispatched task*, while A_{PD} contains monitoring data that describe the *start* and the *end* of each *preemption*. A_{TD} can be exactly calculated during the setup phase (see Section 5.2.3), because in time-triggered systems the points in time are known at which tasks must be activated. A_{PD} can be only estimated. However, the number of expected preemptions (in *worst-case*) can be estimated by an off-line analysis.

ISRs: The occurrences of interrupts are not known at pre run-time, because they have event-triggered semantic. Therefore, we can only estimate the amount of monitoring data within a cluster cycle that contain information over ISRs. However, the basic condition for using of interrupts in these systems is the limitation of interrupt's occurrences, i.e., ISRs must not occur more frequently than allowed by their minimum inter-arrival time. Therefore, we can calculate the A_{WCA} of monitoring data that describe the dispatcher's activity during dispatching of ISRs, because these monitoring data are *sporadic* monitoring data (see Section 4.3.1).

Monitoring at Application Abstraction Level

At the application abstraction level the monitoring system must collect observations over entities that can be found within the local part of the target application. From collected observations the monitoring system can reproduce the run-time behavior of the local part of the target application running on the host of a particular node. The monitoring at the application abstraction level is classified into monitoring at: i) *task* and ii) *function* abstraction level. The resident monitor (see Section 5.2.2) is responsible for collecting monitoring data at both these abstraction levels. However, for gathering of monitoring data at the function abstraction level, *sensors* (see Section 2.2.1) must be inserted into the target application. Furthermore, the resident monitor is responsible for transmitting of collected monitoring data to the central monitor at either abstraction levels.

Monitoring at Task Abstraction Level: In [Gal99] Galla has presented a task model that can be applied in time-triggered systems and which is restricted in following way: tasks receive input messages upon invocation and produce output messages upon completion. Furthermore, in this task model no interim communication or synchronization is permitted. In [Kop97] Kopetz addresses this kind of task model the *S-task model.*

Definition 5.4 - (Monitoring at Task Level): *We define the monitoring of target systems at the* **task abstraction level** *as a monitoring process during which observations over input and output messages of each observed task are collected.*

As above presented, we assume that tasks are considered by the monitoring system during monitoring at the task abstraction level as *black-boxes*, which are started at predefined points in time, and which use predefined input messages and produce output messages. Furthermore, these tasks have timing constraints in form of deadlines, which are known in advance.

The monitoring system's capability for monitoring of a particular node at the task abstraction level is very important, because this capability aids the user in a faulty case to isolate the faulty tasks. After one has detected the faulty task, one can apply monitoring of the particular node at the function abstraction level presented below to locate functions or modules that have caused the faulty behavior.

Monitoring Data. The sources, from which the input and output messages of tasks can be gathered, are classified as follows:

FT CNI: The fault-tolerant (FT) messages are processed by FT tasks [Bau00], which are executed transparently to the application tasks. Periodic FT tasks periodically update FT messages within the FT CNI with the period known in advance, i.e., at pre run-time.

Application: Messages that are exchanged among different tasks, or global variables could be relevant for monitoring systems. An example of such a message is the `M msg` in Figure 5.5, which is produced by application tasks and which is not transmitted to another interconnected nodes in the cluster. The amount of the monitoring data within a cluster cycle that contain observations over these messages can be calculated in advance, because these messages are updated by different tasks with different periods. The update period of these entities can be calculated during the design phase.

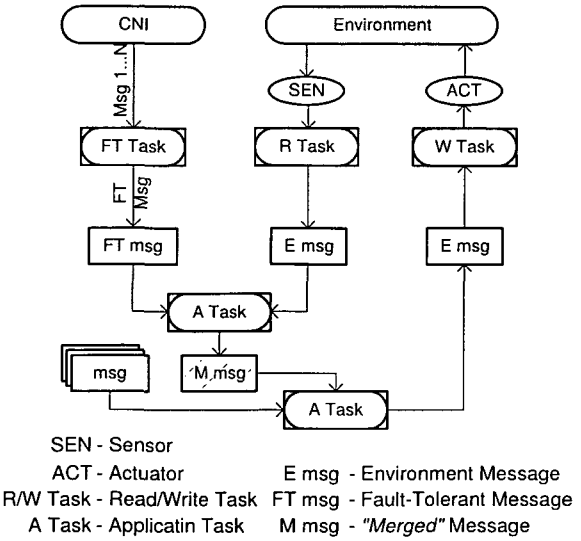


Figure 5.5: Example of Global Node Data in TTA Systems

Environment: While controlling their environment TTA applications have to *read* values from sensors or to *write* values to actuators. These inputs from sensors and outputs to actuators must be gathered for monitoring of target systems at the task abstraction level, because they are inputs/outputs of periodic tasks of TTA applications.

The monitoring data that are collected at the task abstraction level belong to the *node global monitoring data* presented in Section 4.3.2. Since, places are known in advance, where above presented messages are stored, for gathering of these data the OS gathering method presented in Section 4.4.2 is used. This means that the target application does not need to be instrumented, and therefore the monitoring process is done transparent to the target application.

Amount of Monitoring Data. At the task abstraction level the amount of monitoring data collected within a cluster cycle can be exactly calculated in advance, because in time-triggered systems the points in time, at which application tasks are started, are known in advance. This amount can be formally presented as:

$$A_T = \sum_{i=1}^{T} A_{IO} \tag{5.6}$$

where, T denotes the number of application tasks that are monitored, and A_{IO} denotes the input and output bytes of data per task instance within a cluster cycle.

Monitoring at Function Abstraction Level: As presented above, one must monitor the particular node at the function abstraction level after localization of the faulty task(s). This allows the user to localize the incorrect module(s) or function(s), by which the faulty behavior of the target application was caused. Therefore, the monitoring at the function abstraction level is defined as follows:

Definition 5.5 - (Monitoring at Function Level): *We define the monitoring of target systems at the* **function abstraction level** *as a monitoring process during which observations over entities are collected that are not visible outside of functions or modules of the local part of the target application.*

Monitoring Data. At the function abstraction level the monitoring system must collect the *intermediate* values (or *auxiliary outputs*). These monitoring data are called *node local monitoring data*, and are presented in Section 4.3.2. Furthermore, at the function abstraction level the monitoring system has to collect the monitoring data that are generated by the instrumentation code, such as function IDs, etc. These monitoring data are called *pure monitoring data* (see Section 4.3.2). For collection of these monitoring data the target application also must be instrumented.

Amount of Monitoring Data. In time-triggered systems the amount of collected monitoring data within a cluster cycle at the function abstraction level can be only estimated. It is possible in these systems to calculate the A_{WCA} of these data (see Section 4.3.1), because the *execution frequency* of tasks, which contain the instrumented functions, is known in advance. These can be delivered by *path analysis methods* analog to path analysis techniques applied during *WCET analysis* of real-time programs [Pus02a]. This amount can be formally presented as:

$$A_F = \sum_{i=1}^{T} A_{TM,i} \tag{5.7}$$

where T denotes the number of task instances from which the instrumented function is called, while $A_{TM,i}$ denotes the A_{WCA} of data that can be produced by instrumented function(s) within the i^{th} task instance during one cluster cycle.

Characteristics. During monitoring of target systems at the node abstraction level the user has the capability to observe the behavior within single nodes. This means that the nodes at the node abstraction level are considered as white-boxes. At the node abstraction level the target systems are influenced by the monitoring system, because the monitoring system uses the target system's computational, memory and bandwidth resources. However, this influence is taken into account during the setup phase.

5.3.3 Monitoring at Transducer Abstraction Level

As depicted in Section 3.2.2, the application running on the host of a node controls its environment via the controlled object interface (COI). In [Gal99] Galla states that the host uses another shared transmission medium, *field bus*, for its interconnection to the sensors and actuators. In the time-triggered architecture (TTA) the field bus application uses the TTP/A communication protocol [KHE00, KHE01, KLH01] for exchange of their data, and therefore they are called TTP/A applications.

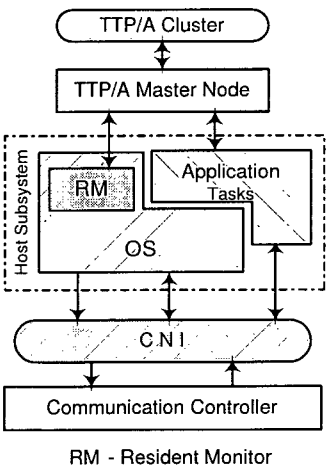


Figure 5.6: Transducer-Level Monitoring

While monitoring of target systems at the *transducer abstraction level*[9] the monitoring system sends monitoring commands to the resident monitor running on the interconnected nodes. The resident monitor translates these commands into the *interface file system* (IFS) read *requests* [OMG01]. These IFS read requests are queued by the resident monitor presented in Figure 5.6 and they are

[9]The monitoring of target systems at the transducer abstraction level is not the objective of this thesis.

transmitted at predefined points in time to the interconnected *TTP/A master* that is a gateway node between the host and the interconnected TTP/A cluster. After processing of IFS read requests the results returned by the TTP/A master are transmitted by the resident monitor to the central monitor, where they are further processed.

5.3.4 Overview of Monitoring at different Abstraction Levels

At the *cluster abstraction level* the monitoring system has to collect the *network monitoring data*. These monitoring data contain *state messages* that are gathered from the message area and *status information* that are collected from the status/control area of the monitoring node. The state messages and the status information are *regular monitoring data* and therefore the monitoring system can exactly calculate the amount of them within one cluster cycle. Furthermore, the network monitoring data contain *event messages* that also are collected from the message area of the monitoring node. However, the observation rate of the event messages is *sporadic* and therefore the monitoring system can only calculate the A_{WCA} of these monitoring data (see Section 4.3.1).

The monitoring at *node abstraction level* is classified into the *OS* and *application abstraction level*. The latter is classified into *task* and *function abstraction levels*. At the *OS abstraction level* the monitoring system has to collect *pure monitoring data* (see Section 4.3.2). Among other things, these monitoring data contain observation over entities that characterize the state of the dispatcher, i.e., the dispatching of either tasks or ISR. The monitoring data that describe the dispatching of tasks are regular and therefore the amount of them within a cluster cycle can be exactly calculate. The monitoring data that describe the ISR are sporadic, which means that only the A_{WCA} can be calculated. At the *task abstraction level* the the monitoring system has to collect *node global data*. These monitoring data contain input and output messages of tasks that can be gathered from the following sources: i) FT CNI, ii) application, and iii) environment of the real-time application (see Section 5.3.2). The amount of these monitoring data within a cluster cycle can be exactly calculated, because they are regular monitoring data. At the *function abstraction level* the monitoring system has to collect *local node* and *pure monitoring data*. These monitoring data contain *intermediate* values (or *auxiliary outputs*), and monitoring data that are generated by the instrumentation code inserted into functions, e.g., *function IDs*. However, we can only calculate the A_{WCA} of these data, because they are sporadic.

At the cluster abstraction level the monitoring data are gathered by the *monitoring-node gathering* method, i.e., the monitoring data are collected from

Abstraction Levels			MD	Gathering Source(s)		Observation Rate	Amount Prediction	Gathering Methods	Instr.	Influen.
Cluster Abstraction Level			Network MD	State Messages (MGA) + Status Messages (SCA)		Regular	Exact Calculation	Monitoring Node Method	No	No
				Event Messages (MGA)		NR / Sporadic	AWCA Calculation			
Node Abstraction Level	OS Abstraction Level		Pure MD	Dispatcher	Tasks	Regular	Exact Calculation	OS Method	OS only	Yes
					ISR	NR / Sporadic	AWCA Calculation			
	Application Abstraction Level	Task Abstraction Level	Global Node MD	FT CNI		Regular	Exact Calculation	OS Method	OS only	Yes
				Application						
				Environment						
		Function Abstraction Level	Local Node and Pure MD	Functions or Modules		NR / Sporadic	AWCA Calculation	In-Line Method	Application	Yes

MGA – Message Area
SCA – Status/Control Area
Instr. – Instrumentation of the Target System
MD – Monitoring Data
NR – Non-Regular MD
WCA – Worst-Case Amount
ISR – Interrupt Service Routine
Influen. – Influence on the Target System

Figure 5.7: Monitoring Abstraction Levels - Overview

the monitoring node's CNI. At the OS and the task abstraction levels the monitoring system has to use the *OS gathering* method for collecting of monitoring data. At the function abstraction level the monitoring system uses the *in-line gathering* method for gathering of needed monitoring data.

During monitoring at the cluster abstraction level the target system is not instrumented and therefore not influenced (no overhead is caused by the monitoring system). At the OS and the task abstraction levels only the OS must be instrumented. At these abstraction levels the resources of the target system are occupied by the monitoring system during gathering of monitoring data. At the function abstraction level the target application must also be instrumented. However, the influence of the monitoring system makes the target system more complex, but this influence is deterministic because system resources (CPU time, memory, communication bandwidth, etc.) are allocated for these purposes during the setup phase. Therefore, this influence does not change the behavior of the target system neither in the temporal nor in the value domain.

An overview of the presented monitoring system at different abstraction levels is presented in Figure 5.7.

5.4 Monitoring of Multiple Clusters

Real-time computer systems are used for controlling of complex physical processes. The complexity of real-time computer systems depends on the complexity of the physical processes which they have to control. In order to reduce the complexity of these real-time computer systems, they have to be decomposed into subsystems, i.e., clusters that are designed to control dedicated parts of the whole complex physical system. Therefore, one of the basic requirements for the presented monitoring system is the capability for simultaneously monitoring of target systems that consist of multiple clusters. The distributed monitoring system must have the capability of monitoring each of the interconnected clusters at the abstraction levels presented in the previous section.

5.4.1 Interconnection Topologies

A prerequisite for successful monitoring of target systems that consist of more than one cluster is their interconnection and their external synchronization. Depending on the fact how multiple clusters are interconnected to each other and how the monitoring system is interconnected to these clusters, we define two different topologies for interconnecting multiple clusters: *parallel* and *cascade* topology. The role of gateway nodes (see Section 3.2) depends on the interconnection topology. This role will be separately explained for parallel and cascade topology.

Parallel Topology

Definition 5.6 - (Parallel Interconnection Topology): *We define the type of the interconnection topology of multiple clusters as* **parallel**, *if the monitoring system is directly connected to each of the interconnected clusters to be monitored.*

In the parallel topology the activities of gateway nodes are limited to two basic activities (see Section 3.2). These activities are:

- exchange of inter-cluster messages between interconnected clusters, and
- external synchronization of interconnected clusters.

In Figure 5.8 an example of a parallel interconnection topology between a target system and the monitoring system is presented. In this example, the target system consists of three clusters, which are interconnected by two gateway nodes. Monitoring system is directly connected to each of the clusters (and gateways do not relay monitoring data).

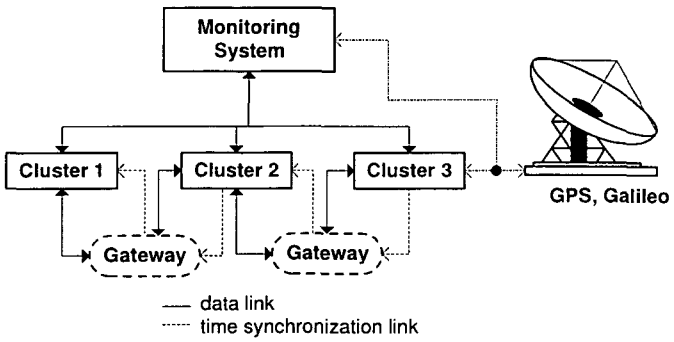


Figure 5.8: Interconnection of Multiple Clusters - Parallel Topology

Advantages. The advantages of a parallel interconnection topology are:

Role of Gateway Nodes: Gateway nodes do not have to exchange *monitoring information*[10] between interconnected clusters.

High Amount/Rate of Monitoring Data: The amount of collected monitoring data within an observation interval is limited only by the capacity of *event channels* of each interconnected cluster (see Section 3.3).

No Additional Impact: Only the clusters that are currently monitored are influenced. This influence depends on the abstraction level at which the target clusters are monitored (see Section 5.3). The other clusters are not influenced at all.

Disadvantages. The disadvantages of a parallel interconnection topology is the high number of links that must be used by the monitoring system for interconnection to all interconnected clusters. The number of data links is equal to the number of interconnected clusters. Furthermore, this high number of needed links can lead to problems of space nature, because in an already assembled target system for example, the user cannot install additional links that are needed by the monitoring system.

Cascade Topology

Definition 5.7 - (Cascade Interconnection Topology): *We say that the interconnection topology of multiple clusters is of type* **cascade**, *if the monitoring system is directly connected to only one cluster and to others via (multiple) gateways.*

[10] *Monitoring information* contain both the monitoring commands and collected monitoring data.

In the cascade topology the monitoring system is interconnected only to one cluster of the target system that consists of multiple clusters. This cluster is called *gate cluster* (in Figure 5.9 the `cluster 1` is the gate cluster), because it is the *gate* of the monitoring system to other clusters of a target system.

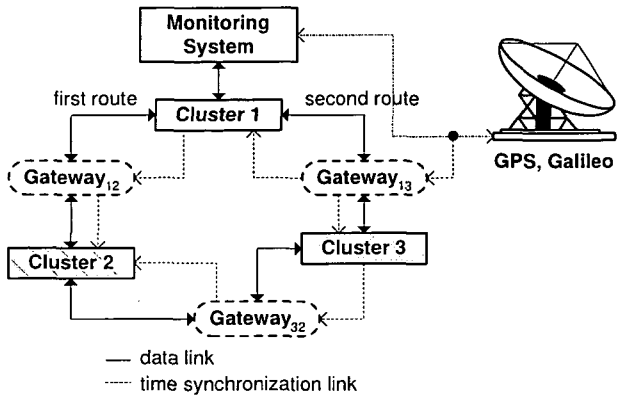


Figure 5.9: Interconnection of Multiple Clusters - Cascade Topology

In Figure 5.9 an example of a cascade interconnection topology between a target system that consists of multiple clusters and the monitoring system is presented. The target system consists of three clusters that are interconnected by three gateway nodes. In the cascade topology gateway nodes are responsible for external synchronization of interconnected clusters and for exchange of inter-cluster messages, which contain not only application messages but also monitoring data and commands.

Definition 5.8 - (Monitoring Route): *Let* **A** *and* **B** *be two different points within the target system. We call the connection between these two points the* ***monitoring route****, if the connection between these two points contains at least one cluster. A monitoring route that contains only one cluster is called* **simple***, while a monitoring route that contains more than one cluster and gateway nodes is called* **complex** *monitoring route (see Figure 5.10).*

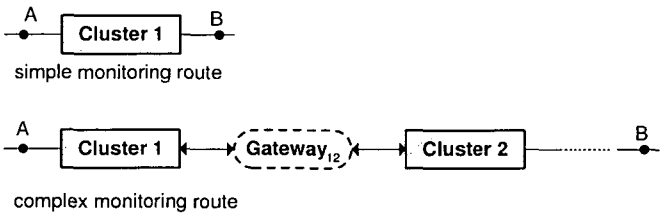


Figure 5.10: Monitoring Routes

The monitoring system uses *monitoring routes* to get monitoring access to the interconnected (destination) clusters of a target system, because it is not directly connected to each cluster. A monitoring route is characterized by two main characteristics: *monitoring capacity* and *propagation delay.*

Capacity of Monitoring Routes: The monitoring capacity is formally presented in Equation 4.9 (see Section 4.2.1). This definition can be applied for description of the monitoring capacity of simple monitoring routes.

For complex monitoring routes we define the capacity as follows:

Definition 5.9 - (Capacity of a Complex Monitoring Route): *For a complex monitoring route that consists of N_r simple monitoring routes the monitoring capacity equals the capacity of the simple route that has the smallest monitoring capacity, i.e.,:*

$$C_r = \min(C_i), i \in (1, N_r) \tag{5.8}$$

C_{r_i} denotes the monitoring capacity of the i^{th} simple monitoring route.

Propagation Delay: The *propagation delay* of a monitoring route is the duration between the point in time when the monitoring system starts a monitoring action on the remote cluster, i.e., it sends a command to the remote cluster, and the point in time when the monitoring system receives the collected monitoring data from the remote cluster through the monitoring route. The propagation delay must be taken into account by the monitoring system, in case different clusters shall be observed within the same time interval. The monitoring system sends sends one command (multicast) to all clusters[11] with an action time at which they shall start sending the collected monitoring to the monitoring system. In TTA systems the propagation delay of a simple route is bounded with the duration of one TDMA round. The propagation delay of a complex monitoring route is defined as:

$$P_d = \sum_{i=1}^{R} P_{sd_i} \tag{5.9}$$

P_{sd_i} denotes the propagation delay of the i^{th} simple monitoring route, R denotes the number of simple monitoring routes.

[11]This command contains an address part that is organized similar to the membership (see Section 3.5), i.e., each cluster has a corresponding bit in the address part. The cluster is considered as addressed if the corresponding bit in the address part is set.

In time-triggered systems both of these parameters are known in advance, because they can be fetched out from the *cluster design database* (see Section 3.2.1). The monitoring system during the setup phase for each interconnected cluster must search the *best* monitoring route(s), which have the sufficient capacity and the shortest length[12], over which the monitoring data will be transmitted. The shorter the monitoring routes are, the lower is the influence of the monitoring system on their clusters. This search is done by the *monitoring router.*

In Figure 5.9 a target system is presented that contains three clusters interconnected by three gateway nodes. The target system contains two different monitoring routes called *first* and *second* monitoring route, over which the monitoring system can reach `cluster 2`. If the capacity of the first monitoring route is not sufficient for monitoring of the `cluster 2`, then the second monitoring route must be also used, even though this will affect `cluster 3` and its neighbor gateway nodes.

Advantages. The main advantage of the cascade interconnection topology is the use of only one connection link from monitoring node to monitored application system to get access to different target clusters.

Disadvantages. The disadvantage of this topology are the bandwidth occupation in different clusters and gateways, and the necessity of relaying of monitoring data in gateways. The amount of monitoring data within an observation interval that can be collected using this interconnection topology is limited by the monitoring capacity of monitoring routes used to get access to the target clusters.

5.4.2 Parallel vs. Cascade Topology

The monitoring system must support both interconnection topologies, *parallel* and *cascade*, presented above. Both of these topologies have their advantages and disadvantages. Therefore, the user must make the decision, which topology he must select. This selection depends on the monitoring intention and on the network architecture of the target system. For example, the parallel topology can be used if the monitoring system can be directly connected to each of the clusters. Otherwise, e.g., if physical access to one of the clusters is not possible (or overly expensive) the cascade topology must be selected. One has to be prepared to deal with bandwidth limitations (may be higher efforts to get necessary monitoring data) if the cascade topology is used.

[12]The transmission of monitoring data can be splitted over different monitoring routes.

Another approach is the support of the *hybrid topology* by the monitoring system. The hybrid topology combines both the topologies presented above.

5.5 Debugging Support

There are two different techniques that can be applied for debugging of time-triggered systems: *distributed breakpoint*, and *deterministic replay*. The objective of this section is to present a brief description of these techniques.

5.5.1 Distributed Breakpoint

The key point of this technique is the *substitution* of the physical real-time through the virtual real-time, the progression of which can be *"stopped"* by the debugger. This substitution is achieved by the debugger *using* the control over the physical clocks of distributed nodes. This substitution is transparent to the target application. The debugger takes the control over the passing of the synchronized global time and therefore is able to allow the user to set breakpoints in distributed time-triggered systems. The target systems on which this debugging technique can be applied must support the concept of the *sparse time base* on which their model of global time must be based. This is the case in the TTA systems. A detailed description of the application of such a breakpoint technique in TTA systems can be found in [SA02].

The main disadvantage of this debugging technique is the need for simulation or emulation of the external environment (sensors and actuators) of the target system. However, it is worth noting that the simulation of the external environment is common practice during the development phase of real-time systems.

5.5.2 Deterministic Replay

The deterministic replay enables the user to deterministically reproduce the run-time behavior of the target system at an given abstraction level based on run-time information that has been gathered in an earlier monitoring process.

Replay at Cluster Abstraction Level

At the cluster abstraction level the monitoring system must be able to visualize all messages that are exchanged among interconnected nodes via the shared communication media. In time-triggered systems the replay process at the

cluster abstraction level is deterministic because these systems are based on a deterministic transmission scheme, i.e., each interconnected node sends its messages at predefined points of synchronized global time. The monitoring system must offer the user the capability for navigation over the time axis in order to display the activities of each interconnected node during the replaying process. Furthermore, the monitoring system must support the visualization of messages on different abstraction levels, e.g., *agreed* and *raw* values of exchanged messages, membership of nodes, etc. (see Chapter 7).

Replay at Node Abstraction Level

The deterministic replay can be used at the *node* abstraction level to help the user debugging the target application's tasks. During the monitoring process the input and output of tasks running at a particular node are collected. During the debug process the debugger redirects the input of tasks to the logging files that contain the input and output information of tasks which have been collected at the run-time. This means that we can debug a task with the same input which the task did have during the monitoring process. The debugger must be able to navigate over the time axis of the global synchronized time with the step equal to the granularity of the synchronized global time. Another requirement of the monitoring (debugging) system is the availability of the monitoring data collected at the cluster abstraction level during the monitoring process.

5.6 Chapter Summary

In this chapter we presented the real-time monitoring for the time-triggered target systems. We started with a presentation of the requirements and the assumptions. In addition, we presented the abstraction levels at which these systems can be monitored: i) cluster, ii) node, and iii) transducer abstraction level. For each abstraction level we presented the monitoring data that need to be collected for successful representation of the run-time behavior of the target system. The implementation of these concepts is the focus of chapters 7 and 8. Moreover, we presented the concepts for monitoring of time-triggered systems that consist of multiple clusters. We showed that there are different topologies for interconnection of the monitoring system to the target clusters. We also showed that one can develop a deterministic monitoring system that is able to predict its interference on the target system in advance, i.e., during the setup phase. This prediction is based on the knowledge about the expected amount of needed monitoring data within the observation interval, and the gathering methods used for collecting of these data at the intended abstraction level.

Chapter 6

Real-Time Triggers

The amount of monitoring data, especially during long-term monitoring, can be enormous. An alternative to storing enormous amounts of monitoring data is the use of the real-time trigger system (RTTS) that allows the user to store data of interest selectively. The RTTS records system operation of target real-time system (RTS) only in time windows of interest, around significant events. It does so by permanently buffering observations and checking for predefined significant events in real-time (RT).

The chapter describes the details of the RTTS and the method for defining and using of its trigger conditions. Section 6.1 presents the objectives and the used terms. Section 6.2 presents the proposed triggers. The *trigger definition language* (TDL) that is used for definition of trigger conditions is also part of this section. The triggered actions are dealt with in Section 6.3. Section 6.4 deals with the evaluation of the defined triggers. A summary closes this chapter.

6.1 Objectives and Terms

The intent of the monitoring system is on the one hand to collect all needed monitoring data and on the other hand to keep small the amount of monitoring data for disk space and bandwidth reasons. As depicted in Chapter 4.3.1, the amount of monitoring data among other things also depends on the length of the observation interval. The longer the observation interval is, the higher is the amount of monitoring data. This could be a problem for long-term monitoring, because the long-term monitoring requires a (very) high disk storage.

An alternative to storing enormous amount of monitoring data is the use of the RTTS that stores data of interest selectively. The RTTS records system operation of the target RTS only in time windows of interest, around significant

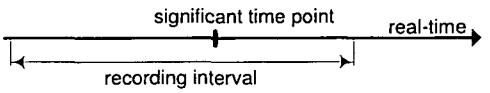


Figure 6.1: Logging Window (Recording Interval)

events. It does so by buffering of observations and looking for significant events in real-time (RT).

A *significant event* is defined as an event of interest, which has to be defined by the user of the monitoring system. Examples of such events are: the *temperature* of a controlled physical system or the *velocity* of a (controlled) car exceed their allowed limits. The time point, at which a significant event is found, is called the *significant time point*. The time interval during which the collected monitoring data are recorded by the RTTS is called *recording interval* or *logging window* (see Figure 6.1).

The RTTS in conjunction with the abstraction levels presented in the previous chapter (especially the *node* abstraction level) can also be used for both (long-term) *on-line diagnosing* and *off-line analyzing* of target systems. Another important usage of the RTTS is the *run-time correctness checking* of the target systems. The significant events the RTTS has to look for in this context are the violations of the system invariants that must be always fulfilled by the target system.

6.2 Triggers

The description of significant events is done by the trigger conditions (see below) that are checked repeatedly by the RTTS. Each time when the evaluation of a trigger conditions yields `true`, i.e., a significant event is found, the associated action is executed [Sma04]. The associated action has to record the monitoring data collected around the significant event, i.e., monitoring data that are collected within the logging window.

6.2.1 Trigger's Attributes

The triggers used by the RTTS consist of four attributes: *name*, *priority*, *condition*, and *action*.

Trigger Name: The name is used for trigger identification and consists of any alphanumeric character combination. A trigger cannot be defined without a definition of its name, because anonymous triggers are not

supported. Each time when the evaluation of a trigger yields `true`, i.e., a significant event is found, the user will receive a message that contains both the significant time point and the name of the trigger.

Trigger Priority: The priority can take any non-negative integer value and it is defined by the user. The smallest priority is equal to 1. If we define more than one trigger, their evaluation precedence depends on their priority, because the triggers are sorted by priority, before they are evaluated. In overload conditions the lower priority triggers are not evaluated. However, in this case the user will be informed about the triggers that were eliminated from the trigger evaluation list.

Trigger Condition: The trigger condition is defined by the user using the trigger definition language presented in Section 6.2.2 (see also Section 7.3.1). The trigger condition describes the significant event that has to be looked for by the RTTS.

Triggered Actions: The triggered actions are executed by the trigger system in case the trigger condition holds. Triggered actions are dealt with in Section 6.3.

6.2.2 Trigger Definition Language

Trigger conditions are defined in the *trigger definition language* (TDL). The syntax and semantic of this simple language has many similarities with the syntax and semantic of the C programming language. Most operators available in C can be used in TDL expressions for defining trigger conditions. TDL expressions can contain the most binary and unary operators found in C. TDL also offers some predefined mathematical constants such as π and e. Furthermore, a TDL expression can contain *system objects*, which are the names of messages that are exchanged among the interconnected nodes, or the names of internal variables of a given node, depending on the intended abstraction level at which the target system is monitored. TDL does not support the following C operators: assignment operators and their derivations: `++`, `--`, `+=`, `-=`, `*=`, `/=`, `%=`, ternary operators: `? :`, indirection and address operators: `*` and `&`.

Furhter, TDL supports an *instance index operator* `@[]` that is not derived from C. This operator allows the RTTS to access instances[1] of system objects from previous *repetition units*[2]. This operator belongs to the unary operators

[1] In case of TTA systems this operator allows the RTTS to access message instances sent in previous TDMA rounds.

[2] In real-time systems tasks are executed periodically. This time period in the context of the RTTS is called a *repetition unit*. In case of TTA systems the repetition unit is equivalent to a TDMA round.

and can be applied only to system objects. It takes a non negative integer n as parameter that refers to the instance of the system object that was sent in the n^{th} repetition unit in the past relative to the current repetition unit. This operator is checked by the RTTS during the trigger definition, and the user will get a warning, if this parameter exceeds the size of the underlying buffer used to hold the collected monitoring data.

In the following example, the use of the instance index operator is shown.

$$msg_1@[1] == msg_1 \tag{6.1}$$

In this example, the content of the previous instance of the msg_1 must be equal with the content of the current message instance. In this case the trigger condition fires, if the content of the last instances of the msg_1 has not been changed. To get the content of the current instance of the msg_1 the instance index operator with zero as parameter can be used, i.e., msg_1 is equivalent to $msg_1[0]$.

TDL expressions may also contain predefined functions. The supported functions are classified into *void*, *unary*, and *binary* functions depending on the number of passed parameters. Most functions that can be found in the C math library can be also used in TDL expressions. The most important predefined functions are *time functions* that return the occurrence time of an actual event. For example the `$time_in_usec` function returns the number of microseconds that passed since the start of the monitoring session.

The syntax of functions supported by the TDL is equal to the C syntax. The passed parameter(s) can be constant or complex expressions, i.e., other functions or nested expressions. Parameters are passed to functions using the `call-by-value` convention.

Predefined Constants

Table 6.1 contains the predefined constants that are supported by the TDL.

Symbol	Description	Value
$PI	π	3.141
$E	e	2.718

Table 6.1: TDL Predefined Constants

Unary Operators

The unary operators that are supported by the TDL are presented in Table 6.2. These operators are listed in descending order of precedence. Their associativ-

ity is `right-to-left`.

Symbol	Description	Example
~	bitwise complement	~ expr
!	logical NOT	! expr
-	unary minus	- expr
+	unary plus	+ expr

Table 6.2: TDL Unary Operators

Binary Operators

The binary operators that are supported by the TDL are presented in Table 6.3. These operators are listed in descending order of precedence. Their associativity is `left-to-right`.

Symbol	Description	Example
*	multiplication	expr1 * expr2
/	division	expr1 / expr2
%	modulus	expr1 % expr2
+	addition	expr1 + expr2
-	substraction	expr1 - expr2
<<	bitwise left shift	expr1 << expr2
>>	bitwise right shift	expr1 >> expr2
<	less than	expr1 < expr2
<=	less than or equal	expr1 < expr2=
>	greater than	expr1 > expr2
>=	greater than or equal	expr1 > expr2=
==	equal to	expr1 = expr2=
!=	not equal to	expr1 ! expr2=
&	bitwise AND	expr1 & expr2
^	bitwise XOR	expr1 ^ expr2
|	bitwise OR	expr1 | expr2
&&	logical AND	expr1 && expr2
||	logical OR	expr1 || expr2

Table 6.3: TDL Binary Operators

Void Functions

Table 6.4 shows the parameterless (void) functions of the RTTS.

Function	Description
`$time_in_usec`	time in microseconds
`$time_in_msec`	time in milliseconds
`$time_in_sec`	time in seconds
`$time_in_min`	time in minutes

Table 6.4: TDL Void Functions

$time_in_usec: This function returns the number of microseconds relative to the start of the monitoring session.

$time_in_msec: This function returns the number of milliseconds relative to the start of the monitoring session.

$time_in_sec: This function returns the number of seconds relative to the start of the monitoring session.

$time_in_min: This function returns the number of minutes relative to the start of the monitoring session.

Unary Functions

Table 6.5 shows the *unary* functions supported by the RTTS.

Function	Description
`$fabs`	absolute value
`$acos`	calculate arccosine
`$asin`	calculate arcsine
`$ceil`	find integer ceiling
`$cos`	calculate cosine
`$cosh`	calculate hyperbolic cosine
`$exp`	calculate exponential function
`$floor`	find largest integer less than or equal to arg
`$log`	calculate natural logarithm
`$log10`	calculate base-10 logarithm
`$_logb`	extract exponential value of argument
`$sin`	calculate sine
`$sinh`	calculate hyperbolic sine
`$sqrt`	find square root

Table 6.5: TDL Unary Functions

Binary Functions

Table 6.6 shows the *binary* functions of the RTTS.

Function	Description	Source
$_hypot	calculates hypotenuse of right triangle	Math library
$logb	calculates base-b logarithm of x [logb(x, base)]	System library
$fmod	finds floating-point remainder	Math library
$pow	calculates value raised to a power	Math library
$__min	returns smaller of two values	Math library
$__max	returns larger of two values	Math library

Table 6.6: TDL Binary Functions

TDL in EBNF Form

The syntax of the trigger definition language (TDL) in an extended Backus-Naur form (EBNF) annotation:

```
/*decimal, float and hexadecimal constants*/
__dec: [0-9]
__float:[0-9]"."[0-9]
__hex: [a-fA-F0-9]
__msg_names: [a-zA-Z0-9]*
__status_info: [a-zA-Z0-9]*

/*predefined constants*/
__pcon: $PI | $E

/*constants*/
const: __dec | __float | __hex | __pcon

/*system objects*/
sys_obj: __msg_names | __status_info | etc.

/*unary and binary operators*/
__uop: "~" | "!" | "-" | "+"
__bop: "*" | "/" | "%" | "+" | "-" | "<<" |
       ">>" | "<" | "<=" | ">" | ">=" | "=="
       | "!=" | "&" | "^" | "|" | "&&" | "||"
```

```
/*unary and binary expression*/
uexp: __uop (expr)
bexp:  (expr) __bop (expr)

/*complex expression*/
expr: uexpr | bexpr | const | sys_obj | func

/*void, unary and binary functions*/
vfunc: const $fname "(" ")"
ufunc: const $fname "(" expr ")"
bfunc: const $fname "(" expr "," expr ")"

/*functions*/
func: vfunc | ufunc | bfunc

/*instance index operator*/
@[]: sys_obj "@[" expr "]"
```

6.2.3 Trigger Conditions

The trigger conditions are used to define the significant events that must be found in real-time by the RTTS. For the rest of this thesis the notions *trigger conditions* and *trigger expressions* will be used alternatively. The trigger conditions are evaluated periodically, i.e., after each *repetition unit*. The result of the evaluation of trigger expressions is always of *boolean* type. The trigger, i.e., the significant event described by the trigger condition, is considered to be found, if the result of the evaluation process of the trigger condition is *true*.

The *trigger cascading* is defined by Eriksson [Eri97] as a process, during which events might be raised that could trigger other rules, when actions are executed, or even when conditions are evaluated. This cascade triggering can continue and might, in the worst case, result in the circular triggering where rules trigger each other infinitely [Eri97]. The RTTS does not support the generation of new events that could cause the evaluation of other triggers. Therefore, the RTTS is immune to the phenomena of trigger cascading. This immunity of the RTTS is the guarantee that the trigger system will never come into an infinite loop. Furthermore, the trigger conditions cannot contain loops, because the TDL does not support them, and therefore the evaluation time of the trigger conditions is always bounded.

The result of trigger evaluation is considered to be false in case an exception is generated (e.g., division by zero, etc.) during the evaluation process of trigger conditions.

6.3 Triggered Actions

Each time the evaluation of a trigger condition yields `true` and its execution is allowed (see Section 6.4) the associated action is executed.

6.3.1 Action Types

Trigger actions are classified into *system* and *recording* actions. System actions can influence the behavior of the monitoring system. Actions that are used for recording of collected monitoring data belong to the recording actions.

System Actions

The following actions belong to this action group:

stop_monitoring: The *stop_monitoring* action stops the monitoring process. This action can be used in conjunction with the `$time_in_usec`(or other time function), to stop the monitoring process after a desired evaluation time. This action is very helpful during automatic testing of the target system, during which different test scenarios have to be automatically executed. The *stop_monitoring* action has one parameter called `future` that denotes the number of repetition units, after which the monitoring process has to be stopped.

mode_change: The *mode_change* action starts the internal re-initialization of the monitoring system, in order to be able to process correctly the monitoring data collected from the target system that has changed its operational mode. This action has no parameter and executes immediately when new operational mode of the target system has been detected.

Recording Actions

Recording actions are used for recording of monitoring data collected during logging windows (see Figure 6.1). Recording actions are classified into three groups: *range_recording*, *start_recording* and *stop_recording*. The atomic unit of collected monitoring data that can be recorded by the RTTS is called *recording unit*[3] and therefore the parameters of the recording actions must be given as

[3]The *recording unit* is the smallest unit of monitoring data that are collected within a *repetition unit*. In case of TTA systems the recording unit is called *round packet*, which contains all monitoring data collected within a TDMA round [Sma02].

(integer) number of recording units. The underlying monitoring system holds the collected monitoring data in an internal ring buffer for a finite time interval, after which the old monitoring data are substituted with new ones. Therefore, the recording actions can store monitoring data collected before the significant time point, if they are available within the internal ring buffer of the monitoring system.

range_recording (past_units, future_units): The *range_recording* action records a range of monitoring data. The boundaries of the recording interval (see Figure 6.1) must be defined by the user. These boundaries are passed to this action as parameters. The parameters of this action are: `past_units` and `future_units` and they represent the number of repetition units that are collected before and after the significant time point.

start_recording (past_units, ∞): The *start_recording* action starts the recording process, which records a finite part of monitoring data that have been collected before the significant time point. The recording process of this action is stopped when either the monitoring session stops or a *stop_recording* action is executed. This action has one parameter, `past_units`, that represents the number of repetition units collected before the significant time point. This action can be considered as action of type *range_recording* with the second parameter of this action type considered to be ∞.

stop_recording (∞, future_units): The *stop_recording* action requires previous *start_recording* action and stops its execution. This action has one parameter, `future_units`, that represents the number of repetition units be recorded after the significant time point. This action can be considered as action of type *range_recording* with the first parameter of this action type considered to be ∞.

After an instance of *start_recording* action has been started, then only *stop_recording* actions are allowed to be executed, because the job of other possible actions of type *range_recording* or *start_recording* will be done by the actual executed *start_recording*. If two or more instances of *start_recording* or *stop_recording* action types overlap, they must be converted to *range_recording* action type, because actions that belong to the same type can be merged into new instances.

6.3.2 Overlapping of Recording Actions

The detection of significant events depends on the correlated trigger conditions. Therefore, it is possible that two or more instances of the same or different recording action types can overlap. Since, all recording action types can be considered as recording actions of *range_recording* type, the overlapping process is explained only for this type.

If two instances of the *range_recording* action type overlap, a new *range_recording* action will result, which has a new range of recording units that must be stored. Let A_1 and A_2 with parameters $A_1[past_1 : future_1]$ and $A_2[past_2 : future_2]$ represent two actions of type *range_recording* and t_1 and t_2 two significant time points at which the corresponding triggers have been detected. The following overlap scenarios can happen:

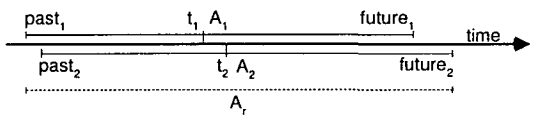


Figure 6.2: Action Overlapping (first scenario)

First Scenario: In this case, the first action A_1 is executed before the second action A_2, i.e., $t_1 < t_2$. After the merging process of these two actions the resulted merged *range_recording* (see Figure 6.2) action has the following limits:

$$A_r[t_1 - past_1 : t_2 + future_2] \tag{6.2}$$

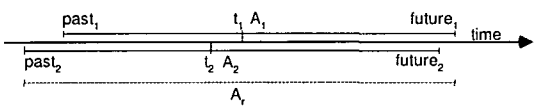


Figure 6.3: Action Overlapping (second scenario)

Second Scenario: The second action is executed before the first action, i.e., $t_2 < t_1$. After the merging process of these two actions the resulted merged *range_recording* (see Figure 6.3) action has the following limits:

$$A_r[t_2 - past_2 : t_1 + future_1] \tag{6.3}$$

Third Scenario: In this case, both triggers share the same significant time point, i.e. the triggers, to which these actions belong, are detected at

the same time. This means that $t_1 = t_2$ (see Figure 6.4). In this case the execution precedence depends on triggers priority. The new merged *range_recording* (see Figure 6.4) has the following limits:

$$A_r \left[\begin{array}{c} min(t_1 - past_1, t_2 - past_2) : \\ max(t_1 + future_1, t_2 + future_2) \end{array} \right] \tag{6.4}$$

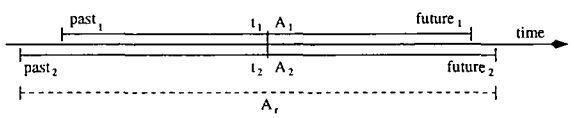


Figure 6.4: Action Overlapping (third scenario)

All recording action types can be considered as actions of *range_recording* type. Therefore, the above presented scenarios can be applied during the overlapping of different recording action types.

6.4 Trigger Evaluation

6.4.1 Trigger Compilation

TDL expressions used for definition of trigger conditions must be compiled *off-line* by the RTTS before the correlated triggers can be evaluated. Furthermore, during this compiling process the RTTS has to generate a *trigger evaluation list*, which is a priority ordered list of compiled triggers. This list is used by the RTTS during evaluation of the defined triggers.

6.4.2 On-Line vs. Off-Line Evaluation

There are two different subsystems of the RTTS by which the defined triggers are evaluated: *on-line logging subsystem*, and *off-line analyzer*.

On-Line Logging Subsystem

The purpose of the on-line logging subsystem is to store monitoring data of interest selectively during monitoring of the target system. This subsystem evaluates the defined triggers on-line and if significant events are found it executes the correlated actions presented in previous section. The correlated actions store selectively monitoring data around the significant events.

The on-line logging subsystem can be used for on-line diagnosing of target systems. It can also be used for checking the correctness of the target system. The user has to define trigger conditions that describe the violations of invariants of the target system that must be always fulfilled.

The on-line logging subsystem monitors the resources of the monitoring system, and eliminates triggers with lower priorities in case resources are not sufficient for their evaluation.

Off-Line Analyzer

In contrast to the on-line logging system the off-line analyzer does not execute the correlated recording actions, because the monitoring data has been already recorded. The purpose of the off-line analyzer is to provide a detailed analysis on the collected monitoring data. During this analysis specific significant events in the stored monitoring data can be found. Furthermore, during this detailed analysis the user has the capability to step forward or backward within the stored monitoring data stream with respect to the significant time point. The step resolution is one repetition unit.

6.5 Chapter Summary

In this chapter we have presented the RTTS, which is an intelligent recording system for real-time monitoring systems. The amount of monitoring data, especially during long-term monitoring, can be enormous. An alternative to storing enormous amount of monitoring data is the use of the presented RTTS that stores data of interest selectively. RTTS records system operation of target real-time system only in time windows of interest, around significant events. It does so by buffering and observation for significant events in real-time. RTTS can also be used for on-line diagnosing and correctness checking of target real-time systems.

Chapter 7

Implementation

This chapter gives a detailed description how *parts* of concepts presented in the previous chapters have been implemented. As presented in Figure 5.1 the distributed monitoring system consists of three main parts: *resident monitors*, *monitoring node*, and *central monitor*. In this chapter only the parts of concepts are presented that can be found in the software product TTPview [Sma02]. The implementation of the other concepts presented in the previous chapters are the focus of the next chapter.

This chapter is organized as follows: Section 7.1 begins with the presentation of the monitoring node's implementation. In Section 7.2 the implementation of the central monitor is presented. Section 7.3 deals with the implementation of the real-time trigger system, which is a client of the presented monitoring system. A summary closes this chapter.

7.1 Monitoring Node

As presented in Section 5.2.2, the *monitoring node* is a gateway between the target system and the *central monitor*. The monitoring node is implemented on a dedicated node, which has the same characteristics as other nodes of the cluster(see Section 3.2.2). The communication controller of the monitoring node is tightly synchronized with other cluster nodes. This synchronization guarantees that the collected monitoring data are temporally consistent (see Section 2.1.2).

The monitoring node is responsible for collecting of monitoring data that are needed during monitoring of TTA target systems at the intended abstraction levels (see Section 5.3). Furthermore, the monitoring node is responsible for sending monitoring commands to the resident monitors running on other nodes of the target system (see Section 5.2.2).

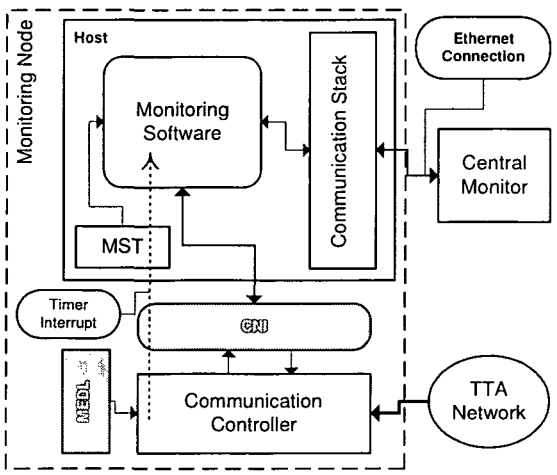


Figure 7.1: Monitoring Node

7.1.1 Monitoring Software

Collecting monitoring data and sending them to the central monitor as well as the receiving of monitoring commands from the central monitor, and sending them to the target nodes, is carried out by the monitoring software running on the monitoring node.

Monitoring MEDL

As presented in Section 3.2.2, the communication controller sends its messages and receives messages broacasted by other nodes at predefined points in time based on the information stored in the MEDL. Like other interconnected nodes, the monitoring node contains a communication controller, which has to collect all TTA messages[1] exchanged among interconnected nodes. To collect all these messages, the monitoring node needs a MEDL called the *monitoring MEDL*. The capability for collecting of all these messages is the difference of the monitoring MEDL to the MEDLs used by other nodes.

The MEDLs of each node, including the monitoring node, are derived from the *cluster database* [TTP02b] (see Section 3.2.1). Some of information that are contained in such a database, and which are important for the monitoring system, are:

[1] *TTA messages* are used by the interconnected nodes to exchange their information among them in a given cluster.

- number of nodes in the cluster of the target system,
- byte-order used by the interconnected nodes for transmission of their data over the shared transmission medium,
- list of messages sent by each node and their time points,
- number of TDMA rounds pro cluster cycle,
- replication level for each message, etc.

Monitoring Schedule Table

The monitoring software running on the host of the monitoring node uses a *Monitoring Schedule Table* (MST) for scheduling of its monitoring activities (see Figure 7.1). Such a table is also derived from the cluster database and contains the points in time when the monitoring software has to start with the collection of monitoring data. The communication controller[2] at these points in time triggers the monitoring software activities, which collect the monitoring data. The MST consists of two or more columns and each column consists of the following items:

- point in time when columns must be processed,
- addresses in the CNI where received data are stored by the communication controller,
- addresses within sending buffers where the collected monitoring data must be stored by the host monitoring software, and
- the sending flag that is used as trigger for sending of collected data to the central monitor.

Each time when the communication controller generates an interrupt, the monitoring software reads the MST table and finds out the item that must be processed at this time. After collecting all monitoring data sent within a particular TDMA round, the monitoring host software checks the sending flag, which is used as a sending trigger. The sending flag indicates that all monitoring data are collected within a particular TDMA round and that they must be sent as an atomic unit to the central monitor. The monitoring node then goes into the silent mode and waits for the next activation event to be generated by the communication controller.

[2]This information is stored in the monitoring MEDL, and the communication controller uses this information to generate a *user interrupt* [TTT02], which triggers the monitoring activities.

Host Software

In order to guarantee that no instance of any TTA message will get lost, the monitoring host software must be able to read out timely all received TTA messages from the monitoring node's CNI (see Figure 7.1). So, the real-time monitoring software uses the *read partition algorithm* (RPA), which divides the CNI of the monitoring node into a limited number of regions. The CNI division is described by the content of the MST. Each item in the MST describes a region in the CNI.

The RPA guarantees that the monitoring host software will read out the collected TTA messages from another region of the CNI, which were received by the communication controller in previous time window(s), while the communication controller writes the currently received TTA messages into next region(s). Thus, the usage of this algorithm guarantees that there will be no conflict between the communication controller and the monitoring host software, which concurrently share the same CNI. Moreover, this algorithm guarantees that as long as the monitoring node's host is fast enough to timely read out the collected TTA messages within a given time window, no instances of TTA messages will get lost. This algorithm requires the CNI to be divided into at least two regions to guarantee the conflict-free readout of the collected monitoring data from the monitoring node's CNI.

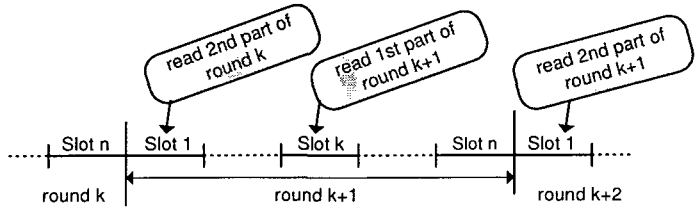


Figure 7.2: Scheduling of Monitoring Activities

In Figure 7.2 an example of scheduling of monitoring activities based on the RPA is presented. In this example the CNI of the monitoring node is divided into two regions. The monitoring host software is triggered by the communication controller at the beginning of the first and the middle slot of each TDMA round. In this case the monitoring host software will be triggered at the beginning of the `slot 1` of the `round k+1`, and it has to read out the TTA messages that are received by the communication controller within the second part of the previous round, i.e., within the `round k`. During this time the communication controller is receiving the TTA messages sent by the interconnected nodes during the first part of the current round, i.e., `round k+1`. On the other hand, when the monitoring host software is triggered at the beginning of the middle slot of the `round k+1`, the monitoring host software has to read out the

TTA messages received during the first part of this round. During this time the communication controller writes the received messages into the second part of the CNI.

7.1.2 Hardware Platforms

As denoted in the previous sections, the presented monitoring system was tailored for the Time-Triggered Architecture (TTA). The key component of this architecture is the Time-Triggered Protocol (TTP) [TTT02]. Therefore, the monitoring node was implemented on two different hardware platforms depending on the supported versions of the TTP [TTT99, TTT02]: *TTPnode* and ^{TTP}Monitoring-Node.

TTPnode - TTP/C-C1 Monitoring Node

The monitoring node implemented using the *TTPnode* [TTP01] is used for monitoring target clusters that use the TTP/C-C1 [AS801] communication controller. TTPnode uses the MC68360 [Mot95b] as a CPU, which is responsible for the execution of the monitoring host software. The monitoring host software is implemented as a standalone application, because it is not executed under the control of any operating system.

The communication stack on this hardware platform is implemented using the Ethernet (10Base-TX) controller MC68160 [Mot95a]. On the basis of such a communication stack, the *monitoring interconnection* which is used for the interconnection of the monitoring node and the central monitor, is implemented. The basic requirement for such an interconnection is a guaranteed transfer rate. The monitoring node must be able to send the collected monitoring data over this connection within a predefined time window, because after that it has to send the new collected one. Therefore, this monitoring node uses a *dedicated Ethernet connection.* We have developed a communication protocol presented in [KSF98, Kuc98], which guarantees a transfer rate up to 4 Mbps. However, the latest version of such an interconnection is implemented using a special developed communication layer based on UDP/IP protocol, which is more reliable than the older version that was based on the raw Ethernet. Especially this new interconnection offers new advantages in the central monitor, because this part can use the well defined socket interface, which is supported by almost all operating systems.

TTP Monitoring-Node - TTP/C-C2 Monitoring Node

The monitoring node implemented using the ^{TTP}Monitoring-Node [MN02] is used for monitoring target clusters that use the TTP/C-C2 [AS898] communication controller. ^{TTP}Monitoring-Node uses the MPC855T [Mot02] as a CPU, which is responsible for executing the monitoring host software. The monitoring host software runs under the control of an *embedded real-time Linux*. The monitoring interconnection to the central monitor is done using the Ethernet connection (100Base-TX), over which the reliable TCP/IP communication stack is built, which is made available by the operating system.

7.2 Central Monitor

The *central monitor* (CM) (see Figure 7.3) is implemented on a *commercial off-the-shelf* (COTS) system. The CM communicates directly with both the user and the monitoring node. Moreover, the CM is responsible for analyzing and planning of the monitoring process, and for managing and processing of monitoring data collected from the target system at the user's intended abstraction level. Managing the collected monitoring data is the job of the *system parts* (see Section 7.2.1), while the *monitoring clients* (see Section 7.2.2) are responsible for the processing of these data. The central monitor presented in this chapter is implemented in the software product TTPview [Sma02]. This tool is being successfully applied in industry for monitoring of distributed hard real-time TTA systems.

7.2.1 System Parts

The *system parts* of the central monitor are responsible for the management of collected monitoring data. The central monitor is implemented on the COTS system, and therefore it can be considered as a *best-effort* system. This part receives monitoring data from the monitoring node and puts them into the internal buffers for further processing. In this chapter only the implementation of the monitoring system is presented, which supports the monitoring of TTA target systems at the *cluster abstraction level* (see Section 5.3.1), and therefore the collected data contain only TTA messages that are exchanged among interconnected nodes via the shared transmission medium. These messages must not be separately time-stamped by the monitoring node, because the monitoring system uses the property of the TTA architecture, in which all activities are known in advance, including the points in time when TTA messages are sent or received. Thus, the monitoring system during time-stamping of the collected

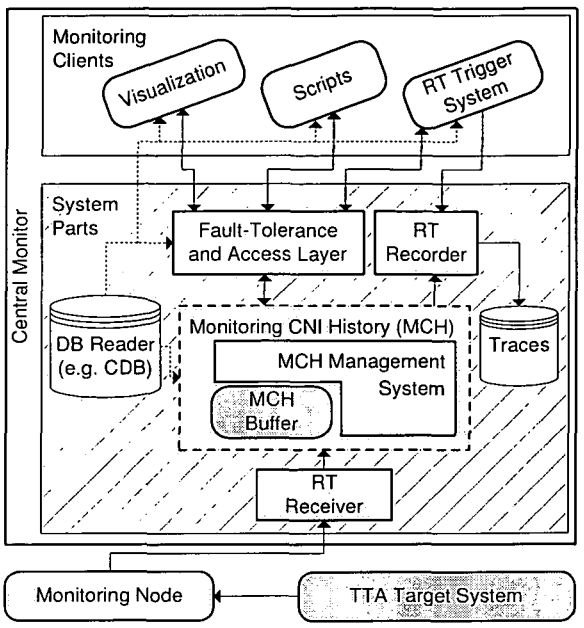


Figure 7.3: Central Monitor

data must only continuously enumerate the TDMA rounds, because the monitoring system can derive the observation times of each TTA message from the given TDMA round. The monitoring system must load the cluster database, in order to successfully interpret and process the collected monitoring data.

The monitoring system needs to know the amount of monitoring data collected within a cluster cycle, in order to be able to allocate the internal buffer before the monitoring process is started. The amount of monitoring data within a cluster cycle is exactly calculated in advance by means of the equation:

$$A_C = \sum_{i=1}^{R}(A_{SCA} + \sum_{j=1}^{N}\sum_{k=1}^{C} Fr_{ijk}) \tag{7.1}$$

where, R denotes the number of TDMA rounds within a cluster cycle, N denotes the number of nodes, C denotes the number of communication channels[3], A_{SCA} denotes the amount of monitoring data collected from the status/control area, and Fr_{ijk} denotes the length of the frame sent during the i^{th} TDMA round by the j^{th} node on the k^{th} channel.

[3]The number of replicated channels in the current implementation of the TTP/C protocol [TTT02] is limited to two ($C = 2$).

RT Receiver

The *Real-Time (RT) Receiver* is the most important system part of the central monitor. The RT Receiver receives all monitoring data sent by the monitoring node. This system part is implemented as an operating system task and has the highest priority in the monitoring system. The performance of the whole monitoring system depends on the performance achieved by the RT Receiver.

RT Recorder

The *Real-Time (RT) Recorder* stores monitoring data in real-time into trace files onto the disk. The stored monitoring data can be used by the monitoring system during off-line analysis. The RT Recorder has to store all received monitoring data. The off-line analysis depends on the availability of the monitoring data recorded by the RT Recorder.

Monitoring CNI History

Monitoring CNI History (MCH) (see Figure 7.3) is the functional part, which stores and manages the collected monitoring data received from the monitoring node. The MCH holds the instances of TTA messages for at least the *availability interval.* Within this interval the monitoring clients can use instances of TTA messages. After this interval the old TTA messages are substituted with the new one.

The smallest accessible unit of the MCH buffer is the *round packet*, which contains all instances of TTA messages sent within a given TDMA round. The header of such a packet contains the round number that is continuously incremented by the monitoring node and is used as a unique packet ID. As denoted above, the time-stamps of TTA messages contained within a given round packet can be derived from the packet ID.

The size of the MCH depends on the following factors:

Target Application: The size of the MCH depends on the target application, i.e., the amount of monitoring data that must be collected within an observation interval depends among other things on the number of entities of the target application that must be observed.

Monitoring Abstraction Level: The monitoring abstraction level at which the target system is monitored also influences the size of the MCH, because at different abstraction levels the amount of monitoring data is different.

Availability Interval: The size of the MCH depends also on the availability interval on which the time of the availability of TTA messages within the MCH depends.

Machine Resources: The size of the MCH buffer is directly dictated by the memory resources of the machine on which the CM is executed.

Since this part of the monitoring software is implemented on top of a non real-time system, it cannot guarantee bounded latency and therefore we cannot use the real-time to serialize the access on the MCH. To guarantee the logical data consistency in the MCH buffer, we have to serialize the access to the MCH using a concurrency control mechanism based on a locking paradigm. The locking mechanism is applied to each round packet. Each transaction locks the packet before it starts using it. There are two types of transactions that access the MCH: *write* and *read* transactions. There is only one write transaction, which is generated by the RT Receiver. The read transactions are generated by *Fault-Tolerance and Access Layer* and *RT Recorder*. The number of read transactions depends on the number of requests generated by the monitoring clients. The read transactions must not lock round packets for an *"unbounded"* time, because the write transaction must not be blocked for an *"unbounded"* time. If a read transaction tries to lock a round packet, which is locked by a write transaction, it has to wait until the round packet becomes free. On the other hand, the write transaction will try to find another free round packet, in the case the round packet is locked by a read transaction. This approach will never block the write transaction for an *"unbounded"* time, from which the performance of the central monitor of the monitoring system depends.

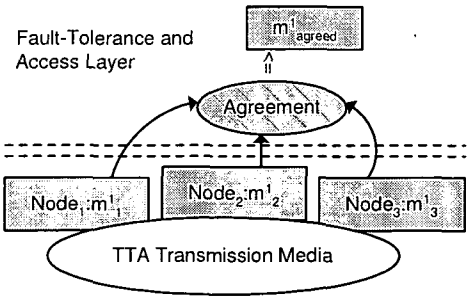


Figure 7.4: TTA Agreed Message

Fault-Tolerance and Access Layer

The *Fault-Tolerance and Access Layer* provides access to TTA messages stored into the MCH in the same way as they are accessed by a normal TTA application running on other nodes. The access to both the *raw* and *agreed* values of

TTA messages is supported. The raw value of a message is the same value as it is sent by the sender via the TTA shared transmission medium. Agreed values are values produced by *agreement algorithms* presented in [KBP01]. Agreement algorithms use different instances of the same message (for example, different sensors used for temperature measuring) and produce an agreed value of the used TTA message. In Figure 7.4 a scenario is presented, where msg_1 is sent by three replica nodes (1-3). The agreement algorithm uses these three instances of the message msg_1 and produces an agreed $msg_{1_{agreed}}$.

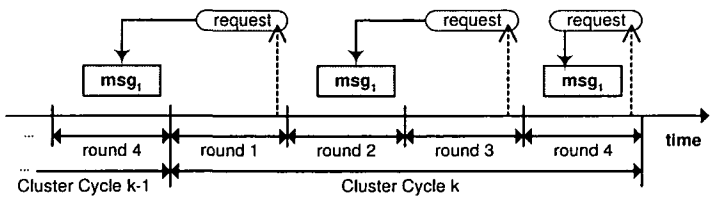


Figure 7.5: Fault-Tolerance Access Approach

Furthermore, this layer provides a means to access different instances[4] of TTA messages that are stored in the MCH. Depending on the time when the access request is made this layer provides the search for the desired instance of a TTA message. Such an approach is presented in Figure 7.5, in which a scenario is given where different instances of the same TTA message msg_1 are sent during the second and the fourth TDMA round of a cluster cycle. If the user of this message makes a request in round four, the actual instance of the message sent during this round is used. However, if the user requests this message in the third round, the value of the message sent during the second round of the cluster cycle is used, because in the third round this TTA message is not sent at all.

7.2.2 Monitoring Clients

The collected monitoring data are processed by the monitoring clients, which are: *visualization*, *scripts*, and *real-time trigger system*.

Visualization Client

The visualization clients provide a graphical presentation of collected monitoring data at different abstraction levels, e.g., the visualization of *raw* messages, *agreed* messages, *raw frames* (as they are sent via the shared transmission

[4]The value, which a TTA message has during the transmission in a given TDMA round, is called the *instance* of the TTA message.

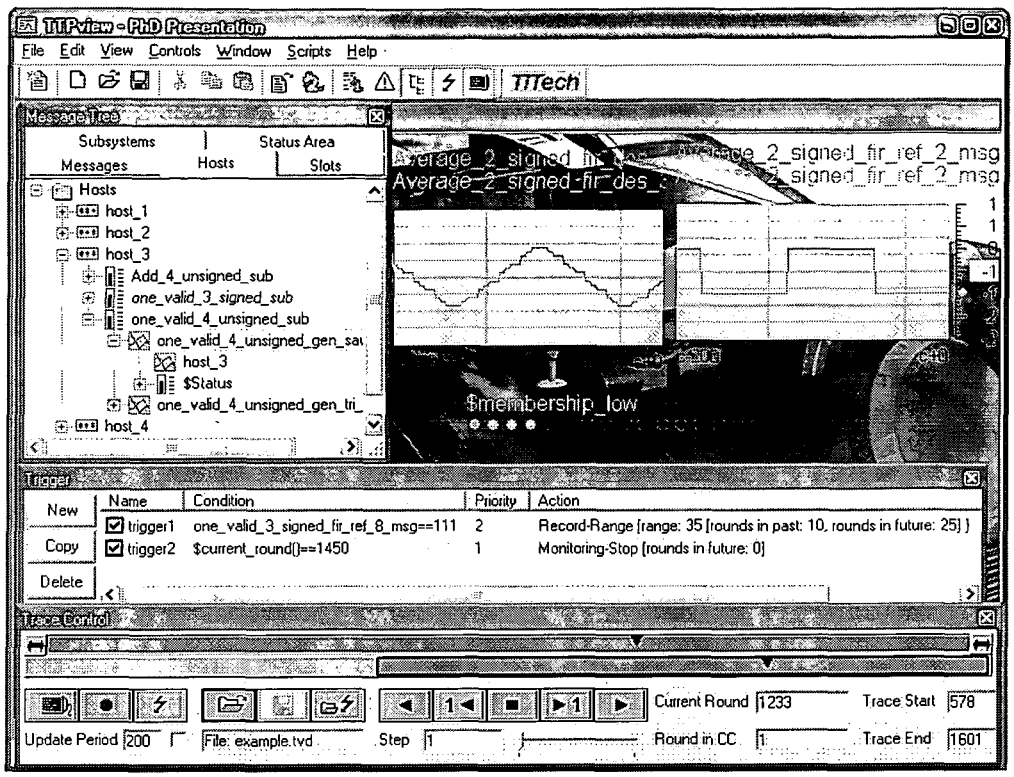


Figure 7.6: TTPview - The Real-Time Monitoring System

medium), *status area objects* (e.g., membership vector), etc. The monitoring tool (TTPview) offers the user an *object browser* (see Figure 7.7), which allows fast and convenient access to the messages and status fields, which can be selected for visualization. There are five *object types* that can be selected to define sort order and hierarchy: *messages*, *hosts*, *subsystems*, *slots* and *status area fields*. The hierarchical tree for every object type starts with the object type name. At the next hierarchy level the selected object type is shown in alphabetical order.

Message Tree: The first hierarchy level shows the messages agreed by the fault-tolerant communication layer. The second level in the message tree consists of the raw values of a message and the message status.

Subsystem Tree: The subsystem tree shows all subsystems of the cluster at the first level. The next hierarchy level shows the message tree of all messages sent by the selected subsystem. In addition the replication level of the subsystem is shown.

Host Tree: The host tree shows the hosts (i.e., nodes) of the cluster at the

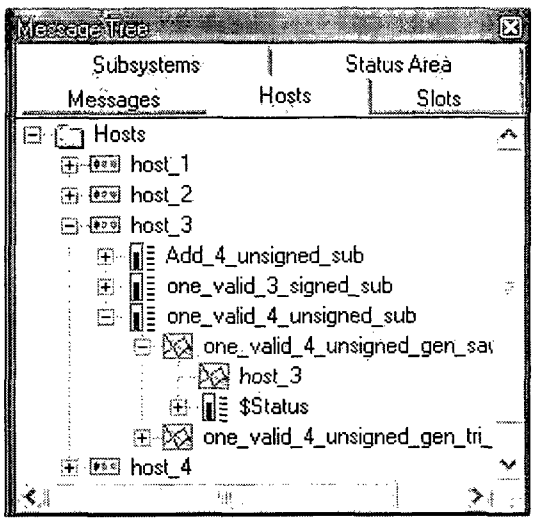


Figure 7.7: Message Tree

top level. The next hierarchy level shows the subsystem trees of the subsystems executed on the selected host, the host membership status and the channel status of the host. The channel status is the number of correct N-frames sent by the host in one TDMA round.

Slot Tree: The slot tree consists of the TDMA round slots at the first hierarchy level. On the next hierarchy level the channel status and the frames sent in the slot are shown. The frames are sorted by round and channel and are visualized as hexadecimal byte fields. After expanding a frame, the message raw values contained in this frame are shown and can be displayed.

Status Area Tree: This tree contains relevant information of the monitoring node's CNI status area, e.g., *membership vector*, *cluster mode* [TTT02]. The snapshot of this area is performed at the end of the TDMA round.

Script Client

The presented monitoring system makes a script interface available, with which the user can access its functionality within the Python programming language. Using this interface the user can use the monitoring system during the testing phase for intensive testing of the target system, because this interface allows automation of the testing approach. The `stop_monitoring` action of real-time triggers can be used for the automation of the testing approach.

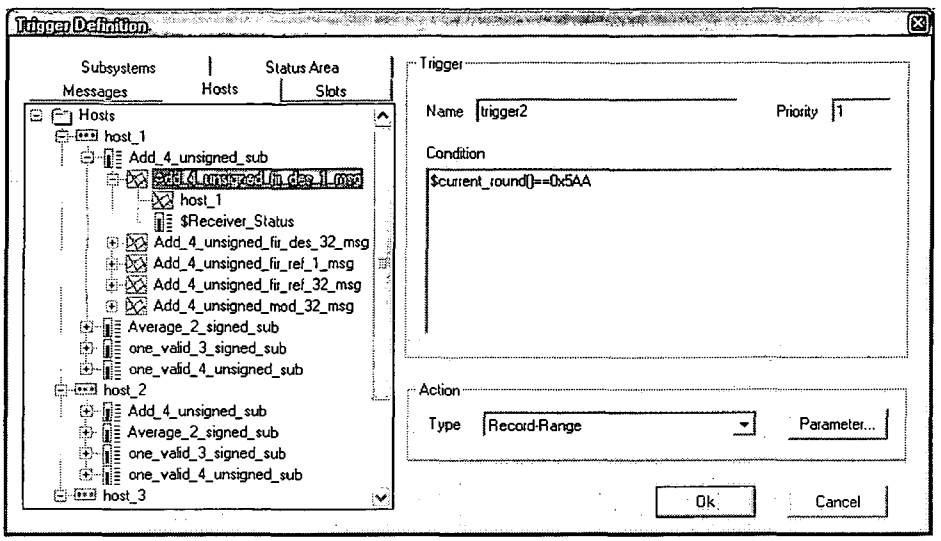


Figure 7.8: Definition of Triggers in TTPview

7.3 Real-Time Trigger System

The real-time trigger system is implemented as a monitoring client on top of the presented monitoring system (TTPview) (see Figure 7.3).

7.3.1 Trigger Definition

For defining triggers in TTPview the user can use either the GUI (see Figure 7.8) or the script interface. After all trigger components are defined, then the condition is parsed by the parser and if all components are correct, then the new trigger will be added into the *trigger list* (see Figure 7.9). This list of defined triggers is used by the parser for generating the *trigger evaluation list* (see Section 6.4.1).

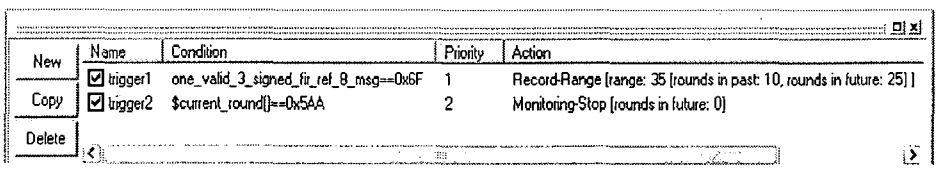


Figure 7.9: Trigger List

7.3.2 Trigger Evaluation

Currently, the RTTS is implemented as an operating system's thread. In case a significant event is found, i.e., if the trigger's condition holds, then a message is displayed that informs the user over the detected trigger. In Figure 7.10

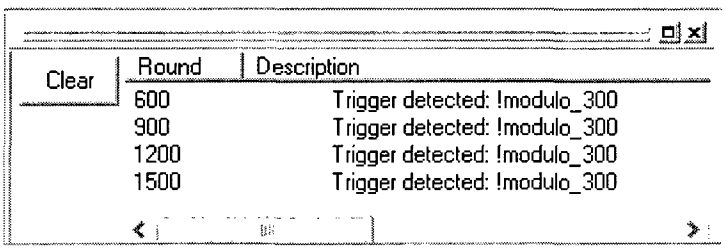


Figure 7.10: Trigger Detection List

the list of detected triggers is presented, during evaluation of the `!modulo_300` trigger by the trigger engine. This trigger has the condition

$$\texttt{\$current\_round}\ \texttt{()}\ \%\ 300\ == 0 \tag{7.2}$$

and it has been evaluated within a trace with limits: 578 and 1600 TDMA rounds. The information messages contain the number of the TDMA round at which the trigger was detected and the name of the detected trigger.

If there are not enough resources needed for the trigger evaluation, i.e., the machine is overloaded or it is not fast enough, the RTTS sends an information message to the user that the defined triggers cannot be evaluated in every round. In this case the trigger with the smallest priority will be deleted from the trigger evaluation list.

Each recording action has to record a logging window, which contains data collected before and after the significant time point. The RTTS closes the actual logging window after all monitoring data that belong to the actual window have been recorded. The closed logging window is then put into the list of the logging windows. This list is appended into the trace file, where the logging windows are recorded. The monitoring system uses this list for interpreting of recorded trace files in the off-line mode.

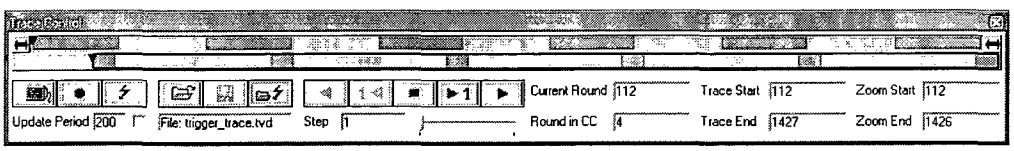


Figure 7.11: Trace Control - List of Logging Windows

Figure 7.11 presents the *Trace Control* which, among other things, is responsible for loading of recorded traces in TTPview (see Figure 7.11). It displays

two different sliders, the lower and the upper, which are alternate colored. Due to triggering, a trace is not a continuous stream of collected monitoring data, but can consist of several blocks, i.e., logging windows. The lower slider shows the trace relative to the point in time of the monitoring start and with temporal correct spaces between logging windows. The upper slider displays the zoomed marked section of lower slider without inter-logging windows spaces.

7.4 Chapter Summary

This chapter described the implementation of the parts of concepts presented in the previous chapters that can be found as features of *TTPview* monitoring tool. We started with the presentation of the monitoring node's implementation, during which we paid special attention to the read partition algorithm. The read partition algorithm is used by the monitoring node's host software for conflict-free reading of monitoring data from the monitoring node's CNI. Furthermore, we presented two different hardware platforms that are used for the implementation of the monitoring node.

Another objective of this chapter was to present the implementation of the central monitor that was implemented in the software product TTPview. TTPview is successfully being applied in industry for monitoring of TTA safety-critical real-time systems.

Chapter 8

Case Study

This chapter describes the concrete monitoring application. This monitoring system serves as a case study for the concepts that have been presented in the previous chapters. It provides the capability to monitor the target systems at the OS and node abstraction level. The brake-by-wire (BBW) application developed by Volvo Technological Development is used as a target application.

The objectives of this chapter are presented in the next section. Section 8.2 starts with the description of the hardware and software setup of the target system and continues with the presentation of preparation steps needed for monitoring the target system. Furthermore, this chapter deals with the collection, processing and presentation of monitoring data collected at the above mentioned abstraction levels.

8.1 Motivation and Objectives

The user can get in insight into the behavior of a particular node by means of the concepts of monitoring of target real-time system at the *node abstraction level* (see Section 5.3.2). There are potential faults in both time and value domain that cannot be detected without the usage of monitoring systems. An example of such faults in the time domain is the deadline violation of tasks during the development of a new real-time system caused, for example, by an implementation bug in a new applied algorithm. In the value domain such a fault can be caused, for example, through an erroneous assumption over the limits of the range of values returned by the applied sensor or algorithm. These faults can be detected by means of the concepts presented in this thesis that enable the visualization of the dispatcher's activity, the measurement of the execution-times and the visualization of inputs/outputs of the scheduled tasks. The implementation of these concepts is the focus of this chapter.

The objective of this chapter is to show that:

- the concepts presented in the previous chapters (that are not implemented in the TTPview[Sma02]) are implementable,
- the expected amount of monitoring data within an observation interval during monitoring of target systems at the OS and the task abstraction level can be calculated in advance, and
- the code that implements these concepts has to use resources that are either reserved for monitoring purposes during the design phase, or an predefined amount of unused resources from an already developed target system. The resources needed for monitoring at each intended abstraction level can be calculated in advance. However, the amount of them is of course limited.

The *long-term diagnosing* and the on-line *correctness checking* of TT target systems can be successfully performed by mean of *concepts* presented in the previous chapters, especially the combination of the *real-time triggers system* (see Chapters 6 and 7) and the *node abstraction level* (see Section 5.3.2). The RTTS has to look for significant events that describe the *invariants* of the system that must be always fulfilled. The violation of invariants causes the detection of a significant events that trigger the recording of the time windows around the significant time points. The investigation of such a violation can be successfully done by mean of these recorded monitoring data.

8.2 System Setup

The target application that is used in this case study is also used as target application in the EU funded IST Project FIT (Fault-Injection for the TTA) [Ade03].

8.2.1 Target Hardware and Software Application

Target Hardware

The target system (see Figure 8.1) used in this *case study* consists of only one cluster. It consists of four nodes that are interconnected by a TTP/C bus[1]. Each of these nodes is a *TTP-Powernode*[2]. A Motorola Controller MPC555 is used as host of these nodes, while the communication controller is of type TTP-C2 (AS8202).

[1] From the monitoring system's point of view there are no differences between those systems that use star and those that use bus topology.

[2] http://www.tttech.com/products/hardware/powernode/overview.htm

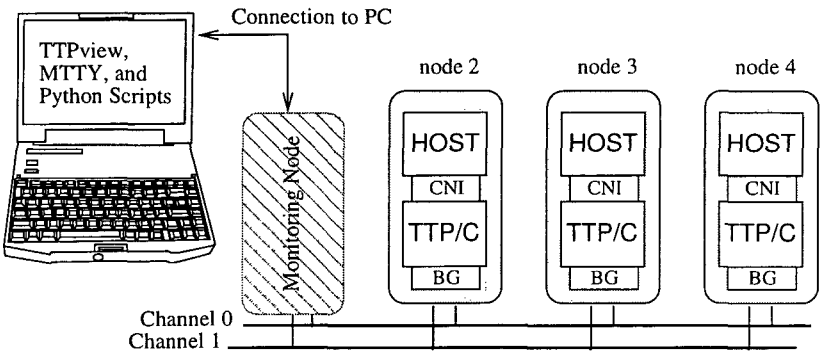


Figure 8.1: Case Study - Setup

The monitoring node is used as an interface between the target system and the PC. The PC sends its monitoring commands to the target system over the monitoring node. Furthermore, the collection of the monitoring data from the target system and the sending them to the PC is done by the monitoring node.

Active Monitoring Node: In the current implementation the monitoring node is not able to send messages to other interconnected nodes via the shared transmission medium. This means that the monitoring node is only a *"passive eavesdropper"* of the activities which take place on the shared transmission medium of the target system. Therefore it is called the *passive* monitoring node.

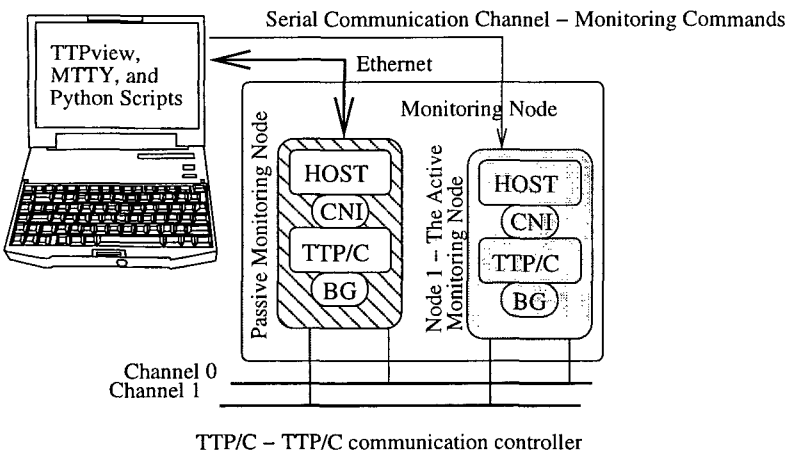


Figure 8.2: Active Monitoring Node

As we need to send *monitoring commands* to the target nodes that will be monitored at the *node abstraction level*. We used one of the target nodes, i.e., `node 1`, in conjunction with a passive monitoring node to simulate

an *active monitoring node* (see Figure 8.2)[3]. In fact, the monitoring node presented in Figure 8.1 consists of two nodes in our prototype setting:

- **Passive Monitoring Node** that is a normal monitoring node which monitors the target systems at the cluster abstraction level and which is connected to the PC over an Ethernet link (see Section 7.1).
- **Active Monitoring Node** is the modified node 1, which receives the monitoring commands from the user (via the *personal computer* - PC) and sends them to the target nodes the user wants to monitor.

Target Software Application

A brake-by-wire (BBW) control system is used as an application workload. BBW is a distributed simulation program designed by Volvo Technological Development. The BBW model consists of two parts, one part modelling the vehicle and the other part modelling a wheel node. The input to the vehicle model is an initial speed value and the brake pedal angle. The vehicle model uses the brake pedal angle to calculate a brake force, which is transmitted to the wheel node. The wheel node calculates the force to be applied on the brake discs. Here, the calculated force is returned to the vehicle model (*BrakeSignal*). The vehicle model calculates the speed reduction caused by the friction force obtained when the brake pad is pressed against the brake disc and then sends new information about the vehicle speed and the speed of the wheel to the wheel node (*VehicleSpeed*, *WheelSpeed*). The wheel node uses the speed of the vehicle and the speed of the wheel to calculate the wheel slip, i.e., the speed difference between the vehicle and the wheel, reducing the brake force if a specified slip level is exceeded. Otherwise, the brake force is increased. This allows the brake force to be adjusted for optimized braking performance. A similar simulation program is used in [AVFJ02]. Both simulation programs run in the I/O controller. The *VehicleSpeed* simulation runs at the node 2 (*data generator node*), and *Wheel* simulation runs at the nodes 3 and 4. The application in the host controller reads the simulation outputs from the I/O controller and calculates the end-to-end checksum.

8.2.2 Software Tools

The software tools that were needed for preparation of the target system of this case study are:

[3]In order to implement an active monitoring node we would have to change the software tool chain such as TTPplan, TTPbuild and TTPload [TTP02b, TTP02a].

TTPplan [TTP02b] is used for allocating of additional *communication bandwidth* needed for transmission of collected monitoring data.

TTPbuild [TTP02a] is used for allocation of additional processing resources needed for monitoring at the node abstraction level. Thus, the fault-tolerance layer (FTL)[4] is modified for handling of the additional added monitoring data.

TTPview [TTP02c] is used for collecting of the monitoring data gathered by the resident monitors running on the target nodes that are monitored at the node abstraction level.

MTTTY [5] is used for sending of monitoring commands given from the user to the monitoring node via the serial communication channel (see Figure 8.2).

8.3 Monitoring Setup

In the target application used in this case study we monitored the *third* node at the *OS* and *task* abstraction level. The *fourth* node was monitored only at the task abstraction level. Additional functionality was assigned to the *first* node (as above presented), i.e., it has taken the role of an active monitoring node.

The monitoring process is initiated by the user using the MTTTY software tool for sending of monitoring commands to the monitoring node that forwards them to the target nodes the user wants to monitor. Therefore, an additional message called *monitoring message* (`n2_u4_mon_cmd`) has been added to the `node 1` (i.e., the *active* monitoring node in Figure 8.2) that is used for carrying out of the monitoring commands. The `nodes 3` and `4` implement additional messages (`n3_a_mon` and `n4_a_mon`). These messages are of type *byte array* [TTP02b] and are used to simulate the *monitoring channels*. The monitoring data collected from the target nodes are carried out by these messages to the monitoring node.

8.3.1 OS Abstraction Level

While monitoring target systems at the OS abstraction level the observations over entities are collected that are not visible outside of the OS. Therefore,

[4] FTL is a middle-ware layer responsible for reception and sending of node's messages.
[5] Available at http://msdn.microsoft.com/library/techart/msdn/serial.htm.

the OS must be instrumented. The activity of the dispatcher is one of the monitoring objectives presented in Section 5.3.2 that is choosen to be implemented. The entities that describe the dispatcher's activities are presented in Equation 5.3 (see Section 5.3.2).

The instrumentation code inserted into the OS uses the data structure presented in Figure 8.3. There are two buffers, *read* and *write*, that are alternated periodically at the beginning of each cluster cycle. This means, that while the observations observed during the i^{th} cluster cycle are sent to the monitoring node, the observations collected during the current cluster cycle, i.e., $(i+1)^{th}$ are written into the write buffer. The collected monitoring data are not sent

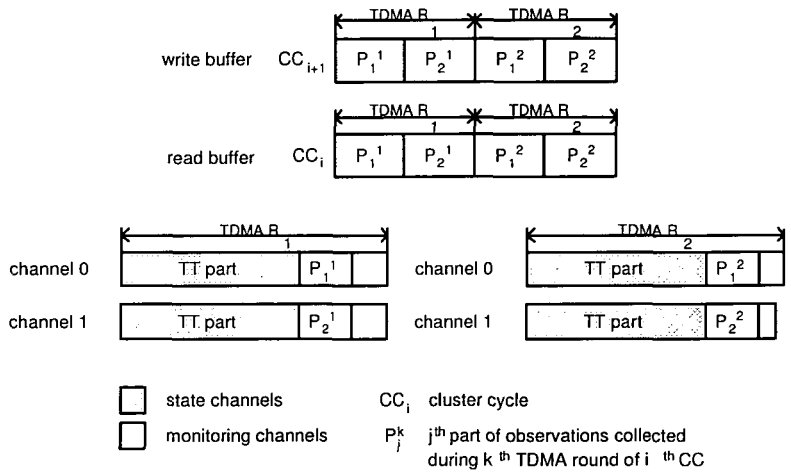


Figure 8.3: Data Structure used by the Instrumentation Code inserted into the OS Dispatcher

redundantly on TTP/C communication channels. Furthermore, the sending of these data is spread out over all TDMA rounds of a cluster cycle. In this case study the cluster cycle consists of two TDMA rounds, i.e., TDMA R_1 and R_2 (see Figure 8.3). The first two parts (P_1^1 and P_2^1) of the monitoring data collected during the first TDMA round of cluster cycle i (i.e., CC_i) are sent during the first TDMA round of cluster cycle $i+1$ (i.e., CC_{i+1}), while the other two parts (P_1^2 and P_2^2) are sent during the second TDMA round.

Monitoring Data. In the current implementation of the monitoring at the OS abstraction level an observation is 3 bytes long:

entity - 2 bits : An entity during monitoring at the OS level can be of type:

S - denotes the start of a particular task instance,

E - denotes the end of a particular task instance, and

PS - denotes the start of the preemption of the currently executed task instance,

PE - denotes the end of preemption, i.e., the preempted task is resumed.

t_id - 6 bits : The `t_id` represents the *ID* of the tasks the state of which is currently changed. In this implementation we can monitor target systems with up to 64 tasks.

t_stamp - 16 bits: The `t_stamp` represents the point in time when the correlated observation was made. The time-stamp must be based on the global time in order to be able to correlate the collected monitoring data with other data collected from other nodes. In the current implementation the 16 bits global time provided by the TTP/C communication controller is used for time-stamping. In case of a counter overflow, the *overflow counter* [TTT02] is incremented. The incrementation of the overflow counter takes place consistently at all participant nodes, inclusive the monitoring node. Therefore, during processing of these collected monitoring data the overflow counter must be read out from the monitoring node, and it must be added to the `t_stamp`.

The amount of monitoring data that can be collected at the OS abstraction level within a cluster cycle is estimated by means of Formula 5.5 (see Section 5.3.2).

$$MD_{WCA} = \sum_{i=1}^{T} T_{D,i} + \sum_{j=1}^{P} Pr_{D,j};\ T_{D,i}, Pr_{D,j} = 2 * len(obs) = 6\ bytes \quad (8.1)$$

where, T denotes the number of tasks within a cluster cycle, while P denotes the estimated number of expected preemptions. Node 3 of the target system used in this case study comes with 16 tasks. The estimated number of preemptions on this node within a cluster cycle is 4. Therefore, the amount of monitoring data within a cluster cycle that are collected during monitoring of the node 3 at the OS abstraction level is equal to:

$$MD_{WCA} = \sum_{i=1}^{16} T_{D,i} + \sum_{j=1}^{4} Pr_{D,j} = 120\ bytes. \quad (8.2)$$

Bandwidth Occupation. In each TDMA round, part of collected observations are transmitted to the monitoring node (see Figure 8.3). Therefore, the bandwidth occupation within a cluster cycle during monitoring of the target system at the OS abstraction level can be calculated as follows:

$$BO = h * MD_{wca}/(N_{TDMA} * N_{ch}) = 1/\Delta t * 120/(2 * 2) = 30\ bytes/\Delta t \quad (8.3)$$

N_{TDMA} denotes the number of TDMA rounds that in case of the used *BBW* target application is equal to 2. N_{ch} denotes the number of physical communication channels. In the current implementation of TTP/C protocol there are only 2 physical channels. The collected monitoring data are not processed before they are transmitted. Therefore, h is equal to $1/\Delta t$ (see Section 4.2.1), and Δt is the time duration of a cluster cycle.

During monitoring of the node 3 of the BBW only 12.5% of the communication bandwidth is occupied:

$$BO_p = (BO/Flen) * 100 \ = (30/240) * 100 \ = 12.5\%; \tag{8.4}$$

BO_p is the percentage of the bandwidth occupation within a cluster cycle, BO is the bandwidth occupation within a cluster cycle, i.e., the number of bytes occupied by the monitoring traffic within a cluster cycle, and $Flen$ is the maximum data field length that can be transmitted over a TTP/C communication channel. In the current implementation of TTP/C this length is 240 bytes.

From the Equation 8.3 we can determine that the bandwidth occupation within a cluster cycle is disproportional with the number of TDMA rounds within a cluster cycle. The higher the number of TDMA rounds within a cluster cycle, the smaller is the bandwidth occupation of the target system.

8.3.2 Task Abstraction Level

As presented in Section 5.3.2 in TTA systems the *S-task model* is used. Tasks in this task model can be considered as *black-boxes* that get their inputs and produce their outputs, because in this task model, tasks receive input messages upon invocation and produce output messages upon completion. Therefore, during monitoring of target systems at the *task abstraction level* (TAL) the inputs and outputs of tasks are gathered.

As depicted in Section 5.3.2, the input and output messages of each task are known in advance, and therefore, we can calculate the amount of monitoring data within a cluster cycle. In the BBW application used in this case study, the node 4 is monitored at the TAL. At this node there are eight application tasks[6]. The monitoring of these tasks is done by instrumentation code inserted within the tasks that are monitored. However, during monitoring at this abstraction level, a monitoring system that is not a prototype would call *monitoring routines* of the *resident monitors* before and after the execution of the task.

[6]In this case study we have monitored only the application tasks. However, the FTL and OS tasks can be monitored in the same way.

8.4 Collection, Processing and Presentation of Monitoring Data

The monitoring data are collected by the monitoring node (see Figure 8.1). These data are sent to TTPview. However, since TTPview does not provide processing and presentation of monitoring data collected at the node abstraction level, in this prototype (of this case study) this job is taken by a software developed in python.

8.4.1 Data Collection

The collection of monitoring data is triggered by the user commands that are sent via the MTTTY software. These commands are then sent to the active monitoring node (`node 1` in Figure 8.1). After they have been processed by the active monitoring node, these commands are sent to the target nodes. As depicted in Section 8.3 these commands are carried out by the additional messages added to `node 1`.

Target nodes[7] check if the *sender status* [TTT02] of the `n2_u4_mon_cmd` is set, which means that the monitoring node has sent a monitoring command. Among other things, these commands contain the `ID` of the target node and the abstraction level at which the target node must be monitored. At the TAL, for example, the `ID` of the task the user wants to monitor is sent with the monitoring command.

After the monitoring command has been received by the target node, it will be checked, and if the target node supports the intended abstraction level, then it starts sending the collected monitoring data at the beginning of the next cluster cycle. These data are carried out by either `n3_a_mon` or `n4_a_mon` depending from which node they are sent. In the implementation presented in this case study the target nodes stop[8] sending their monitoring data after a *predefined*[9] number of cluster cycles.

8.4.2 Data Processing and Presentation

In TTPview the RT triggers (see Chapter 6) are used for recording of only that part of time window, during which the sender status is set. After all these data

[7] In the BBW system only the nodes 3 and 4 are monitored at the node abstraction level.

[8] Monitoring data are always sent, however, their sender status is not set after *predefined* number of cluster cycles are elapsed (i.e., after this period they are marked as invalid).

[9] The *predefined* number of cluster cycles can also be given by the user using the MTTY software tool.

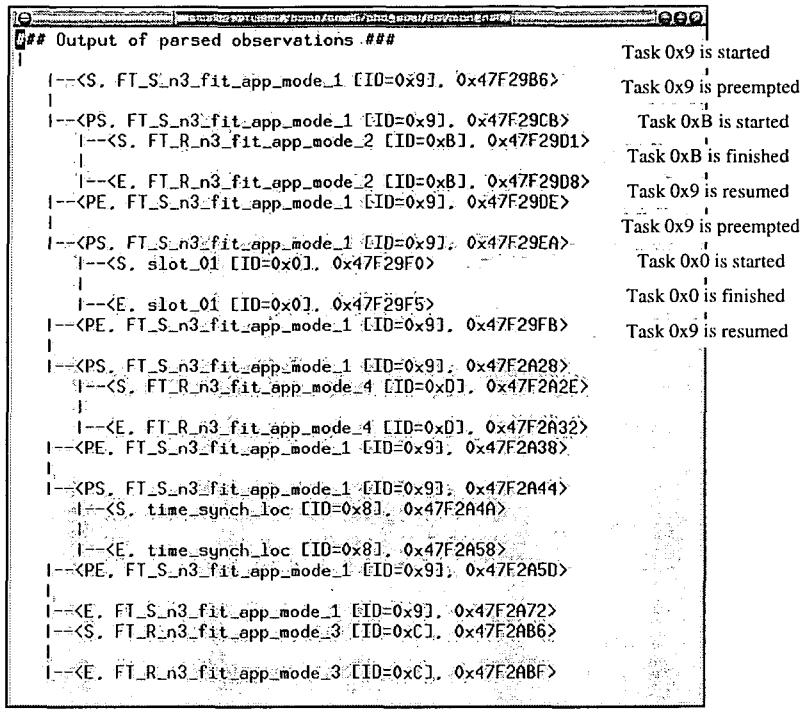


Figure 8.4: Monitoring Data collected at the OS Abstraction Level

have been recorded into the trace file, the export python script is used within TTPview for exporting of the collected monitoring data. The output file of this export process is a *comma-separated value* (CSV) file, that can be used for further processing. However, since the current implementation of TTPview does not support the processing and presentation of monitoring data at the node abstraction level, we have used TTPview only as *real-time recorder* for recording of these data to the disk. These data are processed by python scripts, which have been developed for processing and presentation of the monitoring data collected at the node abstraction level. These python scripts get the exported CSV file as input and after their processing the results are presented into "shell stdout". We have used the shell stdout to avoid the implementation effort needed for implementing of appropriate GUI widgets within TTPview.

OS Abstraction Level.

Dispatcher's Activity. As presented above, one of monitoring objectives at the OS abstraction level is the monitoring of the activity of the dispatcher. In Figure 8.4 we presented a part of the dispatcher activity of the node 3 within a cluster cycle. In this figure we see that the

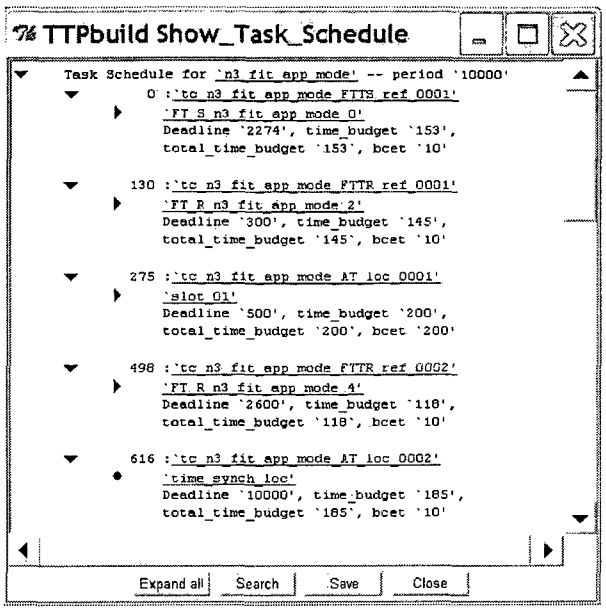


Figure 8.5: Scheduling information generated by TTPbuild

FT task `FT_S_n3_fit_app_mode_1` with `ID 9` was preempted more often than once by another tasks, i.e., by `FT_R_n3_fit_app_mode_2`, `slot_01`, `FT_R_n3_fit_app_mode_4` and `time_synch_loc`. These preemptions cannot be derived from the static schedule produced by TTPbuild's static scheduler (see Figure 8.5). Therefore, the user could find a clue from the visualization presented here, in case these preemptions leads the target system to a faulty behavior in the temporal domain.

```
### Output of calculated Task's Execution-Times (Algortithm ID :2) ###
Task :slot_01 [ID:0] : [bcet :28.8 (us), wcet :29.2 (us)] -- No of MET:74
Task :slot_02 [ID:1] : [bcet :10.8 (us), wcet :18.8 (us)] -- No of MET:74
Task :slot_03 [ID:2] : [bcet :180.8 (us), wcet :186.8 (us)] -- No of MET:74
Task :slot_04 [ID:3] : [bcet :12.4 (us), wcet :72.0 (us)] -- No of MET:74
Task :slot_05 [ID:4] : [bcet :18.8 (us), wcet :19.2 (us)] -- No of MET:74
Task :slot_06 [ID:5] : [bcet :143.2 (us), wcet :143.6 (us)] -- No of MET:74
```

MET - Measurements of Execution-Time.

Figure 8.6: Task Execution-Times

Measurements of Execution-Times. Another important information that can be derived from the observations collected during monitoring of the activity of the dispatcher is the logging of execution-times of the dispatched tasks.

Index	Vector
1	1, 2, 3, 4, 5, 6, 7, 8, 9
2	1, 2, 3, 4, 5, 6, 7, 9, 8
3	1, 2, 3, 4, 5, 9, 8, 7, 6
4	2, 1, 3, 4, 5, 9, 8, 7, 6
5	2, 1, 4, 3, 5, 9, 8, 7, 6
6	9, 8, 7, 6, 5, 4, 3, 2, 1
7	9, 8, 7, 6, 5, 4, 3, 1, 2
8	9, 8, 7, 6, 5, 3, 4, 1, 2
9	8, 9, 7, 6, 5, 3, 4, 1, 2
10	8, 9, 6, 7, 5, 3, 4, 1, 2

Table 8.1: Vector of Integers to be sorted

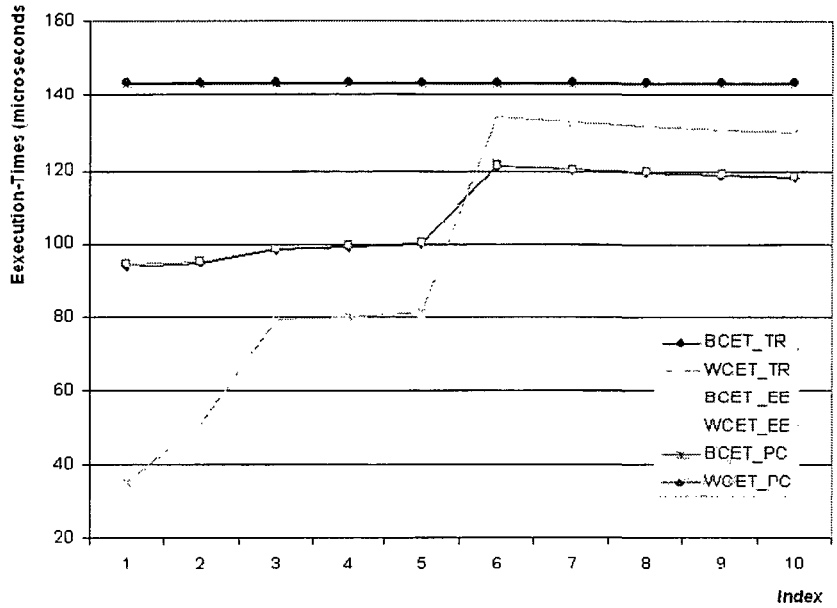
In Figure 8.6 the logged execution-times of the tasks are presented. However, since the time-stamping of the collected observations must be done by using of the global time, the resolution of the time-stamping is in the range of *macroticks* [TTT02], i.e., in the range of μ-seconds.

Bubble Sort: In this case study, we have conducted an experiment, in which the execution-times of different *implementations* of bubble sort algorithms were measured. Different vectors of integers were used as input data for the bubble sort algorithms (see Table 8.1). The following implementations of bubble sort algorithms are used:

- **Traditional -** is a *traditional* implementation of a bubble sort algorithm [Ast03],
- **Early-Exit -** is an implementation of a bubble sort algorithm that returns if there are no more elements to be swapped, and
- **Transformed Pure C** - is an implementation of a bubble sort algorithm that keeps the execution-time of the algorithm independent from the number of elements to be swapped [Pus02b].

During these measurments a local timer[10] with a tick of 0.4μsecs is used for time-stamping. Figure 8.7 presents the execution-times of the above presented bubble sort algorithms for sorting of the integer vectors presented in Table 8.1. The horizontal axis denotes the index on the Table 8.1, while the vertical axis denotes the *best-case* and *worst-case* execution-times in μsecs. For each index in the table the measurments are repeated a constant number of times.

[10]For implementation simplicity we have used the same local timer that also was used by the operating system.



BCET	-	Best-Case Execution-Time
WCET	-	Worst-Case Execution-Time
TR	-	Traditional Bubble Sort
EE	-	Early-Exit Bubble Sort
PC	-	Transformed-Pure C Bubble Sort

Figure 8.7: Execution-Times Measurements of different Bubble Sort Algorithms

From the results presented in Figure 8.7 we can see that the *transformed pure C* implementation of bubble sort algorithm has the most deterministic behavior regarding to execution-times.

Task Abstraction Level.

At the task abstraction level the collected monitoring data represent the inputs and outputs of particular tasks. The timestamps of observations contained by these monitoring data represent the points in time at which the particular tasks have been started and finished, respectively. If the particular node is also monitored at the OS abstraction level, then these timestamps are not gathered (for bandwidth and disk space reasons), because they can be derived from the observations that describe the activity of the OS dispatcher.

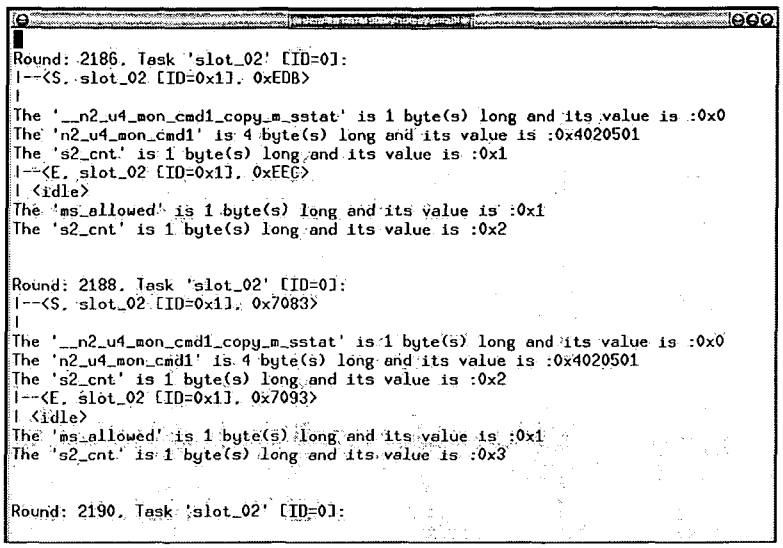


Figure 8.8: Monitoring Data collected at the Task Abstraction Level

Figure 8.8 presents the monitoring data collected from node 4 during monitoring at the task abstraction level. Since, the node 4 is only monitored at the task abstraction level, the collected monitoring data contain the timestamps, which represent the points in time at which the task `slot_02` with `ID 2` has been started and finished. The names of input and output variables and their values before the task has been started and after the task has been finished are also presented in Figure 8.8.

8.5 Chapter Summary

In this chapter we presented a monitoring system that has served as a case study for implementation of the concepts presented in the previous chapters. As a target application the brake-by-wire (BBW) was used, which is a distributed simulation program designed by Volvo Technological Development.

At the *OS abstraction level* the OS of a particular node was monitored. For this case study we have selected to implement only one of the monitoring objectives presented in Section 5.3.2. This is the monitoring of the dispatcher's activity. From the observed information that describe the dispatcher's activity we showed the way how to measure the *execution-times* of the running tasks of a particular node.

At the *application abstraction level* we chose to observe the input/outputs of the particular tasks, i.e., the *task abstraction level*. The monitoring system

provides monitoring of the target system at both abstraction levels in parallel, although for implementation simplicity in this case study we have not monitored the target system at multiple abstraction levels in parallel.

In this chapter we showed that during monitoring of TTA systems: i) the expected amount of monitoring data within a cluster cycle[11] can be calculated in advance, i.e., before the monitoring process has been started, ii) the bandwidth occupation within a cluster cycle is *disproportional* with the number of TDMA rounds within a cluster cycle (in the presented monitoring application the bandwidth occupation within a cluster cycle is less than 12.5%), iii) the code that implements these concepts uses the *pre-planned* resources (based on the knowledge about the amount of monitoring data) which guarantees the determinism of the monitoring system, iv) we can use real-time triggers[12] in conjunction with the monitoring at the node abstraction level to provide long-term diagnosing and correctness checking of target systems.

From the above presented remarks we conclude that the concepts presented in this thesis are implementable, and they are applicable also in systems that are much larger (i.e., complexer) than the used BBW application, i.e., typical industrial applications.

[11] During monitoring of TTA systems the observation interval is equal to the length of a cluster cycle. In these systems the cluster cycle is the smallest periodic cycle (i.e., repetition unit) and its duration is constant.

[12] In the presented monitoring application we used real-time triggers for searching of time points at which the user has started the monitoring of a particular node (see Sections 8.4.1 and 8.4.2).

Chapter 9

Conclusion

The contributions of this thesis are: i) the definition of different types of monitoring data and gathering methods that enable the prediction of monitoring resource requirements in advance[1], and to keep the interference of the monitoring system on the target system deterministic, ii) the design and development of a deterministic real-time monitoring system that can be used for monitoring of time-triggered systems at different abstraction levels, and iii) the design and implementation of the RTTS that records system operation of the target system only in time windows of interests, i.e., around the significant events.

9.1 Monitoring Data

We define monitoring data types that enable the prediction of the influence of monitoring systems on the target systems. This definition makes it possible to predict monitoring resource requirements in advance, i.e., during the monitoring setup phase before the monitoring process is started, and to keep the interference of the monitoring system on the target system deterministic. There are two criteria on which the definition and classification of monitoring data is based. The influence of monitoring systems on target systems depends directly on these two criteria: i) on the way how monitoring data are collected from the target system, and ii) on the amount and the rate of monitoring data being collected within an observation interval.

During monitoring of a target system at the user's intended abstraction level, the prediction of the expected amount of monitoring data within an observation interval is of utmost importance. The amount of the monitoring data that must be collected, determines the resource needs of the monitoring

[1]During the monitoring setup phase, before the monitoring process is started.

system on the target system. These resources are used for collecting, processing, and transmitting the collected monitoring data. They relate to: CPU, memory, communication bandwidth, etc. Based on this information the influence of the monitoring system on the target system can be quantified.

The practical applicability of these concepts is demonstrated in a monitoring application presented in Chapter 8, where they are used for calculation of the expected amount of monitoring data.

9.2 Deterministic Monitoring System

We designed and developed a deterministic real-time monitoring system that is used for monitoring of time-triggered systems. We started with the definition of the basic parts of such a system. Furthermore, we defined the requirements that must be fulfilled by target systems before they can be monitored by the presented monitoring system. In addition, we defined the abstraction levels at which time-triggered systems can be monitored. These abstraction levels are: i) cluster, ii) node, and iii) transducer abstraction level. For each abstraction level we defined monitoring data group(s) needed by the monitoring system for reconstructing the run-time behavior of the target system at these abstraction levels. Moreover, we showed that the monitoring system can either exactly calculate or estimate the amount of monitoring data within an observation interval during the monitoring setup phase, i.e., in advance before the monitoring process is started. This enables the system to predict its influence on the monitoring system and to support the system designer to allocate the resources needed by the monitoring system during the design phase of the target system. This resource allocation makes the influence of the monitoring system on the target system deterministic, i.e., the monitoring system's influence does not change the run-time behavior of the target system neither in time nor in the value domain. In addition, we defined the concepts for monitoring of time-triggered systems that consist of multiple clusters.

The presented concepts are implemented both in the software tool - TTPview (see Chapter 7), and in the monitoring application presented in Chapter 8. TTPview is successfully applied in industry (Honeywell, AUDI, VW, etc.) for monitoring of TTA systems.

9.3 Real-Time Trigger System

We designed and implemented the real-time trigger system (RTTS) that searches in real-time for significant events (i.e., events of interest) during mon-

itoring of a target system. The RTTS selectively records the system operation of the target real-time system, i.e., only in time windows of interest, around significant events. It does so by buffering of observations and looking for significant events in real-time (RT).

The RTTS can be successfully applied during long-term monitoring of real-time systems, because without selection the amount might become unmanageable. They also can be applied during automatic testing of a new developed target system, to automatically switch between monitoring at different abstraction levels of the tested target system. The RTTS can be used for both (long-term) on-line diagnosing and off-line analyzing of target systems. Another important usage of the RTTS is the run-time correctness checking of target systems. The significant events the RTTS has to look for in this context are the violations of the system invariants that must be always fulfilled by the target system. In the context of long-term on-line diagnosing the RTTS can store the selected monitoring data (i.e., around the significant events) on a dedicated non-volatile storage. In the automotive industry this application of the RTTS enables the usage of black-boxes similar to the black-boxes that are used in the aircraft industry. In case a car has a problem, the evaluation of the monitoring data stored in the black-box helps the user to find the reason of the suspected problem.

9.4 Outlook

The work presented in this thesis opens new directions for further research and development in the area of real-time monitoring of TTA systems. These directions can be classified into two different groups:

Development: The first step that needs to be done is the improvement of parts of TTPview (especially the GUI clients), which would facilitate the implementation of monitoring of target systems at the node and transducer abstraction level.

As a further step the design and implementation of gateway nodes has to take place, which facilitates the implementation of multiple cluster TTA systems. The implementation of gateway nodes enables the monitoring system the use of the cascade topology during monitoring of multiple cluster TTA systems.

Another interesting issue is the porting of the real-time trigger system (RTTS) from the central monitor, which is implemented on a COTS system (i.e., PC), to the monitoring node on which they would be executed

under embedded real-time Linux. In this way the real-time capability of the RTTS would be improved.

Further Research: An important research issue is the debugging of real-time TTA systems. The presented real-time monitoring system can be used to help the debugger during debugging of real-time TTA systems. The main idea is to develop a system, which re-executes parts of (or the whole) target application and redirects the inputs of the tasks that the user wants to debug to the trace files containing the run-time information collected by the presented monitoring system. An alternative approach is presented in [SA02]. In this approach the distributed break-points are used, which are achieved by the substitution of the physical time through a virtual time.

Another important issue is the integration of the concepts presented in Chapters 4 and 5 into the design process of the target systems. Currently, the selection of entities, the calculation of the expected amount within an observation interval and the instrumentation process are done manually. The intention is to incorporate the classification of the MD presented in this thesis into the design tools to calculate the needed resources (i.e., MD amount calculation) for monitoring process and to automate the instrumentation process. Examples of design tools are DECOMSYS::Designer[2] or TTP-Plan[3] used for designing of FlexRay or TTA systems. This approach could also be incorporated into design tools for other distributed RTS, e.g., CAN[4], LIN[5], etc.

[2]www.decomsys.com/flyer/DESIGNER.pdf
[3]www.ttagroup.org/ttp/pdf/TTTech-TTP-Plan-Flyer.pdf
[4]http://www.can.bosch.com
[5]http://www.lin-subbus.org

Bibliography

[AAC+94] T. Anderson, A. Avizienis, W.C. Carter, A. Costes, F. Cristian, Y. Koga, H. Kopetz, and et. al. *Dependability: Basic Concepts and Terminology.* International Federation for Information Processing, August 1994.

[Ade03] Astrit Ademaj. *Assessment of Error Detection Mechanisms of the Time-Triggered Architecture using Fault Injection.* PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2003.

[AKMM01] Tankut Akgul, Pramote Kuacharoen, Vincent J. Mooney, and Vijay K. Madisetti. A Debugger RTOS for Embedded Systems. In *Euromicro Conference, 2001. Proceedings*, pages 264–269, 2001.

[AS98] Ehab Salem Al-Shaer. *Hierarchical Filtering-Based Monitoring Architecture for Large-Scale Distributed Systems.* PhD thesis, Department of Computer Sciense, Old Dominion University, Norfolk, VA, December 1998.

[AS898] AS8202NF. *TTP/C-C2 Communication Controller.* TTTech Computertechnik AG, Schönbrunner Straße 7, A-1040 Vienna, 1998. Available at http://www.austriamicrosystems.com/04segments/automotive/as8202.htm.

[AS801] AS8201. *TTP/C-C1 Communication Controller.* TTTech Computertechnik AG, Schönbrunner Straße 7, A-1040 Vienna, 2001. Available at http://www.austriamicrosystems.com/04segments/automotive/as8201.htm.

[Ast03] Owen Astrachan. Bubble Sort: an Archaeological Algorithmic Analysis. In *Proceedings of the 34th SIGCSE Technical Sympo-*

sium on Computer Science Education, pages 1–5, 2003. Reno, Navada, USA.

[AVFJ02] J. Aidemark, J. Vinter, P. Folkesson, and J.Karlsson. Experimental Evaluation of Time-redundant Execution for a Brake-by-wire Application. In *International Conference on Dependable Systems and Networks, DSN 2002*, pages 210 –215, Washington DC, USA, June 2002.

[AW97] David Abramson and Greg Watson. Relative Debugging for Parallel Systems. In *Proceedings of PCW 97*, September 1997. Canberra, Australia.

[Bau00] Günther Bauer. *Transparent Fault-Tolerance in a Time-Triggered Architecture.* PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, April 2000.

[BJHL96] M. Brockmeyer, F. Jahanian, C. Heitmeyer, and B. Labaw. An Approach to Monitoring and Assertion-Checking of Real-Time Specifications. In *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems*, pages 236–243, 1996.

[BJRW94] J.S. Briggs, S.D. Jamicson, G.W. Rondall, and I.C. Wand. Debugging Distributed Ada Programs. Technical report, Real-Time and Distributed Systems Research Group, Department of Computer Sciense, University of York, June 1994.

[BJWL97] Monica Brockmeyer, Farnam Jahanian, Elly Winner, and Constance Heitmeyer Bruce Labaw. A Software Environment for Custom Simulation and Monitoring of Real-Time Specifications. In *Proceedings of High-Assurance Systems Engineering Workshop*, pages 78–84, 1997.

[Blo01] Johnnie Blom. Monitoring of Embedded Distributed Real-Time Systems. Technical report, Department of Computer Engineering, Mälardalen University, Västeras, Sweden, February 2001.

[Bor92] Cristopher B. Borchert. Organization and Management of Distributed Execution Event Histories for Real-Time Debugging. *Southeastcon. 92, IEEE Proceedings*, 1:343–345, 1992.

[BOSS95] T. Born, W. Obelöer, L. Schäfers, and C. Scheidler. The Monitoring Facilities of the Graphical Programming Environment TRAPPER. In *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, pages 555–562, January 1995.

[CJD91] Sarah E. Chodrow, Farnam Jahanian, and Marc Donner. Run-Time Monitoring of Real-Time Systems. In *Proceedings of Twelfth Real-Time Systems Symposium*, pages 74–83, 1991.

[CP98] J.P. Calvez and O. Pasquier. Performance Monitoring and Assessment of Embedded Hw/Sw Systems. *Design Automation for Embedded Systems*, 3:5–22, 1998.

[DMW98] Somnath Deb, Amit Mathur, and Peter K. Willet. Decentralized Real-Time Monitoring and Diagnosis. In *Proceedings of International Conference on Systems, Man, and Cybernetics*, October 1998. San Diego, CA.

[DR92] Paul S. Dodd and Chinya V. Ravishankar. Monitoring and Debugging Distributed Real-Time Programms. *Software-Practice and Experience*, 22(10):863–877, October 1992.

[DR02] Marcio S. Dias and Debra J. Richardson. The Role of Event Description in Architecting Dependable Systems. In *Workshop on Architecting Dependable Systems, ICSE 2002*, May 2002. Orlando, Florida.

[Eri97] Joakim Eriksson. Real-Time and Active Databases: A Survey. In *ARTDB-97, The 2nd International Workshop on Active, Real-Time and Temporal Database Systems, Advance Proceedings*, pages 195–216, September 1997.

[Fid96] Colin Fidge. Fundamentals of Distributed System Observation. *IEEE Software*, 13(6):77–83, November 1996.

[For90] Ray Ford. Monitoring Distributed Embedded Systems. In *Proceedings of the 1990 Symposium on Applied Computing*, pages 237–244, April 1990.

[Gai86] J. Gait. A Probe Effect in Concurrent Programs. *Software Practice and Experience*, 16(3):225–233, March 1986.

[Gal99] Thomas Galla. *Cluster Simulation in Time-Triggered Real-Time Systems.* PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 1999.

[GHS95] Richard Gerber, Seongsoo Hong, and Manas Saksena. Guaranteeing Real-Time Requirements With Resource-Based Calibration of Periodic Processes. *Transactions on Software Engineering*, 21(7):579–592, 1995.

[Gla00] Christian Glawan. Monitoring von Echtzeitbetriebssystemen. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2000.

[Gor91] Michael M. Gorlick. The Flight Recorder: An Architectural Aid for System Monitor. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 175–183, 1991.

[Gus02] Jan Gustafsson. A Prototype Tool for Flow Analysis of Object-Oriented Programs. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 91–100, April 2002. Washington, DC , USA.

[Har02] Timothy L. Harris. Dependable Software Needs Pervasive Debugging. In *Tenth ACM SIGOPS European Workshop*, September 2002. Saint-Emilion, France.

[HKM+94] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed Performance Monitoring: Methods, Tools, and Applications. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):585–598, 1994.

[Hof94] Yigal Hoffner. Monitoring in Distributed Systems. Technical report, ANSA, Poseidon House, Castle Park, Cambridge, CB3 0RD, United Kingdom, October 1994.

[Hof96] Richard Hofmann. Monitoring and Evaluation of Parallel and Distributed Systems, April 1996.

[HS90] Dieter Haban and Kang G. Shin. Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times. *IEEE Transactions on Software Engineering*, 16(12):1374–1389, December 1990.

[HW90] Dieter Haban and Dieter Wybranietz. A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems. *IEEE Transactions on Software Engineering*, 16(2):197–211, February 1990.

[Jah95] Farnam Jahanian. Run-Time Monitoring of Real-Time Systems. *In Advances in Real-Time Systems, Editor, Sang H. Son*, pages 429–454, 1995. Prentice Hall.

[JG90] Farnam Jahanian and Ambuj Goyal. A Formalism for Monitoring Real-Time Constraints at Run-Time. In *Digest of Papers., 20th International Symposium of Fault-Tolerant Computing, 1990. FTCS-20*, pages 148–155, 1990.

[JLSU87] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring Distributed Systems. *ACM Trans. Computer Systems*, 5(2):121–150, May 1987.

[JRR94] Farnam Jahanian, Ragunathan Rajkumar, and Sitaram C.V. Raju. Runtime Monitoring of Timing Constraints in Distributed Real-Time Systems. *Real-Time Systems*, 7(3):247–273, November 1994.

[KB01] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. Research Report 22/2001, Institut für Technische Informatik, Real-Time Systems Group, TU Wien, 2001.

[KBG98] Andreas Kirschbaum, Jürgen Becker, and Manfred Glesner. Run-Time Monitoring of Communication Activities in a Rapid Prototyping Environment. In *Proceedings of the 9th International Workshop on Rapid System Prototyping*, pages 52–57, 1998.

[KBP01] Hermann Kopetz, Günther Bauer, and Stefan Poledna. Tolerating Arbitrary Node Failures in the Time-Triggered Architecture. *SAE 2001 World Congress, March 2001, Detroit, MI, USA*, Mar. 2001.

[KHE00] Hermann Kopetz, Michael Holzmann, and Wilfried Elmenreich. A Universal Smart Transducer Interface: TTP/A. In *3rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2000)*, March 2000.

[KHE01] Hermann Kopetz, Michael Holzmann, and Wilfried Elmenreich. A Universal Smart Transducer Interface: TTP/A. *International Journal of Computer System, Science Engineering*, 16(2), March 2001.

[Kim95] Young-Kuk Kim. *Predictability and Consistency in Real-Time Transaction Processing.* PhD thesis, Department of Computer Science, University of Virginia, May 1995.

[Kla92] Rainer Klar. Event-Driven Monitoring of Parallel Systems. In *Workshop on Performance, Measurement and Visualization of Parallel Systems*, Moravany, Czecho-Slovakia, October 1992.

[KLH01] Hermann Kopetz, Thomas Losert, and Wolfgang Haidinger. Smart Transducers Interface. *Specification in response to the OMG's Smart Transducers Interface RFP (Document orbos/2000-12-13)*, January 2001.

[KLS+02] Moonjoo Kim, Insup Lee, Usa Sammapun, Jangwoo Shin, and Oleg Sokolsky. Monitoring, Checking, and Steering of Real-Time Systems. In *International Workshop on Run-time Verification*, July 2002. Copenhagen, Denmark.

[KO90] Michael J. Kaelbling and David M. Ogle. Minimizing Monitoring Costs: Choosing Between Tracing and Sampling. In *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences*, volume 1, pages 314–320, 1990.

[Kop92] Hermann Kopetz. Sparse Time versus Dense Time in Distributed Real-Time Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 460–467, June 1992. Yokohama, Japan.

[Kop97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, 1997. ISBN 0-7923-9894-7.

[Kop02] Hermann Kopetz. On the Specification of Linking Interfaces in Distributed Real-Time Systems. Research Report 8/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[Kra00] Dieter Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs.* PhD thesis, Johannes Kepler

University Linz, Department for Graphics and Parallel Processing, Altenbergerstraße 69, 4040 Linz, Austria, September 2000.

[Krü97] Andreas Krüger. *Interface Design for Time-Triggered Real-Time System Architectures.* PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, April 1997.

[KS98] Markus Kucera and Christoph Sikula. Application Monitoring in the Time-Triggered Architecture. In *Proceedings of Ninth European Workshop on Dependable Computing*, pages 137–143, May 1998. Gdansk, Poland.

[KSF98] Markus Kucera, Idriz Smaili, and Emmerich Fuchs. A Lightweight Ethernet Protocol to Connect a Time-Triggered Real-Time System to an INTERNET Server. In *European Multimedia, Microprocessor Systems and Electronic Commerce Conference and Exhibition*, September 1998. Bordeaux, France.

[KSS02] Suhee Kim, Sang H. Son, and John A. Stankovic. Performance Evaluation on a Real-Time Database. In *8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002)*, pages 253–265, September 2002. San Jose, California.

[Kuc98] Markus Kucera. *On the Cooperation between Time-Triggered Real-Time Systems and Event-Triggered Internet-Based Systems.* PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, December 1998.

[KV00] Dieter Kranzlmüller and Jens Volkert. Why Debugging of Parallel Programs needs Visualization. In *1-day Satellite Workshop on Visual Methods for Parallel and Distributed Programming held on 14th of September at the IEEE Symposium on Visual Languages*, September 2000. Seattle, Washington, USA.

[KVBA+99] Moonjoo Kim, Mahesh Viswanathan, Hanene Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally Specified Monitoring of Temporal Properties. In *Proceedings of the 11th European Conference on Real-Time Systems*, pages 114–122, June 9-11 1999. York, England, UK.

[LBAK+98] Insup Lee, H. Ben-Abdallah, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. A Monitoring and Checking Framework for Run-time Correctness Assurance. In *Korea-US Technical Conference on Strategic Technologies*, October 22-24 1998. Vienna, VA.

[LM97] Guangtian Liu and Aloysius K. Mok. An Event Service Framework for Distributed Real-Time Systems. In *IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, December 1997. http://citeseer.nj.nec.com/liu97event.html.

[LP85] C. H. Ledoux and D. Stott Parker. Saving Traces for Ada Debugging. In *Ada in Use (1985 International Ada Conference)*, pages 97–108, Cambridge, England, May 1985. Cambridge University Press.

[Lut92] Robyn R. Lutz. Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems. In *Proceedings of IEEE International Symposium on Requirements Engineering*, pages 126–133, January 1992.

[Mah01] Daniel Mahrenholz. Minimal Invasive Monitoring . In *Proceedings of Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 251–258, 2001.

[Mai02] Reinhard Maier. Event-Triggered Communication on Top of Time-Triggered Architecture. In *Proceedings DASC'02 - 21st Digital Avionics Systems Conference*, October 2002. Irvine, USA.

[Maj95] I. Majzik. Software Monitoring and Debugging Using Compressed Signature Sequences. In *Proceedings of the 22nd EUROMICRO Conference*, pages 311–318, 1995.

[May00] Markus Mayer. Design and Implementation of a File-System Based Monitoring Interface for the Time-Triggered Communications Protocol TTP/C under Linux. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2000.

[MH89] C. E. McDowell and D. P. Hembold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[ML97] Aloysius K. Mok and Gunagtian Liu. Efficient Run-Time Monitoring of Timing Constraints. In *Proceedings of Third IEEE Real-Time Technology and Applications Symposium*, pages 252–262, 1997.

[MN02] Monitoring-Node. *TTP Monitoring-Node - A TTP Development Board for the Time-Triggered Architecture based on the TTP Chip C2.* TTTech Computertechnik AG, Schönbrunner Straße 7, A-1040 Vienna, November 2002. Manual Edition: 2.1.00.

[Mot95a] Motorola. *MC68160 - Enhanced Ethernet Interface Transceiver - Technical Data.* Motorola Incorporation, 1995.

[Mot95b] Motorola. *MC68360 QUad Integrated Communications Controller - User's Manual.* Motorola Incorporation, 1995.

[Mot02] Motorola. *MPC855T User's Manual - Integrated Communications Microprocessor.* Motorola Incorporation, April 2002.

[MRW92] Allen D. Malony, Daniel A. Reed, and Harry A. G. Wijshoff. Performance Measurement Intrusion and Perturbation Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, July 1992.

[MS95] Masoud Mansouri-Samani. *Monitoring of Distributed Systems.* PhD thesis, Department of Computing, Imperial College of Sciense, Technology and Medicine, University of London, December 1995.

[MSS92] Masoud Mansouri-Samani and Morris Sloman. Monitoring Distributed Systems (A Survey). Technical Report DOC92/23, Imperial College of Science Technology and Medicine, Department of Computing, 180 Queen's Gate, London SW7 2BZ, September 1992.

[MSS93] Masoud Mansouri-Samani and Morris Sloman. Monitoring Distributed Systems. *IEEE Network*, 7(6):20–30, November 1993.

[MSSP02] Daniel Mahrenholz, Olaf Spinczyk, and Wolfgang Schröoder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 249–256, April 2002. Washington, DC , USA.

[MW94] Frank Mueller and David Whalley. On Debugging Real-Time Applications. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[Mye76] G. J. Myers. *Software Reliability: Principles and Practices.* Published by John Wiley & Sons, USA, 1976.

[NGM98] E. Nett, M. Gergeleit, and M. Mock. An Adaptive Approach to Object-Oriented Real-Time Computing. In *1st IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC '98)*, April 1998. Kyoto, Japan.

[Nos97] Romman Nossal. *An Interface-Focused Methodology for the Development of Time-Triggered Real-Time Systems Considering Organizational Constraints.* PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 1997.

[OMG01] *Smart Transducers Interface.* OMG TC Document orbos/2001-06-03, July 2001. Supported by Technische Universität Wien. Available at http://www.omg.org.

[OPEL01] Roman Obermaisser, Philipp Peti, Wilfried Elmenreich, and Thomas Losert. Monitoring and Configuration in a Smart Transducer Network. In *IEEE Workshop on Real-Time Embedded Systems, 3rd December 2001, London, United Kingdom*, December 2001.

[Ose01] OSEK/VDX Fault-Tolerant Communication Specification 1.0. OSEK/VDX Steering Committee, http://www.osek-vdx.org, July 2001.

[OSS93] D.M. Ogle, K. Schwan, and R. Snodgrass. Application-Dependent Dynamic Monitoring of Distributed and Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762–778, 1993.

[Pet97] Dennis K. Peters. Deriving Real-time Monitors from System Requirements Documentation. In *Proceedings of Third IEEE International Symposium on Requirements Engineering (RE '97) Doctoral Consortium*, pages 89–92, January 1997.

[Pla84] Bernhard Plattner. Real-Time Execution Monitoring. *IEEE Transactions on Software Engineering*, SE-10(6):756–754, November 1984.

[Pol96] Stefan Poledna. Optimizing interprocess communication for embedded real-time systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, Jan. 1996.

[PP00] Dennis K. Peters and David L. Parnas. Requirements-Based Monitors for Real-Time Systems. In *Proceedings of the 2000 International Symposium on Software Testing and Analysis (ISSTA)*, pages 77–85, August 2000.

[PP02] Dennis K. Peters and David L. Parnas. Requirements-Based Monitors for Real-Time Systems. *IEEE Transactions on Software Engineering*, 28(2):146–158, February 2002.

[Pus02a] Peter Puschner. Is worst-case execution-time analysis a non-problem? – towards new software and hardware architectures. In *Proc. 2nd Euromicro International Workshop on WCET Analysis*, Technical Report, York YO10 5DD, United Kingdom, Jun. 2002. Department of Computer Science, University of York.

[Pus02b] Peter Puschner. Transforming execution-time boundable code into temporally predictable code. In Bernd Kleinjohann, K.H. (Kane) Kim, Lisa Kleinjohann, and Achim Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Kluwer Academic Publishers, 2002. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).

[Ram93] Krithi Ramamritham. Time for Real-Time Temporal Databases? In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, June 1993.

[Ram95] Krithi Ramamritham. The Origin of TCs. In *Proceedings of the First International Workshop on Active and Real-Time Database Systems*, pages 50–62, Skövde, June 1995.

[Ram96] Krithi Ramamritham. Where do Time Constraints Come From and Where do They Go? *International Journal of Database Management (invited paper)*, 7(2):4–10, Spring 1996.

[RRJ92] Sitaram C.V. Raju, Ragunathan Rajkumar, and Farnam Jahanian. Monitoring Timing Constrains in Distributed Real-Time Systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 57–67, 1992.

[SA02] Idriz Smaili and Astrit Ademaj. Setting Break-Points in the Distributed Time-Triggered Architecture. In *IEEE International High Level Design Validation and Test Workshop (HLDVT'02)*, pages 57–62, October 2002. Cannes, France.

[Sch94a] Ulrich Schmid. Monitoring Distributed Real-Time Systems. *Real-Time Systems Journal*, 7(1):33–56, 1994.

[Sch94b] Werner Schütz. Fundamental Issues in Testing Distributed Real-Time Systems. *Real-Time Systems Journal*, 7(2):129–157, 1994.

[Sch95] Beth A. Schroeder. On-Line Monitoring: A Tutorial. *IEEE Computer*, 28(6):72–78, June 1995.

[Sev87] R.E. Seviora. Knowledge-Based Program Debugging Systems. *IEEE Transactions on Software Engineering*, 4(3):20–32, May 1987.

[SG94] M. Spezialetti and R. Gupta. Perturbation Analysis: A Static Analysis Approach for the Non-Intrusive Monitoring of Parallel Programs. In *International Conference on Parallel Processing*, volume II, pages 81–88, St. Charles, Illinois, August 1994.

[SG97] Darlene A. Stewart and W. Morven Gentlman. Non-Stop Monitoring and Debugging on Shared-Memory Multiprocessors. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 263–269, May 1997. Boston, MA.

[Shi96] Jun Shih. Debugging Concurrent Programs. Master's thesis, Waterloo, Ontaria, Canada, University of Waterloo, 1996.

[Sik98] Christoph Sikula. A Monitoring System for Real-Time Applications in the Time-Triggered Architecture. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, February 1998.

[SL01] Mohammed El Shobaki and Lennart Lindh. A Hardware and Software Monitor for High-Level System-on-Chip Verification. In *International Symposium on Quality Electronic Design*, pages 56–61, March 2001. San Jose, CA, USA.

[Sma02] Idriz Smaili. A Real-Time Monitoring System for the Time-Triggered Architecture. In *8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002), Poster Addendum to the Proceedings*, pages 9–16, September 2002. San Jose, CA, USA.

[Sma04] Idriz Smaili. Using Triggers to Find Significant Events during Monitoring of Real-Time Systems. In *Workshop on Intelligent Solutions in Embedded Systems - WISES 2004*, June 2004. Graz, Austria.

[Son95] Sang H. Son. *Predictability and Consistency in Real-Time Database Systems.* Advances in Real-Time Systems, S. H. Son (ed.), Prentice Hall, 1995.

[SP04] Idriz Smaili and Peter Puschner. Monitoring Data Types in Distributed Real-Time Systems. In *IEEE International Conference on Computational Cybernetics - ICCC 2004*, August 30 - September 1 2004. Vienna, Austria.

[SS97] T. Savor and R.E. Seviora. An Approach to Automatic Detection of Software Failures in Real-Time Systems. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, pages 136–146, 1997.

[Sto01] Georg Stoeger. The TTA Network - a Safe yet Flexible Communication Architecture for Vehicle Electronics. *SAE - Society of Automotive Engineers, Inc*, March 2001. Detroit, USA.

[TB96] Jeffry J.P. Tsai and Yao-Dong Bi. Debugging for Timing-Constraint Violations. *IEEE Software*, 13(2):89–99, March 1996.

[TBYS96] Jeffrey J.P. Tsai, Yaodong Bi, Steve J.H. Yang, and Ross A.W. Smith. *Distributed Real-Time Systems - Monitoring, Visualization, Debugging, and Analysis.* John Wiley and Sons, Inc., USA, 1996. ISBN 0-471-16007-5.

[TCO91] Kuo-Chung Tai, Richard H. Carver, and Evely E. Obaid. Debugging Concurrent Ada Programs by Deterministic Execution. *IEEE Transactions on Software Engineering*, 17(1):45–63, 1991.

[TFB90] Jeffry J.P. Tsai, Kang-Ya Fang, and Yao-Dong Bi. On Real-Time Software Testing and Debugging. In *Proceedings of Fourteenth Annual International Computer Software and Applications Conference*, pages 512–518, 1990.

[TFC90] Jeffry J.P. Tsai, Kang-Ya Fang, and Horng-Yuan Chen. A Noninvasive Architecture to Monitor Real-Time Distributed Systems. *Computer*, 23(3):11–23, March 1990.

[TFCB90] J.P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging. *IEEE Transaction on Software Engineering*, 16:897–916, 1990.

[TH00] Henrik Thane and Hans Hansson. Using Deterministic Replay for Debugging of Distributed Real-Time Systems. In *Proceedings of the 12th European Conference on Real-Time Systems*, pages 265–272, 2000. Royal Institute of Technology, Sweden.

[Tha00a] Henrik Thane. Design for Deterministic Monitoring of Distributed Real-Time Systems. Research Report 5/9/00, Mälardalen Real-Time Research Center, Stokholm, 2000.

[Tha00b] Henrik Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems.* PhD thesis, Royal Institute of Technology, Stockholm, May 2000.

[TKM88] Hideyuki Tokuda, Makoto Kotera, and Clifford W. Mercer. A Real-Time Monitor for a Distributed Real-Time Operating System. In *Proceedings of ACM Workshop on Parallel and Distributed Debugging*, pages 66–77, 1988.

[TTP01] TTPnode. *TTPnode - A TTP Development Board for the Time-Triggered Architecture.* TTTech Computertechnik AG, Schönbrunner Straße 7, A-1040 Vienna, June 2001. Manual Edition: 1.0.11.

[TTP02a] TTPbuild. *TTPbuild - The Node Design Tool for the Time-Triggered Protocol TTP/C.* TTTech Computertechnik AG, Schönbrunner Straße 7, A-1040 Vienna, July 2002. Manual Edition: 3.2.

[TTP02b] TTPplan. *TTPplan - The Cluster Design Tool for the Time-Triggered Protocol TTP/C.* TTTech Computertechnik AG,

Schönbrunner Straße 7, A-1040 Vienna, August 2002. Manual Edition: 3.5.

[TTP02c] TTPview. *TTPview - The TTP Real-time Monitoring Tool.* TTTech Computertechnik AG, Schönbrunner Straße 7, A-1040 Vienna, December 2002.

[TTT98] TTTech Computertechnik AG. http://www.tttech.com, 1998.

[TTT99] TTTech. *TTP/C Protocol Specification– Version 0.1.* TTTech Computertechnik AG, Schönbrunner Straße 7, A-1040 Vienna, April 1999. Available at http://www.ttpforum.org.

[TTT02] TTTech. *TTP/C Protocol Specification – Version 1.0.* TTTech Computertechnik AG, Schönbrunner Straße 7, A-1040 Vienna, June 2002. Available at http://www.ttpforum.org.

[UB02] David Urting and Yolande Berbers. Runtime Verification of Timing Constraints. Research Report CW 345, Department of Computer Science, K.U. Leuven, België, July 2002.

[vdSvdWA+97] P.D.V. van der Stok, J. van der Wal, A.T.M. Aerts, S.A.E Sassen, and M.P. Bodlaender. Performance Modeling of Real-Time Database Schedulers. *Real-Time Database Systems, Issues and Applications*, pages 251–275, 1997. Kluwer Academic Publishers.

[VW02] Jeffry S. Vetter and Patrick H. Worley. Asserting Performance Expectations. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing*, November 16-22 2002. Baltimore, MD.

[Wat00] Gregory R. Watson. *The Design and Implementation of a Parallel Relative Debugger.* PhD thesis, School of Computer Science and Software Engineering, Monash University, Wellington Road, Clayton, VIC 3168, Australia, October 2000.

[WLS99] Horst F. Wedde, Jon A. Lind, and Guido Segbert. Distributed Real-Time Task Monitoring in the Safety-Critical System Melody. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pages 158–165, June 9-11 1999. York, England, UK.

[WSG96] Wanqing Wu, Madalene Spezialetti, and Rajiv Gupta. Guaranteed Intrusion Removal from Monitored Distributed Applications. In *Eighth IEEE Symposium on Parallel and Distributed Processing*, pages 422–425, 1996. Pasadena, California.

[WSG98] Wanqing Wu, Madalene Spezialetti, and Rajiv Gupta. A Protocol for Removing Communication Intrusion in Monitored Distributed Systems. In *International Conference on Distributed Computing Systems*, pages 120–129, 1998.

Glossary

Note: All terms that are defined in this glossary are put in *italics*. At the end of each entry the section of the thesis that introduces or discusses the term is mentioned in the parenthesis.

Amount of Monitoring Data: The sum of the sizes of observation history of each entity within an observation interval is called the **amount** of monitoring data within the observation interval(4.2.1).

Aperiodic Monitoring Data: We call non-regular MD *aperiodic*, if they contain observations over entities, which are observed non-regularly, and no *minimum time interval* between successive observations exists (4.3.1).

Bandwidth Occupation: The bandwidth of the shared transmission medium that is actually used for transmitting of monitoring data collected within an observation interval is called *bandwidth occupation*(5.3.2).

Cascade Interconnection Topology: We say that the interconnection topology of multiple clusters is of type **cascade**, if the monitoring system is directly connected to only one cluster and to others via (multiple) gateways (5.4.1).

Data Deadline: The time point after which real-time data lose their validity is called *data deadline* (2.1.2).

Debugging: *Debugging* is defined by the ANSI/IEEE glossary as "the process of locating, analyzing and correcting suspected faults", where a fault is defined as an accidental condition that causes a program to fail to perform its required function [BJRW94] (2.3).

Event Record: Whenever the monitor device recognizes an event, it stores a data record (a so called *event record*), which contains the information *what* happened *when* and *where* [HKM+94] (2.2.1).

Event Trace: The sequence of events is stored as an *event trace* [HKM+94](2.2.1).

Event: An *event* is defined by Kopetz in [Kop97] as an occurrence (a state change) that happens at a cut of a directed time line that extends from the past into the future, and from which the flow of real-time can be modelled (2.2.1).

Global Monitoring Data: We call the node MD *global*, if they contain observations over entities that are *globally* visible within a particular node (4.3.2).

Local Monitoring Data: Node monitoring data, which contain observations over entities that are not *globally* visible within a particular node are called *local monitoring data* (4.3.2).

Monitored Application Data: We call MD *monitored application* data if they contain observations over entities that contribute to achieving the computational goal of the target application (4.3.2).

Monitoring at Cluster Level: We define the monitoring of target systems at the *cluster abstraction level* as a monitoring process during which the monitoring system observes entities whose observations are visible within the *cluster scope*, i.e., these observations are exchanged among the interconnected nodes within a given cluster via the shared transmission medium (5.3.1).

Monitoring at Function Level: We define the monitoring of target systems at the *function abstraction level* as a monitoring process during which observations over entities are collected that are not visible outside of functions or modules of the local part of the target application (5.3.2).

Monitoring at Node Level: We define the monitoring of target systems at the **node abstraction level** as a monitoring process during which the monitoring system observes entities whose observations are visible only within the **node scope**, i.e., these observations are not exchanged between the interconnected nodes within a given cluster (5.3.2).

Monitoring at OS Level: We define the monitoring at the OS abstraction level as a monitoring process during which the monitoring system observes entities that cannot be seen outside of the OS, and the run-time behavior of the OS of a particular node must be represented from these observations (5.3.2).

Monitoring at Task Level: We define the monitoring of target systems at the *task abstraction level* as a monitoring process during which observations over input and output messages of each observed task are collected (5.3.2).

Monitoring Capacity: The (rest of the) bandwidth of the shared transmission medium that is not utilized by the target application during an observation interval is called *monitoring capacity* (5.3.2).

Monitoring Data: The set of observation histories that contain observations of an observation interval and which represent the run-time behavior of the monitored target system at the intended abstraction level are called *monitoring data*(4.2.1).

Monitoring Route: Let **A** and **B** be two different points within the target system. We call the connection between these two points the *monitoring route*, if the connection between these two points contains at least one cluster. A monitoring route that contains only one cluster is called *simple*, while a monitoring route that contains more than one cluster and gateway nodes is called *complex* (see Figure 5.10) monitoring route (5.4.1).

Monitoring System: The *monitoring system* is a system that is used for monitoring of *monitored* systems (i.e., target systems) [TBYS96] (2.2.1).

Network Monitoring Data: *Network MD* contain observations over entities, the observations of which are used by tasks running on different nodes, i.e., they are exchanged between interconnected nodes via the shared transmission medium (4.3.2).

Non-Regular Monitoring Data: We call MD *non-regular*, if there is no regular pattern of observations made over the respective entities (4.3.1).

Observation History: The sequence of timely ordered observations that are made over a particular (single) entity within an observation interval is called *observation history* (4.2).

Observation Interval: The *observation interval* is the periodic time interval, which in fact is the smallest interval of the monitoring duration. The the start point and the duration of the observation interval are the same as the start point and the duration of the ARU (see Section 3.4) of the target system (4.2).

Parallel Interconnection Topology: We define the type of the interconnection topology of multiple clusters as **parallel**, if the monitoring system is

directly connected to each of the interconnected clusters to be monitored (5.4.1).

Probe Effect: *Probe effect* is the effect caused by the interference that occurs when a program's execution is monitored [TCO91] (2.2.3).

Probes: *Probes* are code fragments residing within the *resident monitor* (rather than application) [OSS93](2.2.1).

Propagation Delay: The *propagation delay* of a monitoring route is the duration between the point in time when the monitoring system starts a monitoring action on the remote cluster, i.e., it sends a command to the remote cluster, and the point in time when the monitoring system receives the collected monitoring data from the remote cluster through the monitoring route (5.4.1).

Pure Monitoring Data: We call MD *pure monitoring data* if they contain observations over entities that do not provide any contribution for achieving the computational goal of the target application (4.3.2).

Real-Time Data: The type of data that lose their validity with the passage of real-time are called *real-time data*(2.1.2).

Recording Interval: The time interval during which the collected monitoring data are recorded by the trigger's recording actions is called *recording interval* or *logging window* (6.1).

Regular Monitoring Data: We call MD *regular*, if there is a regular pattern of observations made over the respective entities (4.3.1).

RTS Entity: Analog to the RT entity, we introduce a new notion called *real-time system (RTS) entity*, which is a significant "state variable"of the target real-time computer system and which is relevant only for the monitoring system (4.2).

Sensor: A *sensor* is defined in [Sch95] by Schroeder as an entity that observes the behavior of a small part of the application system state space (2.2.1).

Significant Event: A *significant event* is defined as an event of interest, which has to be defined by the user of the monitoring system. Examples of such events are: the *temperature* of a controlled physical system or the *velocity* of a (controlled) car exceed their allowed limits (6.1).

Significant Time Point: The time point, at which a significant event is found, is called the *significant time point* (6.1).

Sporadic Monitoring Data: We call non-regular MD *sporadic*, if they contain observations over entities, which are not observed regularly, but a *minimum time interval* between successive observations exists and is known (4.3.1).

Target Application: The application that is executed on the target system is called *target application* (2.2.1).

Target System: The system that is monitored by the monitoring system is called *target system* [Pla84, PP00] (2.2.1).

List of Abbreviations

AC	-	Absolute Consistency
ARU	-	Atomic Repetitive Unit
BG	-	Bus Guardian
CDB	-	Cluster Design Database
CI	-	Control Interface
COI	-	Controlled Object Interface
COTS	-	Commercial Off-The-Shelf
CM	-	Central Monitor
CNI	-	Communication Network Interface
C-state	-	Local view of the cluster state
DCI	-	Data Sharing Interface
EBNF	-	Extended Backus-Naur Form
ECA	-	Event-Condition-Action
FIFO	-	First-In First-Out
FTcom	-	Fault-Tolerant Communication Layer
FTL	-	Fault-Tolerance Layer
FT CNI	-	Fault-Tolerance CNI
IFG	-	Interframe Gap
IFS	-	Interface File System
ISR	-	Interrupt Service Routine
IT	-	Information Technology
μT	-	Microtick
MCH	-	Monitoring CNI History
MEDL	-	Message Description List
MGA	-	Message Area
MN	-	Monitoring Node
MT	-	Macrotick
NBW	-	Non-Blocking Write Protocol
NDB	-	Node Design Database
OI	-	Observation Interval
OS	-	Operating System
RC	-	Relative Consistency
ReAM	-	Recording Action Manager
RM	-	Resident Monitor
RPA	-	Read Partition Algorithm
RTL	-	Real-Time Logic
RTOS	-	Real-Time Operating System
RTMS	-	Real-Time Monitoring System

RTS	-	Real-Time System
RTTS	-	Real-Time Trigger System
SCA	-	Status/Control Area
SRU	-	Smallest Replaceable Unit
TDL	-	Trigger Definition Language
TDMA	-	Time-Division Multiple Access
TTA	-	Time-Triggered Architecture
TTP	-	Time-Triggered Protocol
TTP/C	-	Time-Triggered Protocol for SEA Class C of applications
TTP/C-C1	-	A prototype version of a TTP/C controller
TTP/C-C2	-	An operational version of a TTP/C controller
TTOS	-	Time-Triggered Operating System
TTPbuild	-	Node Design Software Tool from TTTech AG
TTPplan	-	Cluster Design Software Tool from TTTech AG
TTPview	-	Bus Monitoring Software Tool from TTTech AG
UD	-	User Defined
WCA	-	Worst-Case Amount
WCET	-	Worst-Case Execution Time

List of Publications

1. H. Kopetz, M. Kucera, D. Millinger, C. Ebner, and **I. Smaili**. "Interfacing Time-Triggered Embedded Systems to the INTERNET". *Proceedings of the International Symposium on Internet Technology.* (Taipei, Taiwan. Apr. 1998):Pp.180-186.

2. Markus Kucera, **Idriz Smaili**, and Emmerich Fuchs. "A Lightweight Ethernet Protocol to Connect a Time-Triggered Real-Time System to an INTERNET Server". *European Multimedia, Microprocessor Systems and ElectronicCommerce Conference and Exhibition.* (Bordeaux, France, September 1998)

3. Stefan Poledna, Harald Angelow, Martin Glueck, Manfred Pisecky, **Idriz Smaili**, Georg Stoeger, Christian Tanzer, Georg Kroiss, TTTech. "TTP Two Level Design Approach: Tool Support for Composable Fault-Tolerant Real-Time Systems". *SAE International Congress and Exhibition.* (Detroit, USA, 2000).

4. **Idriz Smaili**. "A Real-Time Monitoring System for the Time-Triggered Architecture". *The 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002)*, September 24-27, pages: 9-16, San Jose, California, 2002.

5. **Idriz Smaili**, Astrit Ademaj. "Setting Break-Points in Distributed Time-Triggered Architecture". *In Proceedings of the 7th Annual IEEE International Workshop on High Level Design Validation and Test*, Cannes, France, October, 2002.

6. **Idriz Smaili**. "Using Triggers to Find Significant Events during Monitoring of Real-Time Systems". *International Workshop on Intelligent Solutions in Embedded Systems - WISES 2004*, Juny 25, Graz, Austria, 2004.

7. **Idriz Smaili** and Peter Puschner. "Monitoring Data Types in Distributed Real-Time Systems". *IEEE International Conference on Computational Cybernetics - ICCC 2004*, August 30 - September 1, Vienna, Austria, 2004.

Curriculum Vitae

Idriz Smaili

October 28^{th} 1971	Born in Ballaban, Prishtinë, Kosova
September 1978 – June 1986	Primary School in Ballaban, Prishtinë, Kosova
September 1986 – June 1990	Secondary School in Prishtinë, Kosova
October 1990 – May 1995	Studies in the Faculty of Electrical Engineering Section of Informatics and Telecommunications University of Prishtina, Prishtinë, Kosova
May 1995	Graduation in the Faculty of Electrical Engineering with distinction
May 1995 – April 1996	Software Engineer Computer System Technologies (CST), Prishtinë, Kosova
Oktober 1995 – April 1996	Teaching Assistant (Computer Languages) University of Prishtina, Prishtinë, Kosova
April 1996 – March 1997	German Language Learning in Vienna, Austria
March 1997 – October 1997	Entrance Examination Real-Time Systems Group, Vienna University of Technology
Oktober 1997 – August 1998	PhD Studies (Real-Time Databases) Real-Time Systems Group, Vienna University of Technology
June 8^{th} 1998 – December 1^{st} 2003	Software Engineer (TTPview Project Manager) TTTech Computertechnik AG (www.tttech.com)
since August 1998	PhD Studies (Real-Time Monitoring) Real-Time Systems Group, Vienna University of Technology