Lehrstuhl für Realzeit-Computersysteme

Scheduling with Message Deadlines for Hard Real–Time SDL Systems

Thomas Kolloch

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr.-Ing. K. Antreich
Prüfer der Dissertation:	1. Univ.-Prof. Dr.-Ing. G. Färber
	2. Univ.-Prof. Dr.-Ing. J. Eberspächer

Die Dissertation wurde am 17.01.2002 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 18.06.2002 angenommen.

Summary

Hard real–time system design requires both, a methodology that allows to control the increasing functional complexity and additionally an a–priori proof that all timing requirements will be met even in worst case situations. This work introduces a design heuristics based on the formal "Specification and Description Language" SDL aiming on the integration of a timeliness verification into an automated design process.

SDL's semantics which is especially suited to event–driven systems relies on non–blocking signal exchange, i.e. a receiving SDL process will be activated on signal arrival. To enforce a deterministic behaviour of a SDL system, the vague semantics of SDL has to be pinpointed to a predictable model of computation. For this, Message based Earliest Deadline First scheduling (MEDF) will be suggested as basis for SDL's processing sequence.

With MEDF, a task inherits the deadline transported by an incoming message, i.e. its dynamic priority is adapted accordingly. Finally, the task bequeathes this deadline to its outgoing messages. For predictability reasons, incoming messages carrying a new deadline now have to be inserted into a task's only queue in a deadline sorted order. Deadline inheritance (DIP), respectively deadline ceiling (DCP) applied to message queues raise the dynamic priority of the receiving task and assure the avoidance of priority inversion effects caused by server tasks responding to multiple requests with unequal urgencies. Since MEDF scheduling implicates an earliest deadline first processing sequence for all tasks, Gresser's schedulability analysis methodology for event–driven real–time systems [Gre93a] can be applied to prove the timeliness of a MEDF system.

With these presumptions, a real–time system's implementation and its worst case timing behaviour can be automatically derived on the basis of its SDL system specification by one strike, if the system model has been extended by timing constraint annotations (deadlines, execution times, timing of system stimuli). Code generation that preserves MEDF processing sequence on the one hand, and mapping of the system specification to a task precedence graph (TPG) mirroring the structural inter–relationships between all resource–sharing components on the other hand, assures conformance between specification, implementation, and analysis model. The latter incorporates all necessary timing properties to perform the final schedulability analysis step and serves as information source to reveal hidden unpredictability and optimization potential in SDL system models.

The MEDF scheduler has been integrated into the real–time operating system RTEMS. An additional test suit validates the correct functionality of the scheduler and demonstrates its timing behaviour. Tool support for the proposed design methodology has been prototypically realized and its applicability has been evaluated with design examples.

Contents

Summary iii

Contents v

List of Acronyms and Variables ix

List of Figures xi

List of Tables xv

1 Introduction 1
- 1.1 Contribution . . . 2
- 1.2 Prerequisites and Restrictions . . . 2
- 1.3 This work's background: Rapid Prototyping . . . 5
- 1.4 Synopsis . . . 7

2 Related Work 9
- 2.1 Real–Time Constraint Specification . . . 9
- 2.2 Code Generation Efficiency . . . 11
- 2.3 Schedulability Analysis . . . 12
- 2.4 Real–Time System Design and Schedulability . . . 14

3 SDL Semantics and Timing Constraints 17
- 3.1 Specification and Description Language SDL . . . 18
  - 3.1.1 Syntax . . . 19
  - 3.1.2 Semantics . . . 20
- 3.2 Abstractions and Non–Determinisms . . . 21
  - 3.2.1 Notion of Time . . . 21
  - 3.2.2 SDL Timers . . . 22
  - 3.2.3 SDL Process Activation . . . 22
  - 3.2.4 Atomicity and Run–To–Completion Semantics . . . 23
- 3.3 Finalization/Restriction of SDL´s Semantics . . . 23
  - 3.3.1 MEDF Semantics . . . 23
  - 3.3.2 Scope of Deadlines . . . 25

3.3.3 MEDF Process Sequencing . . . 25
3.3.4 MEDF Timers . . . 26
3.3.5 Discussion of further SDL Language Constructs . . . 27
3.4 Non–functional Specification . . . 28
3.4.1 Timing Constraints . . . 28
3.4.2 Extensions to the Functional Model . . . 29

4 Code Generation 31
4.1 Decomposition Principles . . . 32
4.1.1 Decomposition by Maximizing Parallelism . . . 32
4.1.2 Decomposition by Timing Constraints . . . 33
4.1.3 Minimizing Interprocess Communication . . . 33
4.2 State–of–the–Art Code Generators . . . 35
4.2.1 SDT's C–Basic and C–Advanced . . . 35
4.2.2 SDT's C–Micro . . . 37
4.3 Code Generation for MEDF Semantics . . . 37
4.3.1 Server Model . . . 38
4.3.2 Activity Thread Model . . . 39
4.3.3 System Time and MEDF Timers . . . 41
4.3.4 System Environment and Software Architecture . . . 42

5 Scheduling with Message Deadlines 45
5.1 MEDF Scheduling Scheme . . . 45
5.2 Predictability of MEDF . . . 46
5.2.1 Task Precedence Constraints . . . 46
5.2.2 Sequencing . . . 50
5.2.3 Message Blocking on Server Tasks . . . 51
5.2.4 Mutual Exclusion on Semaphores . . . 56
5.3 Runtime System Support . . . 56
5.3.1 The EDF Event Object . . . 57
5.3.2 The MEDF Message Queue . . . 58
5.3.3 Priority Inversion Avoidance . . . 59
5.4 Complexity and Execution Times . . . 63
5.5 Optimized Server Model Scheduler . . . 69

6 SDL Real–Time Analysis 75
6.1 Real–Time Analysis . . . 77
6.1.1 Task Model . . . 77
6.1.2 Analysis Algorithm . . . 78
6.1.3 Influence of System Environment and Timer Task . . . 80
6.2 Derivation of the Real–Time Analysis Model . . . 82
6.2.1 Task Precedence Graph (TPG) Synthesis . . . 82
6.2.2 Derivation of the Worst–Case TPG . . . 85

6.2.3 Calculation of Start Times . . . 86

7 Case Study and Evaluation **89**
7.1 Olympus Attitude and Orbital Control System . . . 89
7.1.1 SDL System Model . . . 90
7.1.2 Timing Constraints . . . 91
7.1.3 Computation Times of MEDF Run–Time System . . . 92
7.1.4 System Environment and Timer Task . . . 96
7.1.5 Schedulability Analysis . . . 98
7.2 SDL Style Guidelines for Real–Time Systems . . . 102
7.2.1 General Rules . . . 102
7.2.2 Minimizing Number of SDL Servers . . . 103
7.2.3 Minimizing Length of Blocking Transitions . . . 103

8 Conclusion and Future Work **105**

Bibliography **109**

A Application Examples **119**
A.1 *Olympus* Attitude and Orbital Control System . . . 119
A.2 Code Generation with MEDF Integration . . . 137
A.2.1 SDT's Code Generation . . . 137
A.2.2 MEDF Scheduling Application Interface in RTEMS . . . 138
A.2.3 Example System . . . 139

B MEDF Implementation Details **145**
B.1 Delta Deadline Management . . . 145
B.2 EDF List Insertion . . . 146
B.3 Fasttick Board Support . . . 147
B.4 MEDF Semantics with SDT's Simulator . . . 149

List of Acronyms and Variables

Acronyms

AOCS Attitude and Orbital Control System ... 7
ATM Activity Thread Model ... 11
AT Activity Thread ... 39
BCET Best Case Execution Time ... 5
CASE Computer Aided System Engineering ... 3
CIOP Configurable Input/Output Processor ... 5
CSP Communicating Sequential Processes ... 10
DCP Deadline Ceiling Protocol ... 53
DIP Deadline Inheritance Protocol ... 55
EDF Earliest Deadline First ... 4
EDM Event Dependency Matrix ... 29
EFSM Extended FSM ... 19
ES Event Stream ... 28
FCFS First–Come First–Serve ... 51
FIFO First–In First–Out ... 15
FSM Finite State Machine ... 19
HOOD Hierarchical Object Oriented Design ... 14
HOQ Hand–Over Qeue ... 67
HPU High Performance Unit ... 5
HW Hardware ... 6
IQ Input Qeue ... 67
ISR Interrupt Service Routine ... 25
ITU International Telecommunication Union ... 18
LLF Least Laxity First ... 13
MEDF Message based Earliest Deadline First Scheduling ... 2
MSC Message Sequence Charts ... 10
OMG Object Modeling Group ... 17
OMT Object Modeling Technique ... 15
OOD Object Oriented Design ... 1
OSI Open Systems Interconnection ... 11
PAD Process Activity Description ... 35

PIP	Priority Inheritance Protocol	55
PI	Priority Inversion	15
PMSC	Performance Message Sequence Charts	10
PU	Processing Unit	5
QSDL	Queuing SDL	15
REAR	Rapid Prototyping Environment for Advanced Real–Time Systems	5
RMA	Rate Monotonic Analysis	4
ROOM	Real–Time Object–Oriented Modeling	10
RTAM	Real–Time Analysis Model	6
RTEMS	Real–Time Executive for Multiprocessor Systems	56
RTOS	Real–Time Operating System	14
RTU	Real–Time Unit	5
SA/SD	Structured Analysis/Structured Design	1
SDL/GR	SDL Graphical Representation	18
SDL/PR	SDL Prose Representation	18
SDL	Specification and Description Language	2
SDT	Telelogic's SDL Design Tool	3
SM	Server Model	11
SQ	Server Qeue	67
SW	Software	6
TPG	Task Precedence Graph	76
TPS	Task Precedence System	4
UML	Unified Modeling Language	10
VHDL	Very high speed integrated circuit Hardware Description Language	6
WCET	Worst Case Execution Time	5

Variables

$\mathbf{C(I)}$	computation function, maximum computation time in interval I
$\mathbf{D(T_i)}$	absolute deadline D_i of task i
$\mathbf{D_i}$	absolute deadline D of task i
$\mathbf{E(I)}$	event function, maximum number of events per interval I
$\mathbf{E_m^j}$	jth instance of event E of type m
$\mathbf{F(I)}$	overall computation function of ISRs in interval I
$\mathbf{c}, \mathbf{c_{max}}$	worst case execution time
$\mathbf{c_{min}}$	best case execution time
$\mathbf{d_i}$	relative deadline d of task i
$\mathbf{d_{Tm}}$	relative end–to–end deadline for event E_m
$\mathbf{e_{Tm}}$	time of occurrence of event E of type m
$\mathbf{f_i}$	completion time of task i
$\mathbf{r_i}$	release time of task i
$\mathbf{s(T_i)}$	start time s_i of task i
$\mathbf{s_i}$	start of computation of task i

List of Figures

1.1 Core Phases and Premises in a SDL based Design Methodology 2
1.2 REAR Design Framework 6

3.1 SDL System Syntax (graphical representation) 19
3.2 SDL Process Syntax (graphical representation) 20
3.3 a) End–to–end Deadline vs. b) Triggering Less Stringent Processing 24
3.4 Timeout with MEDF Timer 26
3.5 Event Stream Example 29

4.1 a) Synchronizing Mutual Exclusion and b) Monitoring Shared Function . . 34
4.2 a) Simple SDL System and b) its Tight Integration with SDT's C–Advanced 36
4.3 a) SDL System with Two Timing Constraints and b) its Activity Threads . 39
4.4 Timer Task: Signal and Timeout Handling 41
4.5 Timer Task with Timer Deadline Management 43

5.1 Examples of Task Precedence Systems 48
5.2 Feasible Schedule without a) and with b) Precedence Constraint 49
5.3 a) Server Task and b) Priority Inversion 52
5.4 Priority Inversion Avoidance 53
5.5 Violation of Ceiling Deadline D' without Priority Inversion 54
5.6 RTEMS Priorities, Ready List, EDF Thread List and EDF Event List . . . 56
5.7 Priority Based Thread Queue 60
5.8 Transitive Deadline Inheritance with a) Nested Semaphores and b) Servers 62
5.9 Server Model Task States and Structure of Task Body 66
5.10 Timing: a) EDF Event Create and b) EDF Event Delete 67
5.11 Timing: a) Send (No waiting task) and b) Receive (Message available) . . 69
5.12 Timing: EDF Event Receive (Caller blocks) 70
5.13 Timing: EDF Event Send (Task readied) 71
5.14 List Structures and References for Optimized Server Model Scheduler . . . 72

6.1 Information Flow and Dependencies of Real–Time Analysis 76
6.2 Graphical Representation of a Task Node [Gre93a] and Analysis Parameters 77
6.3 Event Function $E(I)$ and Requested Computation Time $C(I)$ 79
6.4 Busy Periods due to ISRs and Supplementary Tasks 81

6.5 Syntax of Task Precedence Graph and its Annotations 82
6.6 TPG Synthesis Example SDL System 83
6.7 Task Precedence Graph Including All Transition Alternatives 84
6.8 Worst Case Task Precedence Graph 85
6.9 Recursive TPG WCET Calculation (simplified) 86

7.1 Olympus AOCS Sensors and Actuators 90
7.2 AOCS SDL System Survey 91
7.3 Cyclic Event a) without and b) with Jitter 93
7.4 a) AOCS Interrupt Load and b) Busy Period at Interval 0 97
7.5 a) CWS Task Precedence System and b) Real–Time Analysis Model 98
7.6 a) AOCS Overall Computation Function, b) Detailed View in Interval 0 . . 101
7.7 a) AOCS $C(I)$ with $d_{CWS} = 200\,ms$, b) Detailed View in Interval 0 102

A.1 Olympus AOCS System Structure 120
A.2 Olympus AOCS SDL System 121
A.3 Block AOCSController 122
A.4 Block Sensors 123
A.5 Process Gyro 124
A.6 Process IRES 125
A.7 Process Attitude 126
A.8 Process DSS 127
A.9 Process CalibrateGyro 128
A.10 Block ControlLaw 129
A.11 Process Control 130
A.12 Block Actuators 131
A.13 Process MomentumDump 132
A.14 Process ReactionWheels 133
A.15 Process Thrusters 134
A.16 Olympus AOCS Task Precedence Graph 135
A.17 Olympus AOCS Real–Time Analysis Model 136
A.18 SDT C-Advanced System Build Process 137
A.19 Application Example: SDL System 139
A.20 Application Example: ISR Block 140
A.21 Application Example: Interrupt Service Routine 141
A.22 Application Example: SDL Process Network 142
A.23 Application Example: Process 11 142
A.24 Application Example: Server Process S 143
A.25 Generated Task System 143
A.26 Application Example: Organizer View 144

B.1 Deadline Management 145
B.2 Inserting into the Ready List 146

B.3 Insertion with EDF List Algorithm 147
B.4 Deadline Supervision with Clock and Fasttick Support (simplified) 148
B.5 MEDF Simulation based on SDT Tool Suit 149

List of Tables

5.1 Task and Environment Parameters 47
5.2 EDF event send and EDF event receive list dependencies and processing effort 68
5.3 EDF event send and EDF event receive processing sequences with optimized server model scheduler 73

7.1 AOCS Event Streams and Deadlines 92
7.2 Execution times of RTEMS message queue directives 94
7.3 Execution times of MEDF directives for a AOCS Server Model implementation 95
7.4 Processing Times of AOCS Task Precedence Systems and Server Processes 99

A.1 Annotations to the SDL Model 119
A.2 MEDF directives, object modes and attributes 138

Chapter 1

Introduction

In July 1997 the Mars probe *Pathfinder* survived its rubber ball landing on the Martian surface due to its airbag protection. After successful deployment of the *Sojourner* rover, it resumed its mission with gathering environmental data from this distant neighbor planet. Caused by a priority inversion between a sensor, the communication, and the system management task, the spacecraft experienced sporadically total system resets repeatedly leading to a loss of measurement data. Press releases stated the "computer was trying to do too many things at once", but a detailed analysis of the software system showed a watchdog timer had detected an avoidable *deadline* violation for the communication task.

The timing characteristics of this *real–time system* were considered as *soft* during this phase of the mission. A similar failure during the more mission critical phase of approaching the Martian surface would have probably led to a loss of the space vehicle. In this case, the timing constraints and therefore the real–time software system would have been classified as *hard* and a system reset of course would not have been a viable recovery strategy for a deadline miss. Although functionally correct, and although single components had been designed with efficiency in mind, inter–dependencies (resource sharing) of the concurrent tasks had led to a timely incorrectness.

As can be seen, complexity of the whole application makes an analysis of timing requirements difficult. Neither black box simulation, nor performance analysis with state–of–the–art computation models like queuing theory or functional prototyping would had revealed the additional blocking time caused by priority inversion for the communication task in all probability. On the other hand, current practice in schedulability analysis theory is based on very restrictive task models and could have been applied only if the real–time software has been designed accordingly.

Software engineering methodologies like structured analysis/structured design (SA/SD) or object oriented design (OOD) win increasing recognition even in the area of dedicated systems. There exist sophisticated tool support for a multitude of design languages, because it is now generally accepted that software design automation will help shortening development times and thus development costs. But when timing specifications are not met, structural modifications may become necessary. In this case or when design changes occur, e.g. through additional required functionality, a rescheduling of all components and

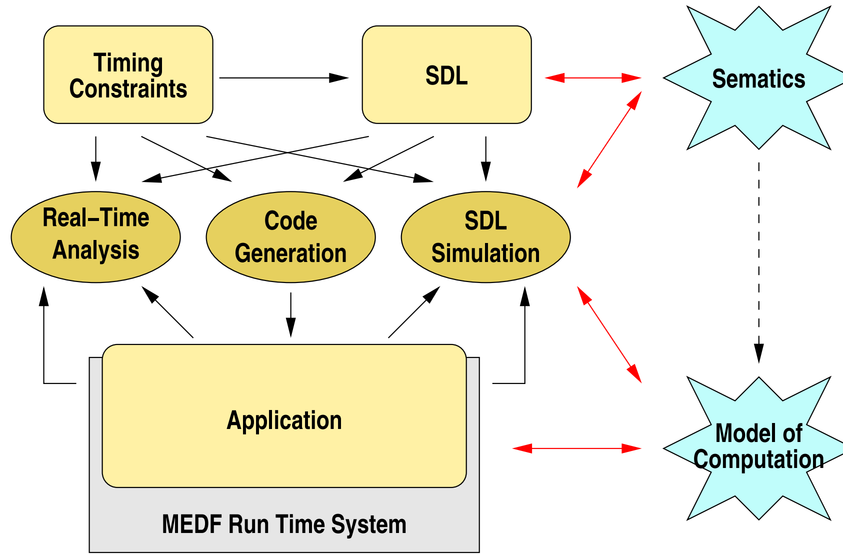

Figure 1.1: Core Phases and Premises in a SDL based Design Methodology

consequently a reanalysis of the temporal properties has to be done. This means, for real–time systems the schedulability analysis should have to be an automated design step as well.

1.1 Contribution

This work introduces a methodology framework based on the "Specification and Description Language" (SDL) aiming on the integration of a timeliness proof into the automated design process for hard real–time systems. For that purpose, SDL's semantics is pinpointed to a predictable model of computation allowing on the one hand to automatically derive analyzable implementations. On the other hand the system specification can now be mapped to a real–time analysis task graph that mirrors the structural inter–relationships between the resource sharing components and is the basis for the final schedulability analysis step. Message based earliest deadline first scheduling (MEDF) serves as a platform for this design process. Therefore MEDF's predictability will be shown and algorithms to evaluate timing properties of MEDF based software systems will be provided.

1.2 Prerequisites and Restrictions

A coarse survey of a design methodology for hard real–time systems based on SDL [Kol98] is given in Fig. 1.1. In the follow section, motives for choosing SDL as the starting point for this framework are outlined. Based on this, restrictions and requirements for automated code generation and the applied scheduling policy are outlined.

System Specification

Design automation in general needs a formal system description to capture the functional and non–functional requirements. Beyond this, automated schedulability analysis requires a formal semantics which suits to the applied analysis task model. Due to the concurrent nature of the embedding environment, concurrency too is used as a means to structure the design of a real–time application. Therefore process models are the basis of most state–of–the–art specification or real–time programming languages. The *Specification and Description Language* SDL as a formal description technique originally developed and standardized [ITU94a] for telecommunication systems is more generally suitable for control flow dominated systems. SDL's message oriented, asynchronous communication and the state–machine like behaviour of its processes matches well the event driven nature of many real–time applications.

Like most specification languages SDL too implies some abstractions and non–determinisms in its semantics due to its intended application in the early phases of the design cycle. This fact is tolerable or even useful during the specification phase, but of course not acceptable in the implementation, respectively during run–time. For this reason, the execution model has to be pinpointed to the chosen scheduling policy to enforce conformance between the language semantics, the model of computation during functional simulation, the schedulability analysis task model, and the implementation strategy. Added to that, there is a need to restrict the usage of some SDL language constructs to avoid unpredictability.

Unlike general purpose systems, the dedicated functionality of *embedded systems* imposes additional constraints like maximal memory usage or maximal power consumption on the design. Therefore the functional specification has to be complemented with a description of non–functional requirements. This work focuses on the analysis of timing properties of hard real–time systems. Thus, the non–functional specification must include a worst case description of the temporal behaviour of the *embedding* system and on the other hand, must specify the *deadlines* to be reached by the final system realization.[1] In case of integrated control systems, deadlines are imposed by the system's environment, i.e. the embedding system that has to be controlled by the application. Consequently, deadlines are understood as attributes of the stimulating events, i.e. they are *end–to–end* constraints. A triggering event has to be responded within a specified deadline interval. The event's characteristics are added as annotations to the functional description.

Code Generation

CASE tools like Telelogic's SDL Design Tool SDT support the mapping of a functional specification to an executable prototype. Unfortunately, the semantical gap between the assumed "each SDL process runs independently on its own processor" behaviour in the specification and the sequential computation on a single–processor target can be the source

[1] As will be shown later (cf. Sec. 3.2.2), SDL's timer mechanism is inappropriate for timing constraint specification.

for inefficiencies in automatically generated implementations preserving the process model. This leads to the effect that a system designer will manipulate the generated target code instead of the system model to achieve the required performance. Thus, system specification and implementation diverge and become inconsistent.

This observation underlines the importance of an efficient decomposition of the system model. This is especially necessary when the application is control flow dominated, i.e. processing time for communication and synchronization of application threads is in the order of magnitude of the application's computation time. Maintainability is gained, if the specification and only the specification is the place for changes. But instead of average or best case throughput and latency, code generation strategies have to concentrate on guaranteeable, worst case performance and *predictability.* Since the model of computation of the system specification and the processing order of the final implementation must correspond exactly to enable a specification based real–time analysis, code generation strategies must also preserve the semantics of the specification language.

Scheduling Policy and Run–Time System

Earliest deadline first (EDF) scheduling is often regarded as dangerous, because under overload, it may show awkward behaviour. However, in a hard real–time system's life cycle an overload situation must not occur, because a deadline miss may lead to a loss of money or even a loss of lives. This fact requires an *a–priori* proof that all timelines will be met even in worst case situations, e.g. when burst events occur simultaneously[2] and all tasks in a system exhaust their worst case processing time budgets.

EDF has been shown to be an optimal scheduling strategy for single processor systems [LL73], i.e. if you can find a schedule with an alternative policy, you will find a feasible solution with EDF too. Furthermore, an existing deadline specification needs not to be transformed into a process priority assignment, like it has to be done with e.g. rate monotonic analysis (RMA, [KRP+93]). Therefore it is easier to integrate into an automated design process. Finally, time–driven schedules for sporadic task activations result in unsatisfactory achievable processor utilization. This fact and their inflexible design speaks for a dynamic management for *event–driven* real–time systems.

Complex process networks in the final realization may emerge through the use of a process based specification language. As a consequence, more than one task will be involved in the processing of a stimulating event and on the other hand one task may serve more than one type of event (with different required response times). Thus, a deadline will span a whole *task precedence system* (TPS). Static partitioning of deadlines and that followed an assignment of sub–deadlines to processes would lead to a loss of laxity, i.e. an over–specification of the system. Consequently, there is a need to dynamically transport a deadline to the succeeding tasks within a TPS. The resulting scheduling scheme is *scheduling with message deadlines* (MEDF).

It will be shown that applying MEDF to a network of communicating processes will lead

[2] called *critical instant*

to an EDF processing sequence of these processes. Thus, feasibility analysis algorithms like [Gre93b, Gre93a], [SSRB98], or [Jef92] may be applied to prove the required timeliness.[3] Unfortunately, there exists no known MEDF implementation in state–of–the–art real–time operating systems. Therefore, necessary run–time system support has to be provided.

Schedulability analysis requires all best (BCET) and worst (WCET) case execution times to be known in advance. There exist different approaches in research, based on measurement or code evaluation, to derive processing time variations caused by e.g. different program paths. Preemption delays caused by cache flushes and pipeline stalls may complicate this problem. WCET/BCET determination is out of scope for this work. Thus, processing times are assumed to be known. For a detailed problem description refer to [PF99, Pet02].

1.3 This work's background: Rapid Prototyping

Rapid prototyping has been proposed as a methodology to find design errors and flaws in the embedded system's requirements at a very early stage of its development cycle. A re–usable, configurable and scalable target architecture serves as a platform to execute a system's specification in the real environment in form of a working prototype. An automated design process is needed to translate the formal system description into an executable in order to rapidly derive such a prototype. The rapid prototyping environment REAR[4] [PMK+99, PMK+00] realizes such a design framework and includes the design methodology as introduced in this work.

REAR's target architecture (8 in Fig. 1.2, [FKMF97]) consists of an heterogeneous multi–processor system complemented with additional field programmable gate arrays, tightly coupled with the microprocessor based units. Its processing nodes (High Performance Unit (HPU), Real–Time Unit (RTU), and Configurable I/O Processor (CIOP)) are specialized according to the "task classification model" [FFKM97], where each type of real–time task corresponds to a best suited type of processing unit (PU), in terms of performance and deterministic execution times. Therefore, a straight–forward allocation of the application tasks to the single PUs can be derived from a task grading according to both, computing complexity and deadline.

Implementing a prototype in software means high flexibility, short design cycles, and good debug facilities, therefore as much functionality as possible will be implemented in software. Since target systems have to fulfill real–time characteristics, HPU and RTU provide means to guarantee predictable limits for run time variations caused by caching or pipelining effects. Application specific hardware on the other hand is very often necessary as a link to the embedding process and as execution unit for processes with deadlines too short to be met in software. A CIOP acts like a pre–processor for external events (CIOP–IO) or may be used for timing measurement respectively trace collection (CIOP–T).

[3] These analysis methodologies take into account inter–task dependencies like precedence constraints or blocking times caused by mutual exclusive access to shared resources.

[4] Rapid Prototyping Environment for Advanced Real–Time Systems

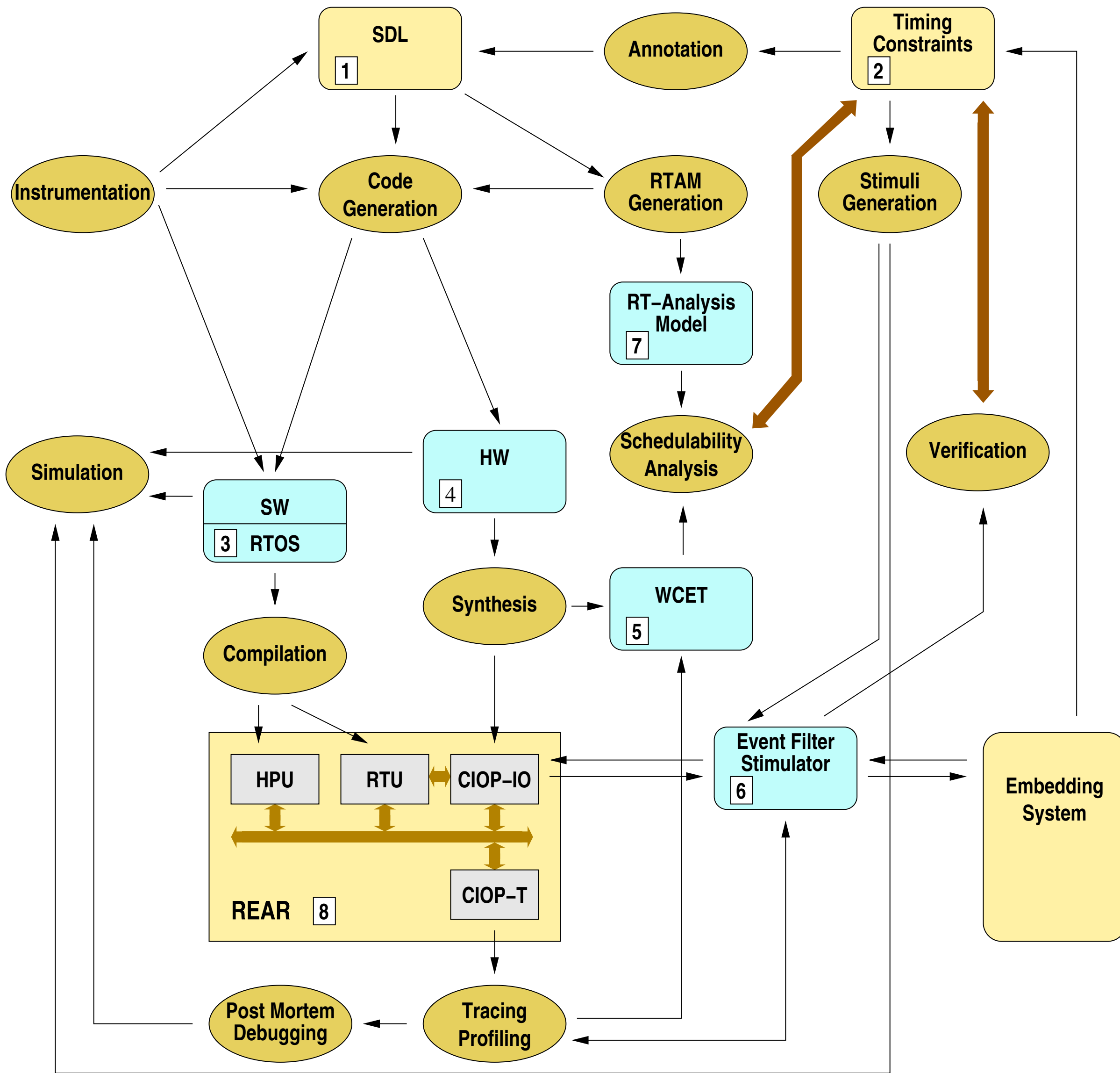

Figure 1.2: REAR Design Framework

A Hardware/Software co–design methodology with SDL 1 as starting point allows to translate the specification into C 3 and VHDL 4 code. C–code generation is partly based on the CASE–tool family SDT (SDL Design Tool) from Telelogic, SDL to VHDL synthesis is part of the research focus [MKMKF00, MF00]. Together with run–time and interface libraries, executables can be compiled and mapped to the dedicated target nodes.

The worst (WCET) and best (BCET) case execution times 5 needed for the final schedulability analysis 7 are on the one hand provided by synthesis tools for the HW–part, on the other hand are determined measuring instrumented code during execution on the target architecture for the SW–part [PF99]. The real–time analysis model (RTAM) automatically generated from the SDL model reveals the structural inter–dependencies

incorporated in the system. It includes the timing constraints 2 that are added to the system specification by annotation. Furthermore, the RTAM is needed as information source for the optimizing code generators. If the final timeliness proof reveals there are no deadlines misses to expect, the prototype can be executed in the real environment. An event filter 6 supervises a minimum distance between events.

In contrast to simulation, the idea of a prototype is to be executed in the real world with real data. It allows to verify by execution if all assumptions were correct and the functional and timing requirements have been modeled correctly. Temporal feasibility is the crucial point in the development of hard real–time systems. Next to the proof that the prototype will meet all specified timing constraints, a first estimation on minimum required processor performance for the final realization can be gained. Furthermore, possible bottlenecks in the SDL system model can be detected, e.g. the negative effect on the responsiveness of a system caused by large areas of mutual exclusion, originating from long state transition times in server processes can be revealed. Finally, the system's predictability can be verified by analyzing the communication structure of the SDL system.

1.4 Synopsis

This work is organized as follows: The next chapter covers state of the scientific and technical knowledge in research topics tangent to this thesis.

Chapter 3 provides a detailed analysis of the SDL's syntax and semantics. It will be shown this language is appropriate for functional specification of hard real–time systems. Its model of computation as well as use of its language features however has to be restricted to gain predictability. A methodology to express non–functional requirements based on the "Event Stream" model is introduced.

Mapping principles allowing the translation of a SDL system model to a final implementation thereby preserving SDL's semantics are discussed in Chapter 4. Requirements to the software architecture imposed by the "Scheduling with Message Deadlines" (MEDF) model of computation are given. MEDF's predictability will be proven in Chapter 5, before implementation alternatives for the necessary run–time system support will be evaluated.

The mapping methodology to translate a SDL specification to a "Real–Time Analysis Model" (RTAM) will be explained in Chapter 6. The RTAM consists of a network of task precedences and shows processes monitoring shared resources. Both is necessary structural information needed by the final scheduling analysis algorithms which will be shortly re–introduced.

The Olympus AOCS case study, i.e. a specification of a satellite's "Attitude and Orbital Control System", serves as sample application to demonstrate the proposed design process and to evaluate the influence of modeling style and code generation strategy on results of the final schedulability analysis step. For this, overhead caused by run-time system directives will be taken into account (Chap. 7). This thesis concludes with a discussion of proposed methodologies and finally outlines possible future work.

Chapter 2

Related Work

The incorporation of schedulability analysis into an automated design process involves the following research areas:

- What languages are used to describe the functionality of real–time systems and how are non–functional requirements expressed?
- What methodologies are deployed to translate a system specification into a predictable realization?
- And finally, what analysis algorithms and heuristics exist to prove the required timeliness of the final implementation?

In the following sections, a closer look to answers provided for these questions will be taken, before holistic approaches comparable with this work are evaluated in detail.

2.1 Real–Time Constraint Specification

It can be generalized that the main focus of all specification languages lies in managing system complexity. For this, they provide means to deal with concurrency and hierarchy as well as communication and synchronization mechanisms to organize data flow and timely inter–dependencies [GVNG94, ELLSV97]. Therefore, common to most languages are process models, whereas at this level of abstraction the term *process* denotes the concept of an autonomous object with its own thread of control.

Synchronous languages like Esterel, Lustre, or Signal [Hal93] idealistically assume that processing of data consumes no time and time progresses only on the occurrence of new external stimulating events. This means actions are instantaneous and the underlying broadcast communication takes zero time as well. Aiming on formal verification of characteristics like causality, liveliness or safeness, this class of specification language necessitates such a strong synchronization. A translation to an efficient implementation keeping this restrictive semantics is difficult, for distributed applications even nearly impossible. This

is even true for Statecharts [Har87, HLN+90], whose timing model is as well synchronous in the sense that time elapses only in equally sized steps.

Asynchronous languages assume that progress of time is continuous, thus actions in processes consume an unspecified but non–zero interval of time. Communication schemes are mostly based on message exchange and may be either blocking, e.g. rendezvous concept in Ada or waiting send in Hoare's CSP [Hoa78], or non–blocking paradigms like the port concept in ROOM (*Real–Time Object–Oriented Modeling* [SGW94]) or SDL's (*Specification and Description Language* [ITU94a]) signal communication. Especially for modeling of event driven real–time applications, asynchronous (non–blocking) message communication schemes are preferable, since they allow a timely decoupling of application threads, often necessary to achieve real–time behaviour.

Although these specification languages are called "real–time", they all lack means to formulate non–functional requirements, e.g. like maximum allowed response times or to describe timing constraints imposed by the system environment, e.g. minimum intervals between external events. In a work–around like approach, expressiveness is improved in combining functional specification languages with real–time temporal logic formulas [Leu95, PB98]. Applicability however remains limited due to the cryptic appearance of temporal logic languages.

In a similar solution comparable to this work, annotations to the system specification are proposed in [SSD97]. Beneath mapping, resource, and cost requirements, the so called *SDL∗* allows to add timing requirements in form of comments. Two types of constraints are distinguished: *duration* and *jitter*. With this, definition of response times is possible, but constraining the behaviour of the system environment in a worst case manner remains unsolved.

The later problem is addressed in *PMSCs* [FLMTS97]. Again annotations in comments or text symbols, but now to Message Sequence Charts (MSC [ITU94b]) are used to express "performance requirements". So called *traffic sources* allow the specification of inter–arrival times of system stimuli. *Durations* and *spans* between marks on the MSC time axis are provided for timing constraint specification. PMSCs may be used as basis for an analysis to calculate mean processing times of system scenarios, i.e. exemplary signal sequences that cover parts of a SDL system [Lam97]. In [BAL97], the general expressiveness of MSCs for timing constraint specification is discussed and an algorithm for verifying the consistency of these timing specifications is presented. Conflicting constraints will be detected by translating basic MSCs into (cycle–free) temporal constraint graphs.

Although there exist combined efforts to incorporate those language features into new versions of SDL [MTMC99], even its latest release SDL'2000 provides no support for a description of non–functional requirements.

A contrary trend can be observed with the Unified Modeling Language (UML [Obj99b]). UML can be seen as a general–purpose specification language latterly applied even for development of real–time applications. There exist different "flavors" of this language depending on the appropriate CASE tool manufacturer [TG00]. UML has still a vague semantics and is itself under ongoing standardization [Kob99], but there are attempts to take care of special requirements imposed by this application area: The "UML Profile for Scheduling,

performance, and Time" [Obj99a] includes so called "mandatory requirements" for future extensions to UML. Its section "Timing Specifications" aims on expressing constraints like deadlines, periods, jitter, inter–arrival times, and so on.

2.2 Code Generation Efficiency

System synthesis and thus automatic translation of a system specification into (executable) code is well understood in the field of "Electronic Design Automation". There exist sophisticated tool support (e.g. [Syn99]) to compile hardware description languages like VHDL [IEE00] and Verilog [IEE95] into networks of target components, whereupon nearly any chip manufacturer is supported. Optimizing synthesis steps allow a translation even of behavioral descriptions into system realizations that are near to an optimal solution. This means uncertainties caused by level of abstraction and granularity of the underlying timing model are successfully resolved by stepwise refinement of the input description.

Unfortunately, the latter does not apply for design processes targeting software. For a detailed problem statement refer to [Mok83] or read the discussion in Sec. 4.1. Research on code generation from SDL is mainly based on implementation strategies developed in the effort to efficiently realize multi–layered OSI protocols [Svo89]. Two different methodologies with different clustering concepts were identified: The *Server Model* centralizes a whole OSI protocol unit into one single task in the final implementation. Inter–layer and thus inter–task communication is based on message exchange. In contrast to this, the clustering concept of the *Activity Thread Model* (ATM) forces all procedures that are involved in serving an incoming request through all protocol layers into one software task. Since inter–layer communication is now based on procedure calls, no overhead due to message queuing or process scheduling can arise. The influence on efficiency when applying these strategies on protocol implementations derived from *Estelle* specifications is evaluated in [HK95]. The semantical correct mapping of SDL specifications to the Activity Thread Model is evaluated in [HKMT97]. [LK97, LK99] present refinements of the initial approach that include serving multiple, concurrent external requests (Extended ATM) and means to ensure a processing order as defined by the model of computation in SDL (e.g. reordering of SDL statements at compile time).

SDL is as well used as a system level specification language for a rapid prototyping design process [PMK+00] with both software and hardware [Mut02] as target. Since this approach partly builds upon the above mentioned design methodology and thus relies on the synchronous language VHDL for its hardware components, a timing paradigm shift during system synthesis is indispensable. Thus, the necessary "hardware run–time library" which provides components for synchronization and communication of concurrent entities on the final chip has to include a certain overhead which is acceptable for rapid prototyping. A performance evaluation of Server Model and ATM based code generation strategies can be found in [MKMKF00, MF00].

2.3 Schedulability Analysis

Especially hard real–time systems require an *a–priori* proof that all specified timing constraints will hold even in worst case situations. Reacting not until a deadline has been missed in an overload situation may have harmful consequences on the system itself or in case of a safety–critical application even on human lives.

There exist two general classes of scheduling strategies for real–time systems: The first category, called *static* scheduling, includes all methodologies that use an off–line calculation to establish a "task calendar". This table defines start times for all task and is read by the run–time system's dispatcher to allocate a certain concurrent task at a pre–defined time to the processor. The task calendar is repeatedly processed leading to cyclic task activations. Start times are referring to the beginning of each round.

Static scheduling is the basis of the fault–tolerant architecture of MARS [DRSK89, SRG89]. Since a single time table can not cope with all operational situations, multiple calendars are prepared before run–time. A "schedule switch" is then initiated in emergency situations or at pre–defined points in time when the operation mode has to be changed. Timeliness of all schedules is achieved by construction. This approach proves to be inflexible since all combinations of possible situations have to be analyzed in advance.

Off–line generation of a parametric calendar is as well proposed by [SGA93]. Their concept of static scheduling allows processing times of single tasks to be in pre–specified $[c_{min}, c_{max}]$ intervals to increase flexibility. Processing order constraints (precedence constraints) lead to so called transactions that consist of partially ordered tasks, but there are no further inter–dependencies between transactions supported. It has been shown that the construction of a time–table for non–preemptive hard real–time scheduling is a NP–complete problem.

Taking into account sporadic tasks that occur seldom but have a short deadline (e.g. initiate the inflation of an airbag in a car crash) leads to a very pessimistic layout of the task calendar. Since polling for the sporadic event is necessary, it will devour the majority of the available processing time. *Dynamic* scheduling approaches utilize this wasted spare time in dispatching a high priority task only in the moment of the sporadic event's occurrence. One differentiates between two approaches, scheduling with *fixed priorities* and scheduling with *dynamic priorities*.

Liu and Layland [LL73] submitted the fundamental theorems to dynamic scheduling. Their assumptions rely on a very simple task model: 1. All requests to tasks are strictly periodic (cycle time T_i); 2. New requests occur only after the previous job has been completed; 3. No shared resources and no precedence constraints are allowed; 4. Run–times have to be constant (execution time C_i). 5. Tasks are perfectly preemptible. Liu and Layland introduce an overall "processor utilization factor" U for a number of m tasks.

$$U = \sum_{i=1}^{m} \frac{C_i}{T_i} \tag{2.1}$$

For fixed priorities, their *rate monotonic analysis* (RMA) algorithm requires priorities to be assigned to tasks in the following way: the higher the task's rate, the higher its priority.

For a set of m tasks they proved that all deadlines will be met if the overall processor utilization U remains below the least upper bound U_{RMA} (sufficient requirement).

$$U_{RMA} = m(2^{\frac{1}{m}} - 1) \tag{2.2}$$

For a large number of tasks, this upper bound goes to $U_{RMA} = ln(2)$ which is quite pessimistic. This utilization bound can be relaxed and even set to $U_{RMA} = 1$, if task periods are harmonic.

Much research has been invested to extend Liu and Layland's restrictive task model and incorporate e.g. release jitter, aperiodic tasks, and precedence constraints into RMA analysis. A good summary is given in [KRP+93]. Special care to analyzing tasks sets that share common resources and the application of the "priority inheritance" protocol can be found in [Raj91]. An interesting approach in minimizing blocking times on shared resources by adapting a "preemption threshold" is described in [WS99].

For dynamic priorities, Liu and Layland proposed in their *deadline driven* scheduling algorithm, to assign priorities to tasks according to their deadline in the *current* request. With this, tasks with the earliest deadline will be dispatched first (EDF). Now, for a fixed set of m tasks they proved that it is sufficient to show that the overall processor utilization U remains below or equals U_{EDF}.

$$U_{EDF} = 1 \tag{2.3}$$

This means, in applying EDF scheduling a 100% processor utilization can be achieved, although periods are not harmonic. Furthermore, EDF has been shown to be an optimal scheduling algorithm. It is optimal in this sense, that if one can find a feasible schedule with a task set with fixed priorities, one will always find a feasible schedule with the deadline driven approach too.

Again, there exist innumerable research papers addressing the relaxation of the task model strictness assumed by Liu and Layland. EDF analysis algorithms to schedule task with precedences, aperiodic task activations and protocols to synchronize tasks with shared resources are summarized in [SSRB98]. Feasibility conditions for sporadic task sets with common resources are presented for example in [Jef92] and [CLB99]. Event driven systems that are triggered by stimuli with sporadic or even burst behaviour are investigated in [Gre93a, Gre93b]. Gresser's EDF scheduling theorems and his "Event Stream Model" for description of task stimuli is the basis for this work.

Astonishingly, although scheduling theory for both approaches is equally sophisticated, tool support is given mostly for rate monotonic analysis. "TimeWiz" (TimeSys), "RT–Architect" (Realogy), "PERTS" (Tri–Pacific) and others provide a real–time kernel as well as a timing analyzer.

Further dynamic scheduling algorithms are e.g. "Least Laxity First" scheduling (LLF) or distance constraint algorithms like the "Pinwheel" strategy [HL97]. The later has been designed with purpose to minimize jitter between task activations and therefore is especially suitable for multi–media applications.

2.4 Real–Time System Design and Schedulability

As has been shown in Sec. 2.1, real–time specification languages lack means to express quantitative timing constraints. Furthermore, there exist only few approaches that integrate a schedulability proof into the system design process as required to construct predictable real–time applications.

"Hard Real–Time Hierarchical Object Oriented Design" (HRT–HOOD, [BW95b]) provides dedicated terminal object types to identify the characteristics of activities within a real–time application. Decomposition rules force the designer to apply a defined calling paradigm when decluttering active objects into terminal objects. Each terminal object possesses certain non–functional properties. *Cyclic* objects are used to represent periodic activities. Timing and criticality attributes for this object type are period, release time offset, worst case execution time, deadline, and priority. *Sporadic* objects allow to specify a minimum interval between successive task releases instead of a period, but are treated equally during timing analysis. *Protective* objects are resource control objects that do not possess an own thread of control, but may be activated by cyclic or sporadic objects. To synchronize concurrent activations, and to minimize blocking times on common resources, a ceiling priority can be assigned to them. Finally, *passive* objects encapsulate functionality, again without a controlling thread, that may be repeatedly used by other active objects.

Unfortunately, HOOD and thus HRT–HOOD lacks a behavioural description for its objects. Instead, release time offsets and deadline splitting is needed to ensure correct processing sequence. To circumvent this, [PB98] proposes to use HOOD objects to specify system structure, but to use Modecharts to describe object details. In contrast to this, [CW95] enhance HOOD with so called *transactions* to specify object behaviour. Lack of a behaviour specification is as well the cause, that execution times in HRT–HOOD apply only to whole objects. Thus, timing resolution of schedulability analysis has to be coarse grained. Burns and Wellings show how a HRT–HOOD specified system has to be mapped to an Ada based software architecture to ensure predictability and provide a heuristic for a rate monotonic analysis based timeliness proof that takes into account the implications of an Ada real–time kernel [BW95a].

Saksena et. al. [SFR97, SK00] provide design style guidelines for ROOM ("Real–Time Object–Oriented Modeling, [SGW94]) specifications to improve schedulability. For a final implementation, they propose either a single–threaded (leading to a non–preemptive schedule) or a multi–threaded software architecture in which several ROOM objects are grouped into one real–time operating system (RTOS) thread. Fixed event priorities have to be manually assigned to inter–thread as well as to inter–object messages. Their schedulability analysis calculates the worst case blocking times of events on objects caused by ROOM's run–to–completion semantics of encapsulated transitions and the run–to–completion processing of each implementation thread. To avoid priority inversion effects, either priority ceiling or the above mentioned preemption threshold [WS99] protocol may be applied and is incorporated into their analysis scheme. Even applying a deadline monotonic approach to find a suitable allocation of priorities to internal and external events need not necessarily

lead to an optimal schedule [SKW00].

Extensions to SDL to support a performance analysis based on queuing theory is proposed in [DHHM95, DHM96]. “Queuing SDL” (QSDL) and the underlying tool QUEST allow to estimate wait time distributions of signals in SDL queues at processes. The later are bound to so called machines that provide one or several services. System stimulations, i.e. triggering events are created by traffic sources. Although a multitude of traffic sources and machine service types are made available, only stochastical predictions on system behaviour can be made. Parameters like a utilization distribution of a machine or a mean response time for a service request are not suitable for hard real–time systems which require a worst case consideration.

Transition priorities instead of normally applied process priorities are proposed for SDL systems in [ADL+00, ADL+99]. A thereof derived rate monotonic analysis is embedded in an object–oriented design process starting with “Object Modeling Technique” (OMT), respectively UML. To avoid priority inversion (PI) on transitions monitoring shared resources, the priority ceiling protocol has to be applied. Due to the run–to-completion semantics of SDL transitions, and due to the retained “first–in first–out” (FIFO) semantics of SDL process queues, i.e. a high priority event will be appended behind low priority events, blocking times of events can be become very large despite the PI avoidance mechanism.[1] Though this effect is covered in their timing analysis, maximum achievable processor load however has to be very pessimistic.

[1] This is also true for the ROOM/RMA integration [SK00].

Chapter 3

SDL Semantics and Timing Constraints

On the one hand, advantages in applying specification languages like SDL or UML/RT[1] for the design of hard real–time systems are increasingly accepted in industrial practice. For example, newest trends in standardization of the "Unified Modeling Language" [Dou98] follow SDL and ROOM language concepts. Furthermore it can be observed that these two languages will be included into the specification and design process as proposed by the "Object Modeling Group" (OMG). State–of–the–art case tools like Telelogic's Tau framework provide a tool chain, covering a continuous design process, starting with UML for requirement specification and concluding with SDL for detailed design and implementation for exactly this application domain.

But on the other hand, applicability of these languages especially for the domain of real–time systems is criticized in theoretical computer science [Hin98b, Hin98a]. Main points of criticism include:

1. Expressiveness to capture real–time requirements that are complementary to the *functional* description of a real–time system is poor in SDL[2] [Leu95].

2. Its asynchronous message passing communication scheme which allows a loose coupling of independent system parts (a sender will never be blocked through a non–ready–to–communicate receiver) is unsuited for real–time systems because the delay until the receiver proceeds its execution can not predictably be bounded [BGK97].

Having this in mind the question arises, what is the general purpose of a specification language, especially its semantics? A language should provide a frame of rules that allow the system designer to describe the application under development in an abstract and implementation independent way. Furthermore the language's strictness and formality should help to avoid design mistakes in advance, e.g. its communication paradigm should

[1]Formerly known as ROOM and used with this name in this work. Ongoing (Spring 2001) lawsuit regarding Rational's new product name.

[2]For this purpose, only a timer concept is supported with SDL.

ensure synchronization on mutual access to common devices or data. In addition to this, its semantics should provide for executability (i.e. system model simulation, even if in an incomplete state), its formality should allow automatic code generation, and finally its predictability and analyzability should enable the proof of certain characteristics, e.g. timeliness.

As can be seen, these are very broad requirements emerging from the fact that a specification language will be used in the very early phases of a design cycle, but likewise may be applied very lately, e.g. even for an implementation description. Thus, it has to support very abstract and therefore probably indeterministic language concepts on the one hand, but has to provide the possibility to pinpoint these non–determinisms finally in the detailed design. As will be shown later in this chapter, this all can be done with SDL.

This work aims on the proof that a system under development, when finally implemented, will show real–time behaviour. This can only be done, if the model itself and the description language used behave predictable. Beyond this, for real–time analysis implementation details like execution times must be known or have to be estimatable. We provide an improved predictable finalization for SDL's semantics (process activation with signal deadlines) which is only a refinement of its original model of computation. It will be applied when approaching the final realization. In order to have real–time analysis results correspond with implementation's behaviour and in order to have a simulation meet the designer's expectations, it is necessary, that all three the semantics of the specification language, as well as a simulation's timing model, as well as the scheduler of the applied real–time operating system behave in a conform way.[3]

SDL's timer concept will be interpreted only as a *functional* language concept and will not be used to express non–functional constraints. For this purpose, we propose to extend a SDL specification with (timing) attributes to capture real–time requirements.

This chapter provides in its first part a short introduction into SDL's syntax and standardized semantics. It proceeds with an evaluation of expressiveness and analyzability of its language concepts. Next section introduces an improved execution model (MEDF scheduling) and a supplementary notation to express timing behaviour of the model itself and its environment. Finally, attributes used to specify real–time constraints are presented.

3.1 Specification and Description Language SDL

Recommended by the ITU–T (formerly CCITT) as design language for telecommunication systems, SDL is a formal specification language with well defined syntax and semantics [ITU94a]. Being continuously extended and improved, its newest version SDL'2000 is under standardization in the moment. All main language concepts are already included in version SDL'92 on which this work focuses. Its object–oriented concepts and thus dynamic process creation must be excluded in this work for predictability reasons (cf. Sec. 3.3.5).

SDL offers two different kinds for its representation, a graphical one (SDL/GR) used in this thesis in all examples because of its expressiveness and understandability, and a

[3] [Für01] realized a MEDF simulator kernel on RTEMS basis for Telelogic's SDL design tool SDT.

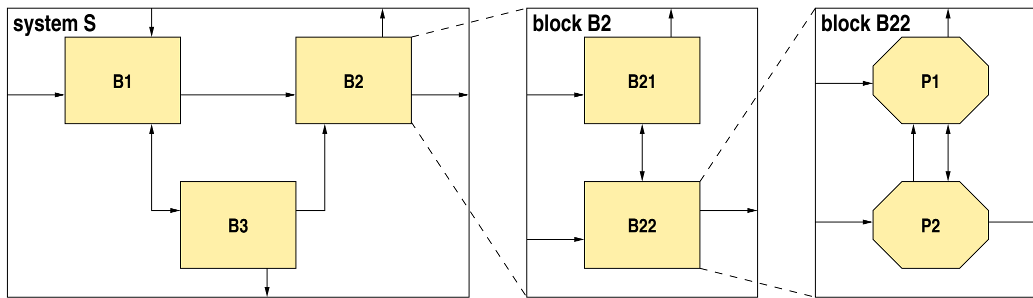

Figure 3.1: SDL System Syntax (graphical representation)

textual one (SDL/PR) used as input language in all parsing tools throughout the design framework[4].

In the following, basic SDL concepts are shortly explained. For a more detailed description of its language features and further recommendations on its use, the reader is referred to [Ols94, BHS91, BH93, ITU94c].

3.1.1 Syntax

SDL system structure: A SDL specification consists of a system model that communicates with the environment. The system model can be decomposed with hierarchically decluttered blocks connected via uni– or bidirectional channels. Blocks can be further decomposed into a set of processes which communicate asynchronously via signalroutes and messages (called signals), whereby signalroutes can be bundled in channels between blocks. All processes are assumed to run concurrently and each SDL process owns its private infinite queue that holds incoming messages according to a first–in first–out strategy[5]. Since there are no shared variables[6] allowed between blocks and even processes, data exchange must be done on message basis. A simple example is given in Fig. 3.1 showing the structure of a SDL system S with a refined block (B_2) and its process network (B_{22}).

SDL system behaviour: System behaviour is expressed through SDL processes. A process itself is specified as an extended finite state machine (EFSM, Fig. 3.2) that consumes incoming signals and in turn produces outgoing messages (signal *output*). Signals are processed in the order of their signal queue position and trigger a state transition. If there is no appropriate *input* statement, the signal will be discarded[7], except it is saved by means of a *save* statement.

SDL allows to add an *enabling condition* to a transition that is tested after signal receival. If the test result is invalid, the incoming signal will be saved. *Continuous signals*

[4] Telelogic's SDL design tool SDT supports SDL/GR to SDL/PR conversion without loss of information. For vice versa additional graphical layout information has to be added, but this conversion is also supported.

[5] except for priority input

[6] SDL provides an explicit *reveal/view* concept for shared variables between processes.

[7] called "*implicit transition*"

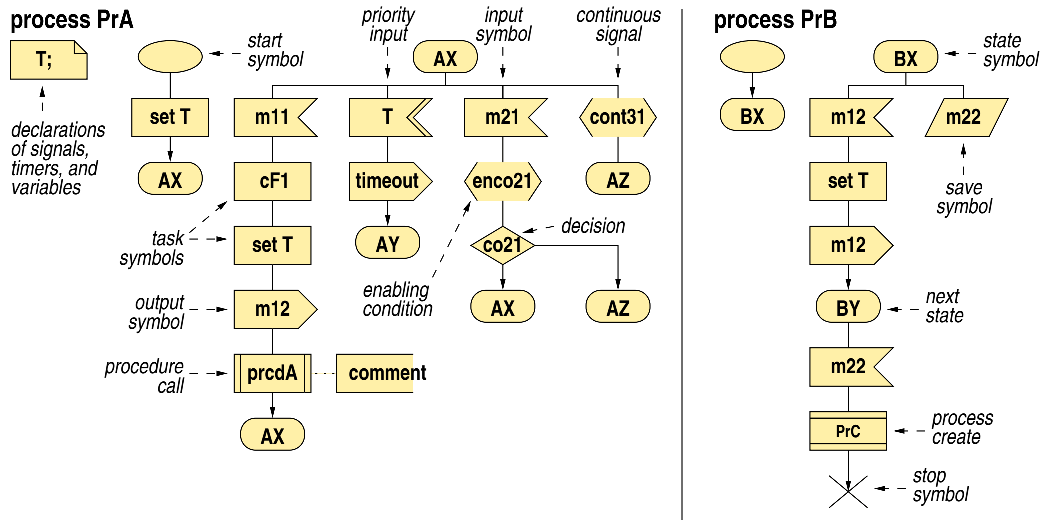

Figure 3.2: SDL Process Syntax (graphical representation)

are a means to express permanent surveillance of boolean expressions which may include external variables from within a SDL process. A transition is initiated when the input queue is empty and the boolean expression computes to true. Incoming signals take precedence over continuous signals.

Manipulation of local data in task symbols, setting and resetting of the processes own timer, and sending of signals may be done during a state transition in arbitrary order. Similar or equal parts of a state machine can be packed into a *procedure*. These sub state machines can be activated from within a transition. Furthermore, SDL's language includes a mechanism to dynamically create processes and provides a symbol for process termination.

Most of the advanced language features like *import/export* or *remote procedure calls* may be substituted through SDL macros, e.g. a synchronous protocol of sending and receiving signals between exporter and importer. SDL provides some shorthands like e.g. asterisk states ($*$ in a state symbol), or asterisk signals ($*$ in an *input* symbol). They can be resolved through an enumeration of all states respectively all signals allowed and known within this process.

3.1.2 Semantics

SDL's computation model can be described as follows. As already mentioned above, all processes run concurrently with equal priorities. They are activated on arrival of an incoming signal.

Communication between processes occurs on base of asynchronous signal exchange with non–blocking send but blocking receive. For this, each SDL process owns one single message queue. Incoming signals are sorted according to their arrival times (FIFO). The signal on first queue position will be processed first, i.e. if two or more signals (e.g. m_{11}

and m_{21} in Fig. 3.2) are available and several transitions may be enabled simultaneously, execution order relies on queue sorting. An exception to this rule may be realized through a *priority input*. With this, a later arrived messages can be preferred (timeout signal T is privileged compared to signals m_{11} and m_{21} in the example).

Neither process priorities nor signal priorities are included in the original Z.100 semantics. Sequence of process activations only depends on arrival order of incoming signals. Signal transmission via channels between blocks may be delaying, but signalroutes between processes deliver instantaneously.

SDL assumes a global system time that can be accessed through the operator *now* from every process. There may exist several SDL *timers* per process. They can be set and reset during state transitions. When a timer expires, a timeout signal is sent to the processes signal queue.

3.2 Abstractions and Non–Determinisms

SDL can be used during the early phases of a system's design cycle for simulation of even incomplete specifications. For this purpose abstract and non–predictable language features like *any decisions* or *spontaneous transitions* are supported.

If a transition including an *any decision* is executed, an arbitrary transition alternative will be chosen non–deterministically from the set of all possibilities. Using the SDL keyword *none* in an *input* symbol enables a SDL process to spontaneously trigger the appropriate transition without an available incoming signal.

These constructs are meant to be replaced when approaching the final implementation. They allow a stepwise refinement of the behavioural part of a system specification. This method of stepwise refinement has to be transferred to SDL's semantics. The computational model must be pinpointed for the final realization to circumvent non–predictable executions in the implementation and to allow an a priori analysis of the system's real–time behaviour. For this, we take a closer look on weaknesses of its timing and computation model before a predictable finalization of its semantics can be introduced in Sec. 3.3.

3.2.1 Notion of Time

Time consumption in processes, respectively state transitions is not clearly specified in Z.100 [ITU94a]. Its proposed interpretation of time depends on application domain, implementation architecture, or purpose of simulation. There are different timing models imaginable. Time may proceed only on expiration of a timer, or it may elapse exclusively in states and thus state transitions are atomic and consume zero time, or it may elapse continuously.[8]

The timing concept of a semantics has to fulfill two tasks. On the one hand, it should allow an implementation independent description of time progress during design and simu-

[8]First and third concept can be found as timing models in SDT's kernels for functional and wall–time (called "real–time") simulation (cf. Sec. 4.2).

lation. On the other hand, it should include possible timing behaviours of final realizations. Since our goal is to proof real–time characteristics of an application's final realization, we need a timing model, that allows to attach certain bounded (best case or worst case) execution times to processes, respectively state transitions. For that, we assume time progresses continuously and state transitions consume an arbitrary, but bounded and non–zero amount of time.

Often rejected for formal verification methodologies, this model can be said to be common understanding in system design and is also proposed in [Hin98b, Hin98a].

3.2.2 SDL Timers

An expired *timer* sends a signal to its own process using the same incoming message queue, normally sorted FIFO. Thus a timeout signal may be appended to already waiting signals. Even if the timer signal is privileged (e.g. through *priority input*), the point in time, when the process will consume this signal, can not be determined. Scheduling of concurrent processes, and thus preemption of the process to be activated by the timeout, will impose an unspecified jitter to this "wakeup" call. This means, even if it has been assured, the process to alarm is idle, duration of waiting time for the signal in the input queue would be unbounded. To emphasize this fact, it even can not be guaranteed, in a system with two processes alarmed by two timers, the more urgent timer signal will be consumed first.

Would process priorities help? Regarding the constellation above with several alarms to consider, for processes with lower priority, it could only be predicted, they will react within an bounded time *after* the receival of the timeout message.

As can be seen, without additional rules for process activation, timers are not applicable for the surveillance of any real–time attributes. This fact is not inherent to SDL systems only, but a problem of all asynchronous communication schemes. Proposed solutions include an "emergency timer" concept [BGK+00] or an exception mechanism for SDL (SDL'2000).

3.2.3 SDL Process Activation

Processing sequence of concurrent processes having all equal rights is unspecified in SDL. This can lead to the effect, signals will arrive in arbitrary non–predictable order at a receiving process. Unless all message sequences are taken into account in the receivers state machine (see process Pr_D in Fig. 4.3(a) on p. 39 as an example), a loss of signals is possible ("*implicit transition*"). To avoid this, signals can be deferred by means of a *save* statement.

In contrast to SDL *signalroutes* which transport signals without delay *channels* may be delaying. Detention of signals on a channel is bound but arbitrary, but order of consecutive signals is maintained. Furthermore, there is no means to broadcast a signal to several processes over block boundaries, making it impossible to determine a common state of a

whole system. For this, explicit synchronization by message exchange is necessary[9]. Since Statechart–like [HLN+90] behavioural concurrency *within* SDL state machines is not part of the language, the only means to directly access state data of parallel processes is to use *reveal/view*. This feature explicitly allows a process to share data with its competitors, but its use is not further recommended in SDL'92[10].

3.2.4 Atomicity and Run–To–Completion Semantics

A further point for discussion is the atomicity level of SDL state transitions. Z.100 [ITU94a] allows statement level, transitions level or no atomicity at all. For real–time applications, preemption is often needed to fulfill required timing constraints by minimizing task latencies. Perfect interruptability is also assumed by most real–time analysis algorithms. Since all data is local to SDL processes and access to a competitor's data has to be explicitly declared with *reveal/view*, thus warning the designer of possible data–inconsistencies, preemption may be allowed without further consequences (cf. Sec. 3.3.3).

Conclusion: SDL's asynchronous communication scheme and its non–privileged process activation rules impose non–determinisms on model behaviour. Predictability can be improved by adding priorities which originally are not part of the language, to SDL processes or signals[11]. Nevertheless, because of message overtaking effects correct behaviour of a functional model should not depend on sort order of incoming signals.

3.3 Finalization/Restriction of SDL´s Semantics

As explained above, signal or process priorities are used in an attempt to assure timely correct behaviour of a SDL system. In this section, an alternative execution scheme, "Scheduling with Message Deadlines" (MEDF) is introduced. Its applicability to SDL is shown with some examples. MEDF code generation is evaluated in Chap. 4, predictability of MEDF is proven in Chap. 5.

3.3.1 MEDF Semantics

An event triggered reactive system has to respond to an external stimulus within a specified deadline. This means, a chain of process activations inside a SDL system has to be processed and completed within a time span, the specified *end–to–end* timing constraint. This *relative* deadline is a non–functional requirement of the system specification and given through the *embedding* system to control.

[9] This again is inherent to all asynchronous communication paradigms.

[10] As will be shown later, use of *reveal/view* is needed for efficiency reasons

[11] SDT's C–Micro kernel provides support for process and/or signal priorities. Its code generator (cf. Sec. 4.2.2) evaluates `#PRIO` keywords attached via comments to process, respectively signal declarations.

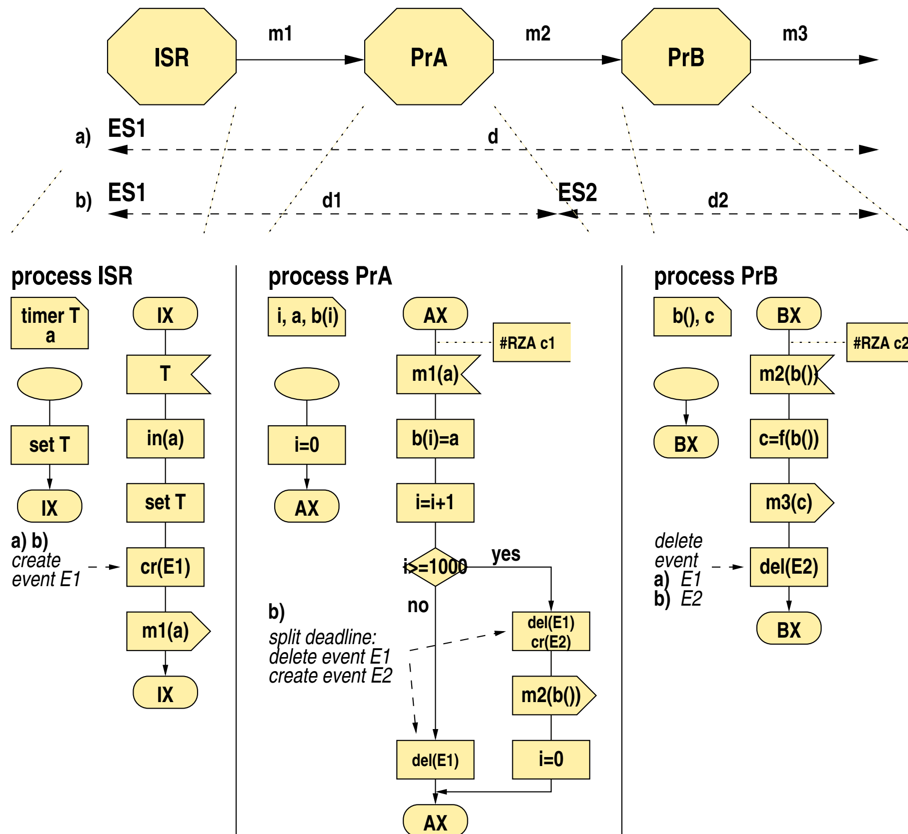

Figure 3.3: a) End–to–end Deadline vs. b) Triggering Less Stringent Processing

With occurrence of an external event an *absolute* deadline can be calculated and will be attached to signals coming in from the environment. A receiving SDL process adopts this deadline and is scheduled accordingly. In turn, outgoing signals get assigned this deadline and thus, successive processes will receive the same absolute deadline. As a consequence, all processes will be dispatched on an Earliest Deadline First (EDF) basis (cf. Chap. 5). To synchronize several incoming signals, a process' signal queue has to be deadline sorted. With this, signals may overtake each other, but as will be shown later, EDF queues are needed to achieve predictability, respectively minimize signal blocking times.

Internal events like timeouts are treated equally to external stimuli, i.e. *timer* signals get assigned their own absolute deadline. Depending on this chosen deadline value, timeout messages will be inserted according to their urgency into the process' signal queue.

To summarize, a deadline token (called "*EDF event*", represents an absolute deadline) is transported with each SDL signal. With signal receival, the consuming SDL process adopts the signal's deadline and operates on this new dynamic priority. Outgoing signals are passing the deadline token on to the subsequent process.

3.3.2 Scope of Deadlines

Deadlines normally span a whole task precedence system. On arrival of an external event, a deadline token is created and passed through a set of consecutive processes. After completion of the final computation in the last process within the communication chain, this EDF event would be deleted, if the deadline was reached.

In some systems, it is necessary to trigger a successive processing with less stringent timing constraint to relax the overall computational requirement. To demonstrate this case, the SDL model of an example application "collection of measurement data" is shown in Figure 3.3.

This very simple system illustrates the need to split up an end–to–end deadline (case a: deadline d) into several deadlines (case b: deadlines d_1, d_2) to achieve required real–time behaviour. Process Pr_A, which assembles measurement data (variable a) to a data array (variable $b(i)$) has to preempt process Pr_B to keep up with the stream of incoming data sent from the interrupt service routine (ISR). Functionality of the ISR which is shown in SDL notation for demonstration purposes only includes cyclic reading of an input value, creation[12] of EDF event E_1 and attaching the appropriate deadline (d in case a and d_1 in case b). With receival of a data block and thus receival of a new deadline, process Pr_B calculates the final result (variable c). After it is sent back to the environment, Pr_B concludes with deletion of the triggering EDF event (E_1 in case a, E_2 in case b).

If there exist non–time critical signals in a SDL system, appropriate EDF events, transported with these signals, will get assigned an infinite deadline.

3.3.3 MEDF Process Sequencing

SDL's Z.100 specification [ITU94a] states, all SDL processes will run independently and concurrently, as if each process is mapped to its own processor. This idealized assumption is now replaced through MEDF scheduling of processes. To better understand the necessity of run–to-completion semantics of state transitions, the following definitions for precedences and concurrency are introduced.

Definition 1 *Precedence*[13]*: If there exists a precedence constraint between two processes $P_1 \rightarrow P_2$, then execution of P_1 will have to be finished before activation of P_2.*

Definition 2 *Concurrency: Two processes $P_1 \| P_2$ will be concurrent, if there does* not *exist a precedence constraint (Def. 1) between them, i.e. if their computational results are independent of processing sequence.*

With Def. 2 and the assumed locality of (state) data in processes, one can easily follow, that state transitions may be preempted through *concurrent* processes only.

[12] For end–to–end deadlines, event creation and deletion may be completely hidden from the system designer, event and thus deadline transportation has to be undertaken by the run–time–system (cf. Chap. 5).

[13] This definition will be extended in Chapter 5 (Scheduling with Message Deadlines) to Def. 3 (Task Precedence System).

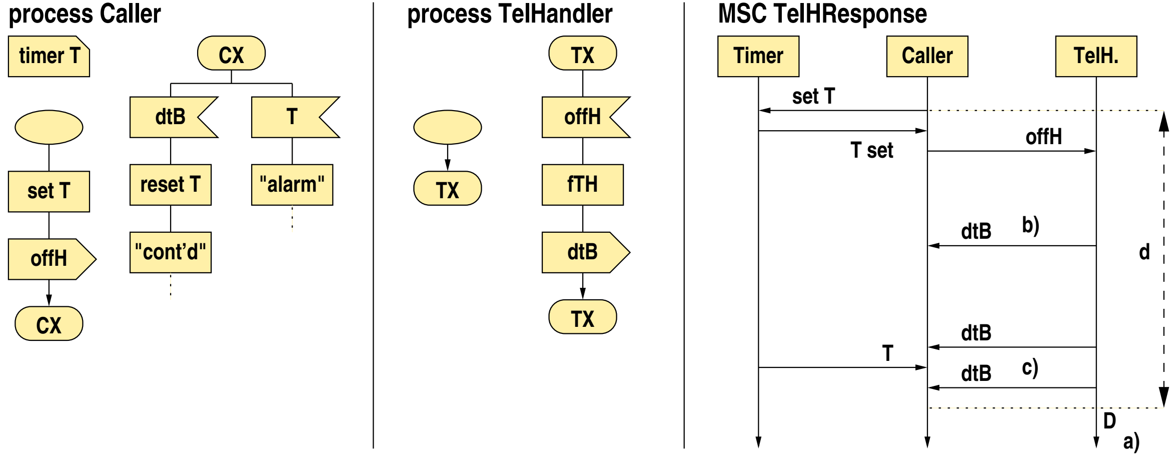

Figure 3.4: Timeout with MEDF Timer

Assuming a process network like $P_{11} \rightarrow P_S \rightarrow P_{13}$, and additionally $P_{21} \rightarrow P_S \rightarrow P_{23}$. Without MEDF and run–to–completion of state transitions, a signal from P_S to P_{13} may activate P_{13}, preempting P_S and leaving its state transition unfinished. A now arriving signal from P_{21} to P_S would either destroy the consistency of P_S state data or will have to be delayed until P_{13} has completed its computation.

As can be seen, run–to–completion of state transitions is necessary in processes serving different requests (signals) transporting deadlines originating of different triggering event types (later called "*server process*"). With MEDF, process precedence constraints (Def. 1) are resolved through process sequencing (cf. Lemma 1 in Sec. 5.2.2), thus a successive process will be only activated after completion of its predecessor.

The described phenomenon of overtaking SDL signals due to non–predictable process activations will not occur anymore. But with deadline sorted signal queues, MEDF allows and needs signal overtaking in message queues.

3.3.4 MEDF Timers

With MEDF semantics, *timer* signals get assigned their own deadline[14] and are inserted into the signal queue according to their (absolute) deadline value. Depending on the urgency of this timer signal, it will be immediately consumed by the process to alarm, but preemption through third party processes may cause a jitter comparable to normal SDL timeout signals.

To demonstrate consequences with an example, a simple telephone handler application [BGK97] is shortly analyzed (Fig. 3.4). There exists the following requirement: Process "TelHandler" has to respond within deadline d to incoming signal *offH*, process "Caller" uses timeout signal T to monitor a timely response of process TelHandler. For expressiveness reasons, a message sequence chart (MSC) is used to illustrate signal exchange between

[14]Deadline administration with timers is explained in Sec. 4.3.3.

Caller's timer, process Caller itself, and process TelHandler. We further assume, deadline $D_T = D_{Alarm}$ is attached to timer T and deadline $D_R = D$ is assigned to signal *offH*.

With these assumptions, different behaviours of the system are imaginable:

a) TelHandler has not achieved to complete its computation and to respond before time D and additionally Caller has not been activated: A deadline violation occurred and will have to be handled by the exception handler of the run–time system.

b) TelHandler responds in time and Caller resets timer T: Everything went well.

c) Timeout signal T and response dtB are sent nearly simultaneously.

To be able to undertake "alarm" actions in Caller, one has to choose the deadlines $D_T < D_R$. Otherwise, if $D_T \geq D_R$ execution order will be forced to be TelHandler *before* Caller and the timeout signal will be ignored. This means, Caller must be released before time D is reached to react to signal T, thus stealing TelHandler processing time to fulfill its task in time. Considering the worst case, TelHandler may complete its computation and respond with signal dtB, but dtB will be appended behind signal T in the signal queue of Caller. This shows, that timeout time and timeout deadline must be chosen wisely.

3.3.5 Discussion of further SDL Language Constructs

The finalization of SDL's semantics to MEDF influences the use of some language features of SDL. Restrictions and changes are discussed below.

SDL–88

priority input : This preferential treatment of some signals according to their importance regarding the processing sequence within a SDL process is not needed anymore since MEDF signals are sorted into the incoming message queue according to their transported deadline.

save : SDL provides the *save* mechanism, to avoid implicit transitions and thus signal loss caused by unexpected signal arrival. But since storage time of a signal *save* can not be bounded without knowledge of all timely inter–dependencies of incoming signals, this language feature is forbidden with MEDF.

enabling conditions : If a transition is guarded by an *enabling conditions* which calculates to "false" on arrival of a triggering signal, the signal is saved. Thus, this construct must also be precluded (→ *save*), at least with this model of computation.

continuous signals allow the surveillance of conditions of variables and will trigger a state transition, if the condition evaluates to "true". In case of an external variable, i.e. if the variable change is caused by the system environment, an additional timing constraint has to be specified to describe the temporal behavior of this trigger.

reveal/view provide a means for unsynchronized data access to variables in concurrent processes. They can be seen as a form of declaration to remind the system designer that data consistency may be at risk. Usage is unrestricted under MEDF.

channels: An arbitrary retardation can be attached to a SDL *channel* in its original meaning. With MEDF, *channels* are non–delaying.

SDL–92

With this 1992–version of SDL, object–oriented concepts were introduced into the language. Most of these language constructs can be understood as a means to further organize the system structure of a SDL system. As long as these features are not resolved at run–time, i.e. if they can be substituted by basic SDL language, their usage is allowed further on. Language constructs like multiple process instances or dynamic process creation by instantiation, anyway not recommended for design of real–time systems, have to be deferred due to their unpredictable behaviour.

3.4 Non–functional Specification

A reactive (event–triggered) real–time system can be generally defined as a system that has to respond to certain external and internal stimuli (state changes) within given timelines. As can be seen, beneath a functional description of the system, a specification of the timing characteristics of the embedding environment as well as of the application part itself is needed to verify that required timelines will hold.

3.4.1 Timing Constraints

A non–functional specification of the embedding system consists of deadlines (i.e. maximum allowed response times), event streams (i.e. a description of the timely behaviour of system stimuli), and finally event dependencies.

Deadlines (d) determine the relative available amount of time for an application to respond to a certain stimulus. They have to be specified for external signals as well as for internal timers.

Event Streams (ES) characterize the temporal behaviour of the embedding process. They are a formal means to express the maximum possible number i of events of a certain type within any arbitrary interval a_{i-1} that will reappear at most with the cycle time z_i [Gre93b]. An event stream consists of at least one or more event tuples $(a_i\ z_i)^T$.

Considering the occurrence of events on the left side of Fig. 3.5, the resulting ES is shown on the right. There are no two simultaneous events in the example, but since at least one event must occur within an instant, the interval in the first event stream tuple

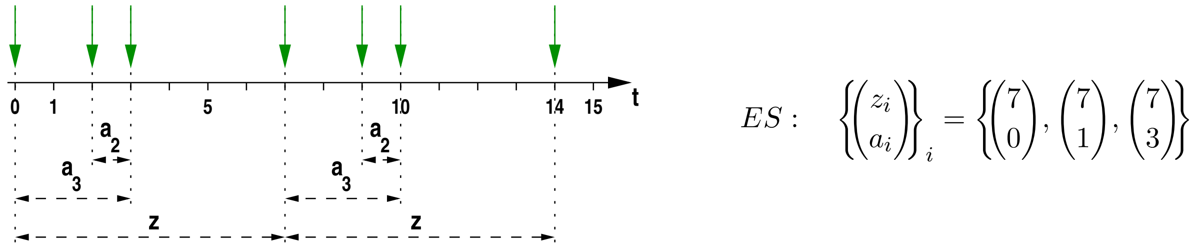

Figure 3.5: Event Stream Example

must be $a_0 = 0$. A maximum of two events occurs in an interval of length one time–unit ($a_1 = 1$) and a maximum of three events in interval $a_2 = 3$. All event occurrences are repeating with a cycle time of $z = 7$.

As with deadlines, ES specifications have to be added to all stimulating external events, i.e. SDL signals coming from the environment, and to all stimulating internal events, i.e. to all timers.

Event Dependencies specify the minimal distance between occurrences of events of *different* types. For each group of dependent events an Event Dependency Matrix EDM may be additionally specified.

$$\mathbf{EDM} = \begin{pmatrix} ed_{11} & ed_{12} \\ ed_{21} & ed_{22} \end{pmatrix}$$

Each element e_{kl} of the EDM denotes the minimal temporal distance between an occurrence of event of type k and a subsequent occurrence of event l.[15]

3.4.2 Extensions to the Functional Model

Further extension to the functional model include the specification of execution times, limits for cycles in transitions or signal loops, process priorities, and system modes.

Minimal and Maximal Calculation Times (c_{min}, c_{max}) describe the expected amount of computation time a processor will need to execute a given task. To simplify the specification process, calculation times can be added to a whole SDL process and thus are valid for all transitions *and* the initial receive. Additionally calculation times can be attached for modeling refinement to single state transitions or even to parts of a transition.

Transition and Signal Loop Bounds (l) define a minimal and/or maximal allowed limit for either loops in transitions or for signals that are sent back to predecessor processes in a precedence system. The lower limits are needed to calculate earliest possible start times of server processes, the upper bounds are used to derive the worst case computation times.

[15] ed_{kk} is the distance between two events of the same type, thus is equal to a_1 given in its event stream.

Priorities (p) are used in a MEDF SDL system specification to divide the system model into hard and soft real–time application parts. Only processes, that execute on a pre–defined hard real–time priority level are scheduled and analyzed using MEDF. Processes running on levels below this priority are considered to be soft real–time.[16]

Timing constraints and model extensions are added to process and signal declarations or are directly specified in SDL *task*–symbols within state transitions. For this, a new keyword `#RZA` is introduced, such that annotations can be embedded in SDL comments within the functional SDL model. An application example showing the use of annotations during the design flow is given in App. A.1. A summary of all possible annotations can be found in Table A.1.

Operation Modes

Real–time analysis (cf. Chap. 6) can not declutter all possible state combinations in all concurrent processes to evaluate the worst case situation of a whole system. A methodology that follows such a concept, e.g. simulation verification or model checking, faces the state space explosion problem [SBMJ01]. Instead, the proposed analysis algorithm relies on an abstraction of the SDL system. A worst case path within a precedence system of SDL processes is evaluated and considered to be the response to an external or internal event. The algorithm then assumes a critical instant of all stimuli to calculate the worst case processing load. Because of this, the analysis result may be very pessimistic, but will always cover the worst case. To weaken this effect, the system designer has several different possibilities:

1. Breaking up deadlines, and specify additional event streams at the beginning of each new deadline (Sec. 3.3.2 and Fig. 3.3).

2. Specification of additional event dependencies, e.g. to take into account setting of *Timers* in state transitions triggered by an external signal.

3. Consideration of mutual exclusive operation modes.

Operation modes take into account that there may exist different processing conditions for the SDL system, either because groups of external or internal events will not occur in the same situation (captured by event dependencies) or groups of state transitions will only be executed in different system scenarios (e.g. during system initialization, normal mode, failure recovery, panic, . . .). To recognize operation modes during the derivation of the worst–case task precedence graph, mode identifiers could be as well annotated to SDL processes and/or transitions. The concept of operation modes is evaluated among other things in Chap. 7.

[16] Tasks on a priority higher than MEDF priority resolve supplementary functions, e.g. the Timer Task. These priorities are not allowed on specification level.

Chapter 4

Code Generation

SDL processes with their explicit parallelism are used for structuring the system under development into functional units, i.e. capsuling functionality into independent design problems. On the other hand, concurrency in the implementation is often needed to achieve real–time performance, e.g. for minimizing task latency or for synchronizing tasks with different, non–compatible timing constraints (e.g. double buffering of data in exercise "Collecting Measurement Data", [MB00]). Unfortunately, granularity as well as structure of concurrent units may differ in system model and system realization. Automatic mapping is made difficult through this fact and therefore is the focus of attention in this chapter.

Code generation is one step within software automation design process and has to fulfill different, often diametrical tasks:

1. Preserve semantics of specification language. Computation within implementation must be the same as execution order specified (and probably simulated) with system model.

2. Provide real–time behaviour and allow timing constraints specified to be met. For this purpose, incorporated overhead needed to synchronize between concurrent processes should be minimal, i.e. code generation must be efficient.

3. Scheduling scheme employed must be analyzable.

4. Generated code should be "human readable" and easy to understand, since test and debugging will be undertaken after target integration and generated code will be used as source[1].

This chapter deals with following topics: At first, possible decomposition principles of system descriptions based on process models are analyzed. In a next step, state–of–the–art generator tools are evaluated and classified accordingly, before code generation strategies for MEDF semantics are presented finally.

[1]Because of its automatic generation, design changes will effect only the system specification. Therefore, maintainability of target code is not the main focus anymore.

4.1 Decomposition Principles

> "Owing to a semantical gap, this translation [to the process model] can be a serious source of inefficiencies during system design and substantially complicates software maintenance later on." [Mok83]

There exist some tradeoffs, which are inherent to decomposition techniques of system descriptions, based on concurrent processes.

Maintainability vs. efficiency tradeoff: A highly efficient system realization may abandon structuring concepts as introduced by the specification language to meet stringent timing requirements. For example, giving up SDL's process capsuling will lead to "unreadable" and therefore unmaintable code. But, e.g. replacing SDL's (asynchronous) message exchange through (synchronous) procedure calls will minimize execution times and help to fulfill timing constraints.

Amount of computation vs. amount of communication tradeoff: Depending on the system's properties, i.e. amount of computation to perform compared to its communication and synchronization effort included, one of the proposed strategies will fit best, i.e. there won't be no overall best strategy for a system's decomposition into concurrent tasks.

Common to all strategies is the minimal granularity used for mapping. Due to the finite state machine characteristics of SDL processes, the obvious rule for system decomposition is to always keep one state transition unfragmented. Depending on the strategy, several transitions will be merged into one task or one process will be split up and its transitions will be integrated into several different concurrent tasks.

The classification below follows Mok's decomposition ontology [Mok83], but takes special care of SDL's communication and computation concepts. To commemorate, SDL's messages are used for both, triggering consecutive operations as well as synchronizing mutual exclusive executions (monitor concept).

The notion "process" is used for a functional unit within the specification language (e.g. SDL process), the term "task" denotes an executional unit within the implementation (e.g. RTOS task), whereas "thread" describes a clustering concept for consecutive computation steps (e.g. activity thread). Actions in SDL state transitions are specified in so called "SDL *task* symbols" in contrast to simply (RTOS) "tasks".

4.1.1 Decomposition by Maximizing Parallelism

Decomposition policy of this straight forward principle is to assign one separate task to each SDL process. Since SDL server processes will be mapped to their own tasks, no additional synchronization mechanisms are required in the implementation.

In case of a multiprocessor platform, this strategy will achieve the most stringent timing constraints[2], because of execution will benefit from application's inherent concurrency. But on a single processor architecture as assumed in this thesis, both types of synchronization concepts, i.e. triggering of the next execution as well as synchronizing mutual executions, will be included in the final realization, bringing in some probably avoidable overhead.

The resulting task architecture will reflect the SDL design with its process structure, and therefore will be easy to understand and maintain.

4.1.2 Decomposition by Timing Constraints

A task is built of a sequence of function calls, each of them representing one single state transition of one SDL process.

Calling sequence of state transitions inside a task is derived from the data path between input and output actions, starting with one triggering input signal. Following send/receive pairs within the SDL system, state transitions, which perform the output to the environment conclude execution of the task. Since state transitions within SDL processes have to share common (state) data, but may be realized through different tasks in the implementation, state variables will become non–local to the corresponding tasks. Therefore, data integrity has to be assured through additional synchronization mechanisms, like semaphores or server queues.

This technique abandons SDL's message exchange for triggering the activation of consecutive processes and resolves SDL's process structure along end–to–end timing constraints. With this, functional capsuling will be lost and the resulting software architecture won't reflect the design problem anymore.

4.1.3 Minimizing Interprocess Communication

Server processes in SDL are a modeling concept used to monitor requests with different computational requirements to shared executions or devices (see Fig. 4.1). The use of additional communication mechanisms (like semaphores or server queues) for resolving this synchronization problem in the implementation can be avoided through the integration of several competing computational requirements into the same task.

Prerequisite for this are compatible timing requirements, i.e. processes have to be activated with same periods[3] (and same activation times) or periods are multiples of a common unit. The task, performing the computation of all processes merged, will be triggered with the greatest common divisor of all periods. In the latter case, scheduling decisions must be generated into the implementation task, since with each activation not all program paths (state transitions) have to be executed, i.e. with every computation cycle, some program paths, respectively state transitions have to be skipped. In case of state transitions belonging to a server process, sequence of their activation has to be determined manually or by

[2]Assumption: Computation predominates communication times.

[3]Assumption: Deadline is equal (or in same order of magnitude) to period.

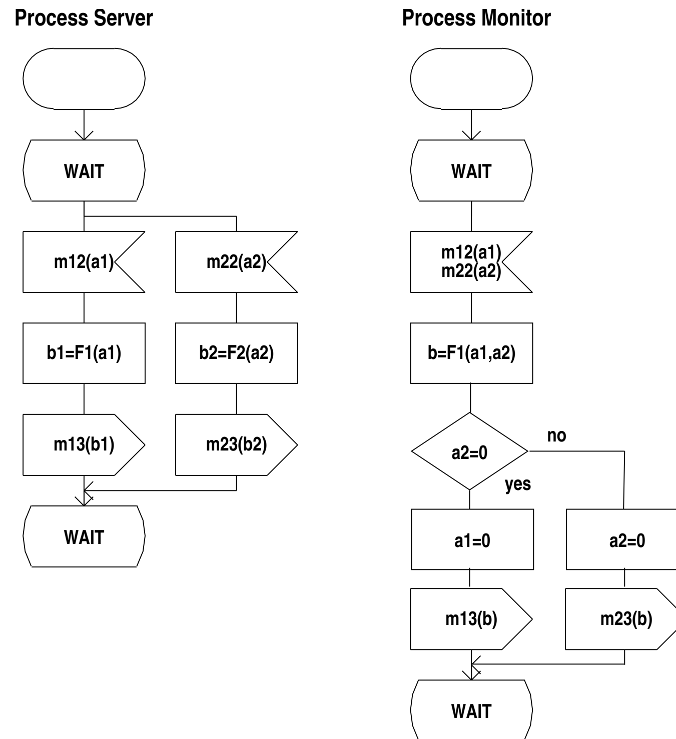

Figure 4.1: a) Synchronizing Mutual Exclusion and b) Monitoring Shared Function

random. If a server process contains only one function, triggered by several different events, with this strategy multiple executions of this monitored function can be economized.

Replacing sporadic computational requirements through their equivalent periodic ones (period will be the same as the smallest minimal activation interval), this technique can be assigned to a whole SDL system. With the consequence, the complete system will be represented through one single task in the final software architecture.

Task latency and thus reaction time, i.e. time till the appropriate state transition for one input event will be activated, will increase.

Conclusion

Each of the decomposition principles has its advantages and drawbacks, depending on system's properties and its timing characteristics. This is summarized in the following:

- Server state machine structure (alternative state transitions with different functionality in each transition, but to be synchronized timely, cf. Fig. 4.1(a)) vs. monitoring shared execution (common functionality, cf. Fig. 4.1(b)): Elimination of redundant computation (Sec. 4.1.3) as deduced in [Mok83] will only occur in the second, the monitoring case, i.e. the body of a monitor process will only be executed once with each activation. This means, an efficiency improvement can only be realized, if there

exists any functionality to be duplicated unnecessarily with strategy "Decomposition by Timing Constraints" (Sec. 4.1.2).

- Strategy "Minimizing Interprocess Communication" (Sec. 4.1.3) is hard to automate, because scheduling information has to be "hand–coded" into the task's body.
- As mentioned above, non–compatible timing constraints and a design problem, which can not be pipelined as described in Sec. 4.1.3 necessitate the straight forward strategy "Maximizing Parallelism" (Sec. 4.1.1) to minimize task latencies.

4.2 State–of–the–Art Code Generators

Telelogic's SDL Design Tool SDT provides different code generators, C–Basic/C–Advanced and C–Micro. These code generators are dedicated for different purposes. On the one hand, for a multitude of simulation techniques, e.g. functional, simulated time, wall time[4] (SDT notion: "real–time simulation") and performance simulation, but on the other hand can be used for target integration as well. With the latter aiming on different targets, embedded as well as general purpose processors.

4.2.1 SDT's C–Basic and C–Advanced

Newest tool release (since SDT Rel. 4.0, [Tel01]) does not distinguish anymore between these two code generators, and therefore the term C–Advanced is used for both tools in the following.

Light Integration Model: The complete SDL model will be integrated into one single operation system task (e.g. an Unix process) for on–host simulation, respectively one single RTOS task on an embedded system. Mapping principle follows "Maximizing Parallelism", since each SDL process will be assigned to its own so called PAD function (Process Activity Description).

The "Light Integration" is based on SDT's proprietary kernels, which implement one single process ready queue for the whole SDL system. Sort order of this ready queue may be FIFO or priority based. Following SDL's rules for process activation, with each incoming, a process readying signal, the appropriate PAD function is called. This results in non–preemptive execution of state transitions.

Since the whole SDL model will be processed in a single main loop, thus polling the system environment, the resulting realization resembles a model interpreter. The main loop consists of several consecutive steps, beginning with an initial call to an input function *xInEnv*, which feds triggering signals into the SDL system part on an environment's state

[4] The notion "wall time" describes the concept of synchronizing an internal system model time to the "real" time of the system's environment. The later may be varying, e.g. the simulator environment (UNIX host), the target's RTOS clock, ...

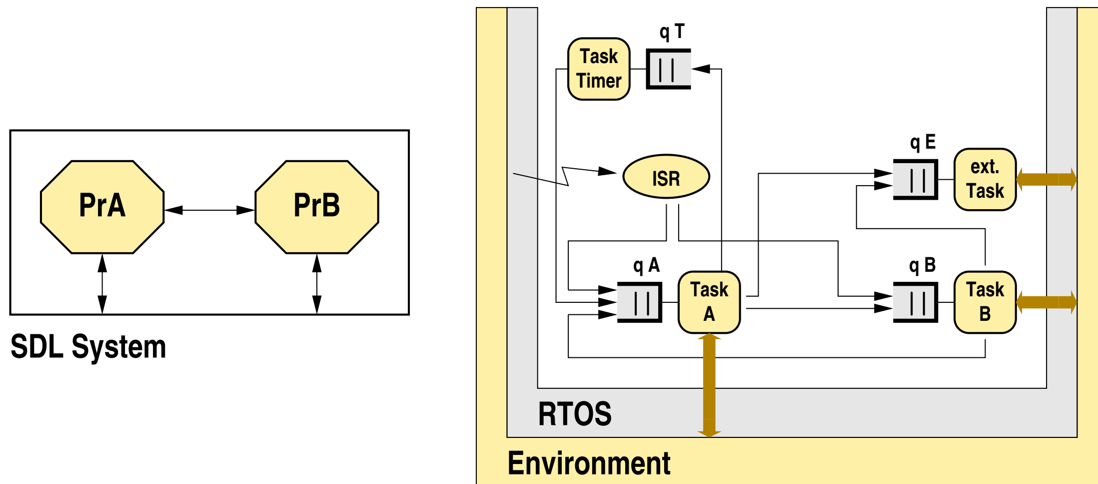

Figure 4.2: a) Simple SDL System and b) its Tight Integration with SDT's C–Advanced

change. In a next step, it will check timers for timeouts. Thereupon the appropriate PAD functions of processes, which are currently ready to run, are called. Depending on the input signal, the corresponding state transition is processed. Output operations are piped through a specific *xOutEnv*–function. Thus, *xInEnv* and *xOutEnv* are building the interface to the system's environment in case of a target integration or to the simulator's user interface (through SDT's postmaster tool).

Tight Integration Model: Again principle "Maximizing Parallelism" is applied, but with a "Tight Integration" (cf. Fig. 4.2) each SDL process is mapped to its own RTOS task and an additional RTOS message queue, thus leaving scheduling and process, respectively task activation to the employed RTOS kernel. With this, many different scheduling schemes are available, but mostly preemptive priority scheduling, with process priorities as specified with a `#PRIO`–directive in the SDL system model, is used. Depending on the scheduler, preemptive as well as non–preemptive execution of state transitions is possible.

Now, a PAD function, i.e. the body's code representation of a SDL process, consists of an eternal loop with a releasing blocking initial receive. SDL semantics is preserved through non–blocking RTOS send and the aforementioned blocking RTOS receive directives.

SDL timer services are assigned to an additional timer task (see Sec. 4.3.3 for details). Interfacing to and from the environment is directly done through RTOS send and receive directives. One proposed implementation suggests a polling external task, which again translates environmental state changes into input signals and performs output actions triggered by output messages. Alternatively, ISRs for input operations and input signaling can be used.

4.2.2 SDT's C–Micro

Restricting SDL to a language subset, the C–Micro code generator mainly aims on embedded targets. Generated code is again built upon a proprietary kernel, the so called C–Micro kernel. Instead of a process ready queue, SDL process scheduling is realized in this core with one or more signal queues, one signal queue for each process priority level, whereas signal queues may be sorted FIFO or by *signal* priorities and process priorities take precedence over signals.

If preemption is enabled, an output statement in a lower priority process' state transition to a process with higher priority will then cause the execution of this transition to be interrupted immediately. Given that a combination of both, signal *and* process priorities is allowed, the resulting scheduling scheme and thus execution order of SDL processes is not predictable.

Two integration models are supported in the moment, "Bare" and "Light Integration", a "Tight Integration" as introduced above is announced for future tool releases. Interfacing to the environment with a Bare Integration, i.e. without run–time support of an additional RTOS is done through calls to *xInEnv* and *xOutEnv* functions, thus polling the environment as explained above or through direct send into the global signal queues (*xMK_Send_Env()*) from within ISRs. Implementations based on a Light Integration, i.e. whole SDL system is processed by one single RTOS task, may be additionally connected through RTOS send or receive directives to the environment.

Since one SDL process is represented through one PAD function in the final code, mapping principle follows again "Maximizing Parallelism".

Conclusion

Both code generators, though employing different scheduling kernels, always apply generation principle "Maximizing Parallelism". Efficiency improvement with C–Micro is gained through a reduction of SDL's language concepts to a smaller subset. For example, a per run–time determination of a receiving SDL process (function *xFindReceiver* in C–Advanced code) is forbidden with C–Micro. Beyond this, C–Micros possibly non–predictable execution scheme violates SDL's semantics. C–Advanced code is in conformance to SDL.

4.3 Code Generation for MEDF Semantics

As outlined earlier, beneath efficiency the preservation of SDL's semantics has to be the main focus of automatic code generation. Following the definition of concurrency (Def. 2) and execution precedences (Def. 1) in Chap. 3.3, MEDF processing can be summarized as follows: A deadline token is transported with each SDL signal. With signal receival, the consuming SDL process adopts the signal's deadline and operates on this new dynamic priority.

4.3.1 Server Model

The Server Model (SM) code generation strategy follows as well principle "Decomposition by Maximizing Parallelism". Thus, this implementation technique maps each SDL process instance to one RTOS task (e.g. SDL process Pr_A to RTOS task A in Fig. 4.2), whereby each RTOS task will additionally get assigned its own (incoming) message queue (q_A). Although preemption of state transitions through a parallel executing task will be allowed, execution order and consequently SDL semantics are preserved by the ordering of messages in the process' queue. This means, as every SDL state transition belongs to exactly one RTOS task, the execution of state transitions is implicitly serialized (a task may not preempt itself). Together with the fact, that there exist no shared variables between the SDL processes and consequently the RTOS tasks, this avoids the necessity for explicit synchronization, which is in contrast to the "Activity Thread Model" (ATM, Sec. 4.3.2).

With MEDF, task preemption will only occur, if a *parallel* task, respectively SDL process will be activated with a more urgent deadline. In a precedence system, all consecutive tasks will have the same deadline (they are processing the same deadline token, transported by the appropriate SDL signals) and therefore state transitions of SDL processes building a precedence system will run to completion. This fact is independently true for the case a transition includes a SDL *output* statement before a SDL *task* statement, which is in conformance to SDL's semantics.

Server processes with MEDF semantics synchronize between messages originating of triggering events of different types. With this, signals in the incoming message queue have to be sorted according to their deadline value. Beyond this, it has to be guaranteed, that blocking times of signals of different types are bounded, presuming a priority inversion avoidance strategy with MEDF. This execution scheme and thus MEDF task scheduling has to be assured through the applied RTOS kernel: The (absolute) deadlines, transported by messages, have to be adopted by the respective RTOS task *in the moment of the signal's arrival* at the incoming queue (cf. Sec. 5.3).

With a modified macro package for SDT's C–Advanced "Tight Integration" [Lar98], an MEDF kernel could be easily included into SDT's code generation process. Using SDT's `#PRIO` and `#CODE`–directives in the SDL system model only to adjust the application for MEDF usage[5], deadline transportation is completely hidden in the functional specification. A more detailed but simple example of the code synthesis and build process can be seen in Appendix A.2.

With MEDF, not only external events, but internal events like timeout signals as well, get assigned their own deadline. Thus SDL's timer service is realized through a special timer task, which has to administrate signal deadlines in addition to normal timer operations (Sec. 4.3.3).

Advantages of the SM implementation technique include the improved testability and maintainability — the software structure resembles the structure of the specification — and the availability of code generators in state of the art CASE–tools.

[5] Only deadline creation and set of task and message queue creation parameters have to be specified explicitly

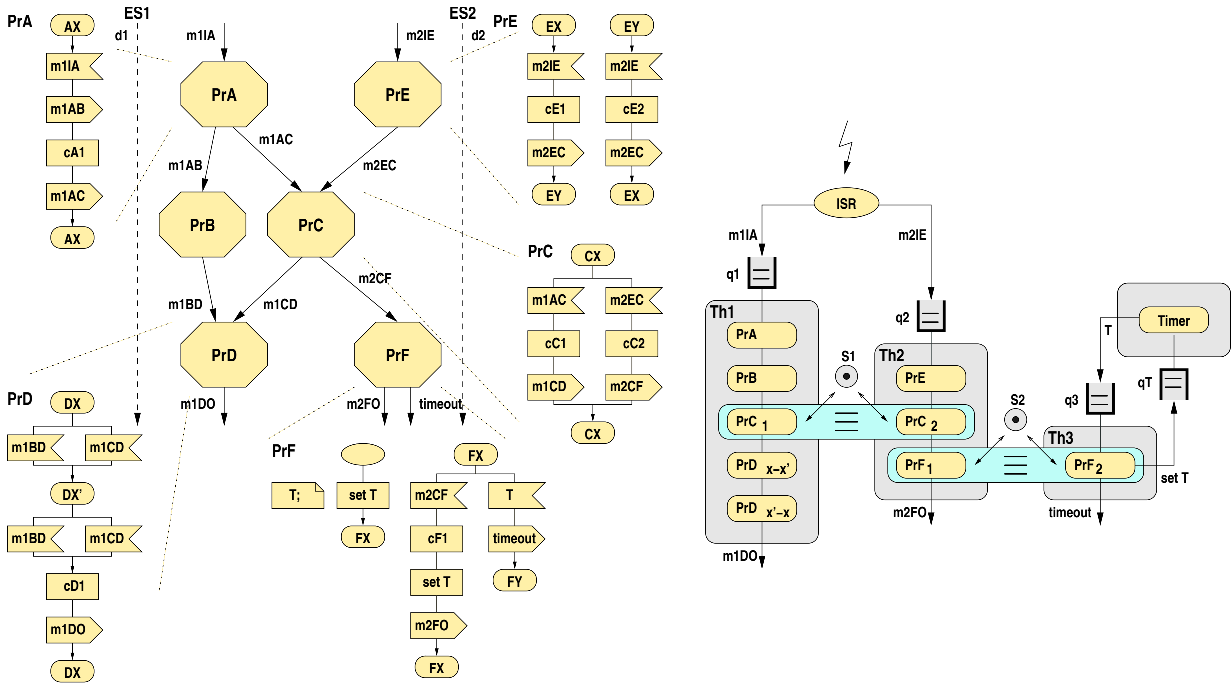

Figure 4.3: a) SDL System with Two Timing Constraints and b) its Activity Threads

4.3.2 Activity Thread Model

Based on principle "Decomposition by Timing Constraints", granularity of concurrency in an Activity Thread Model (ATM) implementation corresponds to the scope of an end–to–end deadline specification. This means, one signal path, responsible for transporting a different event type through a SDL system (the thread of activity, AT) is mapped into one single RTOS task (e.g. Pr_A, Pr_B, Pr_C, and Pr_D of ES_1 in Fig. 4.3(a) are mapped to Th_1 in Fig. 4.3(b)). That part of a SDL process, handling the same message type, is implemented by its own procedure (e.g. SDL processes Pr_A and Pr_C in Fig. 4.3(a) are mapped to the procedures Pr_A and $Pr_{C,1}$ in Fig. 4.3(b)). One single SDL state transition is executed per procedure call, depending on the signal's values, respectively on the SDL state information (e.g. either Pr_E's transition $EX \rightarrow EY$ or $EY \rightarrow EX$ will be processed). Signal communication within ATs, i.e. the sequence of *output* → *input*, *output* → *input* statements, is replaced by a sequence of procedure calls.

Only one event at a time can be handled per AT, therefore consecutive burst events, transformed by an ISR to a message, have to be saved in a FIFO message queue at the thread's beginning (Queues q_1 and q_2 for threads Th_1 and Th_2 for two different kinds of external events in Fig. 4.3(b)). With means of MEDF scheduling, the complete RTOS task representing an AT "inherits" the absolute deadline from the incoming event.

This straight forward mapping algorithm will fail, if the SDL process includes either several *output* statements in one transition[6] or any *output* statement before a *task* statement

[6] Send order of signals in a SDL *output* statement is defined as "arbitrary" in SDL's semantics and

(see transition $AX \rightarrow AX$ of process Pr_A in Fig. 4.3(a)). To guarantee the "run–to–completion" requirement of SDL's semantics, procedure calls of subsequent processes in one AT have to be delayed until the transition currently executed has been finished. To do so, new signals to send, their receivers and parameters transported will be appended at run–time to a FIFO sorted procedure list. Thus, each AT administrates its next procedures to call by himself.

This mechanism is similar to SDT's C–Basic Light Integration scheduling scheme (one process list for the whole SDL system, cf. Sec. 4.2.1). Managing execution order with a FIFO procedure list is likewise the principle applied in the "Basic Activity Thread" implementation scheme in [HKMT97], but again the whole SDL system will be represented with one single AT, leading to large latency times for events with short deadlines.

Observing the resulting calling sequence in thread Th_1, procedures Pr_A, Pr_B, $Pr_{C,1}$ will be activated once (order as displayed in Fig. 4.3(b)), finally procedure Pr_D will be called *twice*, executing the transitions $DX \rightarrow DX'$ and subsequently $DX' \rightarrow DX$. The latter effect is caused by the join of messages m_{BD} and m_{CD} in SDL process Pr_D. As can be seen, this execution scheme is equal compared to the order of computation of single transitions within a MEDF server model implementation. This is due to the fact, that succeeding SDL transitions within a precedence system have the same deadline.

Simultaneous processing of different event types leads to parallel execution of several ATs, therefore state information and internal variables of SDL processes, serving messages of different event sources (Fig. 4.3(b): $Pr_{C,1}$, $Pr_{C,2}$), have to be protected with a semaphore (S_1). By this means sequencing of transitions can be enforced, i.e. preemption of their procedures by transitions of the same SDL process is avoided. To preserve MEDF execution order, ATs blocked on such a semaphore have to be inserted on a deadline sorted task wait list. Analogous to the Server Model, deadlines in this regions of mutual exclusion have to be modified (e.g. deadline inheritance, deadline ceiling) to get rid of priority inversion (cf. Sec. 5.2.4).

Timers are internal events, whose behaviours are described with own event stream specifications. Therefore a timer signal is the triggering source of an AT of its own, i.e. it will hold its own initial message queue (e.g. Queue q_3 in Fig. 4.3(b)). Other SDL language constructs are either not applicable (analyzable) for hard real–time systems (e.g. signal *save*, see Sec. 3.3.5) or have to be translated to their own AT (e.g. cyclic polling of *continuous signals*) as well.

therefore may be non–deterministic. But on the other hand, send order will be pinpointed through the code generators' parser. Regarding a transition like $AX \rightarrow AX$ (Pr_A in Fig. 4.3(a)), but sending signals m_{1AB} and m_{1AC} in one single *output* statement instead, and a join in process Pr_C: Only the specification has to take care of both possible receive sequences of incoming signals. In the implementation, there will be both state transitions included, but because of the specification's interpretation through the code generator only one transition alternative will be executed in all cases. The superfluous transition may reside in the code but will never be processed.

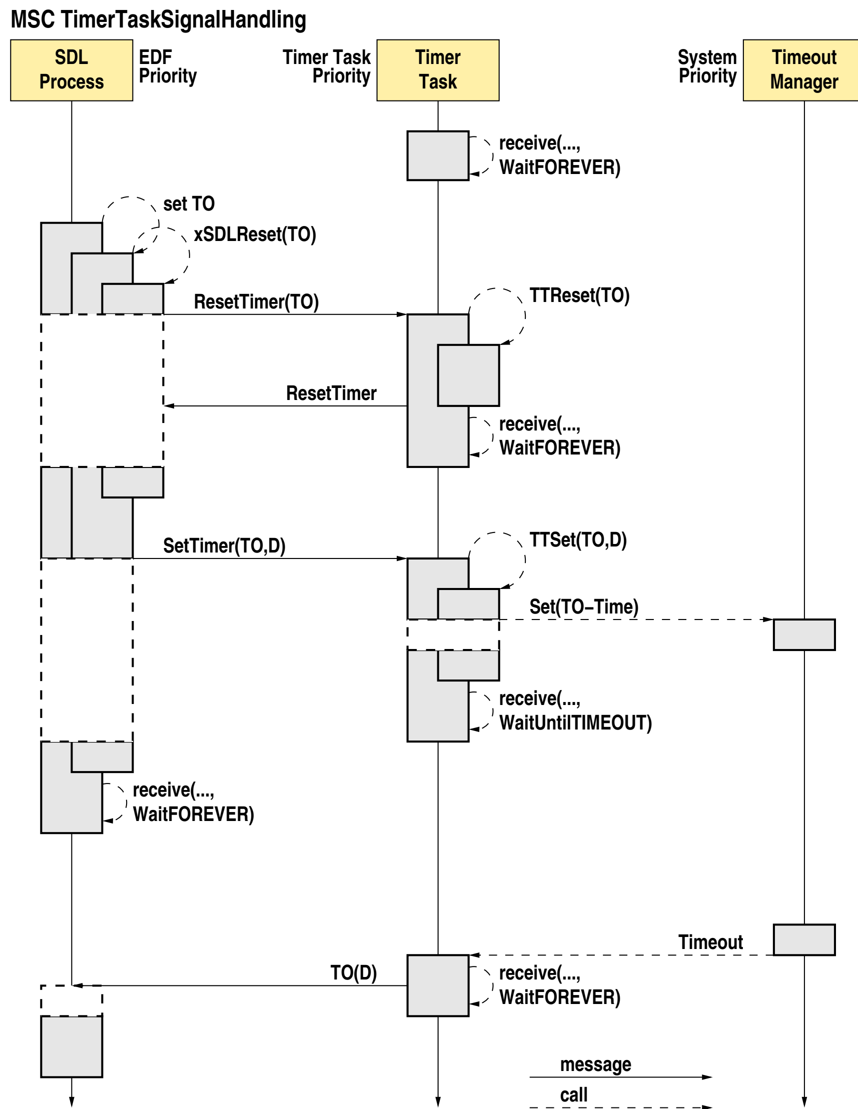

Figure 4.4: Timer Task: Signal and Timeout Handling

4.3.3 System Time and MEDF Timers

SDL's system time, which is expressed through *now* in SDL, has to be connected to the so called *wall clock* [Tel01, Chapters 57, 68]. This wall time is often established through a clock manager within the underlying RTOS kernel. For instance, in VxWorks or RTEMS the clock manager increments a tick counter based on a periodical timer interrupt, thus providing equidistant ticks[7].

Timer service requests will be translated to a protocol of message exchange operations between a client SDL process and a timer task (cf. Fig. 4.4). Therefore, the timer task

[7]Macro `SDL_DURATION_LIT(...)` maps the SDL time to the local RTOS representation of time in SDT's code generation.

will have it's own timer queue (see q_T in Fig. 4.3(b)). At first, SDL semantics requires to reset a formerly set timer (signals `ResetTimer` to and from timer task; includes a removal of a formerly set and already fired timeout signal in the incoming queue of the client SDL process). Subsequently, a send to the timer task with a *set(Timer-x)* request follows and completes the set timer operation. When this timer expires, the timer task will respond to the request with an appropriate timeout *Timer-x* message.

For this, the timer task administrates a list of active SDL timers. This timer list is sorted according to timeout times. Beneath the timeout time and the client's address, each timer list entry now additionally stores a pointer to an EDF event (and thus is linked to a belonging deadline) to implement MEDF timer signals.

Each timer service request transports an EDF event in its data buffer. Since the timer task is a non–EDF task, the deadline of the event transported will be ignored by the scheduler. Timer task priority is higher than EDF priority. Thus servicing a timer request or initiating a timeout are timer task operations that can not be preempted through further MEDF tasks, respectively through further SDL processes.

Timer task structure as can be seen[8] in Fig. 4.5 is derived from SDT's C–Advanced Tight Integration Macro Package [Tel01] and extended with EDF event management for timer deadlines. To avoid periodical activations of the timer task to check for an expired timer, timeout surveillance is realized in the `message_queue_receive` operation[9]. This initial receive solves two tasks. On the one hand, the timer task awaits a new timer service request (timeout time will be set to `WAITFOREVER`, if no timer has been set before). On the other hand, if one or more timers have to be administrated, the receive's timeout time must be adjusted to the next expiration time of the first timer in the timer list.

4.3.4 System Environment and Software Architecture

For both code generation strategies, ISRs are responsible for MEDF event creation. They assign the specified deadline to the newly created EDF event according to its type, and finally feed this triggering token into the SDL system using an "EDF event send" directive. External tasks (external to the SDL system model) build the interface to the system's environment. Their duty is to delete EDF events, i.e. the system has achieved to fulfill the specified deadline and to translate the outgoing SDL signals to output operations.

SDL system's processes, respectively their thereof generated tasks will run in the same priority level under MEDF scheduling. This is again true for both strategies, SM and ATM, whereby a task in ATM implements a whole activity thread and with SM, a task represents only one single SDL process. Additional priority levels are required and used for the timer task and external tasks. Due to the chosen higher priority levels for these

[8] No wall time wrap around management shown.

[9] Alternative mechanisms for SDL Timer service realization, but only for non timeout surveillance functionality (see Sec. 3.4), include e.g. a cyclic task with a `task_wake_after` RTOS system call or a cyclic timer interrupt for polling of state changes in system environment. For this, SDL system model functionality is out–sourced into environment functions (e.g. into an external task or ISR).

```
task TimerTask( ) {

  Initialize_TimerList();

  while( FOREVER ) {
    if ( No_Timer_in_TimerList ) {
      message_queue_receive( TimerTaskQueue, &TimerTaskRequest,
        WAITFOREVER );

      CurrentTime = Now;
    }
    else { /* Timer_in_TimerList */
      CurrentTime = Now;

      if ( First_Timer_in_TimerList->TimerTime <= CurrentTime ) {
        /* Timeout */
        message_queue_send_edf_event(
          First_Timer_in_TimerList->SenderQueue,
          First_Timer_in_TimerList->EDFEvent,
          First_Timer_in_TimerList->TimerSignal );

        Delete_First_Timer_in_TimerList();
      }
      else {
        /* Wait on TimerTaskQueue until next Timeout or TimerTaskRequest */
        message_queue_receive( TimerTaskQueue, &TimerTaskRequest,
          First_Timer_in_TimerList->TimerTime - CurrentTime );
      }
    }

    if ( TimerTaskRequest_is_not_empty( TimerTaskRequest ) ) {
      /* Handle requests "reset timer", "timer x is active",
         "remove timer x", "set timer x" */
      Handle_TimerTaskRequest( TimerTaskRequest );
    }
  }
}
```

Figure 4.5: Timer Task with Timer Deadline Management

tasks, timer as well as output operations will behave like unpreemptible executions seen from tasks within the MEDF priority level (see Sec. 6.1.3 for details).

Conclusion

Execution order in SM is identical to ATM, therefore latency regarding the processing of stimulating events is identical too, but with ATM, communication overhead needed to trigger succeeding transitions may be avoided.

As already mentioned above, a mixing of decomposition principles to achieve the most efficient implementation is possible and probably necessary, depending on the amount of communication compared to the processing effort. ATM implementations could be equally realized by substituting the synchronization mechanism semaphore through SM message queues, thus mixing AT tasks based on procedure calls with server tasks.

A general requirement on the design of SDL server state machines is to keep state transition times as short as possible, thus minimizing blocking times of concurrent input event streams. Necessary deadline modification to avoid priority inversion effects leads to a higher processing performance to be scheduled (cf. Sec. 7.2).

Abandonment of a very strict interpretation of SDL's semantics, particularly the relaxation of the run–to–completion requirement for state transitions, may lead to even more efficient realizations of SDL specifications. This is done by the C–Micro code generation principle, where a send may immediately result in an interruption of the current state transition (call of consecutive transition, respectively its PAD function). Unfortunately, this could lead to complex nestings of process executions. Computational results may nevertheless remain correct, but the arising execution scheme will be difficult to predict and understand.

Chapter 5

Scheduling with Message Deadlines

Earliest deadline first scheduling with deadlines transported on messages (MEDF) is used as a platform for the proposed design process. As mentioned in the introduction, with MEDF a manipulation of specified end–to–end timing constraints with purpose to find a feasible schedule is not necessary anymore.

In order to apply an EDF feasibility algorithm like [Gre93a] or [Jef92] on MEDF scheduling, it will be proven in this chapter that under all circumstances execution order of tasks remain earliest deadline first when scheduled with *message* deadlines. For this purpose, task structures as generated through automatic mapping from SDL specifications are analyzed and, where necessary, design rules for SDL are derived.

Predictability of the execution scheme, but likewise predictable upper bounds for the (worst case) execution times of scheduling directives are needed for real–time analysis. This chapter outlines the MEDF scheduling scheme, takes a closer look at its analyzability and finally sketches an implementation concept and belonging worst case performance considerations and measurements.

5.1 MEDF Scheduling Scheme

An EDF ready list is the basis of EDF task scheduling, which in turn is the platform for the MEDF scheme. Task entries on this EDF ready list are sorted according to the tasks' deadlines. The first entry, the task with the shortest deadline, will be dispatched and executed. Tasks communicate with asynchronous message exchange, i.e. blocking receive directives based on queues and non–blocking send operations. Tasks in the blocked state are managed in an appropriate wait list. Therefore a queue consists of both, the message queue itself and an additional task wait list.

A Task is activated with a message receive. The task's body, implementing the extended finite state machine's (EFSM) transitions, includes zero, one or more send operations to release and ready succeeding tasks and finally the triggering receive is tested again. The task will remain in the ready state, if another message waits in the message queue, otherwise it blocks.

A deadline can not be parameter in a message's (user) data buffer, because the receiving and possibly blocked task has to inherit the deadline *before* it will have to be inserted into the EDF ready list. Instead, the deadline transport mechanism based on messages has to be supported by the run–time system's kernel (cf. Sec. 5.3).

With MEDF, one has to distinguish four different cases:

Send operation:

A No waiting receiver: the message is enqueued according its transported deadline, therefore the message queue will be sorted by deadline value. The sending task proceeds with its operation.

B A waiting (blocked) receiver adopts the transported deadline as well as the new message. It will be unblocked by this operation and has to be inserted into the EDF ready list following its new deadline.

Receive operation:

C No pending message: the receiver is appended to the wait list. Since there is no job to fulfill at the moment, the deadline of this now blocked task is set to infinity.

D A pending message, the first one, i.e. the message with the shortest transported deadline, is dequeued. The receiving task will get assigned the deadline and has to be inserted into the EDF ready list.

As can be seen, in no point of time there may simultaneously reside a message in the message queue together with a blocked tasks in the belonging wait list[1].

5.2 Predictability of MEDF

For the following considerations a fixed number of sporadic tasks is assumed. A triggering event E_m is translated to a message m_{m1} with help of an interrupt service routine (ISR_m) and sent to the responding task T_{m1}. The receival of a message releases the task. Deadlines are specified as end–to–end constraints between the time a stimulus is initiated from the environment and the time, when a response is sent back to the environment. Thus a deadline d_{Tm} will span a whole task precedence system TPS_m. Task model and environment parameters are summarized in Table 5.1.

5.2.1 Task Precedence Constraints

Communication requirements are modeled as precedence constraints among tasks, that is, if a task T_j has to communicate to another task T_k, the pair (T_j, T_k) is introduced in a

[1]Regarding SDL server as well as activity thread model implementations, with only one task listening at a message queue, the depth of the task wait list reduces to one.

Task Parameters	
$T_{m,i}$	ith task T in task precedence system TPS_m
$\mathcal{T}(TPS_m)$	set of tasks $\{T_{m,1}, \ldots, T_{m,n}\}$ in TPS_m
$r_{m,i}$	release time of task $T_{m,i}$
$s_{m,i}$	start of computation of task $T_{m,i}$
$f_{m,i}$	completion time of task $T_{m,i}$
$c_{m,i}, (c_{max_{m,i}})$	worst case computation time of task $T_{m,i}$
$c_{min_{m,i}}$	best case computation time of task $T_{m,i}$
$d_{m,i}$	relative deadline of task $T_{m,i}$
$D_{m,i}$	absolute deadline of task $T_{m,i}$
Environment Parameters	
E_m^j	jth instance of event E of type m triggers $T_{m,1} \in \mathcal{T}(TPS_m)$
e_{Tm}	time of occurrence of triggering event E_m of type m
d_{Tm}	relative end–to–end deadline for triggering event E_m

Table 5.1: Task and Environment Parameters

partial order $\rightarrow$, i.e. if there exists the constraint $T_j \rightarrow T_k$, T_j has to be scheduled preceding the execution of T_k.

To enforce the execution order of two preceding tasks $T_j \rightarrow T_k$, [Bla76] suggests to manipulate the deadlines (d_j of task T_j), such that $\forall j, d_j^* \leq d_j$ and $T_j \rightarrow T_k \Rightarrow d_j^* < d_k^*$. With these shortened[2] deadlines, the EDF scheduler is forced to dispatch T_j before T_k. At run–time, whenever a request of execution for the tasks $T_{m,j} \in \mathcal{T}(TPS_m)$ arrives at e_{Tm}, the value $e_{Tm} + d_j^*$ is assigned statically to the absolute deadlines $D_{m,j}$.

In SDL, a process may respond to several different messages originating from several events of different type. Thus the appropriate task in the implementation will be a member of several task precedence systems in parallel. For instance, T_S in Fig. 5.1 responds to m_{42}, originating from E_4, and to m_{52} with its origin in E_5. Since these events may occur simultaneously ($e_{Tm} = e_{Tn} = e_T$), a statical assignment of absolute deadlines $D_{m,j}$ at e_T is not possible.

A receive triggers the execution of a state transition, thus the appropriate task in the implementation gets released with the receival of a message (time $r_{m,i}$). Since release times $r_{m,i}, \forall i > 1$ are varying during run–time due to non–constant computation times of the single tasks in the precedence system, relative (remaining) deadlines $d_{m,i}$ except $d_{m,1}$ at $r_{m,i}$ are unknown too. This inhibits an on–line, per task determination of absolute deadlines $D_{m,i}$ at release time $r_{m,i}$. A solution for this problem is to transport D_{Tm}, determined at occurrence of the external event, with inter–task messages. The receiving task adopts the transported deadline and is scheduled accordingly (Sec. 5.1).

[2] Reducing deadlines will lead to a loss of laxity in task systems.

Figure 5.1: Examples of Task Precedence Systems

Definition 3 *A set of n communicating tasks $\mathcal{T}(TPS_m) = \{T_{m,1}, \ldots, T_{m,n}\}$ such that $T_{m,i} \rightarrow T_{m,i+1}; \forall i \in \{1, \ldots, n-1\}$ is called a* Task Precedence System *(TPS). The first task $T_{m,1} \in \mathcal{T}(TPS_m)$ is released at time of occurrence e_{Tm} of the external event E_m.*

Example task precedence systems, e.g. TPS_1 can be seen in Figure 5.1.

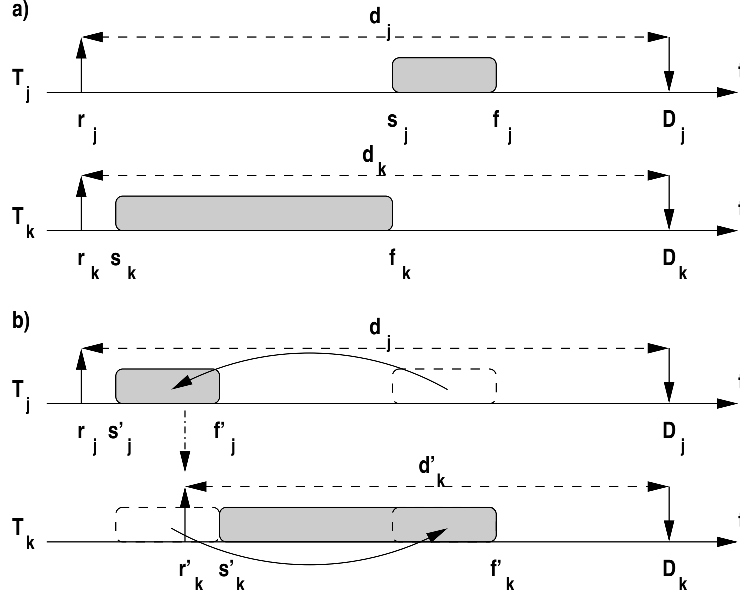

Figure 5.2: Feasible Schedule without a) and with b) Precedence Constraint

Definition 4 *All tasks $T_{m,i} \in \mathcal{T}(TPS_m)$ scheduled under MEDF get assigned the same absolute deadlines $D_{m,i}$ at their release time $r_{m,i}$. $D_{m,i} = e_{Tm} + d_{Tm}; \forall i \in \{1, \ldots, n\}$.*

d_{Tm} is the relative end–to–end response time for event E_m, specified through the non–functional requirements. Preserving this deadline within a TPS leads to the following alternative definition of task precedence constraints (cf. Fig. 5.2(b)).

Definition 5 *Given a precedence constraint $\rightarrow$ on the tasks T_j, T_k:*

$$T_j \rightarrow T_k \Rightarrow r_j < r_k \wedge D_j = D_k$$

With this definition, T_k is released through a message from T_j. Positions of send operations are allowed to be variable in the task's body, that is may be the first action after the triggering receive, hence $r_{m,j} < r_{m,k}$ for two tasks $T_{m,j} \rightarrow T_{m,k}$ and $r_{m,k} = r_{m,j} + \epsilon; \epsilon \rightarrow 0$.

Theorem 1 *$T_j \rightarrow T_k$ is feasible, if there exists a feasible schedule for T_j, T_k without precedence constraint.*

Proof. Assume there exists a feasible schedule compliant with EDF scheduling. That is there may be a couple of tasks T_j and T_k released by the same event but without precedence constraint such that, execution of T_k precedes execution of T_j, i.e. $r_k = r_j$, $f_k < f_j$ (Fig. 5.2(a)). Since the schedule is feasible and there is no precedence relation between these two tasks, it is possible to swap execution of the intervals $[s_j, f_j)$ and $[s_k, s_k + [f_j - s_j))$.

Feasibility is unaffected by this inversion, since the completion time f'_j of task T_j is only be shortened, while the new completion time of task T_k becomes $f'_k = f_j \leq D_j = D_k$. With a finite number of such inversions a feasible schedule for all precedence relations can be found. As shown in Figure 5.2(b), at $t = r'_k$ a message from task T_j may release task T_k, therefore this new release time will always reside in interval $[r_j, f'_j)$. □

5.2.2 Sequencing

In a MEDF task system, an ISR translates an external event into a message together with an specified end–to–end deadline and sends this message to the initial message queue of a TPS. Similar to "event handlers", this message queue buffers stimulating events, provides their timely decoupling, necessary to achieve real–time behaviour.

On asynchronous occurrences of external events, e.g. burst of events of same type, i.e. $E^j_m, j \in \{1, \ldots, l\}$ with deadlines $d(E^j_m) = d_{Tm}, \forall j \in \{1, \ldots, l\}$, triggering the same TPS_m (e.g. TPS_1 in Figure 5.1), the processing of events happens in a sequenced manner.

Lemma 1 *Consecutive events are serialized, such that $T_{m,n} \in \mathcal{T}(TPS_m)$ (last task in TPS_m) will finish its computation released through E^i_m before $T_{m,1}$ (first task in TPS_m) will start the processing of event E^{i+1}_m.*

Proof. The times of occurrence of events in a burst are defined through $e(E^j_m) = e^j_m \mid e^j_m \leq e^{j+1}_m$. With $D^j_m = e^j_{Tm} + d_{Tm}, \forall j \in \{1, \ldots, l\}$ (Definition 4) their absolute deadlines result in $D(E^j_m) = D^j_m \mid D^j_m \leq D^{j+1}_m$. The appropriate messages transporting these triggering events are serialized in first queue $q_{m,1}$ of TPS_m. Since $D(\mathcal{T}(TPS_m)) = D^j_m$ (Definition 4) and subsequent tasks with same deadline are appended to predecessors in EDF ready list and thus get ordered according to their release times, an event within a burst will have to wait until its predecessor has been processed through TPS_m. □

Maximum Message Queue Length With Lemma 1, maximum number of messages $p_{m,i+1}$ in queue $q_{m,i+1}$ between consecutive tasks $T_{m,i}$ and $T_{m,i+1}$ is $p_{m,i+1} = 1; \forall i > 1$.

Remark(s): *[Gre93a]: $p_{m,i+1} = E_m(d_i - d_{i+1})$; with $d_i = d_{i+1} = d_{Tm} \Rightarrow p_{m,i+1} = E_m(0)$. $E_m(t)$ denotes the Event Function of event E_m.*

More complex task systems may contain forks and joins as can be seen with TPS_2 in Figure 5.1 or even loops as in TPS_3.

Forks

Receivers like tasks $T_{2,2,a}$, $T_{2,2,b}$ in Figure 5.1 have same absolute deadline, i.e. messages $m_{m,i+1,a}$ and $m_{m,i+1,b}$ refer to same event E^j_m. EDF ready list is organized as "first come first serve" for subsequent tasks with same deadline. Therefore the processing sequence depends on send operation order in sending task.

Joins

Sort policy for messages transporting references to the same EDF event is "FCFS" as well, i.e. succeeding messages with equal deadline are simply appended to their predecessors. Processing order of events is related to sort order of the incoming message queue, in turn being equal to the execution order of preceding tasks in the TPS.

Bounded Loops

Receival of a loop message in queue $q_{m,i}; \forall i > 1$, i.e. message was sent back from task $T_{m,j}; j \geq i$ within TPS_m. Due to Lemma 1, there is no other event being processed by task $T_{m,i}$ in the moment, therefore no additional blocking times have to be taken into account. Queue length of queue $q_{m,i}$ remains $p_{m,i} = 1$.

Exception: Receival of loop message in first queue $q_{m,1}$. Loop message transports event E_m^i, subsequent events $E_m^j; j > i$ are waiting with their appropriate messages in queue $q_{m,1}$. Task structure is derived from SDL FSM, i.e. initial receive is executed as first *and* last operation within task's body. If several event messages are waiting in the initial queue in case of a burst, the succeeding event to the currently processed one will be consumed with the final (second) execution of the $T_{m,1}$'s initial receive, consequently $T_{m,1}$ is instantly reinserted into the EDF ready list with new deadline $D(E_m^{i+1})$. The same effect can be caused through ISR sends during active periods of tasks $T_{m,i}; i \geq 1$. Thus, the message transporting event E_m^i may have been overtaken from "younger" messages with later deadlines. For this reason, (non–predictable) loop structures with joins at TPS's initial queue $q_{m,1}$ are forbidden.

In contrast to joins into a TPS' initial queue, joins into a server queue are allowed, since preceding tasks sending an event with longer deadline must not preempt the processing of the current event in the loop. If a preceder sends a more critical deadline, preemption will occur and therefore processing order will remain EDF.

Generally, a specification of an upper loop bound is required for predictability and processing times can be easily calculated through loop unrolling.

5.2.3 Message Blocking on Server Tasks

As stated above, there exist no shared resources among SDL processes and consequently no mutual exclusion between tasks in the "Server Model" implementation. To handle I/O–devices or common data the so called *server process* principle is used. Comparable to Hoare's *monitors* [Hoa74], SDL processes observe and manage the access requests on resources. Consider the following scenario in a task system as shown in Figure 5.3.

Scenario 1 *Two tasks $T_S, T_{m,k}$: A task T_S responding on several different messages m_{12}, m_{22} and $T_{m,k}$, a task like T_{31} in TPS_3. Each message $m_{m,i}$ transports a deadline D_{Tm} through TPS_m. At r_{12} task T_S consumes m_{12}, adopts deadline D_{T1} and begins to compute response m_{13}. At r_{22} message m_{22}, transporting $D_{T2} < D_{T1}$, arrives and is inserted into message queue q_S, which belongs to task T_S. At r_{31} the occurrence of an external event,*

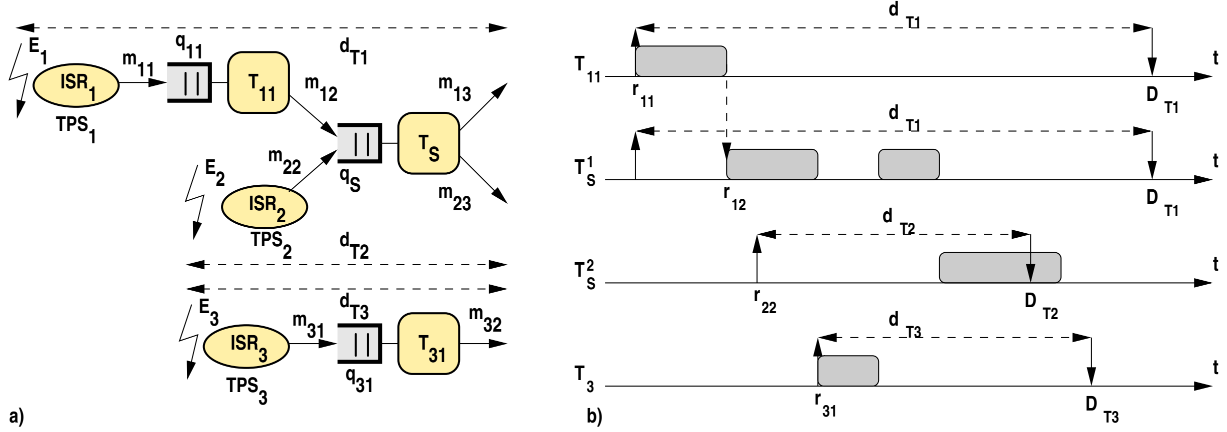

Figure 5.3: a) Server Task and b) Priority Inversion

which triggers TPS_3 *may lead to an activation of task* T_{31}*. In this situation, absolute deadlines may be in relation* $D_{T2} < D_{T3} < D_{T1}$*.* T_{31} *will preempt* T_S*, which can be considered a typical priority inversion, because the most critical deadline* D_{T2} *of message* m_{22} *will not be assigned to* T_S *as long as the whole* TPS_3 *is processed and* T_S *has finished execution of job* m_{12}*.*

Definition 6 *A* Server Task T_S *responds to at least two different message* m_x *respectively* m_y*, received from a common queue* q_S*. The messages originate from different triggering event sources* E_x *and* E_y*.*

In this context, the Server T_S serializes the execution order of its single transitions without respect to EDF ($s(T_S^1) < s(T_S^2)$ though $D(T_S^1) > D(T_S^2)$) and in consequence *Priority Inversion* (PI) corrupts processing order of transitions and any further preempting task with deadline in the middle of D_{T2} and D_{T1}, i.e. sequence does not correspond to EDF anymore ($s(T_{31}) < s(T_S^2)$ though $D_{T3} > D_{T2}$).

As can be seen in Figure 5.3(b), PI can only occur, if $r_{22} > r_{12}$, otherwise any processing of events of type E_2 will preempt tasks of TPS_1 and execution sequence remains EDF. The worst case, i.e. blocking time for T_S^2 is at its maximum, occurs, if $r_{22} = r_{12} + \epsilon, \epsilon \to 0$.

PI can be avoided in any case through a deadline reduction for the blocking task. In the example, any reduction to $D'(T_S^1) = D_{T2}$ will resolve the PI constellation. Following Figure 5.4, $d(T_S^1)$ has to be reduced to at most $d' = c_{11,min} + d_{T2}$. Or more generally:

$$d' = \min_{\forall x}\{s_{T_S^x,min}\} + d_{Ty} \tag{5.1}$$

With $s_{T^x_{S,min}}$ as earliest possible start time[3] for the blocking parts of server T_S and d_{Ty} for the deadline of the blocked part.

[3] To determine the earliest possible start time $s_{T^x_{S,min}}$ for transition T_S^x the sum of all best case execution times of tasks $T \in \mathcal{T}(TPS_x)$ preceding the server can be calculated. Additionally event dependencies can be taken into account.

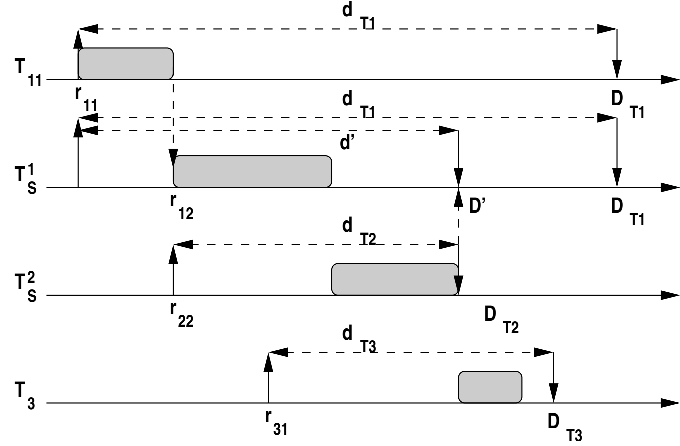

Figure 5.4: Priority Inversion Avoidance

Remark(s): *[Gre93a]: $d' = s_{T_S^x,max} + d_{Ty}$, but with $s_{T_S^x,\mathbf{max}}$ PI can occur, e.g. if $c_{11} < c_{11,max}$.*

Well known strategies for PI avoidance based on priority adaption are *Priority Ceiling* with an a priori raising of the blocking priority and *Priority Inheritance* with an on–line priority manipulation [Raj91]. Adapting these two mechanisms to MEDF leads to the following protocols.

Deadline Ceiling Protocol

Definition 7 *With Deadline Ceiling Protocol (DCP) and MEDF, the deadline $D = D(T_S)$ of a server task T_S is reduced at release time to ceiling deadline D_C. With $D_C = e_{Ts} + d_{QC}, s \in \{x, y, \dots\}$ and $d_{QC} = d' = s_{T_S^x,min} + d_{Ty}$ (Eq. 5.1) the server's queue (q_S) relative ceiling deadline.*

EDF is known as optimal in that sense, if there exists a feasible schedule, EDF will find it. But only with EDF processing order, optimality is given and proven. With DCP, a possible PI is avoided a priori and therefore execution order between the blocked task (resp. transition T_S^2) and all preempting tasks, like task T_{31} in the example is in conformance with EDF. Since a TPS's execution order is administered through MEDF message exchange, deadline reduction within a TPS violates the precedence constraint requirement (cf. Def. 5), i.e. $T_{11} \rightarrow T_S^1$ with $D(T_{11}) = D_{T1} > D(T_S^1) = D_C$ will violate EDF. As a consequence, influence of DCP on tasks preceding a server has to be examined:

Theorem 2 *Using DCP on a server task T_S within a TPS, schedulability of the task set: $T_x \rightarrow T_S$ with d_x, $T_y \rightarrow T_S$ with d_y, and T_z with d_z concerning end–to–end deadlines d_x, d_y, d_z will remain unaffected, if the independent task set, i.e. T_x with d_x, T_y with d_y, T_z with d_z, and T_S with d_{QC} is schedulable (in the worst case).*

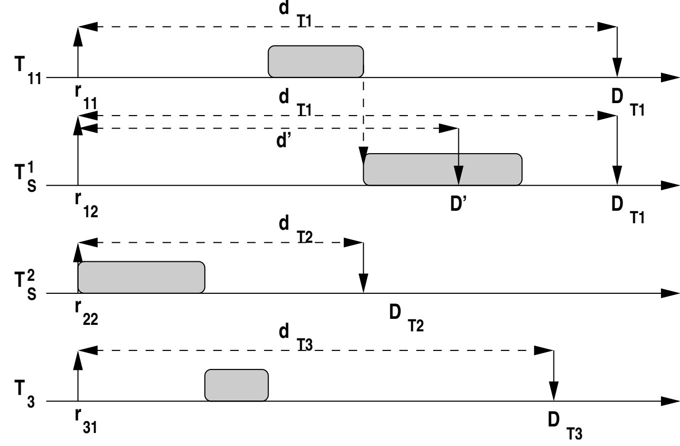

Figure 5.5: Violation of Ceiling Deadline D' without Priority Inversion

Proof. Assumption: There exists a feasible schedule even in the worst case for the independent task set. For the example in Figure 5.4 and 5.5 this means deadlines $d(T_{11}) = d_{T1}$, $d(T_S^1) = d_{QC}$, $d(T_S^2) = d_{T2}$, and $d(T_{31}) = d_{T3}$, but *no* precedence constraint $T_{11} \rightarrow T_S^1$ is given.

1. $r_{22} \leq r_{12}$, PI can not occur. Taking into account the precedence constraint $T_{11} \rightarrow T_S^1$, all tasks, except T_S^1 will finish execution earlier compared to an equal task set without this communication requirement, i.e. these tasks will meet their deadlines (cf. Fig. 5.5). However T_S^1 will at least fulfill its "physical" (specified end–to–end) deadline d_{T1}, because with this deadline, execution order corresponds to EDF and schedulability must be given with the assumption of above and through the additional laxity $d_{T1} \geq d_{QC}$.

2. $r_{22} > r_{12}$. PI is avoided through D'. T_{11} has already completed its processing! Again, the rest of the task set obeys EDF and with the assumption above, all deadlines, even the deadline $d(T_S^1) = d_{QC}$, will hold (cf. Fig. 5.4). □

Summary:

- Due to Theorem 2, it is only necessary to reduce the deadlines of servers within a TPS and leave the preceding and succeeding tasks untouched.

- During run–time, in case of no PI, the ceiling deadline D_C may be missed (cf. Fig. 5.5 and Sec. 5.3.3).

Remark(s): *In "Strategy 2", [Gre93a] suggests to reduce all deadlines of the preceding tasks in a TPS as well, loss of laxity may lead to an infeasible schedule.*

Deadline Inheritance Protocol

Definition 8 *With Deadline Inheritance Protocol (DIP) and MEDF, the current deadline $D = D(T_S)$ of a server task T_S is shortened to $D_I = D_{Ty}$, only when a new message m_y with deadline D_{Ty} is inserted into the message queue q_S and $D_{Ty} < D$.*

Following Equation 5.1 and Figure 5.4, with DIP a deadline of a server task will be reduced to at most $D_{I,min} = e_{Tx} + d_{I,min}$ and $d_{I,min} = d' = \sum_n c_{x,n,min} + d_{Ty}$.

Applying DIP on Scenario 1, at r_{22} task T_S inherits D_{T2} and can not be preempted anymore by tasks of TPS_3, i.e. it finishes the processing of m_{12}, resumes with m_{22} and finally executes transitions of T_{31}.

Lemma 2 *With DIP and MEDF scheduling, each transition τ_i of a server task T_S can be blocked by at most the duration of the longest transition of T_S.*

Proof. Transitions can be considered to be within a critical region, i.e. they behave equivalent to EDF tasks sharing a common resource with mutual exclusion and Priority Inheritance Protocol (PIP). Therefore the proof is analogous to Theorem 6.1 in [SSRB98]. □

Theorem 3 *Using DIP on a server task T_S within a TPS, schedulability of the task set: $T_x \rightarrow T_S$ with d_x, $T_y \rightarrow T_S$ with d_y, and T_z with d_z concerning end–to–end deadlines d_x, d_y, d_z will remain unaffected, if the independent task set, i.e. T_x with d_x, T_y with d_y, T_z with d_z, and T_S with d_I is schedulable.*

Proof. Assumption: There exists a feasible schedule even in the worst case for the independent task set.

1. $r_{22} \leq r_{12}$, PI can not occur (cf. Fig. 5.5), therefore no deadline in the TPS will be shortened at run–time, i.e. the precedence constraint requirement (cf. Definition 5) will not be violated, schedule and schedulability remain unchanged.

2. $r_{22} > r_{12}$. PI is avoided through $D' > D_{I,min}$ (cf. Fig. 5.4). Analogous to Theorem 2, in this case T_x has already completed its processing! Again, the rest of the task set obeys EDF and with the assumption above, all deadlines, even the deadline $d' > d_I$ will hold. □

Conclusion: DCP and DIP are equivalent protocols and have to be treated in the same way during schedulability analysis.

Remark(s): *In "Strategy 5", [Gre93a] suggests to reduce to deadline $d_i = d_{Ty}$ in all cases. Due to this additional loss of laxity, none of both strategies "2" and "5" may outplay the other in advance.*

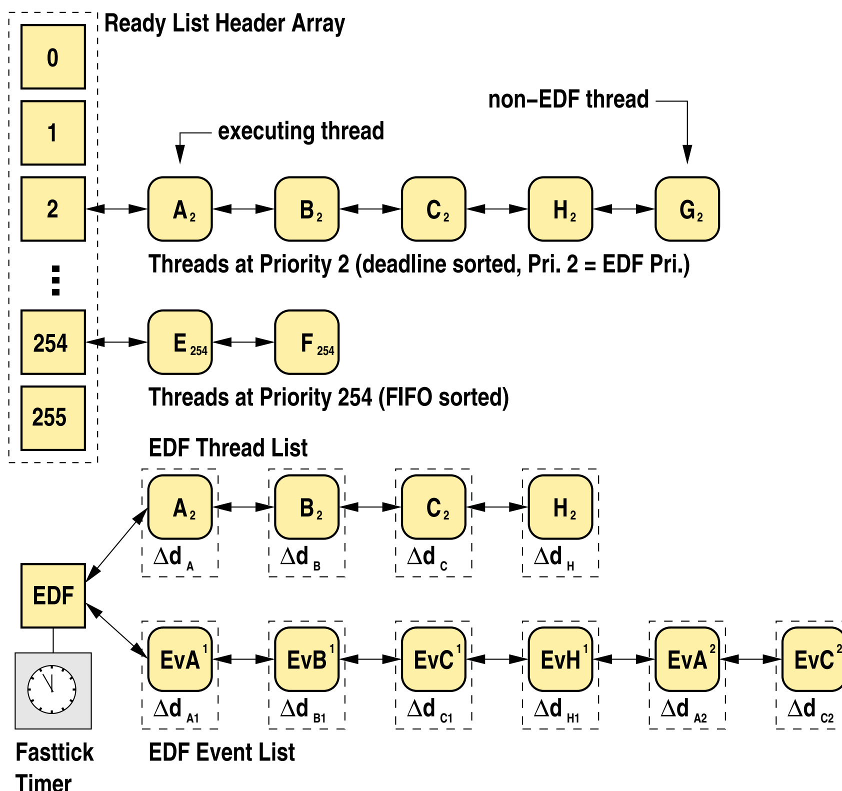

Figure 5.6: RTEMS Priorities, Ready List, EDF Thread List and EDF Event List

5.2.4 Mutual Exclusion on Semaphores

Semaphores are used in ATM realizations of SDL models, when synchronizing mutual access to state data of SDL server processes. Analogous to server queues, PI can also occur. The protocols DCP and DIP are well known with semaphores. A detailed evaluation of the resulting timing behaviour and analysis algorithms can be found in [SSRB98].

5.3 Runtime System Support

This section sketches the realization of MEDF on the basis of “normal” EDF scheduling. Both strategies have been integrated into the Real–Time Executive for Multiprocessor Systems (RTEMS[4] [OAR96]). Implementation details are outlined in Appendix B.

RTEMS is a light-weighted RTOS, i.e. tasks are threads without support of memory protection. It provides so called managers and real–time objects, e.g. a semaphore and a message manager. Each RTEMS object gets assigned an unique object id after its creation.

[4] www.OARcorp.com

5.3.1 The EDF Event Object

Scheduling and thread state management is based on manipulation of doubly linked lists, e.g. a transition from the *ready* to the *blocked* state is handled by removing the thread from the ready list and inserting it into a wait list for the resource. Critical sections, i.e. non–interruptible code segments are protecting data consistency during list adaption.

Fixed preemptive priority scheduling is supported with 256 priority levels (zero is highest). For each of these priority levels the RTEMS thread handler administers an independent *ready list* (Fig. 5.6). Ready list headers are collected in an array for fast indexed access, based on the priority level of a thread. Ready lists only contain threads in the ready state with a single exception: The first thread on the first non empty ready list in priority order is considered to be in the *executing* state. Preemption causes a currently executing thread to be in the ready state again. Execution order is maintained by processing threads from the first to the last node within the ready lists and from the lowest to the last element of the thread list array. EDF scheduling is realized by manipulating the sort order of the ready lists.

EDF thread list. The absolute deadline of a thread T_j is computed at deadline activation a_j: $D_j = a_j + d_j$. To administer these deadlines, an additional *EDF thread list* [Wro97] contains all the currently existing EDF threads in sorted order according to their deadline (cf. Fig. 5.6). The deadline values are set up as delta relative to the previous entry on the list (e.g. Δd_B, Δd_C and Δd_H) or relative to the start of the list for the first thread ($\Delta d_A = d_A$). Adding the deltas, each deadline is correctly defined relative to the list start. To keep track of time, with a delta list only one timer (*Fasttick Timer*) is needed for surveillance of the most critical deadline on the list. A detailed description of deadline management with delta lists is given in Appendix B.1.

EDF ready list. As explained above, the RTEMS scheduler uses an array of ready lists, one list per priority level. Each level is sorted by FIFO, except the EDF level. EDF scheduling influences the sort order of threads in a *single* priority level. Therefore priorities take precedence over deadlines and the application designer has to choose a correct priority level (probably the highest during "normal" operation) to guarantee real–time capability. To handle non–EDF threads without exception, an infinite deadline is assigned to them.

To get this EDF ready list sorted by deadlines, a new EDF thread is appended to all threads on that ready list with less or equal deadlines. New non–EDF threads, with infinite deadlines, are simply appended at the end of the list. Since the EDF thread list is already sorted by deadline order, the search algorithm for the ready list insertion point can rely on this order: Assuming that a thread T is no longer blocked and has to be inserted into the EDF ready list, the insertion point can be found behind all EDF threads on the EDF ready list, which lie before T on the EDF thread list. Therefore, the insertion algorithm (see Appendix B.2 for details) uses a parallel search in both the EDF ready and the EDF thread list.

If the EDF thread list contains n threads, the EDF ready list will contain at most n threads. The linear walks through both lists imply a maximum of $2n$ identity comparisons in total, leading to a complexity of $O(n)$.

EDF event list. As shown in Section 5.2.1, a requirement for analyzability of MEDF is transportation of absolute deadlines. Conceptually, deadlines spanning a TPS belong to stimulating events. The new RTEMS *EDF event manager* provides directives to create and delete *EDF event objects*, which represent the stimulating event and its belonging absolute deadline. Each EDF event object gets assigned its unique object id according to the RTEMS philosophy and at its creation a deadline has to be set. Deadline management of EDF event objects relies on a third list, the *EDF event list* (Fig. 5.6). List administration is identical to EDF thread list's one. This means the EDF event list is as well a delta deadline list and list manipulation works in the same way.

Comparing life cycles of EDF thread list entries and EDF event list entries, one can observe that the number of EDF threads will stay constant but their relative position on the list varies. In contrast to this, the number of EDF events is varying (several occurrences of the same event type, e.g. a burst of the same trigger) but the relative position on the their list remains constant. When the last thread of a TPS fulfills its job, it will delete the triggering event.

Remark(s): *Alternatively, the sending thread could have been calculated an absolute deadline using the RTEMS `clock_get()` directive and sent this point of time to the subsequent thread in the TPS. Since the `clock_get` directive relies on the cyclic RTEMS clock tick mechanism (cf. App. B.3), the resolution of deadlines would have been of the same granularity. To achieve high deadline resolution, one should have had to shorten the clock tick period, leading to a high interrupt load.*

5.3.2 The MEDF Message Queue

Messages are transporting a *reference* to an EDF event list entry. The receiving thread will adopt the message's, respectively the event's deadline when the sender inserts the message into the receiver's queue. With this, the absolute deadline of the receiving thread becomes the absolute deadline of the triggering EDF event. A thread will adopt the deadline by manipulation of its EDF thread list position. After its removal from the EDF thread list (with infinite deadline) follows a re-insertion using the `Add_To_ThreadList_With_EDF` algorithm (Fig. B.3 in Appendix B.2) and the EDF event list as sorted basis. In a subsequent step, the EDF ready list is updated following the new EDF thread list order.

In case of no waiting receiver, the message has to be inserted into a message queue. Sort order of this queue follows likewise the EDF event list order.

Below, the implementation of EDF event send and EDF event receive is outlined in more detail. Again, one has to distinguish four possible cases (refer to page 46):

EDF event send:

A No waiting receiver: the message is enqueued according its transported deadline, therefore the message queue will be sorted by deadline value. The sending task proceeds with its operation.

Implementation and processing sequence:

1. Insertion of message into message queue based on EDF event list (analogous to `Add_To_ThreadList_With_EDF` algorithm)

B A waiting (blocked) receiver adopts the transported deadline as well as the new message. It is unblocked by this operation and has to be inserted into the EDF ready list following its new deadline.

Implementation and processing sequence:

1. Calculate relative deadline due to EDF event list position.
2. Insert receiving task into EDF thread list.
3. Insert now unblocked task into EDF ready list.

EDF event receive:

C No pending message: the receiver is appended to the wait list. Since there is no job to fulfill at the moment, the deadline of this now blocked task is set to infinity.

Implementation and processing sequence:

1. Append task on (empty) thread queue (server model as well as ATM architecture: only one task per message queue) and assign deadline $D = \infty$ (append to EDF thread list).

D A pending message, the first one, i.e. the message with the shortest transported deadline, is dequeued. The receiving task will get assigned the deadline and has to be inserted into the EDF ready list.

Implementation and processing sequence: Identical to case B.

5.3.3 Priority Inversion Avoidance

Thread queues in RTEMS are wait lists that keep non ready threads in either FIFO or priority sorted order. With FIFO queues, adding a thread means appending it to the queue and dequeuing is just removal of the first node.

In contrast to this, priority sorted thread queues have to be searched and manipulated. Multiple threads of the same priority are kept in FIFO order on an additional list, with the first thread on this list as the *head thread* and member of the priority list (Fig. 5.7). Again, dequeuing is just removal of the first node of the priority list. For enqueuing, an

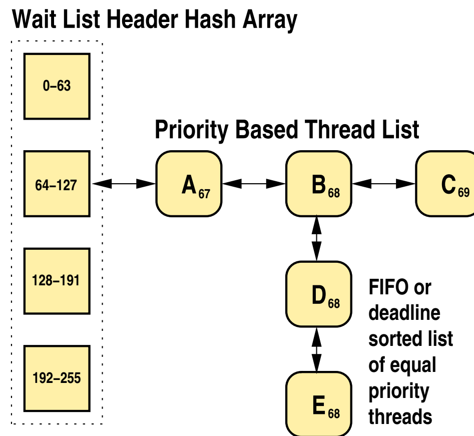

Figure 5.7: Priority Based Thread Queue

insertion point is reached, when a thread with greater or equal priority number is found. In the second case, the thread is just appended to the FIFO list of equal priority threads.

As well as for non–EDF threads, for a blocking EDF thread exists two queuing alternatives, the FIFO sorted or the priority sorted thread queue. Handling of FIFO queues is identical for both kinds of threads, but for an insertion into a priority sorted thread queue, the order of the list of threads in the priority level belonging to EDF must be changed to deadline sorted (cf. Fig. 5.7). Identically to the ready list insertion, comparison of deadlines relies on the sort order of the EDF thread list, therefore the enqueuing procedure uses again the same `Add_To_ThreadList_With_EDF` algorithm (cf. App. B.2).

Deadline Inheritance with Message Queues

For DIP implementation purposes, the message queue control structure holds a pointer to the last receiver thread, the so called *inheritor thread*. Whenever DIP is enabled for a queue, this pointer will be set during dequeuing send or receive operations. On arrival of a new message, the deadline of the transported event is compared to the deadline of the inheritor thread. If the deadline is shorter, the inheritor will adopt the event's deadline.

Inheriting, respectively shortening a deadline of a thread means manipulation of its EDF thread list position. The thread has to be removed and reinserted into the EDF thread list. Its new position is determined from the reference EDF event's list position.

List manipulation for deadline inheritance will only occur, if no waiting receiver is currently blocked at the message queue (EDF event send case A). All three other cases are identical to the remaining "normal" EDF event send and EDF event receive cases C – D.

EDF event send:

A No waiting receiver: the message is enqueued according its transported deadline. An active inheritor thread has to adopt the new event's deadline, in case it is shorter. The sending task proceeds with its operation.

Implementation and processing sequence:

1. Insertion of message into message queue based on EDF event list.
2. Calculate relative deadline due to EDF event list position.
3. Reinsert inheritor thread into EDF thread list.
4. Reinsert now unblocked task into EDF ready list.

Since a server thread will not send its (inherited) deadline to the succeeding thread in the TPS, but a reference to the triggering EDF event, the inherited deadline will not spread over the whole TPS. There exists no deadline restauration with DIP. It is assumed that the inheritor thread either will adopt the new deadline of a just enqueued message after the completion of its current job or in case of no new pending message its deadline will be set to infinity. In the latter case the thread will be appended at the end of the queue's thread list.

Deadline Ceiling with Message Queues

For this purpose an EDF event has to remember its original set end–to–end deadline interval d_{Tm}. With the static ceiling deadline of a queue d_{QC}, the current deadline d and the deadline d_{Tm} stored at events' creation e_m, the adapted ceiling deadline d_C can be calculated at run–time with $d_C = d - (d_{Tm} - d_{QC})$. In the example of Figure 5.4 on page 53: Current (remaining) deadline is $d = D_{T1} - r_{12}$ of thread T_S^1. This d is stored in the delta list of EDF thread list, respectively is the content of fasttick timer FT.

With enabled DCP, the kernel would set the deadline of thread T_S to d_C during dequeuing sends (case B) or receives (case D), i.e. T_S would be inserted with d_C into the EDF thread list. Therefore the processing sequence will be analogous to the non deadline ceiling operations as described in section 5.3.2. Analogous to DIP, there is no need for deadline restauration, when T_S "finishes".

Since d_C may be missed in a scenario without priority inversion, e.g. a thread activation sequence as can be seen in Figure 5.5, deadline surveillance must be disabled for d_C but has to be kept enabled for all other deadlines d_{Tm}. This feature would require complex exception handling in fasttick timer state machine (cf. App. B.3) and therefore DCP is not yet implemented.

Deadline Inheritance with Semaphores

Whenever a task blocks while obtaining a semaphore, the inheritance check is made. If the thread currently holding the semaphore has a longer deadline than the one trying to obtain it, it will inherit the shorter deadline. The original deadline is reinstated when the thread no longer locks the resource. The blocking thread has to be enqueued into the wait list according to its deadline.

In this case, inheriting the deadline of a thread in contrast to adopting it from an event, the inheritor has to be removed and reinserted behind the reference thread in the

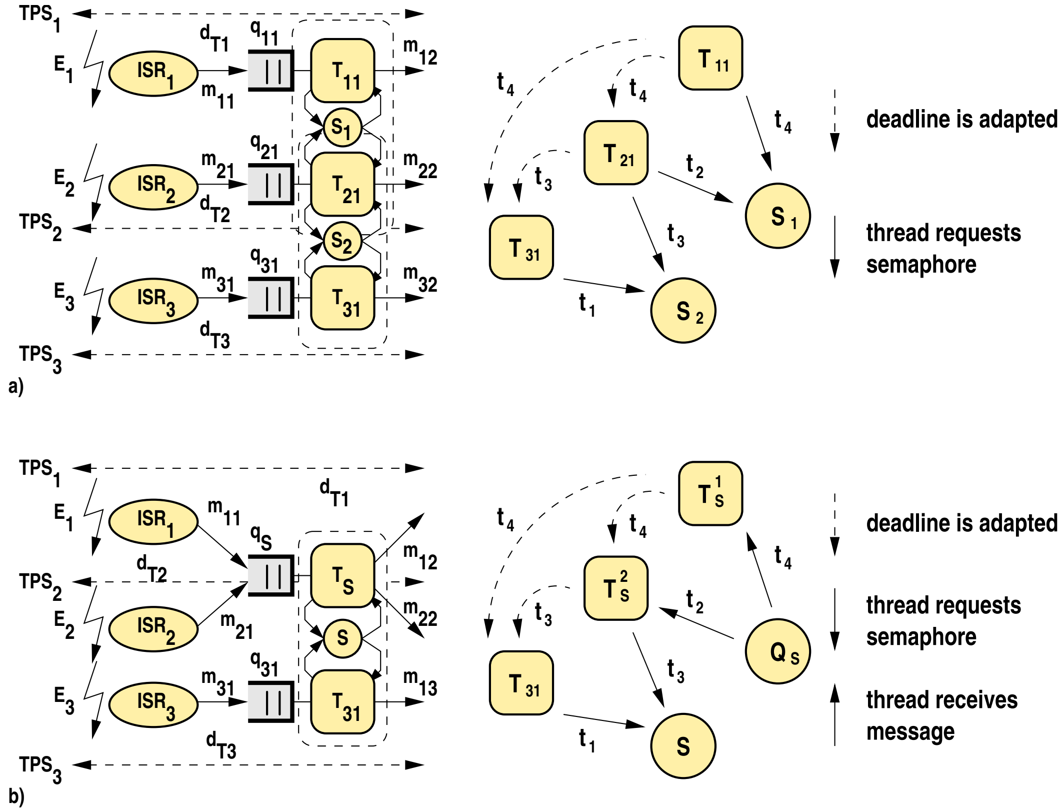

Figure 5.8: Transitive Deadline Inheritance with a) Nested Semaphores and b) Servers

EDF thread list. The delta deadline of the manipulated thread is set to zero. Since it will re-obtain its original deadline after releasing the semaphore, the deadline difference between reference thread and the inheriting thread is summed up during the re–insertion list search walk and stored for future restauration.

Nested Mutual Exclusion

The following example describes a PI scenario caused through a complex mutual exclusion structure in spite of enabled deadline inheritance. Task T_i obtains semaphore S with the operation $T_i.P(S)$, respectively releases S with $T_i.V(S)$.

Scenario 2 *Let T_{11}, T_{21}, T_{31}, T_{41} be four tasks with following deadlines ($D(T_{31}) > D(T_{21}) > D(T_{41}) > D(T_{11})$), i.e. T_{11} is the most critical task in this example. T_{11}, T_{21} are mutual exclusive synchronized with a semaphore S_1, and T_{21}, T_{31} are competing for S_2 (cf. Fig. 5.8(a)). Considering the following sequence $t_1: T_{31}.P(S_2)$, $t_2: T_{21}.P(S_1)$ and at $t_3: T_{21}.P(S_2) \Rightarrow D'(T_{31}) := D(T_{21})$. At t_4 task T_{11} preempts T_{31}, requests S_1: $T_{31}.P(S_2) \Rightarrow D'(T_{21}) := D(T_{11})$ and finally blocks. T_{31} will resume its job. A now activated T_{41} will preempt T_{31} and a PI situation will emerge.*

To avoid this PI, task T_{31} would have to inherit the deadline of task T_{11} at t_4: $D'(T_{31}) := D(T_{11})$ (transitive inheritance [Raj91]). An analogous situation can occur using server tasks together with semaphores (cf. Fig. 5.8(b)):

Scenario 3 *Given are T_{31}, T_{41}, and a server T_S. T_{31} and T_S are synchronized with semaphore S. Considering the following sequence t_1: $T_{31}.P(S)$, at t_2: T_S gets assigned $D_{T2} < D_{T3}$ with the receival of m_{21}, resuming with t_3: $T_S.P(S) \Rightarrow D'(T_{31}) := D(T_S)$. At t_4: ISR_1 preempts T_{31} and sends message m_{11} with deadline $D_{T1} < D_{T2}$ to T_S. Again DIP causes $D'(T_S) := D_{T1}$. Since T_S is blocked, waiting for semaphore S, an additional T_{41} with $D_{T4} < D_{T3}$ will preempt T_{31}. Again a PI situation will emerge.*

Having a closer look at the RTEMS thread queue implementation, one will see that in case of deadline inheritance the deadline of a new thread to be inserted is compared to the semaphore holder's deadline, respectively the inheritor's deadline. If necessary, the deadline of the inheritor will be adjusted as described above. If the resource holder is also blocked, its wait list position will be corrected according to the inherited deadline. Because there is no recursive deadline adjustment for new blocking thread, this implementation strategy is insufficient to fulfill the requirement for transitivity. Therefore PI can occur with RTEMS and nested areas of mutual exclusion.

Fortunately, both code generation strategies SM and ATM do not rely on task systems as shown in Figure 5.8. With SM, there are no semaphores necessary due to the server task as monitor for resources. With ATM, a SDL server process is implemented using a semaphore for synchronizing single transitions, but conflicting parts are only triggered by one event type. This means, there exists no combination of server task and semaphore as in Figure 5.8(b). Due to the sequencing of events within a TPS (Lemma 1), deadlines in queues are in ascending order and a deadline adaption is not necessary.

5.4 Complexity and Execution Times

MEDF as well as EDF scheduling are belonging to the class of dynamic scheduling strategies. Main characteristic of these techniques is the varying execution order of tasks during run–time, leading to the effect of a changing sort order of the necessary ready list. In contrast to EDF or MEDF, there exist other scheduling strategies, which could be implemented through simple list appends, e.g. fixed priority preemptive scheduling in RTEMS, keeping execution times constant. Although the latter property is preferable (and often used as refutation against EDF), it is sufficient even for hard real–time systems, if an upper bound for execution times of scheduling directives used can be given.

This section examines the worst case effort of MEDF processing sequences implemented in the run–time system as introduced in Sec. 5.3. An optimized but specialized (for Server Model software architectures) MEDF realization is introduced as conclusion.

Task structure and thus scheduling operations employed depend on code generation strategy, Server Model or Activity Thread Model:

- SM tasks rely on EDF event receive and EDF event send operations, which enclose the task's body (cf. Fig. 5.9).
- ATM tasks use a releasing EDF event receive, optional EDF semaphore obtain and EDF semaphore release directives to synchronize about mutual exclusive regions.

Common to both strategies (cf. Sec. 4) are ISRs for event creation (EDF event create). EDF event send is used to feed in triggering events into SDL system part. An external task, respectively output tasks or output parts of activity threads delete triggering events with EDF event delete.

The MEDF run–time system implementation as proposed in the previous chapter supports both, the SM as well as the ATM software architecture. Because of its flexibility, supporting EDF message deadlines on the one hand and semaphores with EDF sorted thread queues on the other, processing sequences for the scheduling operations are unoptimized and necessary overhead is incorporated. Execution times of scheduling directives depend on the following:

- State of run–time system, which is given through 5 lists (list types):

  1. EDF ready list.
     Maximum number of elements on ready list (tasks in ready state):
     $$N_{T_{Ready},max} = 2k \tag{5.2}$$
     This is two ready tasks per TPS, one active and one released just now (initial task in TPS); k is the number of different event types, respectively number of TPSs.
  2. EDF thread list.
     Maximum number of tasks with non–infinite deadline on EDF thread list:
     $$N_{T_{D\neq\infty},max} = 2k \tag{5.3}$$
     This is likewise two tasks per TPS, one active and one released just now.
  3. EDF event list.
     This list contains all events simultaneously processed by the system and waiting in initial input queues. In the worst case the maximum calculates to:
     $$N_{E,max} = \sum_{i=1}^{k} N_{E_i,max} + k \tag{5.4}$$
     $N_{E_i,max}$ is maximum number of events of type E_i in initial queue of TPS_i. $N_{E_i,max} = E_i(t_{p,max}) = E_i(d_i)$, $t_{p,max}$: maximum duration of processing of one event E_i in TPS_i. If all timing constraints will hold, $t_{p,max} < d_i$ is valid at all events, which is the definition of real–time processing. There may be one additional event per TPS currently processed. This means in total k additional events on this list.

4. EDF thread queue.

   One for each object (semaphore or message queue). Maximum number of elements depends on way of usage, e.g. maximum thread queue length of a message queue is 1 (one task per message queue in SM and one activity thread listening to one input queue).
   Maximum thread queue length of a semaphore equals the number of activity threads being synchronized for mutual exclusive access to state variables.

5. EDF message list (message queue).

   Depends on way of usage (see below).

- Task's position within communication structure (corresponds to real–time analysis model):

  SM architectures rely on initial (input) tasks, server tasks and "linear" hand–over tasks using three different typed of queues:

  1. Input queues for buffering of bursty events (cf. q_{11} to q_{51} in Fig. 5.1 on p. 48).

     Maximum queue length: $p_{m,1} = N_{E_i,max} = E_i(t_{p,max}) = E_i(d_i)$ (see Eq. 5.4).

  2. Server queues for synchronization of different event streams (q_S in Fig. 5.1).

     Maximum queue length: $p_S = N_{E_S}$. N_{E_S} is number of different event streams to be synchronized on server queue q_S.

  3. "Hand–Over" communication queues (e.g. q_{12}, q_{13} in Fig. 5.1).

     Maximum queue length: $p_{m,i+1} = 1$ is sufficient due to event sequencing (Lemma 1).

  In contrast to SM, ATM architectures only employ initial input queues (cf. queue q_{61}, q_{71} in Fig. 5.1) and semaphores (e.g. semaphore S between tasks T_{62} and T_{72} in Fig. 5.1).

A necessary assumption for the timing considerations below is, the system has already completed its initialization phase. This means, all tasks have already fulfilled their own set–up and have run into their initial triggering EDF event receive (and thus are blocked). The first triggering external events with hard deadlines are only allowed after this initialization. The init phase is irrelevant for timing and schedulability analysis of hard real–time modes.

All timing data[5] presented in following sections have been collected on a MIPS R4650 processor based Galileo 4 Evaluation Board [Gal95, Gal96] with 50 MHz clock frequency, using the processor's cycle counter as time base (resolution 20 ns).

[5]Run–time variations based on caching or pipelining effects are out of scope of this work.

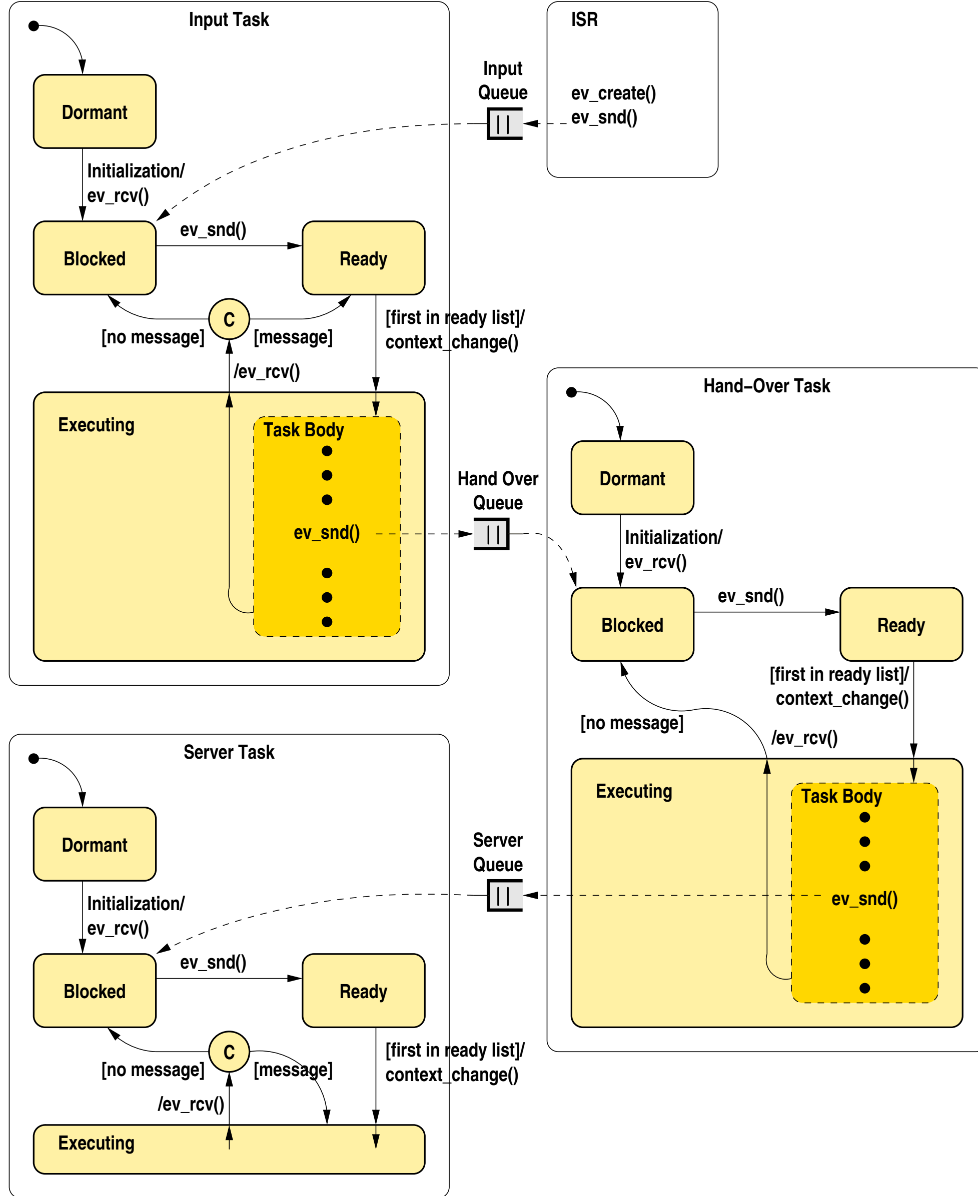

Figure 5.9: Server Model Task States and Structure of Task Body

EDF event create is called from within ISR. As described above, there exists only one single EDF event list for all events in proposed implementation. One linear list walk is needed for EDF event insertion. Upper bound of execution time depends on maximum number of all events, being simultaneously in the system: $N_{E,max}$ (cf. Eq. 5.4).

As a prerequisite to derive $N_{E,max}$, real–time analysis has to proof timely correct behaviour ($t_{p,max} < d_i$). To solve this bootstrapping problem, the following method is proposed:

1. Assumption: System fulfills timing requirements, therefore all processing times of

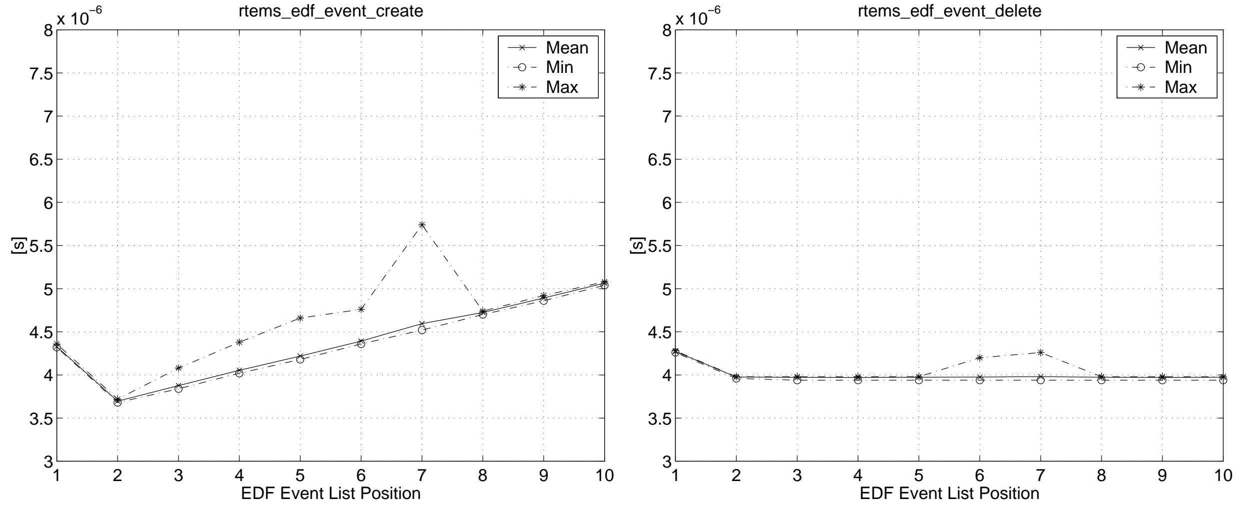

Figure 5.10: Timing: a) EDF Event Create and b) EDF Event Delete

events in all TPS will be equal or lesser than deadline!

2. Estimation of execution times with this assumption and re–check of real–time behaviour.

The influence of EDF event list position on EDF event's creation execution time can be seen in Fig. 5.10 (a). Overhead on list position zero is explained by time needed for EDF timer adjustion (deadline surveillance).

EDF event delete is called from within external task or from output transitions within last TPS task. Execution time will be constant, simply because only well defined list manipulation is needed to delete one event from EDF event list. Timing can be seen in Fig. 5.10 (b). Again overhead on list position zero can be explained by time needed for EDF timer adjustment.

EDF event send and EDF event receive are responsible for transporting triggering events with their deadlines between concurrent respectively consecutive tasks. Both directives depend on list manipulations to fulfill the stipulated EDF execution order. Since the proposed implementation for MEDF scheduling is based on only one type of EDF message queue, the "Server Queue" (SQ), send and receive operations for all queue types will independently rely on the `Add_To_ThreadList_With_EDF` algorithm (cf. App. B.2), which is a linear list walk. A summary of lists involved and processing effort to be derived thereof is given in Table 5.2.

IQ–A First task in TPS (T_{i1}), respectively task representing activity thread is active, i.e. there is no waiting receiver. Thus insertion of message into message queue depends on EDF event list including one linear list walk with a maximum of $N_{E,max}$ steps. An exemplary timing can be seen in Fig. 5.11(a).

Type[a]	Case[b]	EDF event send	EDF event receive	Case[b]
IQ	A	EDF event list ($N_{E,max}$).	Append receiver to (empty) thread queue.	C
	B	EDF event ($N_{E,max}$), thread ($N_{T_{D\neq\infty},max}$), and ready list ($N_{T_{Ready},max}$).	EDF event ($N_{E,max}$), thread ($N_{T_{D\neq\infty},max}$), and ready list ($N_{T_{Ready},max}$).	D
HOQ	A	—[c]	Append receiver to (empty) thread queue.	C
	B	EDF event ($N_{E,max}$), thread ($N_{T_{D\neq\infty},max}$), and ready list ($N_{T_{Ready},max}$).	—[c]	D
SQ	A	For message queue insertion: EDF event list ($N_{E,max}$). For DIP: EDF event list ($N_{E,max}$), EDF thread list ($N_{T_{D\neq\infty},max}$), and EDF ready list ($N_{T_{Ready},max}$).	Append receiver to (empty) thread queue.	C
	B	EDF event ($N_{E,max}$), thread ($N_{T_{D\neq\infty},max}$), and ready list ($N_{T_{Ready},max}$).	EDF event ($N_{E,max}$), thread ($N_{T_{D\neq\infty},max}$), and ready list ($N_{T_{Ready},max}$).	D

[a] IQ: Input Queue, HOQ: Hand–Over Queue, SQ: Server Queue
[b] A) no waiting receiver, B) receiver readied, C) no pending message, D) message available (cf. page 46)
[c] will/must not occur

Table 5.2: EDF event send and EDF event receive list dependencies and processing effort

*Q–C (IQ–C, HOQ–C, SQ–C): Server as well as Activity Thread Model: Only one task per message queue, i.e. thread queue used as wait list will be empty in all cases and for all queue types. Simple append is sufficient, processing effort will be constant. Magnitude of execution time can be seen in Fig. 5.12 (thread queue position 1).

*Q–B (IQ–B, HOQ–B, SQ–B): Receiving task (e.g. in case of an input queue, first task in TPS T_{i1} or task representing activity thread) is blocked. This means, task is waiting for an triggering event. Processing involves all three EDF lists and hence processing effort depends on $N_{E,max}$, $N_{T_{D\neq\infty},max}$, and $N_{T_{Ready},max}$. The EDF event send's dependency on ready list position on the one hand and thread list position on the other is shown in Fig. 5.13.

IQ–D Message available, caller remains ready, but will be preempted through succeeding task in TPS with shorter deadline (processing of an older event, see "Input Task" state diagram in Fig. 5.9). Processing effort is analogous to IQ–B. Execution time dependency on the EDF thread list position is displayed in Fig. 5.11(b).

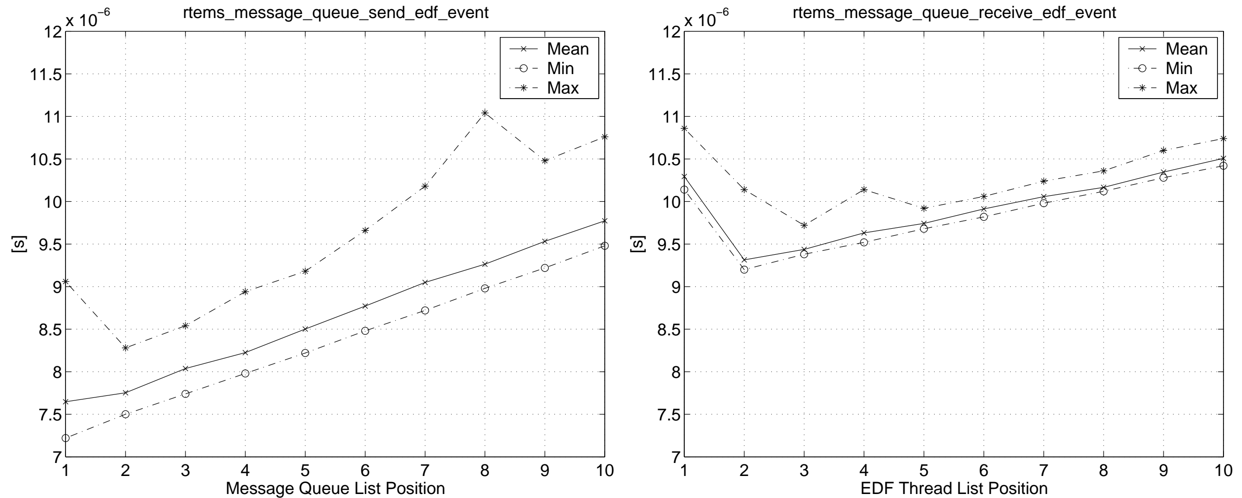

Figure 5.11: Timing: a) Send (No waiting task) and b) Receive (Message available)

HOQ–A Due to sequencing of events of the same type (Lemma 1), there will always be a waiting receiver on an “Hand–Over Queue” (see HOQ–D for details).

HOQ–D When an “Hand–Over Task” finishes the processing of an event, it will conclude its processing with the initial EDF event receive and block. In no case there will be a further consecutive message available, because these events are all stored in an initial input queue (see “Hand–Over Task” state diagram in Fig. 5.9). Thereof it follows, necessary message queue length for an HOQ is zero.

SQ–A Insert message into message queue. DIP (or DCP) must be enabled for predictability, therefore push the receiver’s deadline to deadline to inherit (sender’s deadline) in case it is shorter. Message insertion effort is analogous to IQ–A, DIP deadline manipulation analogous to IQ–B due to the necessary re-insertion of the inheritor thread.

SQ–D Analogous to IQ–D, but receiver will remain in the “Executing” state (cf. “Server Task” state diagram in Fig. 5.9). Processing effort is equal to IQ–B.

5.5 Optimized Server Model Scheduler

The run–time support system as proposed in the previous sections supports both, the SM as well as the ATM software architectures. Since the latter needs an additional EDF thread list, to park blocked threads with non–infinite deadlines, e.g. on semaphores, and due to the straight–forward implementation of the EDF event list as linear delta–deadline list, scheduling operations incorporate some avoidable overhead. Therefore upper bounds for execution times of scheduling directives have to be quite pessimistic. In contrast to this, a scheduler designed exclusively to support only one single code generation strategy will

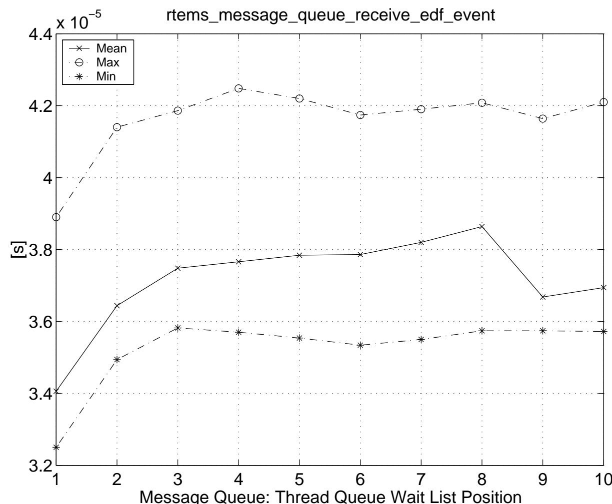

Figure 5.12: Timing: EDF Event Receive (Caller blocks)

have the opportunity to be more optimized. But in consequence, the loss of flexibility has to be taken. This section introduces a scheduler implementation specialized for Server Model architectures.

For predictability reasons, neither semaphores nor additional non–Server–Model tasks will be allowed with this scheduler (and will be obviously unnecessary with the server resp. monitor concept for devices or shared resources to be synchronized). Therefore, one single list structure for threads in the "Ready" state (with non–infinite deadlines) and blocked threads (with infinite deadlines) is sufficient (cf. "Ready List" in Fig. 5.14).

The concept of an additional EDF event list for end–to–end deadline administration is kept, but burst events will now be stored on separate event type lists. Each of them structured as FIFO buffer. Delta deadline management (only with root nodes, e.g. EvA^1, $EvB^1, \dots$) and time–out surveillance (EvA^1 has shortest deadline) will be done exclusively with this doubly linked list. EDF ready list sort order will be kept up–to–date with each EDF event send and EDF event receive operation. For this purpose references between EDF events and EDF threads on the ready list are established resp. broken up during the MEDF operations (see Tab. 5.3 for details).

This SM scheduler provides three different queue types: an input queue (to save bursty events), a hand–over queue (for communication within TPS) and a server queue (for synchronization at shared resources). Different implementations for each queue type provide optimized processing sequences tailored for their dedication.

EDF event create is used to assign an end–to–end deadline to a new EDF event object on the occurrence of a stimulating event. One has to differentiate between two different cases:

1. New EDF event is first or sole instance of this event type:

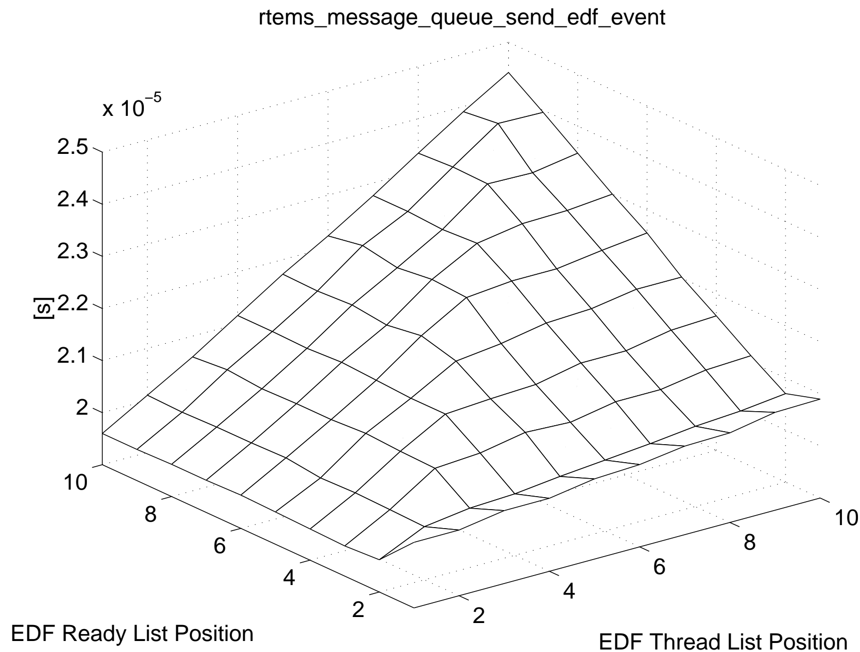

Figure 5.13: Timing: EDF Event Send (Task readied)

Insert new EDF event into deadline sorted EDF event list. This insertion can be done with one linear list walk. The upper bound for processing steps needed depends on maximum length of EDF event list. Since only root nodes have to be searched, this calculates to number of different EDF event types k, i.e. number of different stimulating external or internal event sources.

2. New EDF event is burst event, this means there already exists an EDF event root node of the same event type:

   Append new burst event on FCFS list for this event type (e.g. append EvA^2 on root node EvA^1 in Fig. 5.14). Execution time to be scheduled will be constant.

EDF event delete destroys an EDF event object, which has normally already been processed by the SDL task system. Again two different cases are distinguished:

1. Deletion of bursty events:

   Deletion from FCFS list is of constant complexity. This operation is very unlikely, because burst events should still be unprocessed.

2. Deletion of first event in burst list (root node):

   This operation requires a re-insertion of the consecutive, i.e. now new first event into the EDF event list (e.g. event EvA^2 in Figure 5.14). Again processing effort will be $O(n)$, because only one linear list walk is needed, upper bound of processing steps is k again.

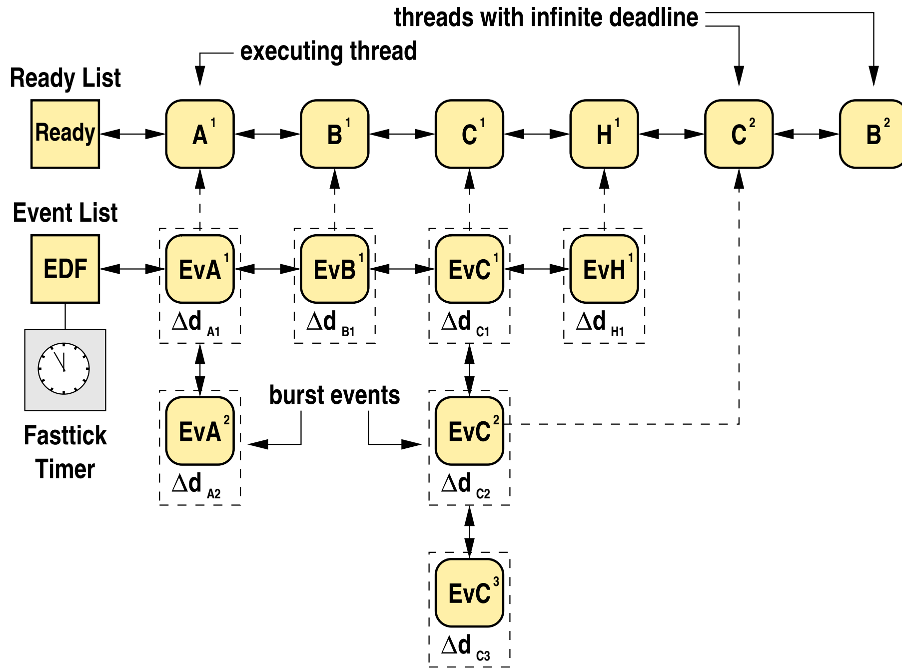

Figure 5.14: List Structures and References for Optimized Server Model Scheduler

EDF event send and EDF event receive A survey of necessary processing steps, required for EDF event send and EDF event receive operations is given in Table 5.3.

IQ–A Sequencing of subsequent events of the same type (Lemma 1). Append burst events to message queue. Absolute deadlines of succeeding events of the same type will be increasing, therefore no comparison with EDF event list is needed.

*Q–C (IQ–C, HOQ–C, SQ–C): Only one task per message queue is allowed with Server Model. Append receiver to empty thread queue and set its deadline to infinity.

*Q–B (IQ–C, HOQ–C, SQ–C): Assumption: EDF event create and EDF event send are consecutive and non–interruptible operations within ISR. With every new event, one have to differentiate between two cases:

1. EDF event to be sent is first event or event with shortest deadline in task system:

   Receiver will automatically be executing thread, i.e. first thread on EDF ready list. Processing steps needed are: Build reference to receiver (Pointer $EvA^1 \rightarrow A^1$) and simply append receiver to empty ready list.

2. There exist already events with more urgent deadlines:

   Reference of transmitted event's left neighbor to its processing thread must have already been built. Insert receiver behind left immediate neighbor event/thread pair. Example of Fig. 5.14: Left neighbor event of EvH^1 is EvC^1. Its processing thread C^1 can be derived through pointer $EvC^1 \rightarrow C^1$. Insert H^1 behind C^1.

Type[a]	Case[b]	**EDF event send**	**EDF event receive**	Case[b]
IQ	A	Append message to message queue.	Append receiver to (empty) thread queue.	C
	B	Insert receiver behind left immediate neighbor event/thread pair.	Receive subsequent (burst) event.	D
HOQ	A	—[c]	Append receiver to (empty) thread queue.	C
	B	Insert receiver behind left immediate neighbor event/thread pair (respectively, append to sender).	—[c]	D
SQ	A	Message queue insertion: EDF event list (with a max. of k root nodes). For DIP: Move receiver **behind** sender.	Append receiver to (empty) thread queue.	C
	B	Insert receiver behind left immediate neighbor event/thread pair (respectively, append to sender).	No ready list manipulation is needed.	D

[a]IQ: Input Queue, HOQ: Hand–Over Queue, SQ: Server Queue
[b]A) no waiting receiver, B) receiver readied, C) no pending message, D) message available (cf. page 46)
[c]will/must not occur

Table 5.3: EDF event send and EDF event receive processing sequences with optimized server model scheduler

Amount of execution time needed for insertion point determination and insertion into ready list will be constant.

For a HOQ or a SQ, receiver could be alternatively inserted directly behind sender (who is processing same event with same deadline).

IQ–D Append receiver for the time being with infinite deadline to EDF ready list. Establish reference from burst event to receiver (cf. event EvC^2 and thread B^2 in Fig. 5.14). On deleting the first event (event EvC^1 in the example), the new first event will have to be reinserted into the EDF event list (see EDF event delete) and the ready list position of its associated thread will be updated too (analogous to IQ–B).

HOQ–A see case HOQ–A on p. 68.

HOQ–D see case HOQ–D on p. 69.

SQ–A A Server Queue unites events of different types, resp. event sources. Therefore deadlines of consecutive events may be unsorted with regard to their deadline value. In consequence, sort order of message queue has to rely on EDF event list, which may have a maximum of k root nodes in total. DIP must be enabled. In case of a shorter deadline of a new incoming event (exactly, if message must be inserted on message queue position number 1), deadline inheritance must take place and receiver (inheritor thread) will have to be moved behind sender thread. Deadline value will be set to the sender's one.

SQ–D Since DIP must be enabled, receiver (inheritor thread) will already have adopted deadline of succeeding event (see SQ–A).

Only EDF event create and EDF event delete will initiate (re–)insertion and manipulation of the EDF event list (list search needed). This can be done with one linear list walk and complexity will depend only on number of different event **types**. Both other directives EDF event send and EDF event receive will rely on list appends, respectively the point for list insertion can be directly derived without list search from the list neighbor of the event transported. Therefore scheduling complexity and hence execution time to be expected will be constant. Single exception: The DIP case of EDF event send, which can be done likewise with a maximum effort of k steps.

As can be seen, MEDF scheduling processing effort is identical to normal EDF task scheduling complexity, but MEDF allows separation of functional design (SDL process network respectively RTEMS task structure) and timing constraint specification with end–to–end deadlines and message communication.

Chapter 6

SDL Real–Time Analysis

Although SDL is widely used in industry for construction of real–time and communication systems, there comes little or no support from methodologies or tools for proving real–time behaviour of an application designed with SDL.

> "Verifying that a time deadline is always met obviously requires a model of real resources. [. . .] Verifying the timing (that is — proving that it is correct) would not be an easy task, even with sophisticated tools." [Ree98]

Most of commonly accepted methodologies which can be found in well known literature rely on mean and/or peak load considerations of computing nodes involved or propose to compare processing performance of the underlying hardware architecture with timing intervals to be achieved.

> "As a rule of thumb the mean peak load on a single computer should not exceed 0.3 in order to give room for statistical peak loads." [BH93, p. 247]

> "Allocate processes to computers such that the mean load on a single computer not exceeds 0.3 *Erlang* of its total capacity. [. . .] Consider the implications of real–time requirements. Calculate the response times for time critical functions and check that requirements will be satisfied. Use priority to ensure fast responses." [ITU94c, p. 106]

> "Whether the time constructs of SDL are sufficient for specific real–time requirements depends as much on the implementation of time–constructs in the final system, as it depends on the particular constructs in the language. [. . .] As a first rule of thumb for implementation, it is possible to use the constructs for system design if the tolerances on the time intervals are ≥ 100 times the average instruction time of the CPU used for implementation. [. . .] These observations are without any theoretical evidence: they are only based on experience with some implementations of SDL and similar timing schemes." [Ols94, p. 101]

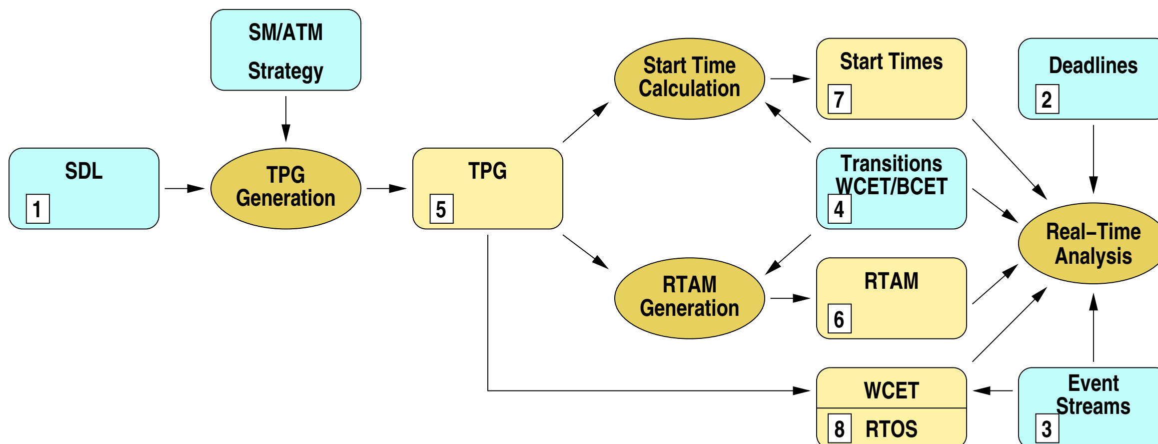

Figure 6.1: Information Flow and Dependencies of Real–Time Analysis

The reader will surely agree these approaches to be insufficient for hard real–time systems, where a prove that timing constraints will be met even in worst case situations is inevitable.

With the goal to enable the automatic verification of real–time requirements, SDL's process activation rules and thus its semantics has been pinpointed to MEDF scheduling. In Sec. 5.2 it has been shown that all processes under MEDF will be activated according to EDF scheduling policy, therefore an EDF analysis methodology can be applied to task systems, automatically generated from SDL specifications, in case MEDF semantics has been preserved (cf. Sec. 4.3).

For this purpose the feasibility prove as introduced by Gresser [Gre93a, Gre93b] can be applied. Gresser's approach suits well to event driven systems due to its capability to take into account even a sporadic behaviour of external stimuli. In the following section Gresser's assumptions regarding the underlying task model and the analysis algorithm itself are shortly explained and, where necessary, adapted according to the predictability requirements of MEDF.

The second part of this chapter deals with the automatic transformation of a SDL system to a network of task nodes as a preparation for the final analysis step. The intermediate actions and the interdependencies of all information necessary to perform the verification can be seen in Fig. 6.1.

In a first step, the so called task precedence graph (TPG, 5) is automatically derived from the signal exchange between SDL processes. Since it includes *all* possible execution paths and thus all alternatives, i.e. all transitions even if they are mutual exclusive due to exclusive start states, the next step uses the worst case execution times (WCET, 4) of transitions to refine the TPG to a network of analysis tasks (real–time analysis model, RTAM, 6). On the other hand best case execution times (BCET, 4) of transitions are used to determine the earliest possible start times of tasks (7) within areas of mutual exclusion. As has been shown in Sec. 4.3, the chosen code generation strategy influences the number of tasks in the final implementation and controls the RTOS directives used. The maximum

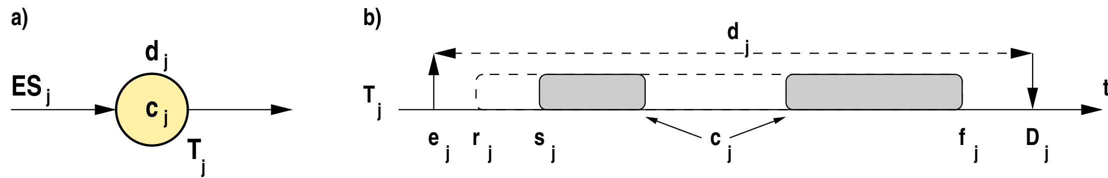

Figure 6.2: Graphical Representation of a Task Node [Gre93a] and Analysis Parameters

number of tasks as well as the maximum possible number of events processed concurrently in the system in turn affect the upper bounds of execution times of RTOS directives (8).

6.1 Real–Time Analysis

In this section, Gresser's feasibility analysis for event driven real–time systems [Gre93a, Gre93b] is shortly introduced. For this, the assumptions regarding the underlying task model, the methodology itself, and the environment's influence on the system's real–time behaviour are surveyed.

6.1.1 Task Model

A task of a software system is represented through an analysis task node that holds the following parameters[1]: A single stimulating event E_j (time of occurrence e_j) releases the task T_j (*release time* r_j) and, when dispatched, it starts with its computation (*start time* s_j). Preemption caused by concurrent tasks may occur repeatedly before the processing of this task will finally end at *completion time* f_j. For each task, an upper bound (*worst case execution time* c_{max}) and a lower bound (*best case execution time* c_{min}) for the overall computation time without preemption as well as a worst case response time (relative *deadline* d) must be given. As can be seen in Fig. 6.2(b), the timing constraint d will be fulfilled, if the task completes its processing before the absolute deadline $D_j = e_j + d_j$. Fig. 6.2(a) shows the graphical representation of an analysis task node with the most important parameters.

Conceptually, Gresser assumes a fixed internal task structure, consisting of non interruptible sections at the beginning and at the ending of each analysis task, enclosing a fully preemtable task body. The opening section may consist of blocking operating system directives like *semaphore obtain* ($P()$–operation) and/or *receive* operations. The closing section in turn may only include non blocking directives like *send*, *semaphore release* ($V()$–operation), calls of timer services and finally *input* or *output* operations. Thus, the resulting task communication is as well asynchronous and consequently comprises SDL's message exchange paradigm.

[1] As can be seen, these parameters are identical to task model and environment parameters used to prove the predictability of MEDF in Sec. 5.2 (cf. Tab. 5.1 on page 47), but are shortly re-introduced for readability purposes.

Comparing these assumptions to the structures of tasks, server model task as well as activity threads, the following differences can be observed:

- Position of terminating send/out operation in Server Model tasks (cf. Fig. 5.9) is varying and may reside within task body (*send* before *task* symbol in SDL).

  Since a MEDF send is non blocking and since the now released receiver task will be activated only after the sender blocks again at its initial receive (MEDF Sequencing, Lemma 1), the send position has no influence on scheduling order. Thus, the introduced analysis task model will fit to SM tasks too.

- Pairs of semaphore obtain and release operations protect mutual access to common state data in Activity Thread Model tasks within task body.

  This structure originates when several state transitions of consecutive SDL processes are packed into one single task (the activity thread) to avoid unnecessary message exchange. The data synchronization of server processes will be maintained by semaphores enclosing a complete state transition. In this case, an ATM task has to be split up into several analysis task nodes, i.e. one node for the section before the $P()$–operation, one node for the state transition protected by the semaphore and finally one node for the section after the releasing $V()$–operation.

With this, the resulting analysis model for ATM task systems will resemble an analysis task network of the appropriate SM task system. Only difference: Analysis nodes of TPSs triggered by one single event type, e.g. TPS_1 to TPS_3 in Fig. 5.1 may be merged to one single analysis node.

6.1.2 Analysis Algorithm

The schedulability analysis algorithm requires the *Event Streams* ES_j (cf. Sec. 3.4) to be transformed into an *Event Function* $E_j(I)$, expressing the maximum number of events per interval I (m event types with $n_j, j \in [1, m]$ event tuples each.).

$$E_j(I) = \sum_{i=0}^{n_j-1} \begin{cases} 0 \; ; & I < a_{i_j} \\ \left\lfloor \dfrac{I - a_{i_j}}{z_{i_j}} + 1 \right\rfloor \; ; & I \geq a_{i_j} \wedge z_{i_j} < \infty \\ 1 \; ; & I \geq a_{i_j} \wedge z_{i_j} = \infty \end{cases} \tag{6.1}$$

Figure 6.3 shows $E(I)$, derived from the sample Event Stream[2] in Figure 3.5.

The *computation function* $C(I)$ is defined as maximum computation time requested and due within interval I. For a single task T_j $C_j(I)$ can be calculated easily from $E_j(I)$ by shifting by the deadline d_j and multiplication with the WCET $c_{max,j}$ (Fig. 6.3). The resulting $C(I)$ for a number of *independent* tasks on a computing node is simply the sum of all $C_j(I)$ functions

[2] $ES = \left\{ (0 \quad 7)^T, (1 \quad 7)^T, (3 \quad 7)^T \right\}$

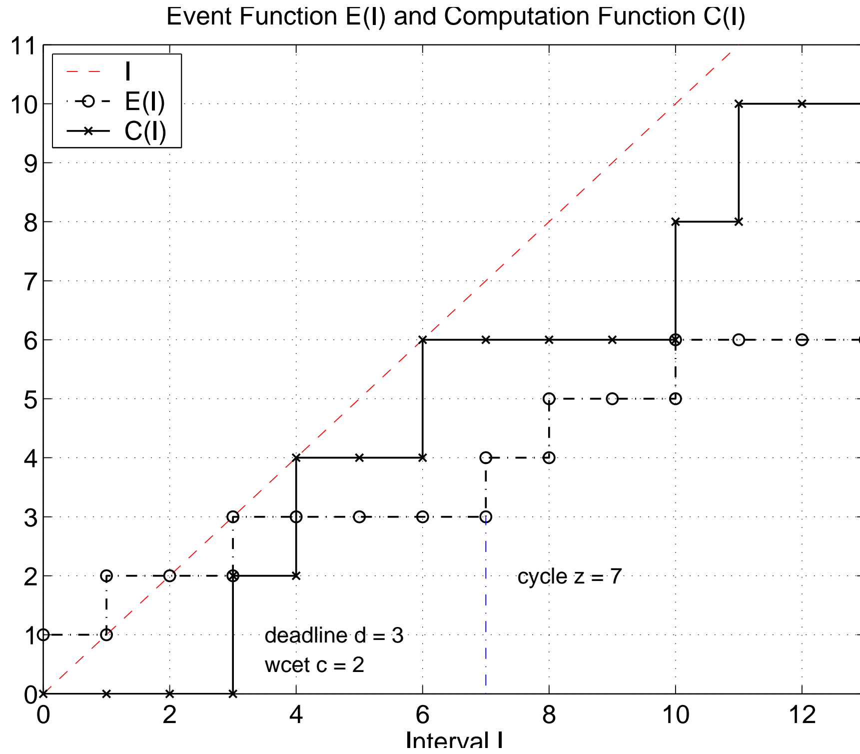

Figure 6.3: Event Function $E(I)$ and Requested Computation Time $C(I)$

$$C(I) = \sum_j C_j(I) = \sum_j E_j(I - d_j) \cdot c_{max,j} \quad (6.2)$$

For EDF Gresser proved that all tasks on one processing node meet their deadlines, if the resulting $C(I)$ always runs under the bisector which specifies the available computing time in each interval.

$$C(I) \leq I \quad \forall \quad I \geq 0 \quad (6.3)$$

Task Dependencies Communication between SDL processes and thus within the resulting task networks is responsible for task dependencies like precedences and mutual blocking. Precedence relationships between tasks under MEDF leave the schedulability uninfluenced (Theorem 1 in Sec. 5.2.1), and thus do not need further consideration during the analysis. On the other hand, dependencies through message blocking on server tasks[3] can be resolved by independently taking into account all execution times of all competing parts of server tasks (i.e. all state transitions) plus an additional necessary deadline manipulation. For this, the deadlines of the resulting analysis nodes that are possibly blocked

[3] The same is true for semaphores (Sec. 5.2.4).

must be adapted at most to the deadline of the most urgent transition[4] (Theorem 2 for DCP, Theorem 3 for DIP in Sec. 5.2.3).

Event Dependencies Task dependencies will tighten the computation time requirements to be scheduled. In contrast thereto, event dependencies can be used to give a closer specification of the worst case behaviour of triggering events. Without these additional minimum intervals between (two) event types, a simultaneous occurrence of events would have to be considered. The methodologies provided by [Gre93a] transform dependent event streams into one event stream with less stringent computation time requirements.

A delay between two analysis tasks and thus as well a delay caused by a timer signal with its set operation depending on an external event can be considered as an event dependency between triggering external event and the delayed internal event stream, i.e. the ES of the timer signal.

6.1.3 Influence of System Environment and Timer Task

Overall available computation time per interval $C(I)$ must be shared between tasks that are generated automatically from the SDL system model (running in the MEDF priority level) and supplementary tasks and ISRs to connect the SDL system part to the environment. Activations of all necessary ISRs as well as the worst case expirations of all set timers are given through the timing constraint specification or can be derived thereof.

- The triggering of ISRs needed to translate external events (k event types) to MEDF signals is specified through the given timing constraints (ES_1, ES_2, ..., ES_k).
- The cyclic clock ISR to increment the clock tick is activated with period z_{Cl}, thus owns the event stream $ES_{Clock-ISR} = (0 \quad z_{Cl})^T$.
- Activations of the timer task by expired timers with a subsequent send of the appropriate timeout signal are described by an event stream that consists of a concatenation of all single timer ES specifications.

In contrast to MEDF tasks, ISRs and supplementary tasks are scheduled with fixed priorities. ISR priorities are given through HW interrupt priorities and obviously are as well higher than the MEDF priority level, thus disabling preemption for MEDF tasks. From the MEDF tasks point of view, an ISR or supplementary task completely seizes the processor for an interval that is of equal length to its processing time (cf. c_{ISR_1}, c_{ISR_2} in Fig. 6.4). Assuming a simultaneous occurrence of all events (i.e. the critical instant), the (longest possible) busy period I_{BP} of all ISRs and high priority tasks will emerge

[4][Gre93a] additionally provides a methodology for deadline manipulation to deal with nested mutual exclusion in complex task structures. Since nested mutex constellations can not occur with SDL server processes, this methodology needs not be introduced here.

Figure 6.4: Busy Periods due to ISRs and Supplementary Tasks

starting with interval $I = 0$. In this case, busy period[5] denotes the fact that all available computation time is consumed by ISRs or further supplementary tasks. It is necessary to take into account I_{BP} as a minimum amount of laxity to take care of delays caused by disabled preemption to fulfill all specified deadlines.

For example, the occurrence of an external event during a timer set or reset operation will be immediately recognized by an ISR, but the thereof activated SDL task will be delayed until the timer task has completed its operation (cf. Fig. 4.4).

As can be seen in Fig. 6.4, I_{BP} is already included in the ISR's computation function $F(I)$. Thereof it follows, all deadlines of MEDF tasks will be met, if their overall computation function $C(I)$ requests less processing time than the remaining available computation time [Fär00].

$$C(I) \leq I - F(I) \quad \forall \quad I \geq 0 \tag{6.4}$$

[5]Starting with the highest priority ISR, one has to iteratively consider the processing times of ISRs/tasks with higher priority preempting the ISR/task under evaluation until the lowest priority ISR/task can complete its processing and no further bursty events are pending.

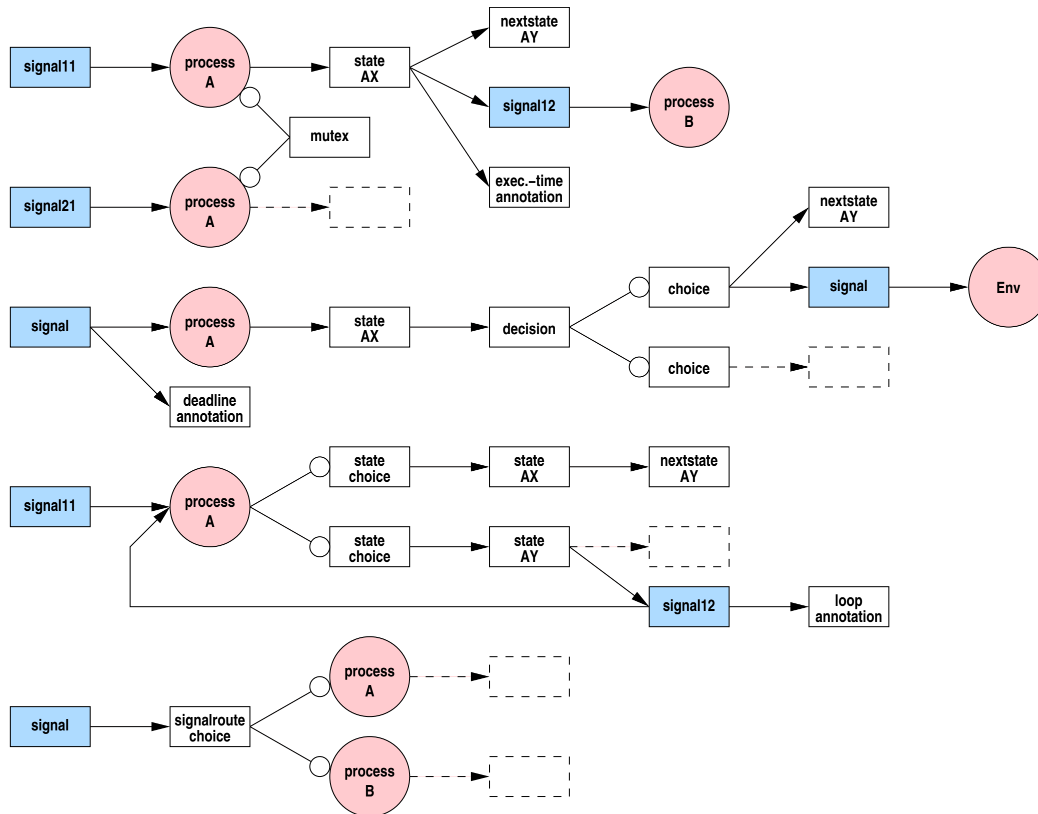

Figure 6.5: Syntax of Task Precedence Graph and its Annotations

6.2 Derivation of the Real–Time Analysis Model

The automatic generation of the analysis task network has to be split up into two phases. In a first step, all information of an application's SDL system model, like structure, behaviour and the interface to the environment is used to derive a Task Precedence Graph (TPG). Thereupon, worst case execution times of processes, respectively state transitions are evaluated to find the complete Real–Time Analysis Model (RTAM), i.e. the worst case path in the TPG.

In contrast to the RTAM derivation, the best case execution times of transitions can be used to calculate earliest possible start times of server processes to alleviate the effects of deadline reduction necessary to consider priority inversion avoidance during real–time analysis.

6.2.1 Task Precedence Graph (TPG) Synthesis

Gresser's real–time analysis model consists of a network of analysis task nodes and thus only shows the communication and synchronization structure of a software system. In contrast to this, the Task Precedence Graph (TPG) as introduced here additionally includes behavioural information. State transitions are attached to nodes representing a

Figure 6.6: TPG Synthesis Example SDL System

SDL process respectively a resulting analysis task (cf. Fig. 6.5). The signal(s) sent to trigger consecutive processes are shown within these transitions.

To build up the TPG, a SDL parser flattens the hierarchical SDL blocks into a network of "leaf" SDL processes, whereby only processes in the priority level reserved for hard real–time are analyzed. The main translation principle is to map one SDL state transition to one analysis task node. The transformation algorithm ([KF98], [Lar98]) starts at each triggering external event or each triggering internal SDL Timer signal and traces the transported deadline tokens through the complete SDL system. Consecutive SDL processes are identified by means of *send* statements. The correlating analysis tasks, respectively state transitions, can be resolved by the corresponding *receive* statements. The algorithm will

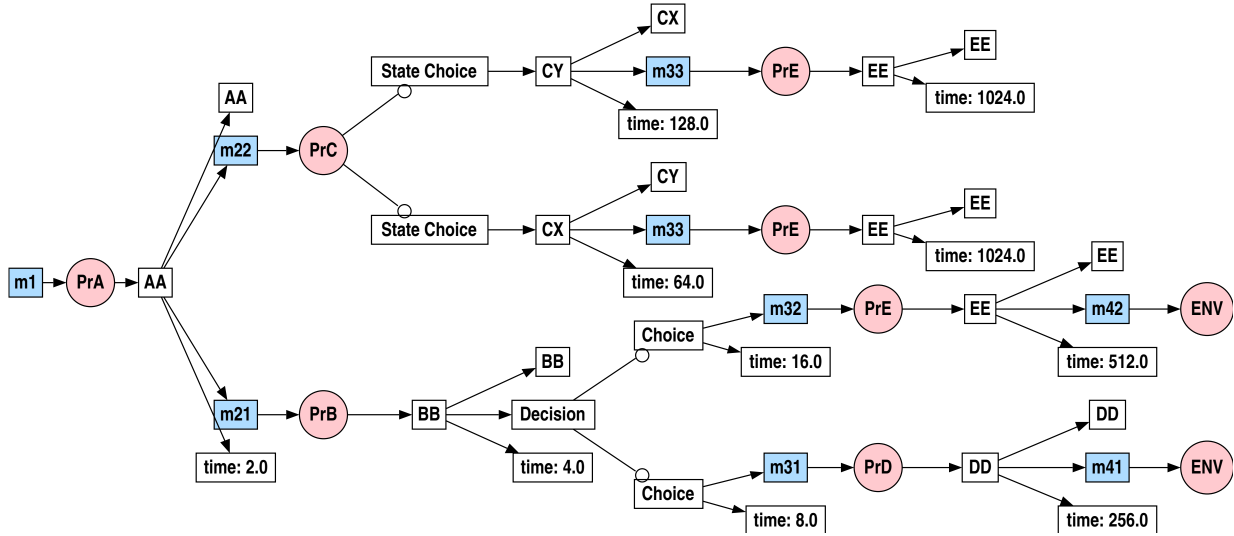

Figure 6.7: Task Precedence Graph Including All Transition Alternatives

terminate, if no more *send* statements can be found in state transitions, or if a *send* is addressed to the system environment (endpoint of end–to–end deadline).

SDL server processes, i.e. processes responding on several different messages, stimulated by different *ES*s are mapped onto several analysis nodes (number of nodes corresponds to number of different event types to be responded). These nodes are additionally marked to be processed mutual exclusively (mutex annotation in Fig. 6.5).

There are two kinds of forks possible within the TPG, enumerations or choices. The first kind will occur, if several *send* statements[6] are included within the same state transition. The processing times of all these branches contribute altogether to the overall computation time. Choices will be caused by *decisions* in state transitions, if the same message is consumed in different states (state choice), or if several *signalroutes* appear on the receiver side of a block to block communication versus a *channel* (signalroute choice). Forks in SDL *signalroutes* lead to TPG choices, since only one of several possible receivers will (non–deterministically) process the message sent. Processing times belonging to branches originating from a TPG choice only have to be taken into account alternatively in the real–time analysis (cf. Sec. 6.2.2).

Loops in a TPG will emerge, if signals are sent back to predecessor processes. To uncover this effect, each already parsed state transition will be marked. When this transition will be re-reached during the TPG synthesis, the message currently traced by the parser must be tested, whether it is in the set of sent signals of the current TPG. If this is the case[7], the actual task node corresponding to the marked state transition will be signed as a loop start– and endpoint.

[6] Not necessarily with different receivers. In this case, the succeeding TPG is simply triggered repeatedly and therefore must be replicated several times.

[7] Otherwise, this message belongs to a different event type and thus, the task node under investigation is simply a server process.

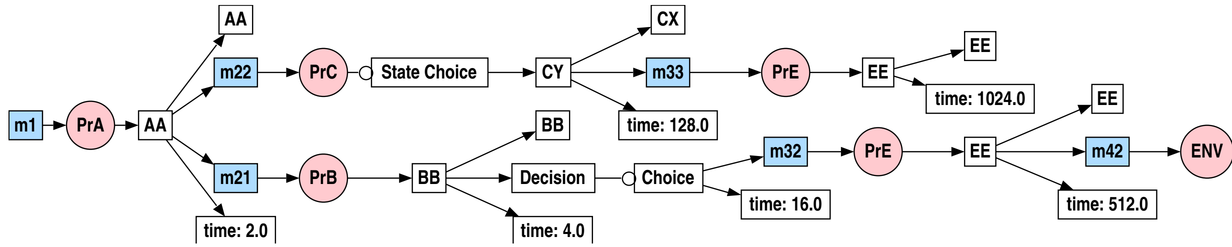

Figure 6.8: Worst Case Task Precedence Graph

The resulting TPG will consist of several independent chains of analysis tasks, each being triggered by a different type of ES. The single branches of the TPG share regions of mutual exclusion. All analysis nodes in one precedence system have the same deadline, leading to the effect that different SDL state transitions originating from the same SDL (server) process may have different timing constraints.

As an example, the generated TPG of a simple exemplary SDL system (Fig. 6.6) is depicted in Figure 6.7.[8] The sending of messages m_{21} and m_{22} in the state transition $AA \rightarrow AA$ in process Pr_A has to be mapped to the two main TPG branches (type enumeration). The *decision* ($a < b$) in process Pr_B leads to two TPG choices, headed by the SDL signals m_{31} and m_{32}. A state choice emerges due to the alternative *receive* statements in states CX and CY in process Pr_C. As can be seen, there exists *no* area of mutual exclusion for process Pr_E, since message m_{32} as well as m_{33} are triggered from the same event stream ES_1, thus process Pr_E is no server process.

6.2.2 Derivation of the Worst–Case TPG

Branches in the TPG, caused by state, decision and signalroute choices are resolved by evaluating the specified (or measured) execution times to find the worst case path in the TPG. This means, all state transitions belonging to a signal path that do not contribute to the overall worst case computation time are eliminated.

Starting with the root node, i.e. the triggering signal, execution time annotations of all sub-graphs will be recursively summed up. With this, each child returns its WCET to its parent node. Depending on the type of the parent node, choice or enumeration, one has to add the maximum of all sub-node's WCETs (node Pr_C and node *decision* in Pr_B in TPG of Fig. 6.7) in a choice. In case of an enumeration, the sum of all returned WCETs has to be calculated and added to the node's own WCET (e.g. node Pr_A).

An advantage of this recursive approach (cf. Fig. 6.9) is its ability to handle loops in the TPG as well. If the function `TPG_next_subnode()` is called on a node connected by an arc that is directed to an ancestor node in the TPG, the annotated loop count of the arc will be decremented. After reaching zero, this arc and thus its connected subtree will not be considered for the WCET calculation anymore.

[8]TPG visualization is based on the graph drawing tool *daVinci* [FW94].

```
max_ct Worst_Case_Execution_Time( TPG_node node ) {
  max_ct ct  = 0;
  max_ct lct = 0;

  while ( TPG_node_has_subnodes( node ) {
    lct = Worst_Case_Execution_Time( TPG_next_subnode( node ) );

    if ( TPG_node_is_of_type_ENUM( node ) ) {
      /* sum of all subnode WCETs */
      ct = ct + lct;
    } else if ( TPG_node_is_of_type_CHOICE( node ) ) {
      /* find maximum WCET of all subnodes */
      if ( lct > ct ) {
        ct = lct;
      }
    }
  }

  ct = ct + node.ct;
  return ct;
}
```

Figure 6.9: Recursive TPG WCET Calculation (simplified)

Figure 6.8 shows the worst case TPG, that is the Real–Time Analysis Model, derived from the TPG of the example SDL system (Fig. 6.6 and Fig. 6.7). As can be seen, there is only one branch of each choice left.

6.2.3 Calculation of Start Times

Deadline reduction during real–time analysis is necessary for tasks that will be processed mutual exclusively to take into account possible additional blocking times caused by priority inversion avoidance strategies. Since shortening deadlines will decrease the slack time available to schedule the requested computation, earliest possible start times of server tasks can be used to minimize these necessary deadline reductions (see Equation 5.1).

For this purpose, the best case execution times of all tasks prior to a server task will simply be added up. In case of several possible activation chains, the minimum resulting start time has to be chosen ($s_{T_S^x,min}$). A known dependency e_{xy} (minimum interval) between an event E_x triggering the blocking and an event E_y triggering the blocked part of a server task can be finally added to $s_{T_S^x,min}$.

Conclusion

As has been shown above, real–time analysis of SDL systems is threefold. On the one side, processing time requirements (the overall computation function $C(I)$) resulting from the SDL system itself can be derived automatically, if worst and best case execution times of all system model parts are given.

In addition to computation times of state transitions, processing times of the underlying RTOS directives must be known as well. Upper bounds of their execution times depend on the configuration of the final software system, i.e. implementation model and scheduler used, number of tasks involved and maximum number of stimulating events.

Finally, tasks and functions of the infrastructure, necessary to connect the SDL model to its environment are influencing the schedulability of the whole real–time application. Their processing impose additional blocking times to the model tasks and have to be taken into account in the analysis. The interface to the embedding system is more seldom subject to changes during the design process compared to the application core itself. Due to this fact, execution times of these supplementary tasks only will have to be measured once.

It must be stressed that the designer's modeling style may have harmful consequences on the schedulability of the whole system. A fine granular system structure (design goal: "emphasizing the data flow") will lead to complex message exchange between SDL processes. State transitions then will be very simple and the portion of the application to be scheduled will be small or at most in the order of magnitude compared to the overhead of the run–time system (cf. Sec. 7.2). This negative effect may be weakened through the application of the activity thread model during code generation, however the overhead imposed through the environmental functions (ISRs etc.) will remain the same.

Chapter 7

Case Study and Evaluation

This chapter provides a step by step demonstration of all phases necessary to evaluate the real–time characteristics of a SDL system. Its organization follows the proposed modus operandi as shown in Fig. 6.1.

The "Attitude and Orbital Control System" (AOCS) of the communication satellite "Olympus" exemplifies an application with tight timing requirements. Its real–time properties have been analyzed by Burns et. al. [BWBF93] in a similar case study. Showing a HRT–HOOD [BW95b] specification of the AOCS, deadline monotonic schedulability analysis has been used to prove the timeliness of an Ada implementation on a Motorola 68020 board. Real–time analysis requires that processing times of the application itself, i.e. execution times of algorithms and state transition times are known. The cited evaluation as well as the case study in this chapter rely on results of a performance analysis for Motorola 68020 based processing units. In [BWBF94] execution time estimations of the AOCS are given.

7.1 Olympus Attitude and Orbital Control System

The "Olympus", launched in July 1989, was planned as a platform for telecommunication applications, as well as for television and radio broadcasting over Europe. The three–axis–stabilized satellite suffered its most serious in-orbit problems when during 29th of May 1991 it lost its earth lock and automatically entered "Sun Acquisition Mode". Due to thereof initiated improper operator commands, the satellite turned its solar panels side–on to the sun and froze. Fortunately, according to a fall–back strategy the thrusters fired autonomously and control could be regained. This incident gave reason to reevaluate the real–time properties of Olympus' control system and lead to the former cited case study.

As with most geostationary satellites, an "Attitude and Orbital Control System" (AOCS) has the task to maintain the spacecraft's position and orientation on its orbit, such that broadcast antennas remain directed to earth. This is achieved by forcing roll, pitch as well as yaw angle of the satellite to zero during this so called "Normal Mode" operation.

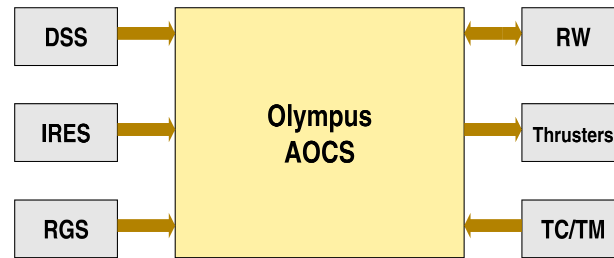

Figure 7.1: Olympus AOCS Sensors and Actuators

On "Olympus", necessary angular momentums can be either created by accelerating respectively decelerating four reaction wheels (RW) or by thruster firing. An "Infra–Red Earth Sensor" (IRES) provides the earth vector (roll, pitch). Yaw angle derivation is based on an integration of rate–gyrometer data (RGS, gyro data is received approximately once per 100 ms) and thus is subject to a constant drift. Therefore, for two spells per day, when sun reaches a specified position, the gyro has to be re–calibrated using the "Digital Sun Sensor" (DSS) based yaw. A survey view of AOCS input/output data flow can be seen in Fig. 7.1.

Beneath the Normal Mode control loop with a cycle time of as well 100 ms, additional attitude control functions have to be processed.

- When the speed of any reaction wheel exceeds a preset threshold, momentum dumping is initiated. Burst of thruster firings compensate the speed reduction of the wheel in question.
- Ground operators have the possibility to influence the satellite's behaviour through a telecommand function. It allows to preset control values, to choose a dumping mode (zero dumping vs. normal dumping) or to trigger the gyro calibration by hand (minimum inter–arrival time is 190 ms). Telemetry requests satellite equipment status at most every 62.5 ms.

In contrast to the original hardware setup, where a serial bus connects sensors and actuators to the AOCS, a memory mapped access to input and output data is assumed for simplicity. Interrupts are released on gyro data receival or telecommand and telemetry status data request. Further sensor data as well as RW speed is polled by either control loop or additional cyclic tasks (Timer interrupt).

7.1.1 SDL System Model

The SDL system of the AOCS models only the control structure of the application layer. This means, that process decomposition and data exchange as well as activation rules are shown, but algorithms and attitude control laws itself are hidden in comments, respectively in calls to appropriate procedures.

A survey view of the AOCS SDL system with its first and second level decomposition included can be seen in Figure 7.2. DSS, IRES and RW data readings as well as process

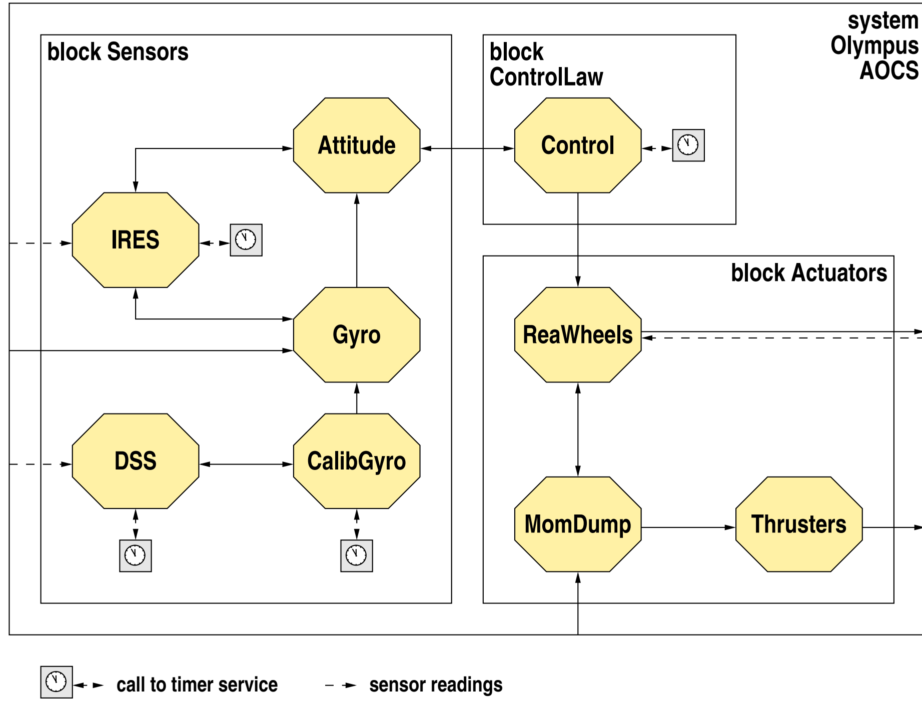

Figure 7.2: AOCS SDL System Survey

activations by timer expiration can not be expressed with SDL on a structural level, i.e. system or block layers, but are incorporated for clarity reasons.[1] Signal names are not shown. The complete AOCS system with all SDL process descriptions can be seen in App. A.1.

7.1.2 Timing Constraints

Table 7.1 summarizes event streams of AOCS input data and shows the appropriate deadlines to fulfill. Gyro data is received without request. Since this sensor has a different clock there may be some jitter[2] ($j \leq 1$ ms) on the incoming data (cf. Fig. 7.3 with $x = z - 2j$). IRES roll and pitch angles are polled with a frequency of once every 100 ms.

Momentum dumping is initiated very rarely. Therefore, the minimum inter–arrival time of the "Critical Wheel Speed" event is considered to be by order of magnitude longer than the longest cycle time in the system ($z_{DSS} = 1000$ ms) and thus modeled with $z_{CWS} = \infty$.

The DSS announces sun presence twice a day and subsequently initiates "Calibrate Gyro". Thereof it follows, both events can not occur simultaneously.

[1] Additions are not conform to SDL syntax.

[2] Without jitter, a cyclic event is described by a single event stream tuple $\left\{\binom{z}{0}\right\}$ with cycle time z (Fig. 7.3 (a)). Considering jitter, the first tuple $\left\{\binom{\infty}{0}\right\}$ of ES_{Gy} specifies a single event in an interval 0 with consecutive events not earlier than $\Delta x, \Delta x + z, \Delta x + 2z, \ldots$ leading to an additional event tuple $\left\{\binom{z}{x}\right\}$ with $x = z - 2j$ (Fig. 7.3 (b)).

Event Type	Event Stream	Deadline
Gyro Raw Data[a]	$ES_{Gy}=\left\{\binom{\infty}{0}\binom{100}{100-2j}\right\}$	$d_{Gy}=100$
IRES Data Processing	$ES_{IDP}=\left\{\binom{100}{0}\right\}$	$d_{IDP}=100$
Control Law	$ES_{CL}=\left\{\binom{200}{0}\right\}$	$d_{CL}=200$
Critical Wheel Speed	$ES_{CWS}=\left\{\binom{\infty}{0}\right\}$	$d_{CWS}=100$
Digital Sun Sensor	$ES_{DSS}=\left\{\binom{1000}{0}\right\}$	$d_{DSS}=1000$
Calibrate Gyro	$ES_{CG}=\left\{\binom{1000}{0}\right\}$	$d_{CG}=1000$
Telecommands	$ES_{Tc}=\left\{\binom{190}{0}\right\}$	$d_{Tc}=190$
Telemetry[b]	$ES_{Tm}=\left\{\binom{62.5}{0}\right\}$	$(d_{Tm}=62.5)$

[a] Cyclic occurrence with jitter j.
[b] Functionality is not covered by SDL model, but influence on interrupt load will be considered.

Table 7.1: AOCS Event Streams and Deadlines

Finally, the main control loop has to be executed within a 200 ms cycle and deadline. The "bursty" behaviour of operator commands (Telecommand) could be taken into account with further event tuples to decrease the processor demand imposed through interrupt load.

7.1.3 Computation Times of MEDF Run–Time System

Worst Case Execution Times of MEDF directives considering a scheduler[3] as proposed in Section 5.3 mainly depend on three parameters: maximum number of EDF events $N_{E,max}$ being processed simultaneously, maximum number of tasks $N_{T_{Ready},max}$ in a ready state, and maximum number of tasks $N_{T_{D\neq\infty},max}$ with non–infinite deadline on EDF thread list (cf. Sec. 5.4).

Number of different event types k in this example is $k = 8$ (cf. Tab. 7.1), but can be reduced to $k = 7$, when taking into account mutual exclusive processing modes "DSS"/"Calibrate–Gyro". Taking into consideration that momentum dumping will be initiated on threshold overrun from within the control law activity thread, i.e. the "Control Law Timer Event" will be just forwarded, k further reduces to $k = 6$. Since telemetry requests are considered soft real–time, no EDF events will be created here, i.e. $k = 5$.

Each message queue (of type IQ) saves a maximum of $N_{E_i,max} = E_i(d_i - c_{max,i})$ incoming

[3] This case study concentrates on the MEDF scheduler supporting Server Model as well as Activity Thread Model implementations. MEDF send/receive directives of the "Optimized Server Model Scheduler" (cf. Sec. 5.5) would have constant run–times, except EDF event create or EDF event delete. Processing times of the later depend only on the maximum of the EDF event list length.

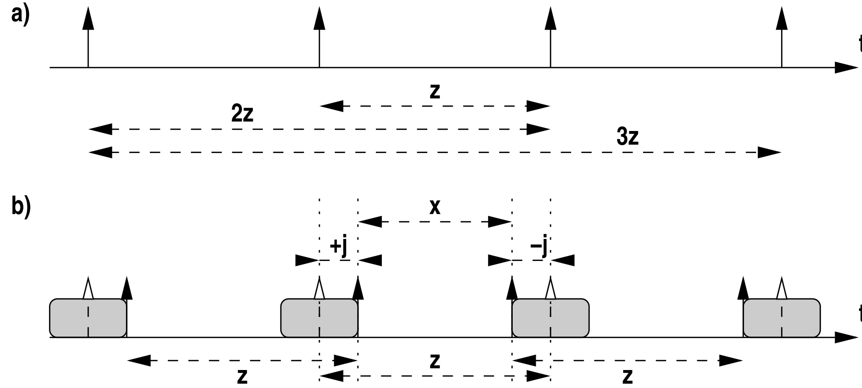

Figure 7.3: Cyclic Event a) without and b) with Jitter

events. With given event streams (only one single event tuple per event type) and deadlines, this can be simplified to $N_{E_i,max} = E_i(d_i) - 1 = 1$. This means, there will be at most one additional event saved per incoming message queue, because there exist no event bursts and processing of each event in the appropriate activity thread will be finished before subsequent events arise. This will be even true for gyro raw data events, if the jitter is smaller than the worst case processing time in this activity thread.

With this, the total number of events calculates to $N_{E,max} = 10$ (Eq. 5.4). For simple periodic task activations this number can be regarded as very pessimistic. Only if the processor is charged to or near its maximum, the processing of a few events will not be completed before their successor event will be created.

Assuming a Server Model implementation strategy, the final software architecture would contain 5 sensor tasks, 1 control task, and 3 actuator tasks (cf. SDL model in Sec. 7.1.1), i.e. a total number of 9 MEDF tasks. With an Activity Thread Model realization the overall task number would reduce to 6 MEDF tasks, with either "DSS" or "Calibrate Gyro" exclusively executing.

Taking a closer look to the resulting task precedence systems (cf. Fig. A.16), the maximum number of tasks waiting on the schedulers ready list as well as the maximum possible number of tasks with non–infinite deadline on the EDF thread list can be derived to $N_{T_{Ready},max} = N_{T_{D\neq\infty},max} = 9$, which is below the worst case to be assumed by Eq. 5.2 and Eq. 5.3. This is due to the fact that TPS_{IDP} as well as TPS_{TC} consist of only one task node and TPS_{DSS} and TPS_{CG} operate in mutual exclusive modes only.

In the following, a closer look to the single MEDF directives will be taken. Upper bounds for their processing times in the AOCS application are derived on the one hand from the measurements[4] as carried out on the Galileo Evaluation Board (Sec. 5.4) and on the other hand from the influencing parameters as calculated above.

To evaluate the amount of overhead imposed through the MEDF run–time system itself

[4]For this case study, the mean times of measurement data are used. Influence of jitter on the run–times caused by cache flushes, pipeline stalls or ISR preemption will be ignored. Prediction and worst case estimation of these effects are not focus of this work.

RTEMS Message Queue Operation	Galileo Cycles[a]	Galileo [μs]	M68020 [μs][b]	$\frac{c_{M68020}}{c_{Galileo}}$
rtems_message_queue_create	5167	103.34	207	2.00
rtems_message_queue_delete	643	12.86	85	6.61
rtems_message_queue_send				
no waiting tasks	233	4.66	103	22.10
task readied — returns to caller	422	8.44	107	12.68
task readied — preempts caller	956	19.12	129	6.75
rtems_message_queue_receive				
message available	180	3.60	87	24.17
message not available — NO_WAIT	103	2.06	51	24.76
message not available — caller blocks	1100	22.00	119	5.41
rtems_message_queue_flush				
no messages flushed	59	1.18	36	30.51
messages flushed	83	1.66	46	27.71
rtems_message_queue_urgent				
no waiting tasks	231	4.62	103	22.29
task readied — preempts caller	955	19.10	130	6.81
task readied — returns to caller	441	8.82	107	12.13

[a] MIPS R4650 processor clock frequency is 50 MHz.
[b] Motorola MVME135 with a 20 MHz 68020 CPU according to [OAR96].

Table 7.2: Execution times of RTEMS message queue directives

and necessary additional I/O (environmental) functions like ISRs and timer facilities, the Galileo processing times will be translated to M68020 processing times by multiplication with a constant delay factor.

As can be seen in Table 7.2, which compares execution times of RTEMS message queue directives on both architectures as provided by the RTEMS timing test suite, ratio of run–times varies widely. This is due to the different characteristics of the underlying processing units on the one hand and different memory layouts, internal bus structures with different clock frequencies on the other. For simplicity reasons, only a constant (average) delay factor $d = c_{M68020}/c_{Galileo} = 10$ will be assumed which is comparable to the Mips R4650[5] to Motorola M68020[6] performance ratio. A summary of all estimated execution times is given in Table 7.3.

[5] Galileo Board, Mips R4650 CPU with 50 MHz clock frequency, RISC architecture with 1 cycle per instruction, performance 50 MIPS.

[6] Motorola 68020 CPU with 20 MHz clock frequency, CISC architecture with about 6 cycles per instruction, performance 5.2 MIPS [Boy96].

	Galileo [μs]	M68020[a] [μs]
EDF event create	5.2	52
EDF event delete	4.3	43

Type[b]	Case[c]	**EDF event send**			Case[c]	**EDF event receive**		
		Galileo [μs]	$c_{max_{A,B}}$ [μs]	M68020[a] [μs]		Galileo [μs]	$c_{max_{C,D}}$ [μs]	M68020[a] [μs]
IQ	A	7.7	24.0	240	C	34.1	34.1	341
	B	24.0			D	10.4		
HOQ	A	—[d]	24.0	240	C	34.1	34.1	341
	B	24.0			D	—[d]		
SQ	A	31.7	31.7	317	C	34.1	34.1	341
	B	24.0			D	10.4		

[a] Processing times estimated with delay factor $d = 10$.
[b] IQ: Input Queue, HOQ: Hand–Over Queue, SQ: Server Queue
[c] A) no waiting receiver, B) receiver readied, C) no pending message, D) message available (cf. page 46)
[d] will/must not occur

Table 7.3: Execution times of MEDF directives for a AOCS Server Model implementation

EDF event create and EDF event delete

Regarding both operations, only execution times of EDF event create are varying. As can be seen in Fig. 5.10 (a), processing times c_{cr} only depend on $N_{E,max}$. On Galileo, c_{cr} exceeds the initial additional processing time $c_{cr,1}$ for $N_{E,max} \geq 6$ and constantly increases further on. For $N_{E,max} = 10$, this results in an upper bound $c_{cr,10} = 5.2\ \mu$s.

In contrast to the create operation, EDF event delete run–times c_{dl} are independent of list search operations and thus remain constant for all $N_{E,max}$ (cf. Fig. 5.10 (b)). One only has to be aware of the initial timer adjustment that dominates the worst case run–times of c_{dl}. With this the upper bound derives to $c_{dl} = 4.3\ \mu$s.

EDF event send and EDF event receive

Processing times of EDF event send and EDF event receive operations depend not only on the list parameters as evaluated above, but also depend on state and on type of the queue involved, respectively on the situation the task under consideration currently faces. This makes a distinction of cases (A and B for send directives, C and D for receive operations) necessary. Worst case run–times then result from respective maxima (columns 4 and 8 in Tab. 7.3).

*Q–B (IQ–B, HOQ–B, SQ–B) Send with receiver readied depends on search walks on both lists, ready list and EDF thread list, processing time increases linearly with both

parameters. As can be seen in Fig. 5.13, an upper bound for EDF event send with $N_{T_{Ready},max} = N_{T_{D\neq\infty},max} = 9$ derives to $c_{snd,B} = 24.0\ \mu$s.

IQ–A Send into an input queue with no receiver waiting means inserting a message into an empty queue for this application example, with $N_{E_i,max} = 1$ for all event types. Thereof it follows, $c_{snd,IQ-A} = 7.7\ \mu$s (cf. Fig. 5.11 (a)).

HOQ–A Must not appear.

SQ–A Send into a server queue with DIP enabled consists of both, adding a message to a queue (similar to case IQ–A) and adapting the deadline of the task currently active. Maximum number of messages N_k already waiting in a server queue is at most the number of different event types sent to this queue. For example, $N_k = 3$ for the IRES server queue. But deadline inheritance and thus the worst case regarding the execution time will only occur, if the currently arriving signal transports the shortest deadline, i.e. place of insertion is first place in message queue ($c' = c_{snd,IQ-A} = 7.7\ \mu$s). Coding of DIP, the second part, is analogous to case "Send with receiver readied" (SQ–B). Thus, an overall upper bound calculates to $c_{snd,SQ-A} = 31.7\ \mu$s.

*Q–C (IQ–C, HOQ–C, SQ–C) Receive and caller blocks in a Server Model is built of appending to an empty thread queue. With Fig. 5.12, the processing time derives to $c_{rcv,C} = 34.1\ \mu$s for all queue types.

IQ–D
SQ–D Receive and message available initiates as well a search walk on ready and EDF thread list. An upper bound for the processing time for $N_{T_{Ready},max} = N_{T_{D\neq\infty},max} = 9$ can be found at $c_{snd,D} = 10.35\ \mu$s (Fig. 5.11 (b), $c_{snd,D}$ exceeds threshold at $N > 8$).

HOQ–D Must not occur.

7.1.4 System Environment and Timer Task

Tasks that result from code generation share the processor with Interrupt Service Routines (ISRs) and additional functionality needed to ensure full SDL language support. In case of the AOCS system the later mainly means system time, which is provided by a cyclic Clock ISR and Timer Service, realized by the Timer Task. This pre–load $F(I)$ has to be taken into account in real–time analysis. Basis for the following considerations are either the above estimations or run–times measurements as given by the timing test suite on a M68020 [OAR96].

RTEMS Clock Tick ISR increments RTEMS internal time and is additionally responsible for MEDF deadline surveillance (cf. App. B.3). Since SDL Timer services rely directly on wake–up calls to the Timer Task within a RTEMS integration, Clock Tick resolution is

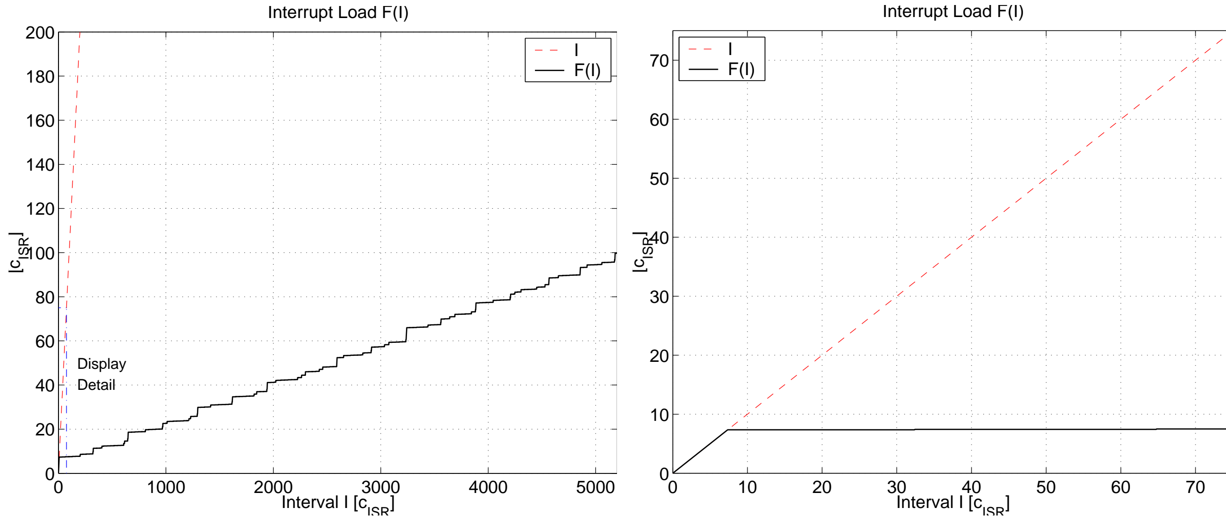

Figure 7.4: a) AOCS Interrupt Load and b) Busy Period at Interval 0

set to a cycle time $z_{Cl} = 10$ ms, leading to an event stream $ES_{Clock-ISR} = \left\{\binom{10}{0}\right\}$. Execution time for `Clock_isr` according to [OAR96] is $c_{Clock-ISR} = 20\ \mu$s.

Context switch times are included in EDF event send and EDF event receive measurements. RTEMS interrupt entry and exit overhead on M68020 is specified with $9\ \mu\text{s} + 8\ \mu\text{s} = 17\ \mu$s. Incoming Gyro Data, Telecommand functions, and Telemetry status each initiate their own ISR. In the former case, an ISR consists of an EDF event create and a subsequent EDF event send to an input queue. Both operations together with ISR overhead consume processing time of $c_{ISR} = 309\ \mu$s. Telemetry ISR only triggers soft real–time tasks. Thus, no deadline management and with this no event creation is needed. ISR processing time calculates to $c_{ISR,Tm} = 257\ \mu$s.

All of Control Law, DSS respectively Calibrate Gyro, and IDP activity threads are released with the receival of a timeout signal. Execution times of *set(Timer)* operations are included in state transition times (cf. Fig. 4.4). In contrast to this, Timer Task processing on a timeout additionally has to be taken into account as extra time to $F(I)$. On a timeout the Timer Task initiates an EDF event send before it blocks on the attempt to receive a new request on its empty message queue. Therefore its processing time for this case is assumed to be $c_{TT} = 460\ \mu$s.

With the event streams as specified in Table 7.1 the Interrupt Load $F(I)$ can be derived. The result is shown in Fig. 7.4 (a), intervals I as well as execution times are normalized with c_{ISR}. Since the overall load $F(I)$ is very small compared to the available computation time per interval (function $f(I) = I$), y–axis scale is heavily enlarged.

A detailed view of interval $I = [0, 80]$ can be seen in Fig. 7.4 (b). It displays the busy period that arises from a critical instant including all event types and the clock tick ISR. Busy period length calculates to $c_{busy-period} = 2275\ \mu\text{s} = 7.4 \cdot c_{ISR}$.

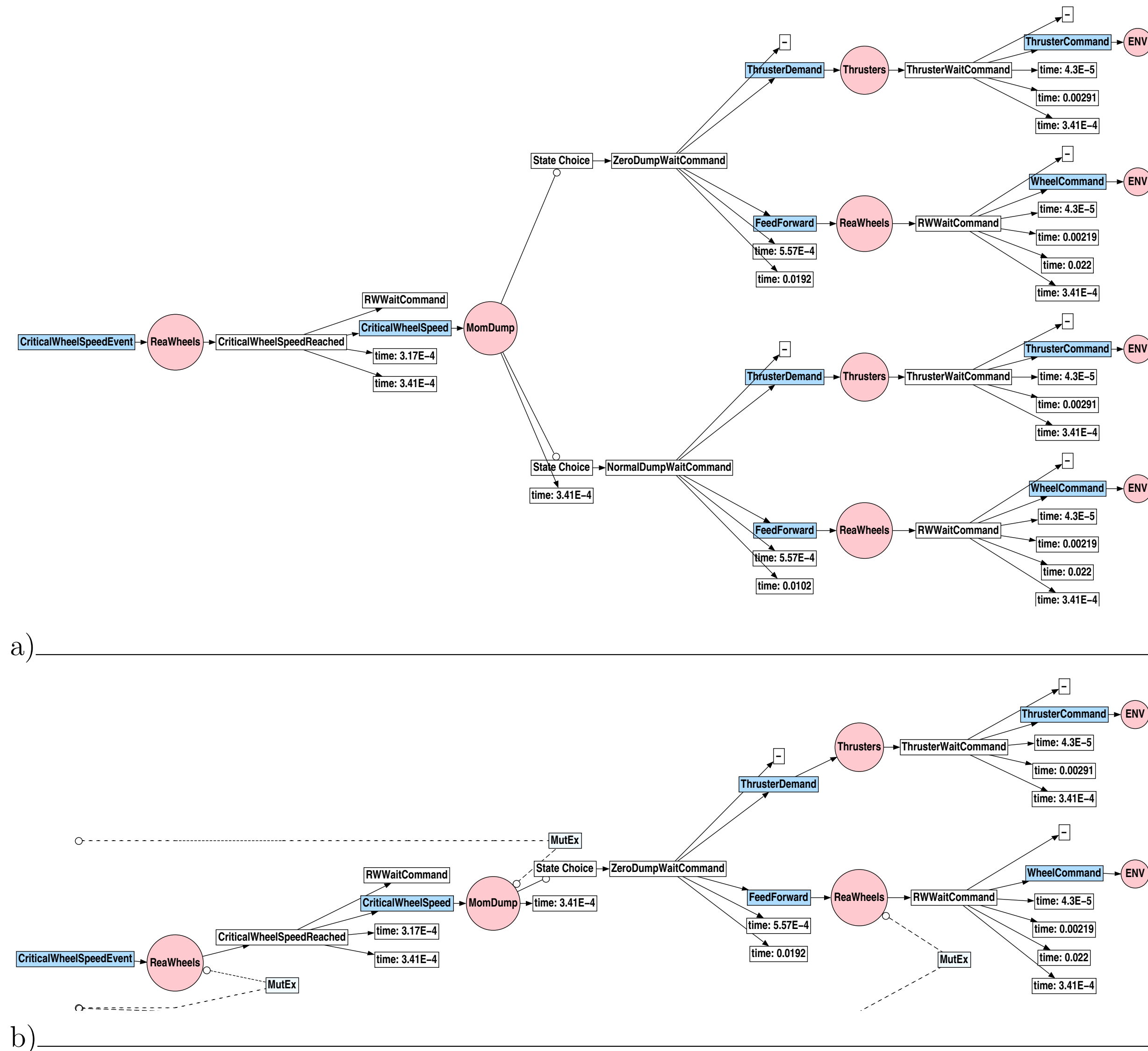

Figure 7.5: a) CWS Task Precedence System and b) Real–Time Analysis Model

7.1.5 Schedulability Analysis

First step for the final real–time analysis is the generation of the Task Precedence Graph (cf. Sec. 6.2.1). A full view of the AOCS TPG with all triggering events included can be seen in Fig. A.16. For clarity reasons, only the Task Precedence System of the "Critical Wheel Speed Event" is shown here (Fig. 7.5 (a)).

The structure of this TPS is dominated by the state choice in process "MomDump" that will be resolved when evaluating the worst case execution times of transitions and send/receive operations as annotated[7] to each node (cf. Sec. 6.2.2). The resulting worst

[7]Multiple annotations per node consist of EDF event receive WCETs (c_{rcv} = 34.1 μs, cf. Tab. 7.3),

TPS		**Server Processes in TPS**						d_{TPS} [ms]	c_{TPS} [ms]
	Abbr.	Tc	Gy	CWS	CL	IDP	CG		
	d_{DIP}	190	100	100	200	100	1000		
Telecommands	Tc	X	Mom. Dump					190	0.34
Gyro Raw Data	Gy		X			(IRES)		100	5.73
Crit. Wheel Speed	CWS			X				100	48.62
Control Law	CL		IRES, Gyro	Rea. Wheel	X	IRES		200	105.19
IRES Data Proc.	IDP		(IRES)			X		100	8.55
Calibrate Gyro	CG						X	1000	13.57

Server Process	TPS	d_{TPS} [ms]	d_{DIP} [ms]	c_{Server} [ms]	s_{Server} [ms]
Mom.Dump	Tc	190	100	0.34	0.00
IRES	CL	200	100	0.58	1.56
Gyro	CL	200	100	1.96	1.56
Rea.Wheels	CL	200	100	46.67	58.51

Table 7.4: Processing Times of AOCS Task Precedence Systems and Server Processes

case graph is shown Fig. 7.5 (b). Processes “MomDump” and “ReaWheels” are servers and therefore additionally connected to MutEx nodes that are shared with server processes in concurrent Task Precedence Systems (clipped and not shown here). The complete Real–Time Analysis Model of the AOCS can be seen in App. A.1, Fig. A.17. The TPS for Digital Sun Sensor data processing is suppressed in the overall RTAM, since computation of the mutual mode “Calibrate Gyro” poses the more stringent computation requests to the worst case.

Adding up all timing annotations in each TPS, the total worst case processing time c_{TPS} for each event can be derived (rightmost column in Tab. 7.4(a)). Together with timing constraints as summarized in Tab. 7.1 and the assumption all processes to run independently (no blocking on servers), the overall computation function $C(I)$ can be calculated.

As can be seen in Fig. 7.6, at interval $I = 647 \cdot c_{ISR} = 200\,ms$, laxity $L(I) = I - F(I) - C(I) = 40.3 \cdot c_{ISR} = 12.5\,ms$ reaches its global minimum.[8]

and EDF event send or EDF event delete WCETs (e.g. $c_{dl} = 43\ \mu s$), and one or more (for compound transitions) additional state transition times.

[8]This is true for all intervals $I \in \mathbf{R}^+ : C(I) > 0$. For intervals within busy period, laxity is zero per

Next step during schedulability analysis is to take into account possible mutual blocking of state transitions within server processes by reducing the deadline of the blocking part of the server process. The matrix in Table 7.4(a) gives a survey of servers shared by different TPS (cf. Fig. A.17), whereas table entries occur only, when a server process may cause a blocking due to its longer deadline (E.g. process “MomDump” in TPS “Telecommand” with deadline $d_{Tc} = 200\,ms$ may block state transitions released by events of type “Gyro Raw Data” with deadline $d_{Gy} = 100\,ms$. Process “IRES”may cause blocking in the “CL” TPS, but is scheduled with unshortened deadlines for “IDP” and “Gy”. Due to equal deadlines $d_{Gy} = d_{IDP}$ no blocking is possible.). Server process parameters are summarized in Tab 7.4(b), c_{Server} identifies the amount of computation to be scheduled with shortened deadline, i.e. the blocking part of the server process.

With deadline reduction for servers as deduced above, the overall computation function $C(I)_{DIP}$ can be derived. This time Fig. 7.6 shows a possible deadline violation for $C(I)_{DIP}$ at interval $I = 323 \cdot c_{ISR} = 100\,ms$. This results in a negative laxity $L_{DIP}(I) = -50.1 \cdot c_{ISR} = -15.5\,ms$.

For simplicity, no distinction between worst case and best case execution times has been made in annotations to the AOCS SDL system model. To demonstrate the influence of start times to decrease the necessary deadline reduction for servers in real–time analysis (cf. Eq. 5.1), timing as given in the AOCS Task Precedence Graph (Fig. A.16) is used. In applying the algorithm as described in Sec.6.2.3, earliest possible start times s_{Server} of AOCS servers (see Tab 7.4(b)) can be derived.

As can be seen in Fig. 7.6(b), computation requests $C(I)_{DIP,s}$ are relaxed in this way that minimum laxity will occur again at interval $I = 647 \cdot c_{ISR} = 200\,ms$, i.e. $L(I)_{DIP,s} = L(I)$. Particularly, processing load caused by the “ReaWheels” server is shifted to interval $I = d_{Gy} + c_{RW} = 513 \cdot c_{isr} = 158\,ms$, thus all deadlines can be guaranteed.

Conclusion

This case study demonstrates the influence of message blocking at SDL server processes on processing load and shows how, by considering server process start times, the processor demand can be re–relaxed. On the other hand, deadline reduction caused by priority inversion avoidance need *not*–necessarily lead to a tightening of processing requests. This depends on the application, respectively the timing constraints involved. Considering an AOCS with a Critical Wheel Speed Event deadline of e.g. $d_{CWS} = 200\,ms$, minimum laxity $L(I)$ will not change (see Fig. 7.7 (b)) when taking into account deadline inheritance. Worst case occurs, when signals with deadlines of different magnitudes of order are synchronized at a server that consists of “long” state transitions in the blocking part.

In this example, communication structure, transition times of the application itself, and their deadlines have been chosen to force a deadline violation. Relying on start times as single means to ensure that all deadlines will hold would certainly be questionable in a

definition; $L(I) = 0, \forall I \in [0, I_{BP}]$.

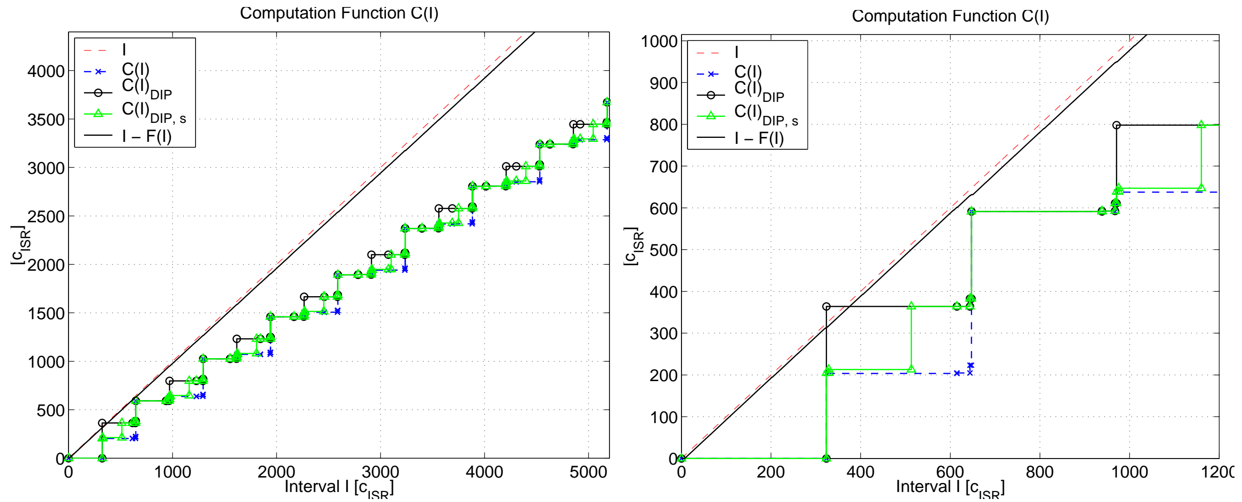

Figure 7.6: a) AOCS Overall Computation Function, b) Detailed View in Interval 0

real–world application, but could be used as a means to increase a requested safety margin.

With this AOCS SDL system model, control part compared to application part is almost negligible. Transition times compared to run–time system execution times are in magnitude of 1000, overall execution time spent in transitions is $c_{App} = 171.2\,ms$, whereas overall execution time spent in run–time system directives is $c_{RTS} = 10.9\,ms$. For this reason, this case study analyzed only a pure server model implementation. Regarding an Activity Thread Model implementation, the number of tasks implementing an activity thread would be nearly equal to the number of overall processes, respectively to the number of SDL servers.[9]

Assuming nearly equal execution times for semaphore obtain and release operations (compared to EDF event send and EDF event receive directives) as needed in an ATM implementation for SDL servers and substituting receives and sends from/to Hand–Over Queues with function calls, the overall execution time spent in run–time system directives would reduce at best to $c_{RTS} = 6.8\,ms$. Naturally, this improvement depends on application type and modeling style. Influence of ISRs and supplementary functions is equal for SM and ATM realizations.

More positive effect on schedulability analysis than an optimized implementation promises the consideration of operation modes as already shown with “DSS” vs. “Calibrate–Gyro”. In the example above, the crucial blocking in process “ReaWheels” can only occur, when it must be assumed that “Critical Wheels Speed Events” have to wait due to processing of “Control Law” events. This means, momentum dumping is allowed to be performed concurrently with normal mode control.

To take into account these mutual exclusive operation modes, one would have to add additional control structure to the AOCS system model, since this functionality is not

[9]There can be only one real activity thread identified in the AOCS, the “Control Law” loop.

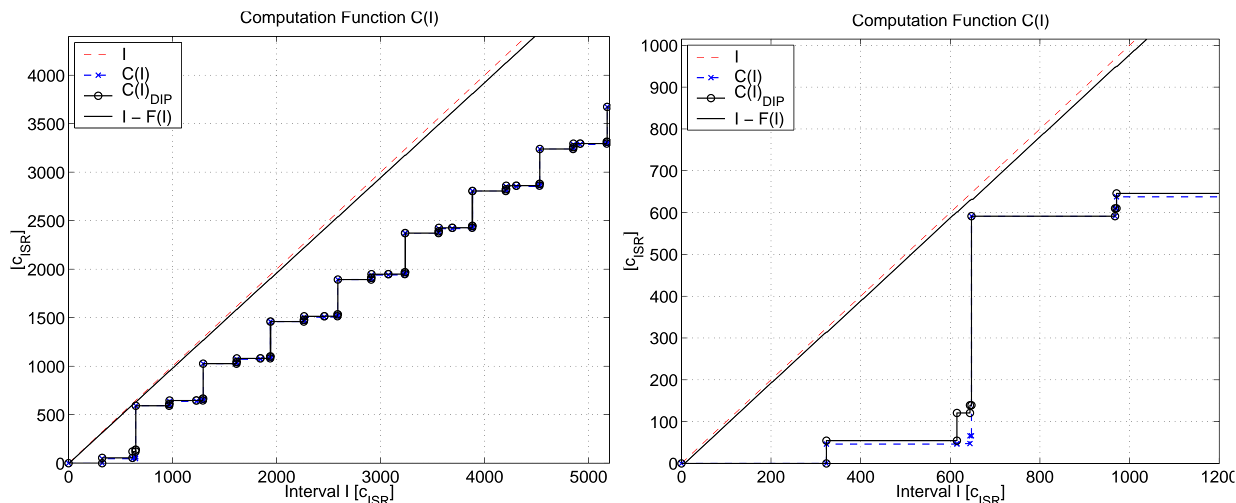

Figure 7.7: a) AOCS $C(I)$ with $d_{CWS} = 200\,ms$, b) Detailed View in Interval 0

covered in the current SDL system. The achievable gain would be twofold. On the one hand, processing times of only either CL–TPS or CWS–TPS would add to overall $C(I)$ calculations: On the other hand, the necessary deadline reduction for analysis of process "ReaWheels" would cease to exist.

7.2 SDL Style Guidelines for Real–Time Systems

Regarding the experiences gained in the AOCS case study, the following design style guidelines for real–time systems based on SDL and MEDF scheduling can be derived.

7.2.1 General Rules

The identification of mutual exclusive operation modes seems to be of utmost importance to confine the pessimism in schedulability analysis. Process actions belonging to different modes should be placed in different state transitions (triggered by different signals or originating in different states) or even placed in different SDL *services* (SDL-92). The avoidance of state or modes variables is strictly recommended.

Signals in SDL should only be used to model the control structure of a system to restrict the overhead imposed by MEDF scheduling. This means, SDL process granularity should follow the "natural" concurrency that characterizes a system. In any event, SDL process syntax is not adequate for modeling of algorithms.

Of minor importance is the avoidance of "dead code" in SDL transitions. Transitions that are never executed can occur through the discrepancy between SDL semantics and MEDF run–time system implementation (see Chap. 3). Since processing order of multiple consecutive sends in a state transition is a–priori known with MEDF, and thus scheduling order on receiver side is known too, order of receives in a process synchronizing several in-

coming signals must cover only the original sender's order (cp. process "Attitude", Fig. A.7 in contrast to process "Pr_D", Fig. 4.3). With this, resulting superfluous forks in TPG and RTAM are avoided.

7.2.2 Minimizing Number of SDL Servers

Beneath system modes and MEDF scheduling, possible blocking caused by SDL servers may lead to an undesirable intensification of processor demands to be planned. With this, a substitution of SDL message exchange through *reveal/view* may be preferable, if data consistency at time of variable readings can be assured (e.g. collecting of measurement data in "Gyro" or "IRES" processes, Figures A.5 and A.6). An alternative may be to combine server processes with equal timing constraints and merge them into one single SDL process, thus giving up modularity for the benefit of efficiency.

7.2.3 Minimizing Length of Blocking Transitions

Comparable to the negative effect of uninterruptable regions to scheduling latency or the general real–time design rule to minimize amount of computation within critical regions, state transitions in server processes should be as "short" as possible, at least in the blocking part, i.e. the transition processed with a longer deadline. If possible, complex state transitions should be subdivided. This can be carried to extremes by out–sourcing of computation intensive state transition parts to consecutive SDL processes. The later is only possible if transitions do not share common variables or access a common resource.

Chapter 8

Conclusion and Future Work

This thesis introduced message based earliest deadline first scheduling (MEDF) as a model of computation for hard real–time SDL systems. With MEDF, each task is assumed to have its own message queue. A task inherits the deadline transported by an incoming message and bequeathes this deadline to its outgoing messages. To assure predictability when dispatching tasks according to message deadlines, incoming messages carrying a new deadline have to be inserted into this queue in a deadline sorted order. Priority inversion avoidance mechanisms must be integrated into send and receive directives to avoid unbounded blocking times for high priority messages caused by server tasks responding to multiple requests with unequal urgencies. For this, deadline inheritance (DIP) and deadline ceiling (DCP) at message queues were proposed. These protocols raise the dynamic priority of the receiving task according to the deadline of the succeeding message that is stored in the incoming queue (DIP), respectively is expected to arrive (DCP).

Two implementation alternatives to realize the run–time system for MEDF scheduling were suggested. The first strategy allows tasks to be blocked either by receive–operations on empty queues or by P–operations on seized semaphores, i.e. permits the use of additional semaphores beneath asynchronous message exchange to synchronize concurrent tasks. This fact makes a differentiation of ready and deadline surveillance list (EDF list) within the MEDF scheduler necessary and leads to concatenated list search walks to derive correct processing order. The second alternative prohibits the usage of semaphores, thus relying on server tasks only to monitor access to shared resources. This allows to unify ready and EDF list, thus leading to simpler list operations (mostly list appends) and a low overhead implementation of the MEDF scheduler. Common to both run–time system layouts is the provision of EDF event objects. They represent an end–to–end deadline interval that is transported by messages and has to be inherited by the receiving task. It could be shown that MEDF scheduling effort equals EDF scheduling. In effect, MEDF can be seen as an event scheduling (in contrast to EDF task scheduling), i.e. processing order of tasks is predetermined on the occurrence of a stimulating event.[1]

Since it could be demonstrated that MEDF scheduling implicates an earliest deadline

[1] In case of burst events, a re–scheduling of tasks will be initiated when the succeeding event will be processed.

first processing sequence for all tasks, Gresser's schedulability analysis methodology for event–driven real–time systems [Gre93a] can be applied to prove the timeliness of a MEDF system. Improvements to his verification algorithms were made in dealing with resource control tasks (servers). The necessary deadline reduction to schedule server tasks which results in a more straining processor demand could be confined. This means, necessary over–dimension of processing power to cope with load peaks in worst case situations could be reduced.

Construction of a software architecture that does not hide any pitfalls (e.g. synchronization deadlocks or communication bottlenecks) and is efficient at the same time is not an easy task. The idea of this thesis is to automatically derive a real–time system's implementation and its timing analysis model on the basis of its SDL system specification by one strike. For that purpose, SDL's semantics had to be pinpointed to the predictable MEDF model of computation, restricting the usage of few language features (e.g. *signal save* is prohibited), but enhancing the applicability of its timer concept.[2]

Two code generation strategies were evaluated to translate a SDL model to a final task system. The so called "Server Model" preserves MEDF by a one–to–one mapping of SDL processes to MEDF tasks, relying on signal exchange as a single means for inter–task synchronization. To minimize communication overhead, the "Activity Thread Model" replaces inter–process messages by procedure calls. Thus, processing sequence of consecutive state transitions is scheduled at compile time. Monitoring of shared resources, achieved by a single (server) task with the Server Model, now has to be fulfilled in a classical way by semaphores. In the Activity Thread Model, MEDF queues are only needed to cope with bursts of triggering events. The later strategy is superior to the Server Model especially in implementations of SDL systems that incorporate only few server processes and in system models with an application part rather small compared to its control part ("short state transitions").

On the other hand, the system specification extended with timing characteristics of embedding as well as embedded system can now be mapped to a task precedence graph (TPG) that mirrors the structural inter–relationships between all components of the final implementation. State transition processing times as well as worst case execution times of MEDF directives (annotated to the TPG nodes) can be evaluated to derive a worst case task graph, the so called real–time analysis model (RTAM). It incorporates all necessary timing properties to perform the final schedulability analysis step, and serves as information source to reveal design flaws and optimization potential. On the basis of the RTAM, earliest possible start times of server tasks can be calculated and included to confine the necessary deadline reduction when proving the schedulability of the whole task system.

Upper bound estimation for execution times of MEDF directives has to be performed manually in the moment. Their dimension depend on the chosen code generation strategy (number of resulting tasks, realization of MEDF scheduler) and the timing characteristics of the stimulating events. Although this step could be automated as well,

[2] Timer signals transport as well a signal deadline. Hence, timer signal consumption is subject to deadline surveillance and is performed according to the timer's urgency.

this modus operandi seems to be justifiable, since the interface to environment changes more seldom than the SDL model itself and timing requirements (events and deadlines) are predetermined by the application to be controlled and thus will not change during the development. The same information as used to calculate these execution time estimates is needed to configure the real–time operation system. This means, the specification of objects (tasks, queues, semaphores, EDF events) involved could be automatically derived from the system specification. Furthermore, the interface to the environment consisting of interrupt service routines and external tasks could be generated without additional effort.

Scheduling with message deadlines was introduced in this thesis as a platform for development of hard real–time SDL systems targeting single processor hardware. Future work emerges when restrictions to this type of application are abolished:

- *Mixed soft and hard real–time applications.* Application parts with soft real–time requirements are assumed to run with fixed priorities on levels below the MEDF priority level. Although interference of the soft real–time part is included in the analysis algorithm for the hard real–time part, a response time estimation for the soft real-time tasks has to be developed.

  In the moment, the boundary between soft and hard real–time has to be drawn on process level, since priorities can only be assigned to whole SDL processes. The possibility to mix hard and soft real–time requirements within a single process, i.e. a boundary on state transition level (transition priorities), would be desirable.

- *Tightly coupled multi–processor based target systems.* At first sight, MEDF scheduling with message deadlines spanning processing unit boundaries seems to be an intuitive way to synchronize multi–processor systems. Taking a closer look, the following problems arise:

  – How can resulting event streams at processing unit boundaries be derived in spite of jitter due to dynamic processing order on the predecessor processing unit and varying execution times of tasks?
  – How much laxity has been used up until the message reaches its new processing unit, i.e. how long is the remaining deadline? Or alternatively, what is an optimal algorithm for pre–splitting of deadlines and their assignment to the single processing units?[3]
  – Without pre–splitting of deadlines, the EDF analysis methodology as introduced in this work has to be enhanced for multi–processor systems.

  Even if deadlines spanning processing unit boundaries are avoided, predictable means to synchronize between application parts on different processors have to be developed (e.g. global semaphores or spin locks on shared memory to divide server processes in ATM realizations).

[3] Deadline splitting would artificially increase the urgency of the message.

- *Distributed systems.* The former facts are true for tightly coupled multi–processor systems but apply all the more for distributed systems. Furthermore, a bus arbitration protocol for communication between processing units has to be found that suits MEDF scheduling. Time division media access (TDMA) strategies (e.g. Time Triggered Protocol) or bus arbitration according to fixed priorities with non–preemptive messages (e.g. CAN bus like arbitration) would disrupt the MEDF philosophy.

At the latest since development complexity of embedded systems can not be handled anymore by one single programmer, control software development has been recognized as an engineering discipline. Therefore, software engineering methodologies based on standardized specification languages and supported by CASE tools have been developed. Astonishingly, ad–hoc attempts in real–time system design still prevail in industry. This is due to the fact that capturing non–functional requirements in a system specification is poorly supported until now and analysis methodologies to verify timing properties are not integrated into the design process for real–time applications. The methodology framework as suggested in this thesis shows how a SDL based design methodology can be enhanced by incorporating a timeliness proof as an automated step into the design process for hard real–time systems, allowing to generate a quality product that will meet not only all functional but as well all timing requirements.

Bibliography

[ADL+99] J.M. Alvarez, M. Díaz, L.M. Llopis, E. Pimentel, and J.M. Troya. Integrating schedulability analysis and SDL in an object-oriented methodology for embedded real–time systems. In *Proc. of the 9th SDL Forum*, Montreal, June 1999. 15

[ADL+00] J.M. Alvarez, M. Díaz, L.M. Llopis, E. Pimentel, and J.M. Troya. SDL and hard real–time systems: new design and analysis techniques. In *Proceedings of Workshop on SDL and MSC (SAM2000)*, Grenoble, Col de Porte, France, June 2000. 15

[BAL97] Hanêne Ben-Abdallah and Stefan Leue. Timing constraints in message sequence chart specifications. In *Tenth International Conference on Formal Description Techniques FORTE/PSTV'97*, Osaka, Japan, 1997. 10

[BGK97] M. Broy, R. Grosu, and C. Klein. Reconciling real–time with asynchronous message passing. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *FME'97, 4th International Symposium of Formal Methods Europe, Graz, Austria, Lecture Notes in Computer Science 1313*. Springer, 1997. 17, 26

[BGK+00] M. Bozga, S. Graf, A. Kerbrat, L. Mounier, I. Ober, and D. Vincent. SDL for real–time: What is missing? In *Proceedings of Workshop on SDL and MSC (SAM2000)*, Grenoble, Col de Porte, France, June 2000. 22

[BH93] Rolv Bræk and Øystein Haugen. *Engineering Real Time Systems*. Prentice Hall, 1993. 19, 75

[BHS91] Ferenc Belina, Dieter Hogrefe, and Amardeo Sarma. *SDL with Applications from Protocol Specification*. Prentice Hall, 1991. 19

[Bla76] J. Blazewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In E. Gelenbe and H. Bellner, editors, *Modeling and Performance Evaluation of Computer Systems*, Amsterdam, 1976. North–Holland. 47

[Boy96] Robert Boys. *comp.sys.m68k Frequently Asked Questions (FAQ)*, January 1996. Rev. 22. 94

[BW95a] Alan Burns and A. J. Wellings. Engineering a hard real–time system: From theory to practice. *Software — Practice and Experience*, 25(7):705–726, July 1995. 14

[BW95b] Alan Burns and Andy Wellings. *HRT–HOOD: A Structured Design Method for Hard Real–Time Ada Systems*. Elsevier Science B. V., Amsterdam, The Netherlands, 1995. 14, 89

[BWBF93] A. Burns, A. Wellings, C. Bailey, and E. Fyfe. The Olympus attitude and orbital control system: A case study in hard real–time system design and implementation. In *Ada sans frontiers, Proc. of the 12th Ada–Europe Conf.*, Lecture Notes in Computer Science, pages 19–35. Springer–Verlag, 1993. 89

[BWBF94] A. Burns, A. Wellings, C. Bailey, and C. H. Forsyth. A performance analysis of a hard real-time system. Technical report YCS 224, University of York, Department of Computer Science, 1994. 89

[CLB99] Marco Caccamo, Giuseppe Lipari, and Giorgio Buttazzo. Sharing resources among periodic and aperiodic tasks with dynymic deadlines. In *Proc. of the IEEE Real–Time Systems Symposium*, Phoenix, AZ, December 1999. 13

[CW95] Pete Cornwell and Andy J. Wellings. Transaction specification for object-oriented real-time systems in HRT-HOOD. In *Ada-Europe*, pages 365–378, 1995. 14

[DHHM95] M. Diefenbruch, E. Heck, J. Hintelmann, and B. Müller–Clostermann. Performance evaluation of SDL systems adjunct by queueing models. In *SDL'95 With MSC in CASE, Proc. of the 7th SDL Forum*, pages 231–242, Oslo, Norway, September 1995. 15

[DHM96] M. Diefenbruch, J. Hintelmann, and B. Müller–Clostermann. The QUEST–approach for the performance evaluation of SDL–systems. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX, Theory, application and tools, Kaiserslautern, Germany*, pages 229–244, London, UK, 1996. Chapman & Hall. 15

[Dou98] Bruce Powel Douglass. *Real–Time UML — Developing Efficient Objects for Embedded Systems*. Addison–Wesley, 1998. 17

[DRSK89] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz. The real-time operating system of MARS. *ACM Operating Systems Review, SIGOPS*, 23(3):141–157, 1989. 12

[ELLSV97] Stephen Edwards, Luciano Lavagno, Edward Lee, and Alberto Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, mar 1997. Special Issue on Hardware/Software Co–Design. 9

[FFKM97] Georg Färber, Franz Fischer, Thomas Kolloch, and Annette Muth. Improving processor utilization with a task classification model based application specific hard real–time architecture. In *Proceedings of the 1997 International Workshop on Real–Time Computing Systems and Applications (RTCSA'97)*, Academia Sinica, Taipei, Taiwan, ROC, October 27–29 1997. 5

[FKMF97] Franz Fischer, Thomas Kolloch, Annette Muth, and Georg Färber. A configurable target architecture for rapid prototyping high performance control systems. In Hamid R. Arabnia et al., editors, *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, volume 3, pages 1382–1390, Las Vegas, Nevada, USA, June 30 – July 3 1997. 5

[FLMTS97] Nils Faltin, Lennard Lambert, Andreas Mitschele-Thiel, and Frank Slomka. An annotational extension of message sequence charts to support performance engineering. In *SDL'97 Time for Testing – SDL, MSC and Trends, Proceedings of the Eighth SDL Forum*, Paris, France, September 1997. 10

[Fär00] Georg Färber. Script Real–Time Systems, 2000. Handout Lehrstuhl für Realzeit–Computersysteme. 81

[Für01] Armin Fürst. RTEMS als Simulatorkernel für das SDL Design Tool SDT, 2001. Interdisziplinäres Projekt am Lehrstuhl für Realzeit–Computersysteme, Technische Universität München. 18, 150

[FW94] Michael Fröhlich and Mattias Werner. Demonstration of the interactive graph visualization system *daVinci*. In R. Tamassia and I. Tollis, editors, *Proceedings of DIMACS Workshop on Graph Drawing, Princeton (USA)*, volume 894 of *Lecture Notes in Computer Science*. Springer–Verlag, 1994. 85

[Gal95] Galileo Technology, San Jose, CA USA. *Galileo–4: 64–bit R4XXX Evaluation and Development System*, November 1995. Rev. 1.0. 65, 148

[Gal96] Galileo Technology, San Jose, CA USA. *GT–64010A: System Controller with PCI Interface for R4XXX/R5000 Family CPUs*, December 1996. Rev. 1.1. 65, 148

[Gre93a] Klaus Gresser. *Echtzeitnachweis ereignisgesteuerter Realzeitsysteme*. Number 268 in Fortschrittsberichte VDI, Reihe 10. VDI–Verlag, Düsseldorf, 1993. Dissertation am Lehrstuhl für Prozessrechner, Technische Universität München. iii, xi, 5, 13, 45, 50, 53, 54, 55, 76, 77, 80, 106

[Gre93b] Klaus Gresser. An event model for deadline verification of hard real–time systems. In *Proc. Fifth Euromicro Workshop on Real Time Systems*, pages 118–123, Finland, June 1993. IEEE. 5, 13, 28, 76, 77

[GVNG94] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. *Specification and Design of Embedded Systems*. Prentice Hall, Englewood Cliffs, NJ, 1994. 9

[Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Dordrecht, Netherlands, 1993. 9

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. 10

[Hin98a] Ursula Hinkel. *Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL*. PhD thesis, Dissertation am Institut der Informatik IV, Technische Universität München, 1998. 17, 22

[Hin98b] Ursula Hinkel. Verification of SDL specifications on base of a stream semantics. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC (SAM'98)*, Berlin, Germany, June 1998. Humboldt-Universität zu Berlin. 17, 22

[HK95] T. Held and H. König. Increasing the efficiency of computer–aided protocol implementations. In Son T. Vuong and Samuel T. Chanson, editors, *Protocol Specification, Testing and Verification XIV, Vancouver, Canada*, pages 387–394, London, UK, 1995. Chapman & Hall. 11

[HKMT97] R. Henke, H. König, and A. Mitschele-Thiel. Derivation of efficient implementations from SDL specifications employing data referencing, integrated packet framing and activity threads. In *Proc. of the 8th SDL Forum, SDL'97 Time for Testing SDL, MSC and Trends*, pages 397–414, Evry, France, September 1997. Elsevier Science Publishers B.V. 11, 40

[HL97] Chih-Wen Hsueh and Kwei-Jay Lin. Schedulability comparisons among periodic and distance–constrained real–time schedulers. In *Proceedings of the 1997 International Workshop on Real–Time Computing Systems and Applications (RTCSA'97)*, Academia Sinica, Taipei, Taiwan, ROC, October 27–29 1997. 13

[HLN+90] D. Harel, H. Lachover, A. Naamad, et al. Statemate. A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), April 1990. 10, 23

[Hoa74] C.A.R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974. 51

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978. 10

[IEE95] IEEE. *The Institute of Electrical and Electronics Engineers: "IEEE Standard Description Language Based on the Verilog(TM) Hardware Description Language" (IEEE-1364-1995*, 1995. 11

[IEE00] IEEE. *The Institute of Electrical and Electronics Engineers: "IEEE Standard VHDL Language Reference Manual" (IEEE-1076-2000*, 2000. 11

[ITU94a] ITU–T. *ITU–T Recommendation Z.100: CCITT Specification and Description Language (SDL)*, June 1994. 3, 10, 18, 21, 23, 25

[ITU94b] ITU–T. *ITU–T Recommendation Z.120: Message Sequence Chart (MSC)*, September 1994. 10

[ITU94c] ITU–T. *ITU–T Z.100 Appendices I and II: SDL Methodology Guidelines and SDL Bibliography*, June 1994. 19, 75

[Jef92] Kevin Jeffay. Scheduling sporadic tasks with shared resources in real–time systems. In *Proc. of the IEEE Real–Time Systems Symposium*, pages 89–99, Phoenix, AZ, December 1992. 5, 13, 45

[KF98] Thomas Kolloch and Georg Färber. Mapping an embedded hard real–time systems SDL specification to an analyzable task network — a case study. In F. Müller and A. Bestravros, editors, *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, Lecture Notes in Computer Science, pages 156–165, Canada, June 19–20 1998. Springer–Verlag. 83

[Kob99] Cris Kobryn. UML 2001: A standardization odyssey. *Communications of the ACM*, 42(10):29–37, October 1999. 10

[Kol98] Thomas Kolloch. SDL/MSC design methodology for hard real–time systems. Workshop on Performance and Time in SDL and MSC, Technical Report TR–SMD98, IMMD VII, University of Erlangen-Nürnberg, Erlangen, Germany, February 1998. 2

[KRP+93] Mark Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael González Harbour. *A Practitioner's Handbook for Real–Time Analysis: Guide to Rate Monotonic Analysis for Real–Time Systems.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993. 4, 13

[Lam97] Lennard Lambert. Bewertung von MSC–Spezifikationen mit Task–Graphen. In *Formale Beschreibungstechniken für verteilte Systeme, 7. GI/ITG–Fachgespräch*, Berlin, June 1997. 10

[Lar98] Gunnar Larisch. Integration des Spezifikationswerkzeugs SDT in den Entwicklungsprozeß der *REAR* Rapid Protoyping Umgebung, 1998. Diplomarbeit (masters thesis) am Lehrstuhl für Prozessrechner, Technische Universität München. 38, 83

[Leu95] Stefan Leue. Specifying real–time requirements for SDL specifications — a temporal logic–based approach. In *Protocol Specification, Testing, and Verification PSTV'95*. Chapman & Hall, 1995. 10, 17

[LK97] Peter Langendörfer and Hartmut König. Improving the efficiency of automatically generated code by using implementation-specific annotations. In *Proceedings of the Third International Workshop on High Performance Protocol Architectures (HIPPARCH '97)*, Uppsala, Sweden, June 12–13 1997. 11

[LK99] Peter Langendörfer and Hartmut König. Automated protocol implementations based on activity threads. In L. Palagi and B. Werner, editors, *Proceedings of the 7th International Conference on Network Protocols (ICNP'99)*, pages 3–10, Toronto, Canada, 1999. IEEE Computer Society. 11

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard–real–time environment. *Journal of the ACM*, 20(1):46–61, 1973. 4, 12

[MB00] Annette Muth and Gregor Burmberger. Tutorial Real–Time Systems, 2000. Handout Lehrstuhl für Realzeit–Computersysteme. 31

[MF00] Annette Muth and Georg Färber. SDL as a system level specification language for application-specific hardware in a rapid prototyping environment. In *Proceedings of the 13th International Symposium on System Synthesis (ISSS'2000)*, Madrid,Spain, September 20–22 2000. IEEE Computer Society Press. 6, 11

[MKMKF00] Annette Muth, Thomas Kolloch, Thomas Maier-Komor, and Georg Färber. An evaluation of code generation strategies targeting hardware for the rapid prototyping of SDL specifications. In *Proceedings of the 11th IEEE International Workshop on Rapid Systems Prototyping (RSP'2000)*, Paris,France, June 21–23 2000. IEEE Computer Society Press. 6, 11

[Mok83] Aloysius Ka-Lau Mok. Fundamental design problems of distributed systems for the hard–real–time environment. Technical Report MIT-LCS-TR-297, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Massachusetts, USA, May 1983. Ph.D. Thesis. 11, 32, 34

[MTMC99] Andreas Mitschele-Thiel and Bruno Müller-Clostermann. Performance engineering of SDL/MSC systems. *Journal on Computer Networks and ISDN Systems*, 31(17):1801–1815, 1999. 10

[Mut02] Annette Muth. *SDL–based Design of Application Specific Hardware for Hard Real–Time Systems*. 2002. Dissertation am Lehrstuhl für Realzeit–Computersysteme, Technische Universität München. 11

[OAR96] OAR On-Line Applications Research Corporation, Huntsville, Alabama, USA. *RTEMS — Real–Time Executive for Multiprocessor Systems*, 1996. Rev. 3.6.0. 56, 94, 96, 97

[Obj99a] Object Management Group. *UML Profile for Scheduling, Performance, and Time, Request for Proposal*, March 1999. OMG Document: ad/99-03-13. 11

[Obj99b] Object Management Group. *Unified Modeling Language Specification, v. 1.3*, June 1999. OMG Document: ad/99-06-08. 10

[Ols94] Anders Olsen. *Systems engineering using SDL–92*. Elsevier, Amsterdam, 1994. 19, 75

[PB98] D. Priddin and A. Burns. Integrating real–time structured design and formal techniques. In Anders P. Ravn and Hans Rischel, editors, *Formal Techniques in Real–Time and Fault–Tolerant Systems, Proceedings of the 5th International Symposium (FTRTFT'98)*, Lecture Notes in Computer Science, pages 92–102, Lyngy, Denmark, September 14–18 1998. Springer–Verlag. 10, 14

[Pet02] Stefan Petters. *Real–Time Systems Utilizing State of the Art Processors*. 2002. Dissertation am Lehrstuhl für Realzeit–Computersysteme, Technische Universität München. 5

[PF99] Stefan M. Petters and Georg Färber. Making worst case execution time analysis for hard real–time tasks on state of the art processors feasible. In *6th Int. Conf. on Real–Time Computing Systems and Applications (RTCSA'99)*, Hongkong, ROC, December 13–15 1999. IEEE, IEEE Computer Society Press. 5, 6

[PMK+99] Stefan Petters, Annette Muth, Thomas Kolloch, Thomas Hopfner, Franz Fischer, and Georg Färber. The REAR framework for emulation and analysis of embedded hard real–time systems. In *Proc. of the 10th IEEE Int. Workshop on Rapid Systems Prototyping (RSP'99)*, pages 100–107, Clearwater, Florida, June 16–18 1999. IEEE Computer Society Press. 5

[PMK+00] Stefan Petters, Annette Muth, Thomas Kolloch, Thomas Hopfner, Franz Fischer, and Georg Färber. The REAR framework for emulation and analysis of embedded hard real–time systems. *Design Automation for Embedded Systems*, 5(3):237–250, August 2000. 5, 11

[Raj91] Ragunathan Rajkumar. *Synchronization in Real–Time Systems. A Priority Inheritance Approach.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991. 13, 53, 63

[Ree98] Rick Reed. Timing restriction. SDL–News (mailing list), SDL Forum, December 29 1998. Author is ITU–T SG10 rapporteur for SDL (Q.6/10). 75

[SBMJ01] D. A. Stuart, M. Brockmeyer, A. K. Mok, and F. Jahanian. Simulation–verification: Biting at the state explosion problem. *IEEE Transactions on Software Engineering*, 27(7):599–617, July 2001. 30

[SFR97] M. Saksena, P. Freedman, and P. Rodziewicz. Guidelines for automated implementation of executable object oriented models for real-time embedded control systems. In *Proc. of the IEEE Real–Time Systems Symposium (RTSS'97)*, San Francisco, CA, December 1997. IEEE Computer Society Press. 14

[SGA93] Manas Saksena, Richard Gerber, and Ashok Agrawala. Scheduling with relative timing constraints. In *10th IEEE Workshop on Real–Time Operating Systems and Software*, pages 6 – 10. IEEE Computer Society Press, May 1993. 12

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real–Time Object–Oriented Modeling.* John Wiley & Sons, Inc., 605 Third Avenue, New York, 1994. 10, 14

[SK00] M. Saksena and P. Karvelas. Designing for schedulability: Integrating schedulability analysis with object-oriented design. In *Proc. of the Euromicro Conference on Real-Time Systems (ECRTS'2000)*, Stockholm, Sweden, June 2000. IEEE Computer Society Press. 14, 15

[SKW00] M. Saksena, P. Karvelas, and Y. Wang. Automatic synthesis of multi–tasking implementations from real–time object–oriented models. In *Proc. of the 3rd IEEE Int. Symposium on Object–Oriented Real–Time Distributed Computing (ISORC'2000)*, Newport Beach, CA, USA, March 2000. IEEE Computer Society Press. 15

[SRG89] W. Schwabl, J. Reisinger, and G. Grunsteidl. A survey of mars. Research report, Institut für Technische Informatik, Technische Universität Wien, October 1989. 12

[SSD97] Simone Spitz, Frank Slomka, and Matthias Dörfel. SDL* — an annotated specification language for engineering multimedia communication systems. In *Proceedings of the Sixth Open Workshop on High Speed Networks*, Stuttgart, Germany, October 1997. 10

[SSRB98] John A. Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C. Buttazzo. *Deadline Scheduling for Real–Time Systems — EDF and Related Algorithms*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998. 5, 13, 55, 56

[Sta97] Richard Stamvik. Integrating SDT generated code with target platforms. Technical report, Telelogic, Sweden, 1997. 137

[Svo89] Liba Svobodova. Implementing OSI systems. *IEEE Journal on Selected Areas in Communications*, 7(7):1115–1130, 1989. 11

[Syn99] Synopsys, Mountain View, CA, USA. *Design Compiler Reference Manual*, 1999. 11

[Tel01] Telelogic, Kungsgatan 6, S-20312 Malmö, Sweden. *SDT Reference Manual*, 2001. Release 4.1. 35, 41, 42

[TG00] François Terrier and Sébastien Gérard. Real time system modeling with UML: current status and some prospects. In *Proceedings of Workshop on SDL and MSC (SAM2000)*, Grenoble, Col de Porte, France, June 2000. 10

[Wro97] Heinz Wrobel. Implementation of deadline scheduling and deadline inheritance into RTEMS and port to the MC 68332 based NF 300 board, May 1997. Diplomarbeit (masters thesis) am Lehrstuhl für Prozessrechner, Technische Universität München. 57, 140

[WS99] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings of the 6th International Conference on Real–Time Computing Systems and Applications (RTCSA'99)*, Hongkong, ROC, December 13–15 1999. IEEE, IEEE Computer Society Press. 13, 14

Appendix A

Application Examples

A.1 *Olympus* Attitude and Orbital Control System

This appendix contains the complete AOCS SDL system model. Syntax of annotations needed to specify non–functional requirements within the SDL model is shown in Table A.1. All extensions follow either the keyword `#RZA` or keyword `#PRIO` and have to be placed within SDL comments.

Timing Constraints	
`#RZA ES`$\{ [m]\,(z\ \ a_0)\,[,\ \ [m]\,(z\ \ a_i)]* \}$	event stream ES
`#RZA ED` $k\ \ l\ \ ed$	event dependency ed_{kl}
`#RZA Deadline` d	deadline d
Extensions to the Functional Model	
`#RZA [Max]CalculationTime` c	calculation time c_{max}
`#RZA MinCalculationTime` c	calculation time c_{min}
`#RZA [Max]LoopBound` l	loop limit l_{max}
`#RZA MinLoopBound` l	loop limit l_{min}
`#PRIO` p	process priority p

Table A.1: Annotations to the SDL Model

Fig. A.16 and Fig. A.17 present the AOCS task precedence graph and its appropriate analysis model.

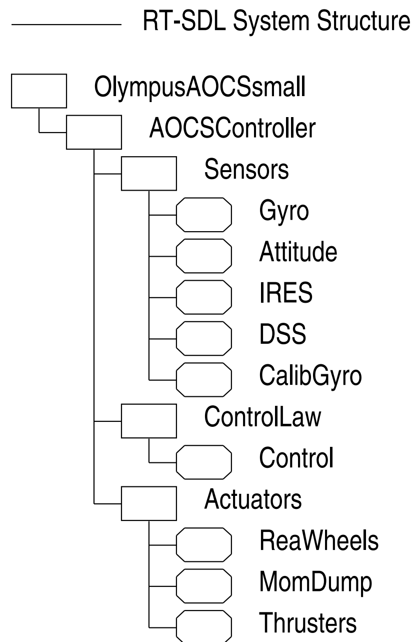

Figure A.1: Olympus AOCS System Structure

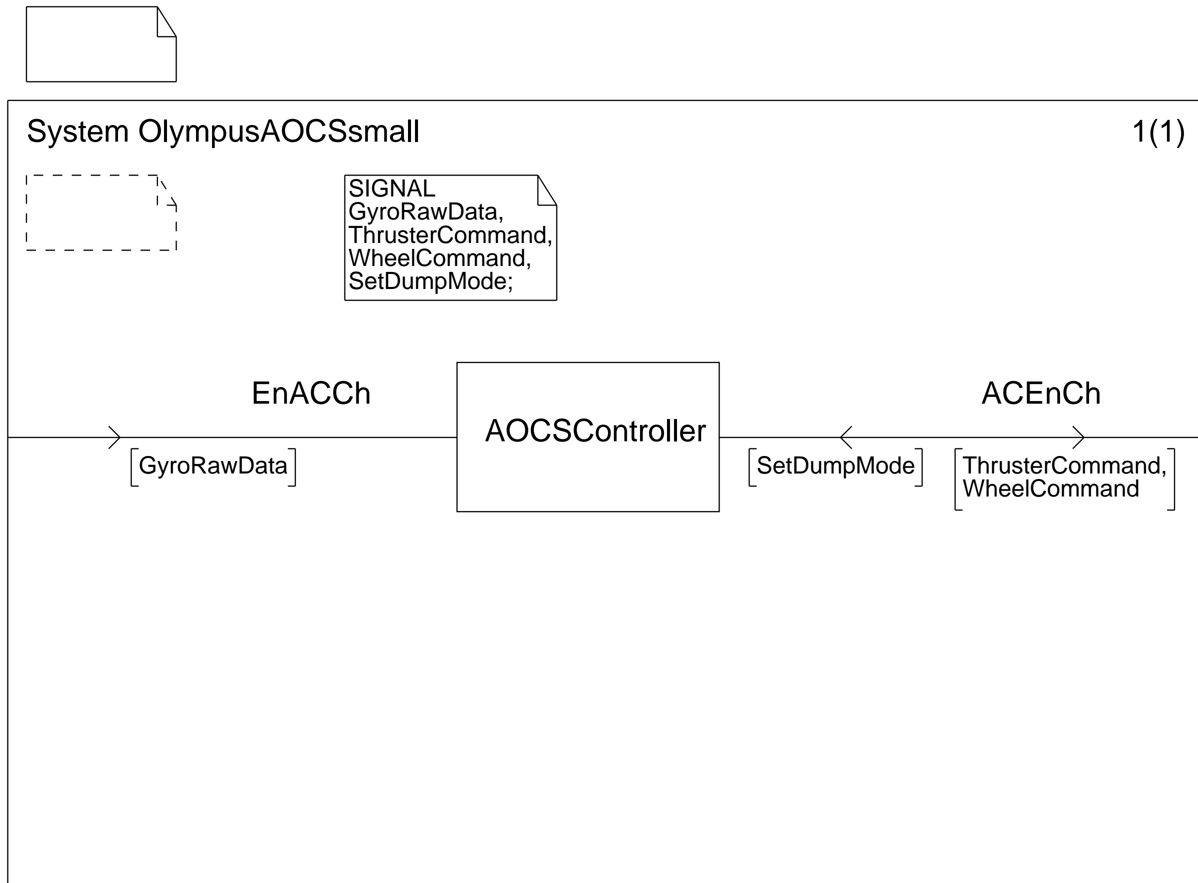

Figure A.2: Olympus AOCS SDL System

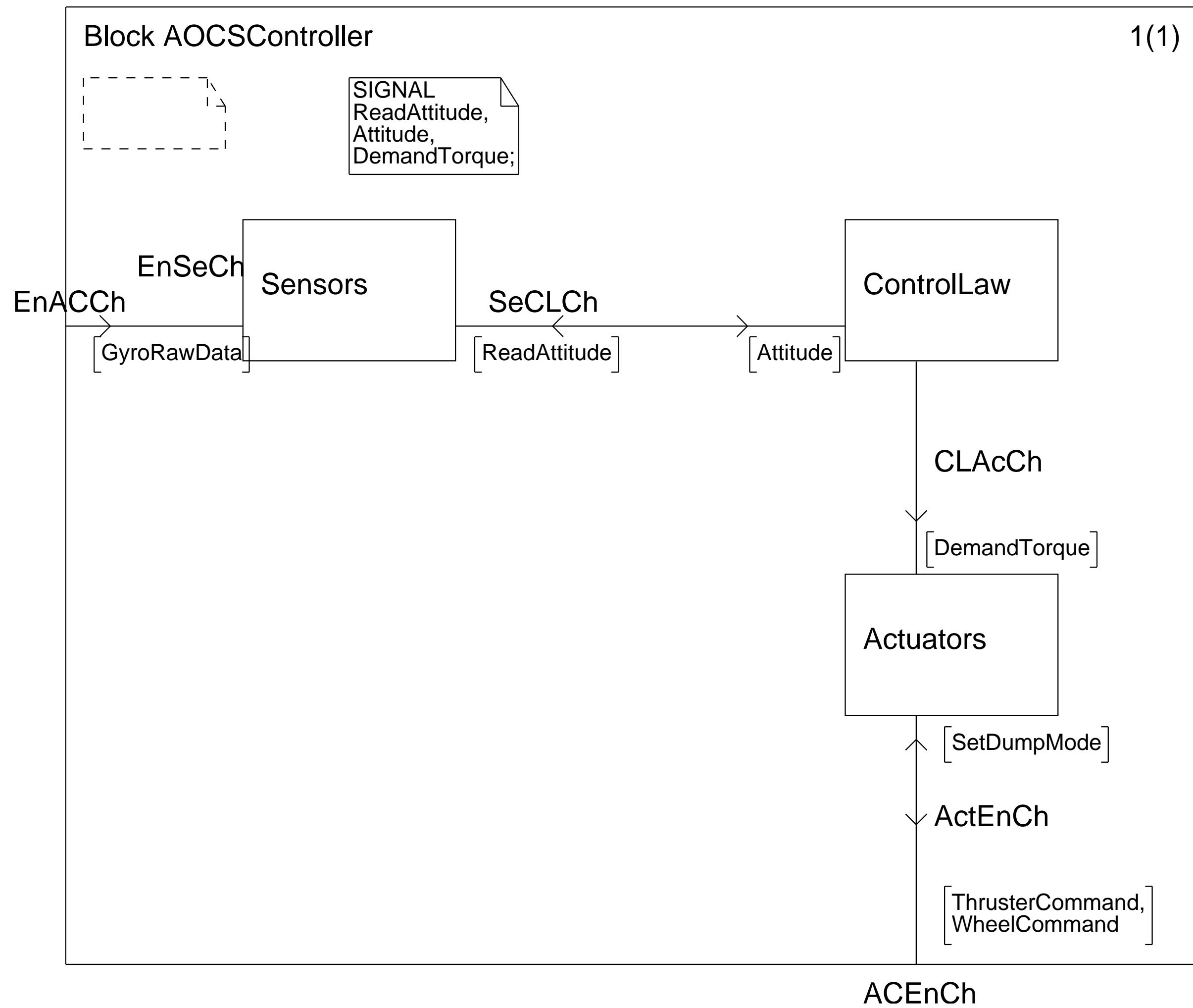

Figure A.3: Block AOCSController

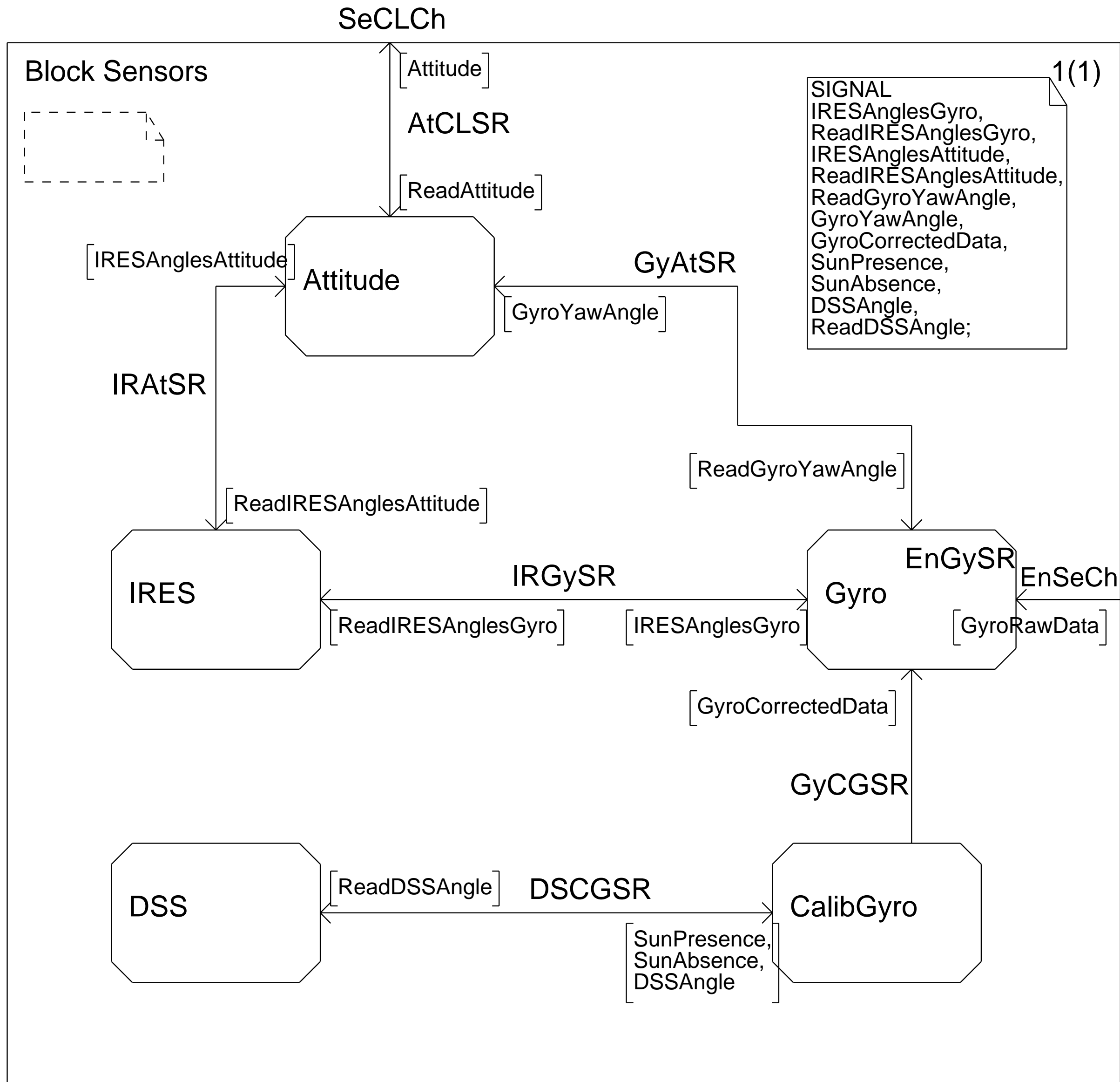

Figure A.4: Block Sensors

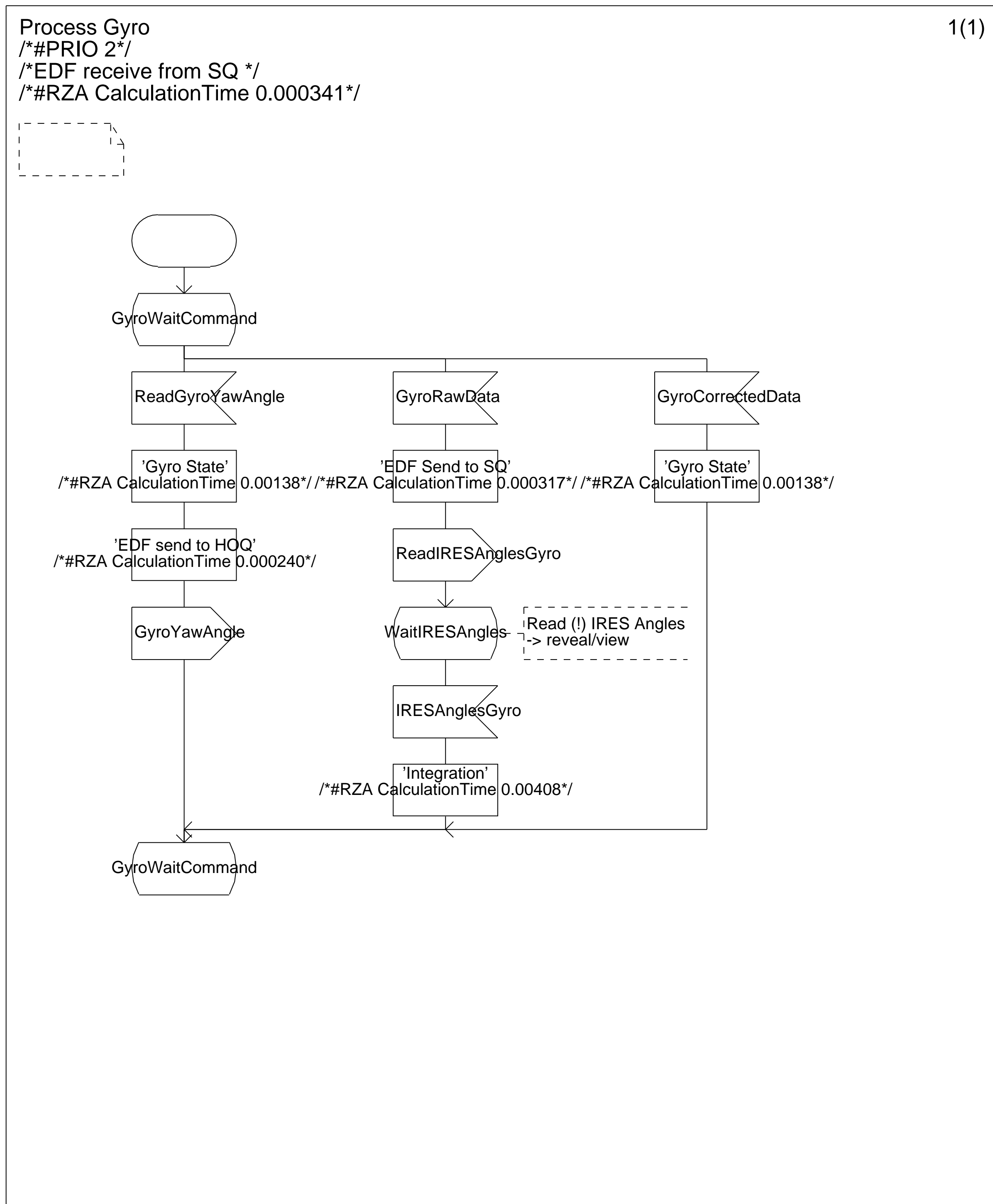

Figure A.5: Process Gyro

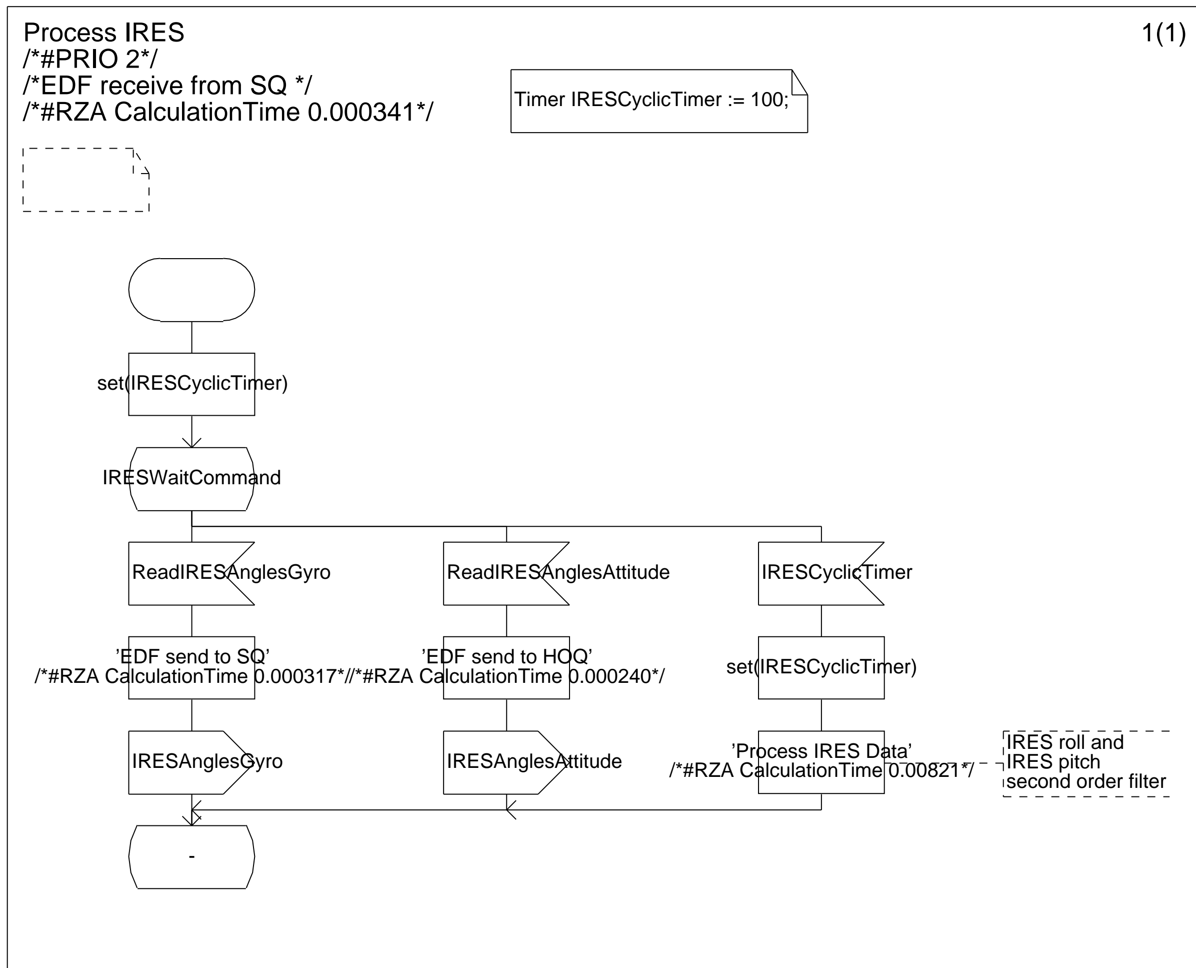

Figure A.6: Process IRES

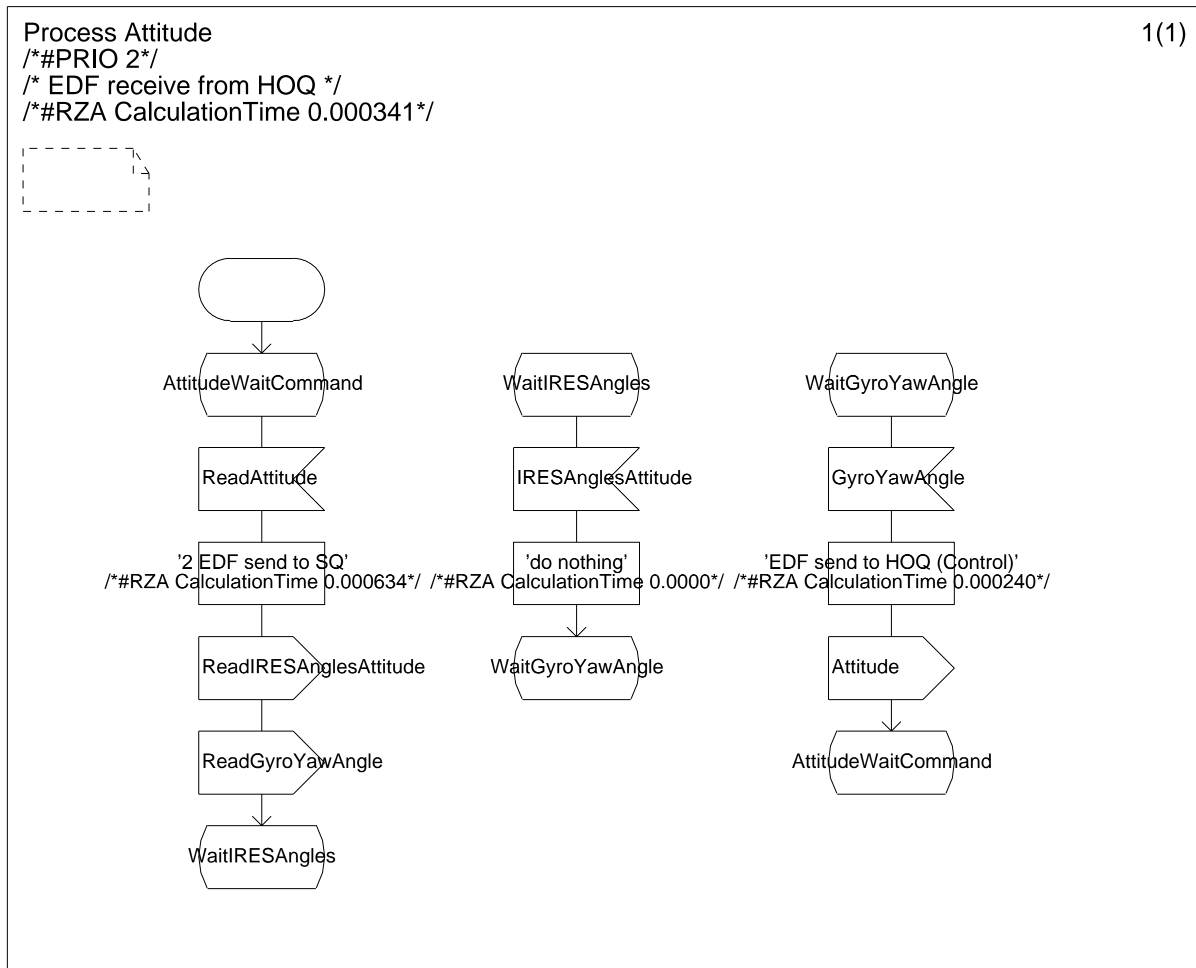

Figure A.7: Process Attitude

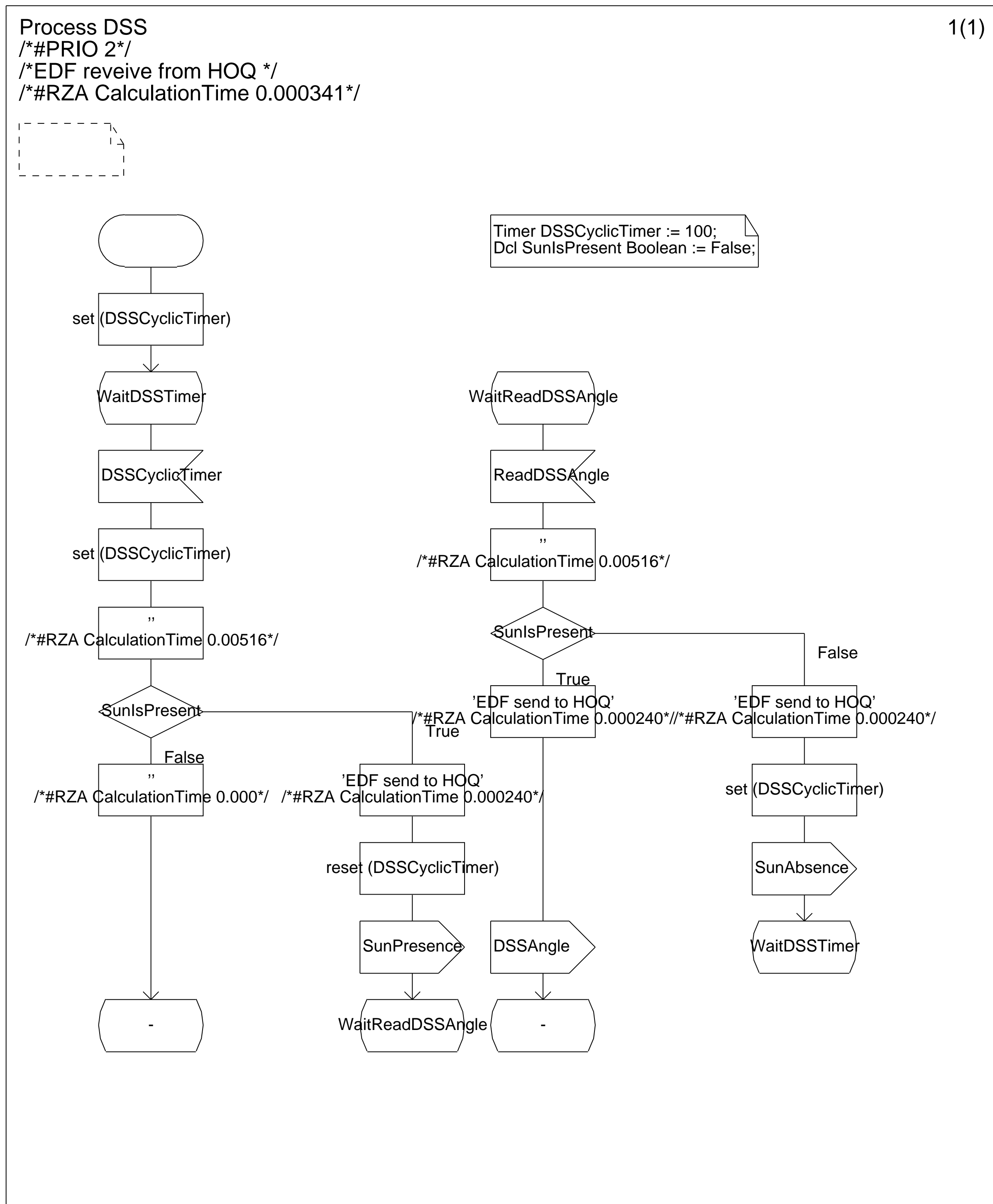

Figure A.8: Process DSS

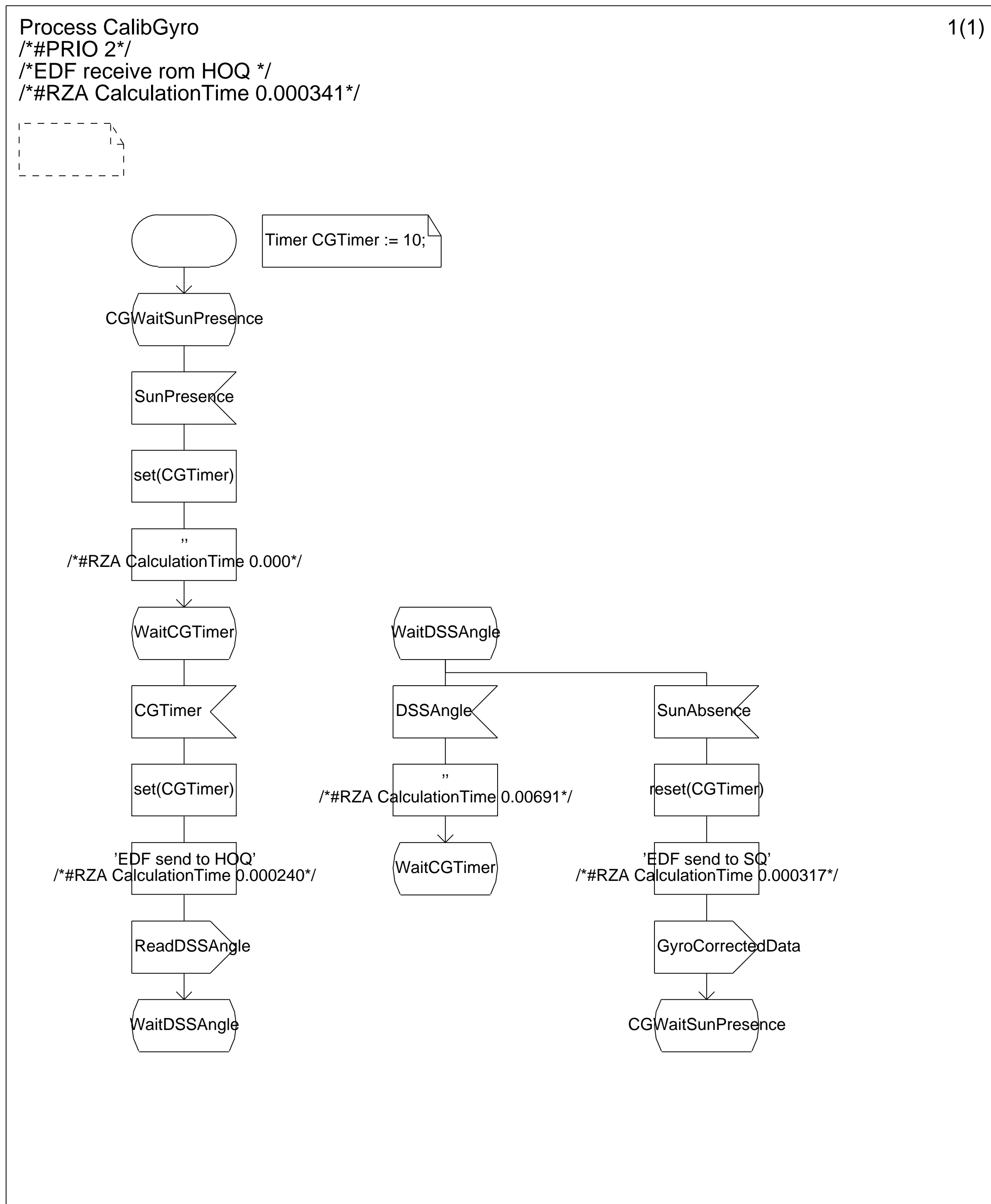

Figure A.9: Process CalibrateGyro

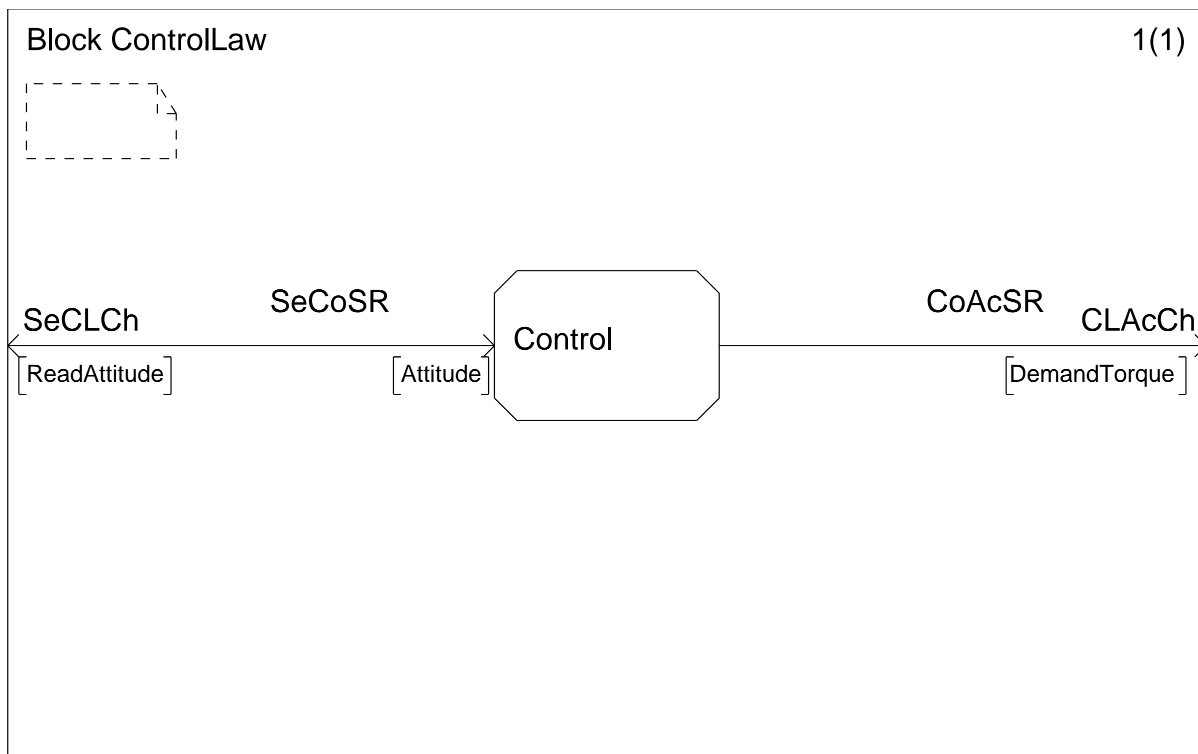

Figure A.10: Block ControlLaw

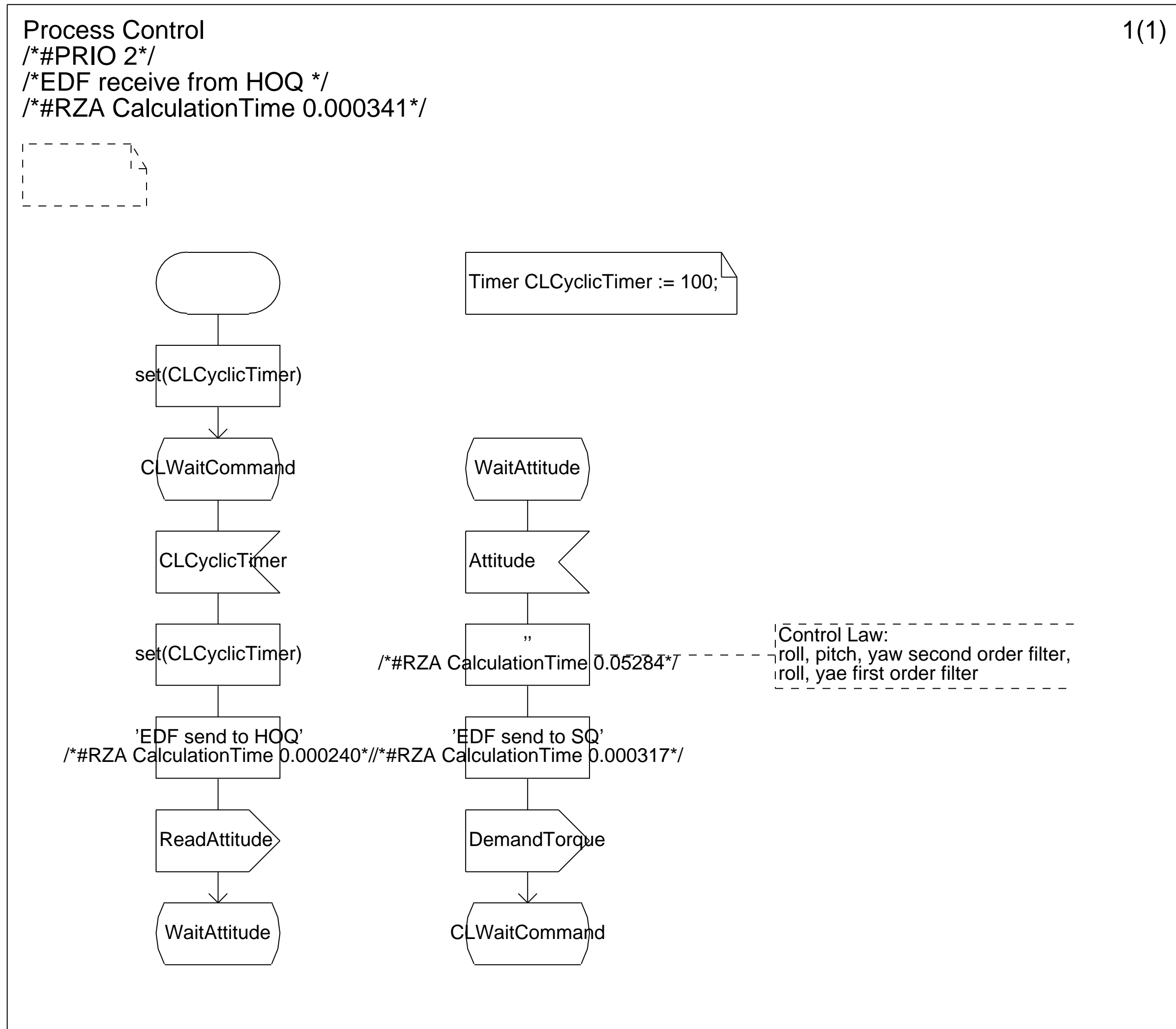

Figure A.11: Process Control

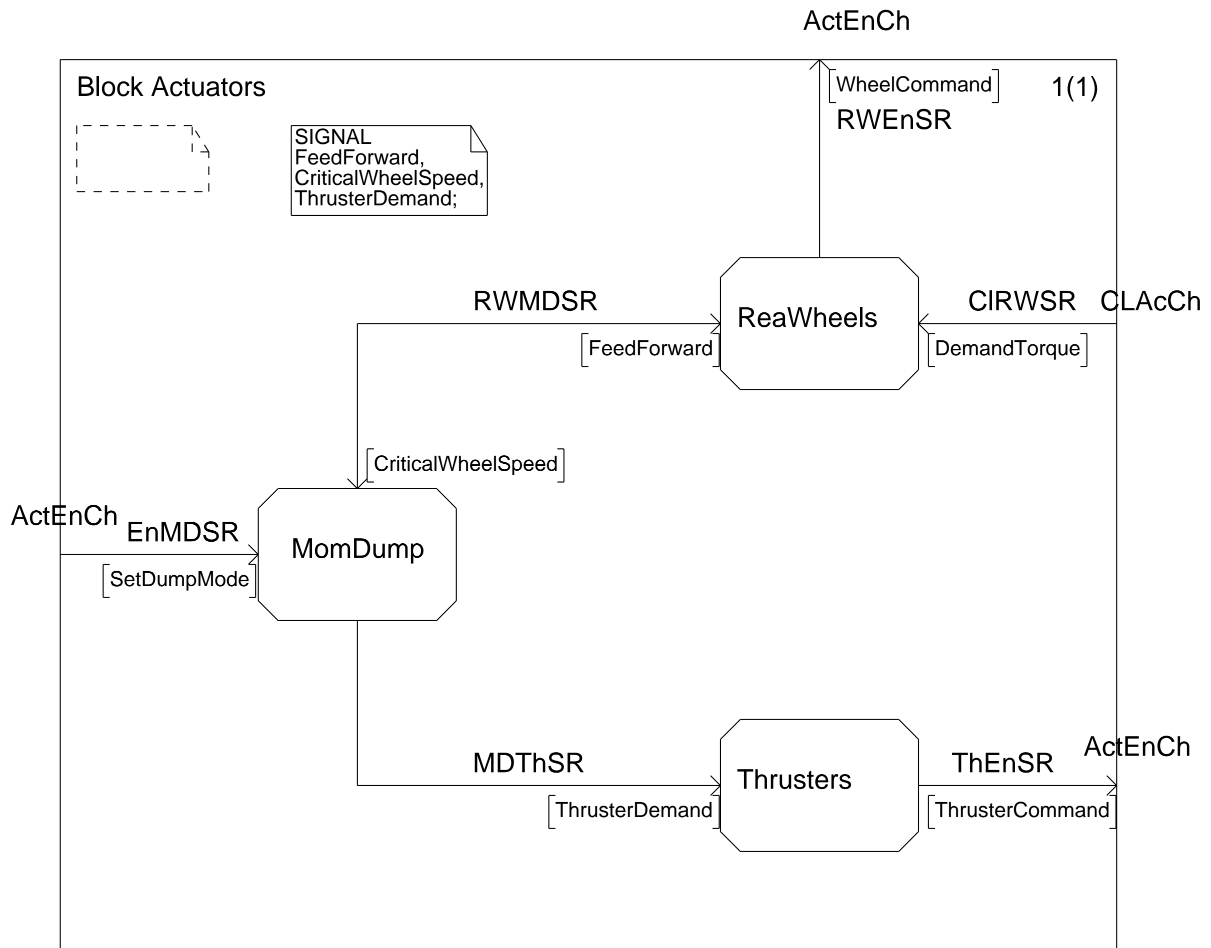

Figure A.12: Block Actuators

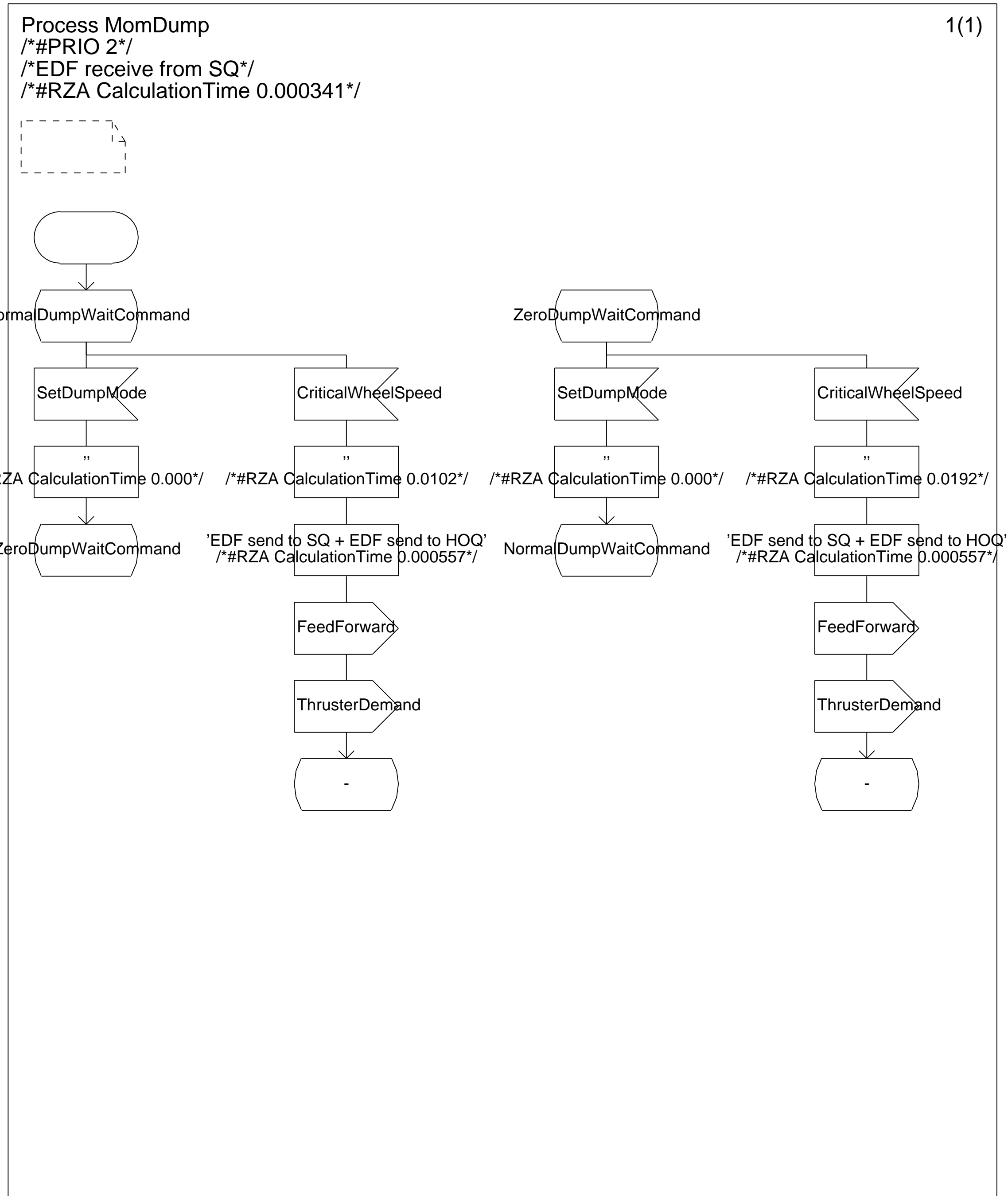

Figure A.13: Process MomentumDump

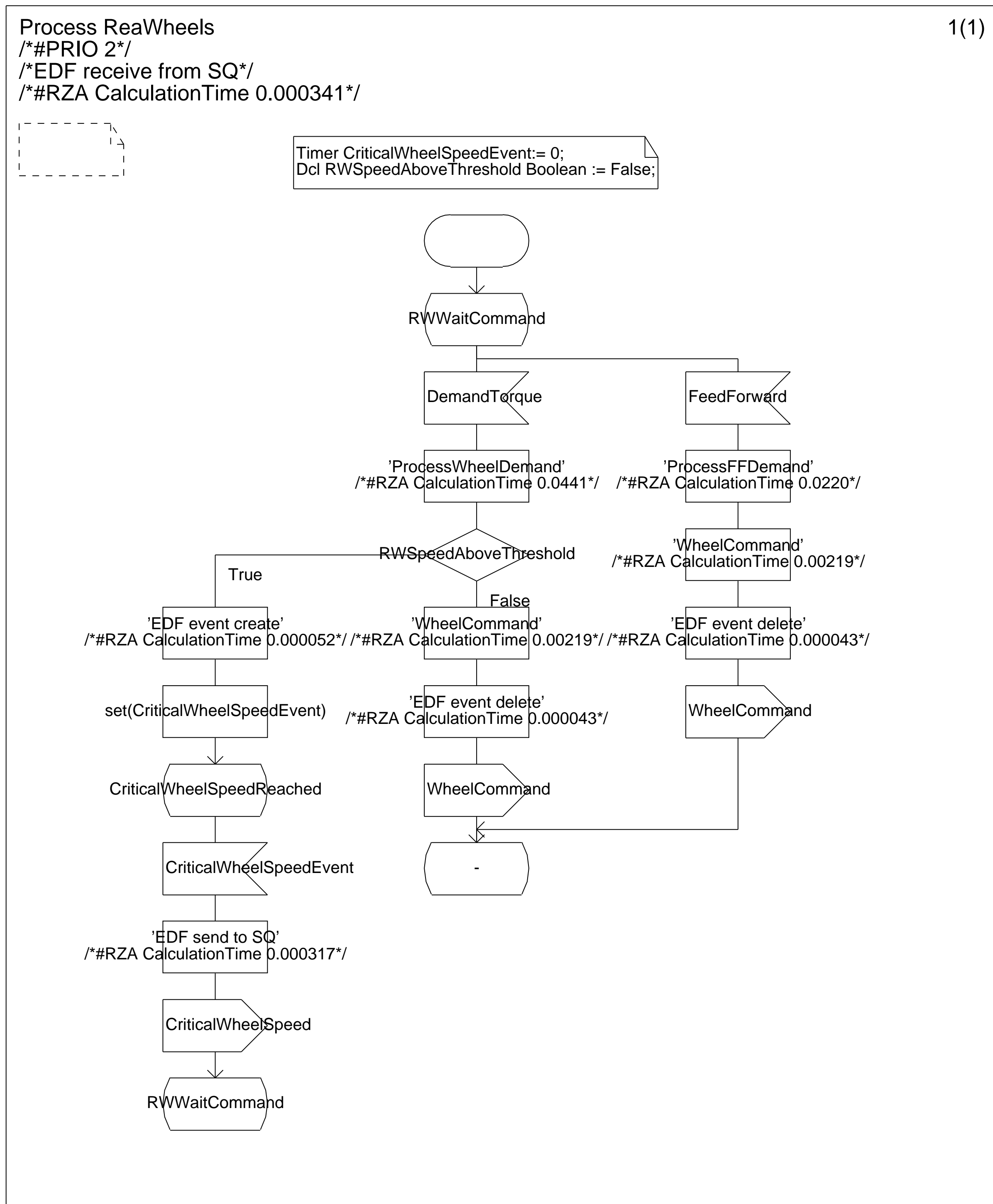

Figure A.14: Process ReactionWheels

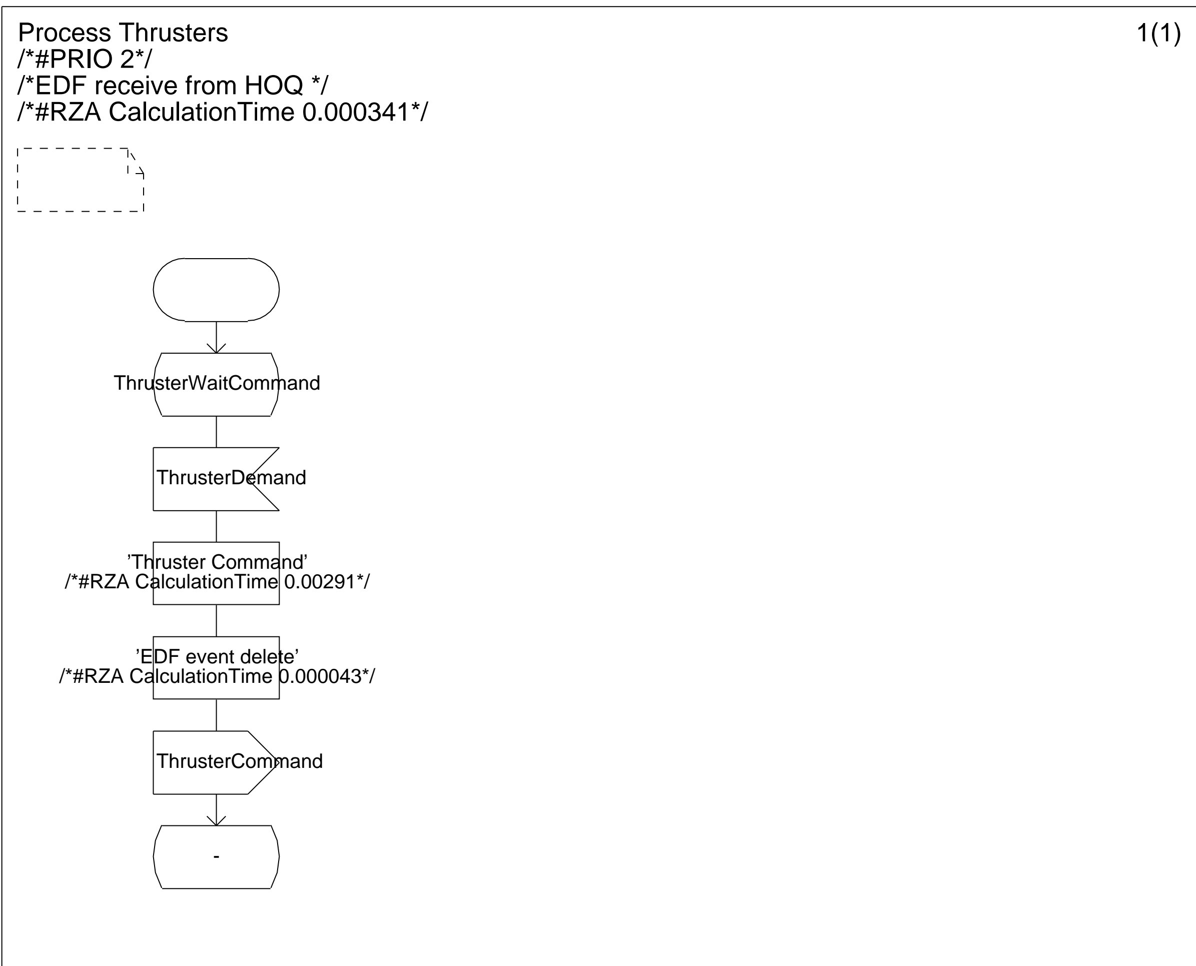

Figure A.15: Process Thrusters

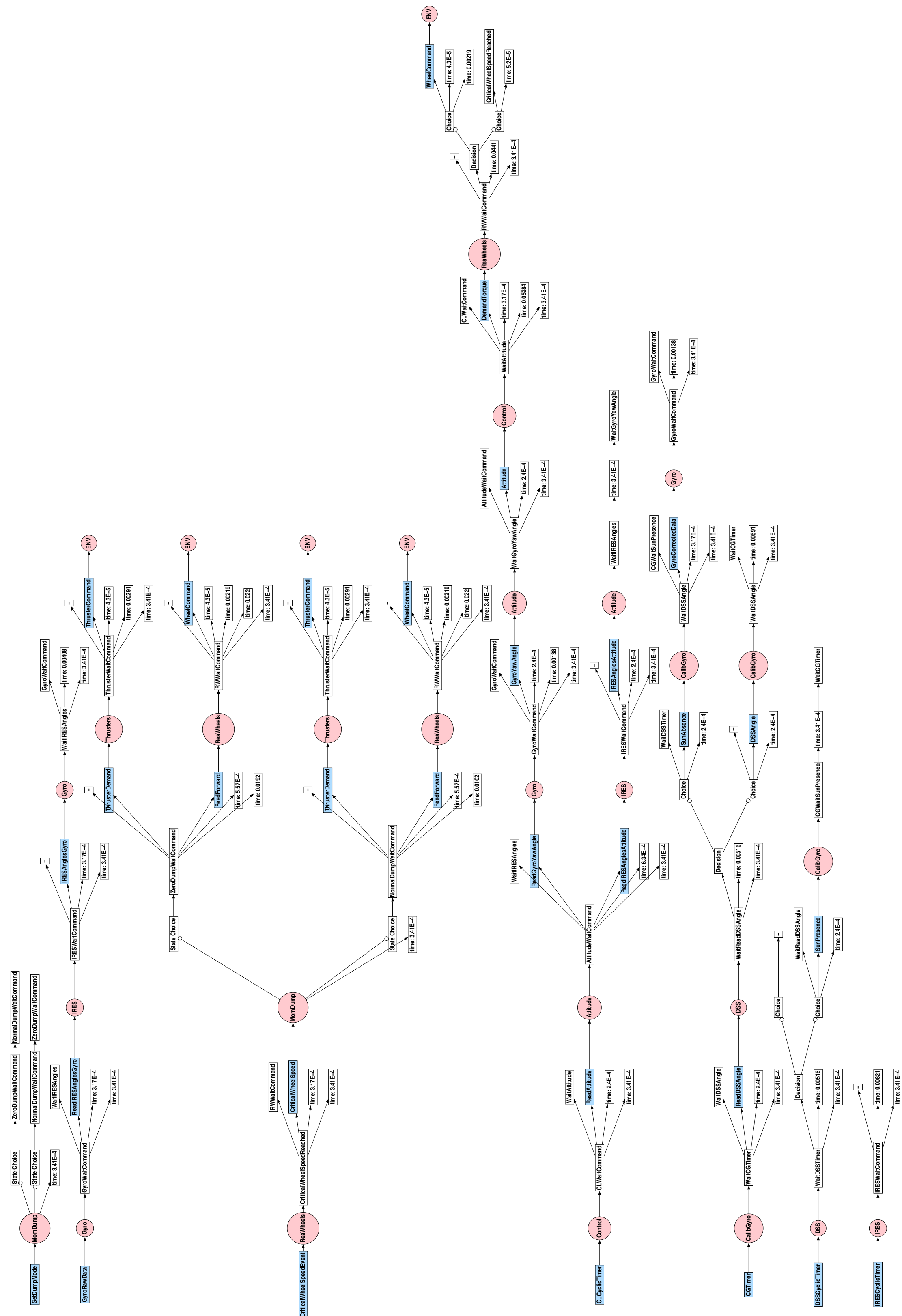

Figure A.16: Olympus AOCS Task Precedence Graph

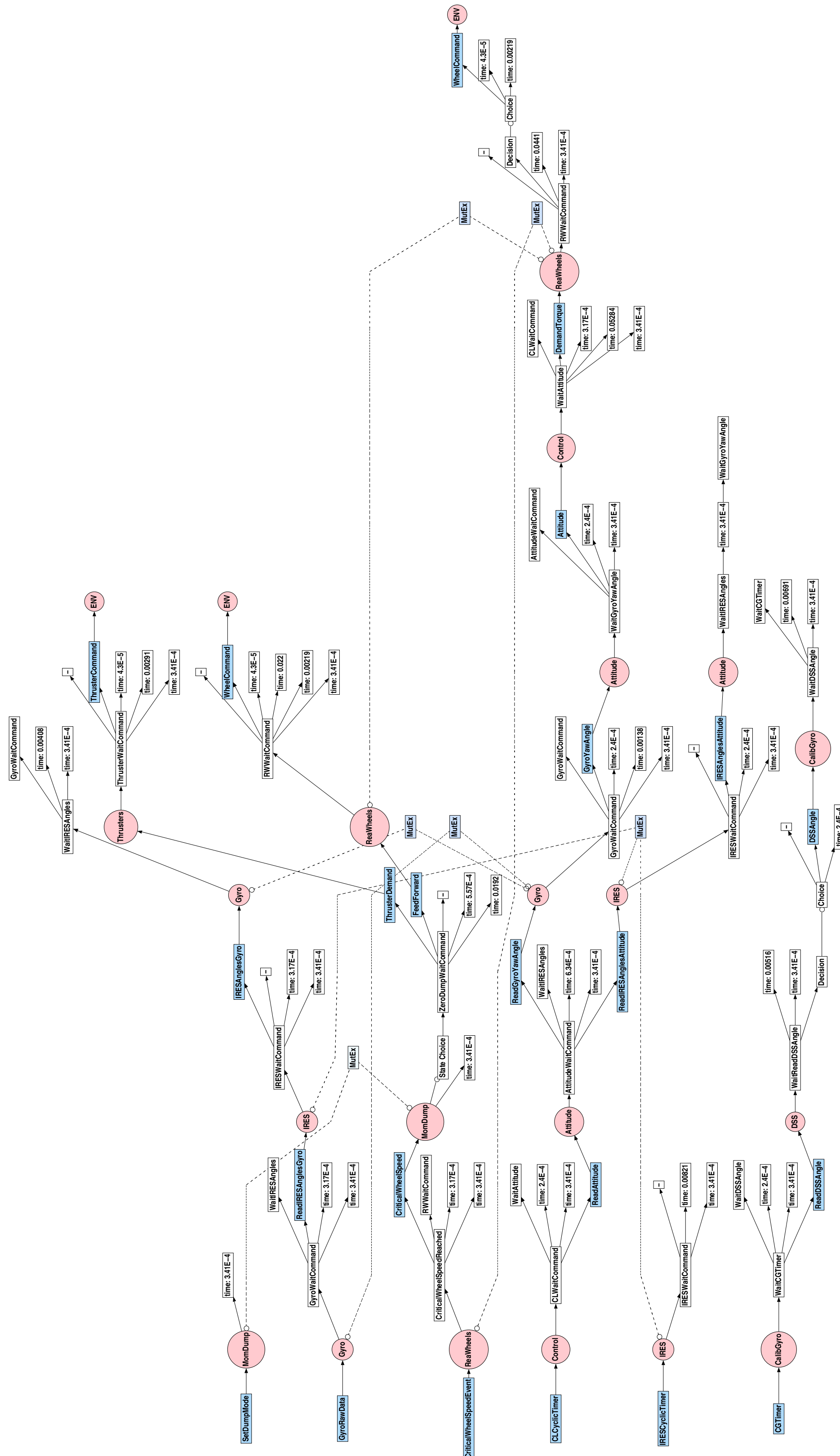

Figure A.17: Olympus AOCS Real–Time Analysis Model

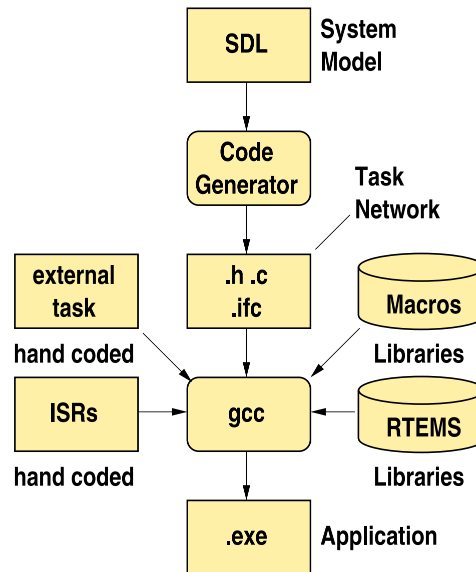

Figure A.18: SDT C-Advanced System Build Process

A.2 Code Generation with MEDF Integration

This section demonstrates how to incorporate MEDF scheduling into a tight integration with SDT's C–Advanced code generator (Telelogic).

A.2.1 SDT's Code Generation

SDT's different code generation priciples have been evaluated in detail in Sec. 4.2.1. As can be seen in Figure 4.2, tight integration maps one SDL process to one RTOS task and one additional RTOS message queue. Scheduling of SDL processes, respectively RTOS tasks is managed by the RTOS kernel (in contrast to the light integration model) [Sta97]. SDL signal exchange is replaced by RTOS send and receive directives. To connect to the SDL system environment, hand coded interrupt service routines have to translate external events to messages, which will be sent into the generated task system and a hand coded external task translates SDL signals to the system environment to output functions.

Generated tasks together with ISRs and the external task are linked to the final application (Fig. A.18). A likewise synthesized interface file (.ifc) that contains types, contents and names of signals from and to the environment assures consistency at modul boundaries. RTOS primitives, responsible for SDL process and signal management, are hidden with multi level C–macros, allowing an easy porting to other real–time operating systems. Telelogic provides an integration for e.g. VxWorks, PSOS and OSE. RTEMS macros were derived from the very similar VxWorks package. RTEMS itself is linked in form of a library to the application.

User directives in a SDL specification allow the application designer to add implementaion details to the system model.

`#PRIO` This keyword will be mapped to C–macros (mainly `RTOSSDL_STATIC_CREATE`) that influence the priority and modes of a task and the queue modes during their creation.

`#CODE` Code following this directive in SDL state transitions, signal or process declarations will be included unchanged in the generated C–code.

A.2.2 MEDF Scheduling Application Interface in RTEMS

Instead of attaching a deadline as an absolute point of time to a SDL signal, a so called *EDF event object* is transported by the underlying RTEMS message. Each unique EDF event represents a relative deadline starting with the point in time of the EDF event's creation. Sending a deadline message, the receiving task will adopt the transported deadline, respectively the deadline of the transported EDF event and will be scheduled accordingly, i.e. will be inserted into the earliest deadline first (EDF) sorted RTEMS ready list. RTEMS directives, object modes and attributes provided by the different RTEMS managers are summarized in Table A.2.

EDF task manager	
`RTEMS_EDF` (task mode)	enforces EDF scheduling
`set_deadline( tid, deadline )`	sets deadline of a task
MEDF event object manager	
`edf_event_create( name, deadline, &eid )`	creates an event and sets its deadline interval
`edf_event_delete( eid )`	deletes an event
`edf_event_ident( name, &eid )`	returns object id of an event
`edf_event_get_deadline( eid, &deadline )`	returns remaining deadline of an event
MEDF message queue manager	
`RTEMS_EDF` (queue mode)	enforces MEDF scheduling
`RTEMS_INHERIT_DEADLINE` (queue attribut)	enables deadline inheritance
`send_edf_event( qid, eid, ... )`	attach event to message and send it to queue
`receive_edf_event( qid, &eid, ... )`	receive message and adopt deadline from event

Table A.2: MEDF directives, object modes and attributes

RTEMS administrates 256 different priority levels. For each priority level a single ready list is kept up to date. Since the EDF scheduling is integrated into RTEMS by modifying the sort order of one single ready list of a dedicated priority level, it is up to the user to choose an appropriate priority for the tasks with EDF mode (obviously the highest priority in normal mode operation).

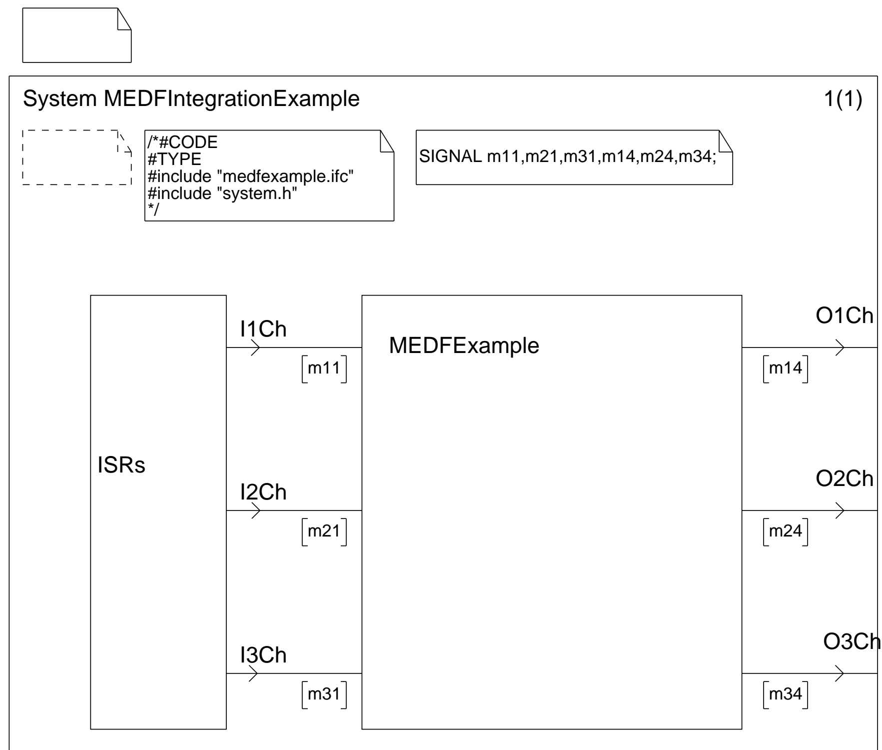

Figure A.19: Application Example: SDL System

A.2.3 Example System

This very simple application example consists of a SDL process network with 3 precedence systems, i.e. there exist 3 stimulating external events. Event recognition is done by interrupt service routines (ISRs). For simplicity and demonstration reasons the ISRs are modelled in SDL too (Block `ISRs` in Fig. A.19 and Fig. A.20). SDL process `ISR1` in Figure A.21 shows the deadline specification in a transition including a `#CODE` directive with the statement `XSIGNALDEADLINE(1000)`. The `XSIGNALDEADLINE` command is a new C–macro, defined in the RTEMS integration macro package. It will be replaced with a `rtems_edf_event_create` directive.

```
#define XSIGNALDEADLINE( SIGDL ) \
{ \
   rtems_status_code status; \
   status = rtems_edf_event_create( \
                ++rtems_name_count, \
                SIGDL, \
                &yVarP->SignalEvent); \
   XOS_ERROR("CREATE", \
             "rtems_edf_event_create", \
              status); \
}
```

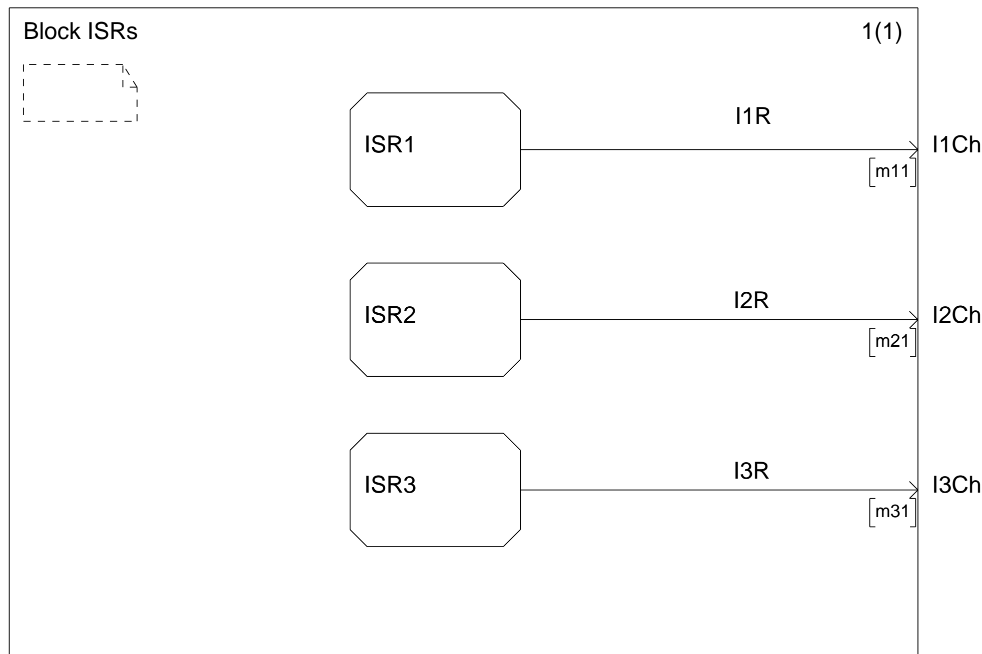

Figure A.20: Application Example: ISR Block

`SIGDL` specifies the new event's deadline in *ticks*, respectively so called *fastticks* [Wro97], `rtems_name_count` is used instead of a real name. The process variable `SignalEvent` holds the unique RTEMS object id for the EDF event and will be passed to the `rtems_message_queue_send_edf_event` within the SDL *output m11* command in the same SDL transition. Therefore the deadline transportation within the SDL specification is hidden from the user.

The second macro in this transition, `XSIGNALPERIOD` as well as the endless loop structure would not appear in a real ISR implementation and is only used for the RTEMS unix simulator realization.

```
#define XSIGNALPERIOD( SIGPERIOD ) \
{ \
   rtems_status_code status; \
   status = rtems_task_wake_after( \
                SIGPERIOD ); \
   XOS_ERROR("PERIOD", \
             "rtems_task_wake_after", \
             status); \
}
```

The main application is encapsulated within the `MEDFExample` block (cf. Fig. A.22). It shows a network of 8 SDL processes, all except process S with a state machine structure identical with process 11 in Figure A.23. A state transition is triggered with a SDL *receive* command (signal $m11$). After some action (empty SDL *task* symbol in this simple example)

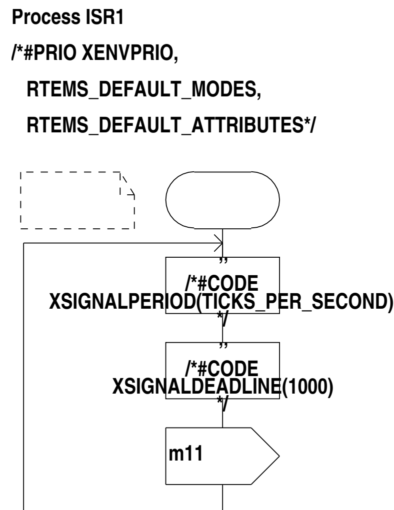

Figure A.21: Application Example: Interrupt Service Routine

a new SDL signal (signal $m12$ in process 11) is forwarded together with the attached EDF event. As mentioned above, receiving, passing and sending the EDF event is hidden from the user and has not to be modelled.

SDL process S (Fig. A.24) acts like a server process, i.e. it responses on serveral different requests (signals $m12$ and $m22$). To avoid priority inversion with this message queue, the attribut `RTEMS_INHERIT_DEADLINE` is enabled during the message queue's creation. For this purpose, one can use the `#PRIO` directive in the process' name specification field. The (modified) syntax of this directive is defined as:

```
#PRIO( task priority,
       task creation modes,
       queue creation attributes )
```

In this example, two different task priorities are used: One level for all processes with EDF message scheduling (process 11 to process 33), selected with the macro `XEDFPRIO` and one priority level for the external task and the pseudo ISRs (ISR tasks in the UNIX simulation, macro `XENVPRIO`). According to this, the EDF processes must be created with `RTEMS_EDF` mode enabled and for the queues the attribut `RTEMS_EDF_EVENT` has to be switched on.

The hand coded external task finally receives all messages sent to the environment ($m14$, $m24$ and $m34$) and deletes the transported EDF events at the end of each task precedence system (external task is not shown in Fig. A.25).

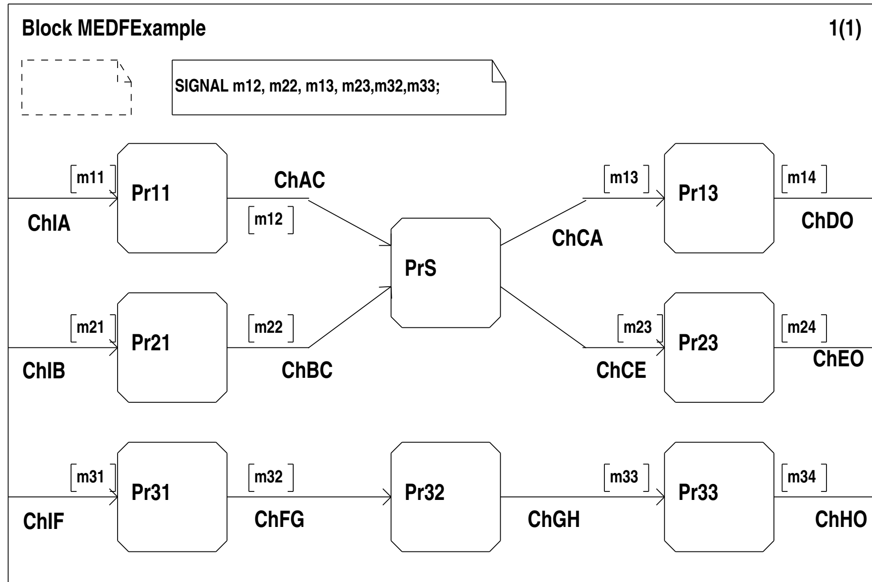

Figure A.22: Application Example: SDL Process Network

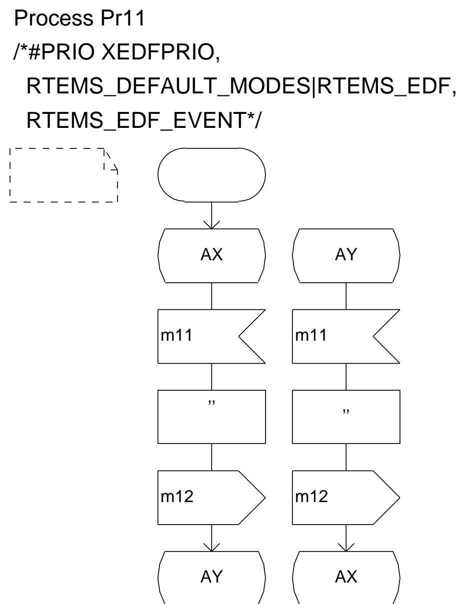

Figure A.23: Application Example: Process 11

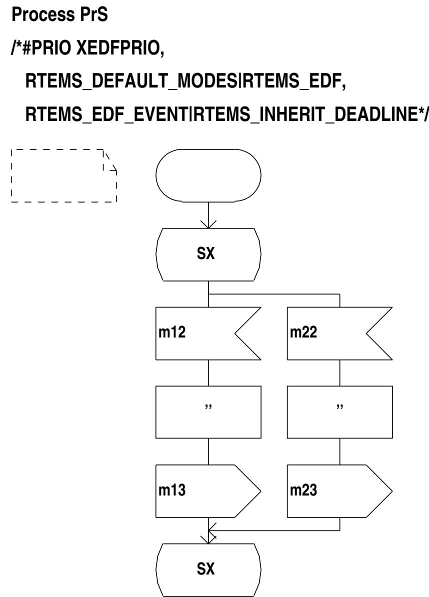

Figure A.24: Application Example: Server Process S

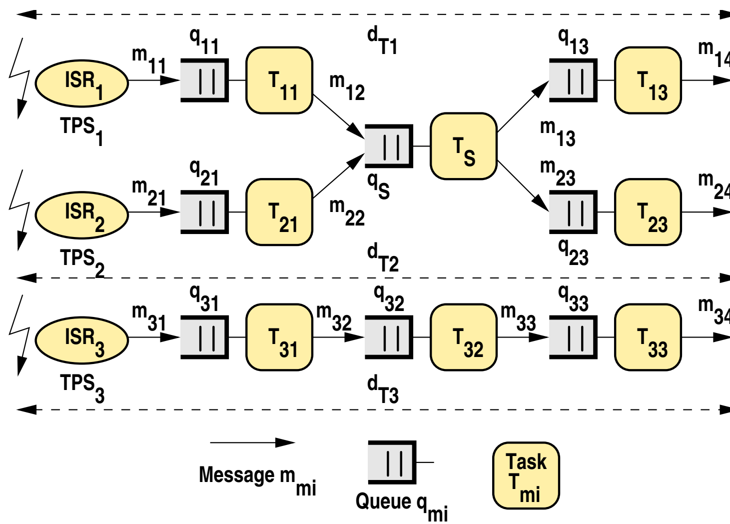

Figure A.25: Generated Task System

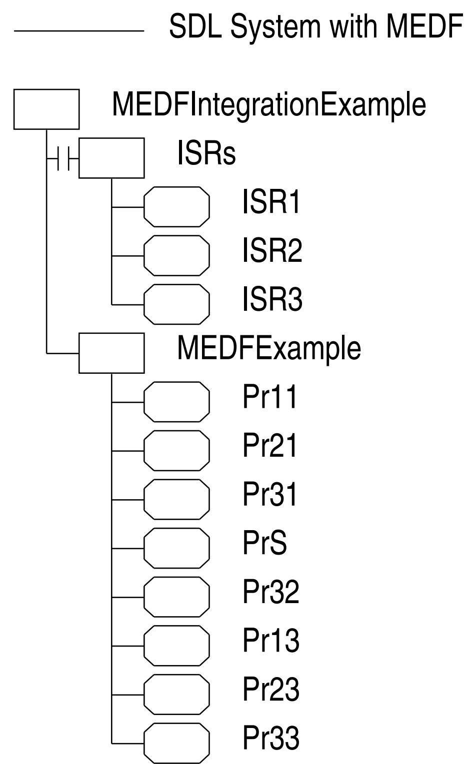

Figure A.26: Application Example: Organizer View

Appendix B

MEDF Implementation Details

B.1 Delta Deadline Management

To keep track of time (cf. Fig. B.1), with a delta list only one timer (*Fasttick Timer FT*) is needed for surveillance of the most critical deadline d_f on the list. Adding a thread to the EDF thread list, one has to distinguish two cases:

1. For a new later deadline $D_l = a_l + d_l$ and $D_l \geq D_f$, i.e. $d_l + FT \geq d_f, FT = a_l - a_f$ (at $a_f : FT = 0$): $\Delta d_l = D_l - D_f = FT + d_l - d_f$. More generally, while walking from the first item $\Delta d_0 = d_f$ to the end of the EDF thread list, deltas are subtracted from Δd_l as long as they are smaller. As soon as the new deadline value gets smaller than the current delta, the correct insertion point has been found. $\Delta d_l = FT + d_l - \sum_i \Delta d_i$ until $\Delta d_l < \Delta d_{i+1}$. After insertion the calculated new delta value needs to be subtracted from the next thread's delta deadline to keep delta continuity: $\Delta d'_{i+1} = \Delta d_{i+1} - \Delta d_l$.

2. For a new earlier deadline $D_e = a_e + d_e$ and $D_e < D_f$, i.e. $d_e + FT < d_f, FT = a_e - a_f$: $\Delta d_e = D_f - D_e = d_f - (FT + d_e)$. The new first entry becomes $d'_f = d_e$, the corrected second $\Delta d'_1 = \Delta d_e$. Finally, the deadline timer has to be restarted $FT = 0$.

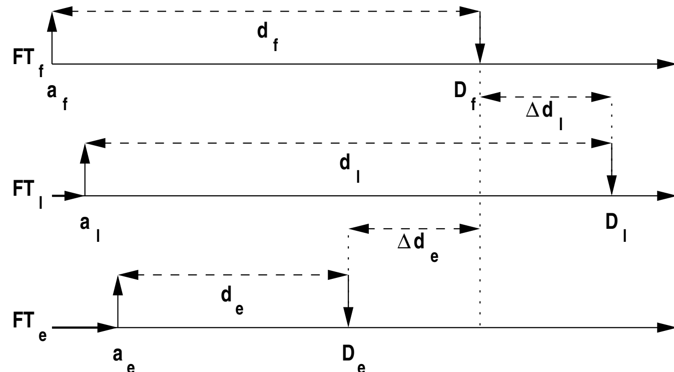

Figure B.1: Deadline Management

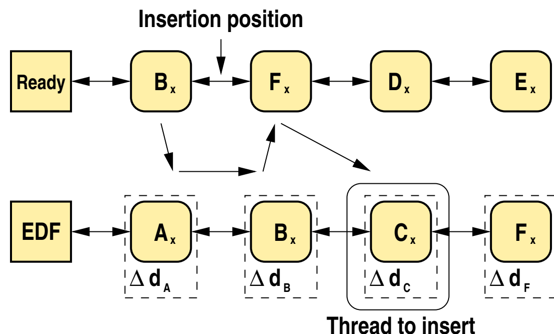

Figure B.2: Inserting into the Ready List

When a thread completes or its EDF mode has been cleared to turn off EDF scheduling, it has to be removed from the EDF thread list. If this thread is not the first thread on the EDF thread list, only its deadline value will have to be added to the delta deadline of the immediately following thread on the list. If the first thread is removed, the deadline timer interrupt will be cancelled during removal. The timer itself will keep on running, but additionally has to be adjusted (cf. Sec. B.3) to the new first thread on the list.

B.2 EDF List Insertion

This section details the list insertion algorithm with EDF sort order, in this example insertion into EDF ready list, based on the *EDF Thread List*.

A new EDF thread to insert is appended to all threads on the EDF ready list with less or equal deadlines. New non–EDF threads, with infinite deadlines, are simply appended at the end of the list. Since the EDF thread list is already sorted by deadline order, the search algorithm for the ready list insertion point can rely on this order (cf. Fig. B.3). The EDF thread list may contain more threads than the EDF ready list, as some EDF threads may currently be blocked. On the other hand, the EDF ready list may contain more than EDF threads because some of the threads may be normal non–EDF threads (e.g. threads D and E in Fig. B.2).

Assuming that thread C in Figure B.2 is no longer blocked and has to be inserted into the ready list, the insertion point of C can be found behind all EDF threads on the ready list, which lie before C on the EDF thread list. The algorithm uses a parallel search in both the ready and the EDF thread list. Starting with the first thread on the ready list (B), the search seeks this thread on the EDF thread list. Remembering this look–up EDF thread (B), the search advances to the next thread on the ready list (F). If the thread to insert (C) is detected before the thread under investigation (F in the example) on the EDF thread list, the correct insertion point is found before the current thread (F) on the ready list. Exceptions are an empty ready list or if the search advances to a non–EDF thread on the ready list. For the first case, the thread may be inserted directly, for the second case it will be inserted before the non–EDF threads with infinite deadlines.

```
Add_To_ThreadList_With_EDF( node insert_thread,
  list thread_list, list EDF_list ) {

  if ( list_is_empty( thread_list )
    Append( thread_list, insert_thread ); return;

  EDF_node    = EDF_list->first;
  thread_node = thread_list->first;

  for ( entire thread_list ) {
    if ( not is_EDF( thread_node ) ) break;

    for ( EDF_list beginning with EDF_node ) {
      if ( thread_node == insert_thread )
        Insertion_position_found = True; break;

      if ( thread_node == EDF_node )
        EDF_node = EDF_node->next; break;
    }

    if ( Insertion_position_found ) break;
  }

  Insert( insert_thread before thread_node );
}
```

Figure B.3: Insertion with EDF List Algorithm

B.3 Fasttick Board Support

The RTEMS `clock_tick()` function is called once per timer interrupt within the board support package (BSP). “Tick” resolution is set up by the application’s designer by telling RTEMS about how many ticks will occur in a second. Common periods are in the magnitude of 10 ms, which leads to 100 timer interrupts per second. As explained above, EDF lists require together one single deadline timer to keep track of time. This deadline timer will be loaded with the shortest deadline intervall, an expiration would indicate a deadline violation. Using the clock tick, a deadline timer would at best have the same precision. Higher resolutions of the clock tick would increase timer interrupt load significantly.

For this reason, an additional *fasttick* timer is introduced. For list management the fasttick has to support the following operations: Beneath intervall setting and resetting, it must provide on request the time used up of the set intervall (“deadline so far”) and must allow an adjustment of the deadline currently set to a new deadline without stopping the timer to achieve deadline continuity.

The clock device driver for the processing unit, a MIPS R4650 processor based Galileo–

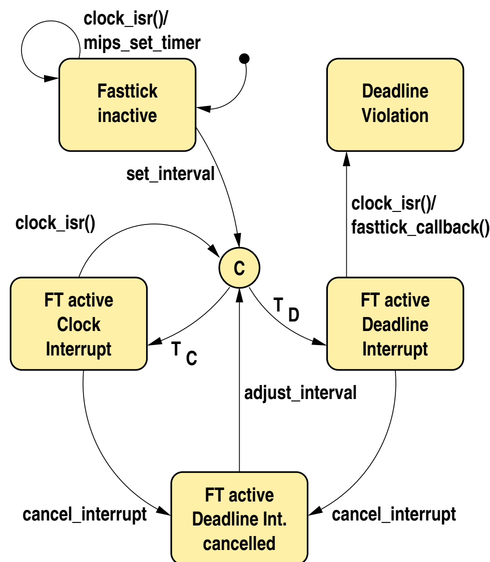

Figure B.4: Deadline Supervision with Clock and Fasttick Support (simplified)

4 Evaluation Board [Gal95, Gal96], uses the processor's cycle counter and compare register interrupt to generate the clock tick. For the fasttick an additional hardware timer within the Galileo controller could have been used, but the interrupts from controller to processor are already occupied by the IPC layer of the multiprocessing environment. Alternatively a cooperative tick/fasttick implementation was realized with a software statemachine (Fig. B.4) on top of the processor's cycle counter. Transitions between states can be triggered either by the fasttick interface or by the `clock_isr()`. In transitions to fasttick timer active states, the current deadline intervall will be compared (decision C in Fig. B.4) to the number of cycles to the next clock interrupt. If smaller, the next interrupt will be an expiring deadline timer (transition T_D), otherwise a clock interrupt will be expected (transition T_C).

An interrupt in state "FT active Deadline Interrupt" signals a deadline violation and causes the `clock_isr()` function to call the RTEMS fatal error manager. A connected user extension may then take appropriate measures to react on this situation, e.g. switch the system into a fail safe state.

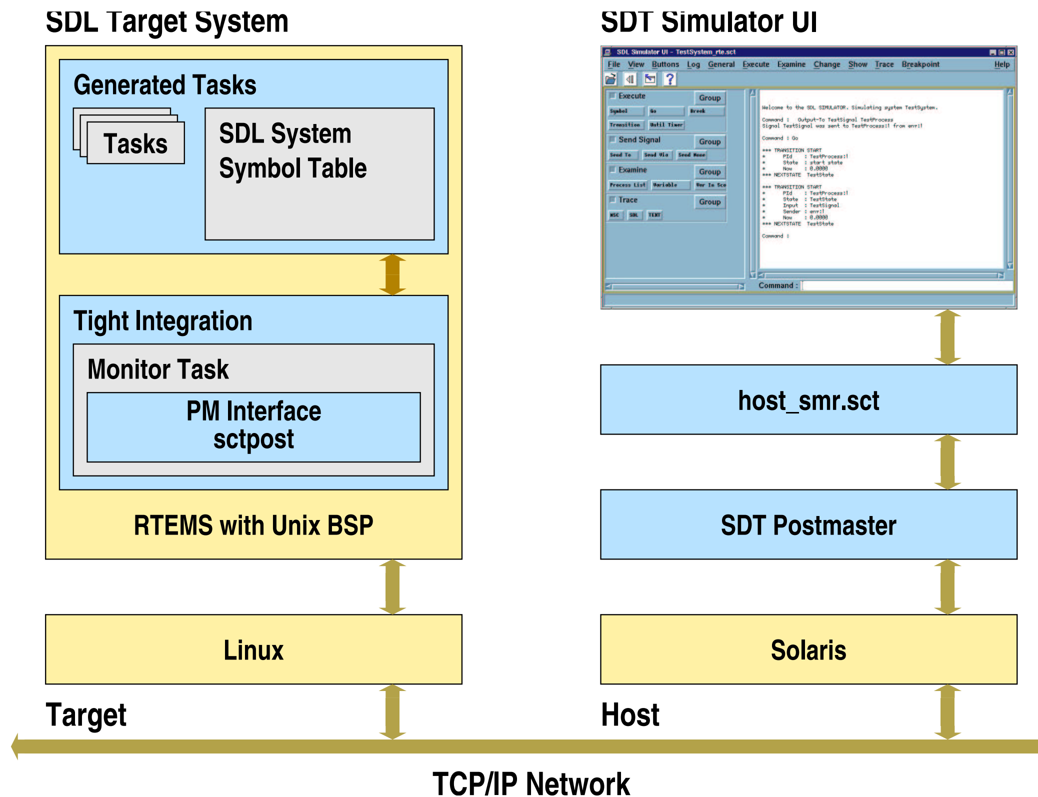

Figure B.5: MEDF Simulation based on SDT Tool Suit

B.4 MEDF Semantics with SDT's Simulator

All three, the semantics of the specification, the behaviour of the final implementation, and of course the processing sequence of the simulation have to match, to allow a system designer to verify the functionality of his SDL model. Telelogic's SDT tool suite provides different models of computation during simulation[1], but regarding scheduling policies only fixed process priorities are supported. For this reason, a MEDF simulator kernel has to be integrated into the SDL design framework.

MEDF simulation uses different software packages: SDT's remote–target simulation via TCP/IP sockets, SDT's C–advanced code generation with tight integration onto RTEMS, RTEMS itself with a so called "Unix Board Support Package" (RTEMS Simulator running as a stand–alone process under Linux) and an additional RTEMS monitor task (see Fig. B.5). SDT's remote–target simulation normally requires the "Master Library" to be included in the target, i.e. the master library's own monitor, which is responsible for the scheduling of all SDL processes in the implementation, assures communication of simulation control commands from and to SDT's Simulator User Interface (Communication via modul "Postmaster Interface", modul "sctpost", TCP/IP socket, "SDT Postmaster", "SimUI"). Since a tight integration with RTEMS is needed to enforce MEDF processing

[1] Mainly different interpretations of time progress (see Sec. 3.2.1).

during simulation, the above mentioned monitor of the master library has to be replaced by a monitor that leaves scheduling of tasks to the RTOS kernel, i.e. the monitor now has to be an RTOS tasks for itself.

Again, the monitor tasks establishes connection via TCP/IP to SDT's postmaster using moduls "Postmaster Interface" and "sctpost". Furthermore, the monitor realizes the following basic simulation commands [Für01]:

Tracing allows as usual to observe process creations, state transitions, set and reset operations of timers, and signal receive, respectively signal output. Since signals now transport a deadline interval, the appropriate MEDF event object id and the remaining response time will be shown on the Simulator User Interface. Tracing has to be enabled by compiler switches within the tight integration macro package.

Signal This command allows to feed signals with a deadline interval attached into the system model. By dialog, the user will be prompted for a target process and may select a valid signal.

Break This directive stops and resumes simulation time. During a break the global system clock is halted, i.e. state of SDL timers as well as deadlines will not change, consequently deadline surveillance is disabled.