

Time-Triggered Organic Computing

DISSERTATION
zur Erlangung des Grades eines Doktors
der Ingenieurwissenschaften

vorgelegt von
Msc. Mario Qosja

eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen
Siegen 2026

Betreuer und erster Gutachter
Prof. Dr. Roman Obermaisser
Universität Siegen

Zweiter Gutachter
Prof. Dr. Mathias Pacher
Duale Hochschule Baden-Württemberg Karlsruhe

Tag der mündlichen Prüfung

10 June 2026

Statutory declaration

I hereby declare in lieu of an oath that I have drawn up the present work without any undue assistance by third parties and without using any aids other than the ones specified. The data and concepts taken, either directly or indirectly, from any other sources have been marked, indicating the source. The work has not been submitted to any other examining authority neither in Germany nor abroad and neither in the same nor in any similar form. Use of the services of any PhD mediation institute or of any similar organisation has not been made. In addition, any family relationship, first-degree relationship, marriage, civil partnership or cohabitation to the proposed members of the PhD Commission do not exist.

Signed:

Location, Date:

Acknowledgements

I would like to thank Prof. Dr. Roman Obermaisser for giving me the opportunity to be part of this project, with his continuous help and directions throughout all the duration of this thesis.

Thanks to all my colleagues of the Embedded Systems chair at the University of Siegen, specially the Organic Computing team composed of Simon, Sabikun and Utkarsh.

To my family, thank you for all the sacrifices and for the infinite support during this journey.

Contents

List of Figures	9
List of Tables	10
Listings	11
List of Abbreviations	12
Abstract	16
1 INTRODUCTION	17
1.1 Motivation and Goal	17
1.2 Contributions	20
1.3 Organization	22
2 STATE-OF-THE-ART	23
2.1 Organic Computing	23
2.2 Time-Triggered Communication	26
2.3 Research Gap	27
2.4 Requirements / Challenges	30
2.5 Thesis contribution	31
3 BASIC CONCEPTS AND TECHNOLOGIES	32
3.1 Organic Computing	32
3.1.1 Observer/Controller Architecture	34
3.1.2 Artificial DNA	35
3.1.3 Artificial Hormone System (AHS)	36
3.2 Embedded Real-Time Systems	38
3.2.1 Requirements	41
3.2.2 Classification	43
3.2.3 Distributed Architecture	44
3.2.4 Global Time Base	45
3.2.5 Dependability	47
3.3 Real-Time Communication	50
3.3.1 Communication Network	50

3.3.2	Real-Time Network	50
3.3.3	Time Sensitive Network	51
3.4	Discrete-Event Simulations	54
3.4.1	SystemC	55
3.4.2	OMNeT++	55
4	TIME-TRIGGERED ORGANIC COMPUTING ARCHITECTURE	57
4.1	System Architecture	57
4.2	System State	60
4.2.1	Normal State	61
4.2.2	Reconfiguration State	61
4.3	Time-Triggered Schedule	66
4.3.1	Timing Model	66
4.3.2	Scheduling Algorithm	67
4.4	Communication Controller	72
4.5	Task Dispatcher	73
4.6	Time Sensitive Network	74
5	SIMULATION ENVIRONMENT FOR TIME-TRIGGERED ORGANIC COMPUTING	75
5.1	Simulation Architecture	75
5.2	Simulated PE	77
5.2.1	Old Simulated PE	77
5.2.2	New Simulated PE	78
5.2.3	SystemC Thread	82
5.3	Simulation of Communication Network	84
5.3.1	Dynamic TSN simulation	84
5.3.2	Message Construction	85
5.3.3	Gate Control List	86
5.3.4	Layer 2 services	87
5.4	Simulation Controller	88
5.5	Control and Data Plane	89
5.5.1	Message Queue (MSQ) System Design	89
5.5.2	Message Flow Between Tasks	90
6	EVALUATION	92
6.1	Evaluation Goals	92
6.2	Simulation Environment	94
6.2.1	Simulated Processing Element	94
6.2.2	OMNeT++ Network Simulator	94
6.2.3	Simulation Controller	94

6.3	Scalability	97
6.4	Fault Tolerance	99
6.4.1	Babbling Idiot Failure	99
6.5	Predictability	101
6.5.1	Delay	101
6.6	Dynamic Reconfiguration	103
7	Result Analysis	105
7.1	Simulation Environment	106
7.1.1	Simulated Processing Element	106
7.1.2	TSN Simulation	107
7.1.3	Simulation Controller	108
7.2	Scalability	109
7.3	Fault Tolerance	110
7.4	Predictability	111
7.5	Dynamic Reconfiguration	112
8	CONCLUSION AND FUTURE WORK	113
8.1	Conclusion	113
8.2	Future Work	115
	References	126
A	TTOC Simulatioir	127
B	Co-simulation Interface	150
C	OMNeT++ environment	162

List of Figures

3.1	Central OC Observer/Controller architecture	34
3.2	Basic element	35
3.3	Hormone Loop of AHS	37
3.4	Real-Time System	38
3.5	Scheduling hierarchy	39
3.6	Example of DAG graph	41
3.7	Distributed Computer System	45
3.8	Gate Control List of TAS	53
3.9	Discrete-event simulation event list	54
3.10	Example of discrete-event simulation event list	54
3.11	OMNet++ simulations architecture [120]	56
4.1	Inside view of TTOC processing element	58
4.2	Processing element state	60
4.3	Example of System Timing Model	66
4.4	DAG graph	67
4.5	Schedule table for task execution	70
5.1	Time-Triggered Organic Computing Simulation Architecture	76
5.2	DNASIM Architecture	78
5.3	Circular Linked List	79
5.4	Updated Circular Linked List	79
5.5	Execution flow of three processing elements	80
5.6	Inside view of TTOC processing element	82
5.7	Reconfiguration Module in INET/OMNeT++	85
5.8	TTOC TSN network device in INET/OMNeT++	86
5.9	StreamTable of the TSN Switch	87
5.10	Co-simulation logic	88
5.11	Message Queue Architecture	90
6.1	TSN PE in TTOC with one application	99
6.2	TSN PE in TTOC with two applications	100
7.1	Execution flow in SystemC simulator	106

7.2	Execution flow in SystemC simulator	106
7.3	Execution flow in SystemC simulator	107
7.4	Execution flow in SystemC simulator	107
7.5	Application tasks affected by PE failure	112

List of Tables

3.1	ADNA description of an Embedded System	36
3.2	Real-Time Scheduling Algorithms	40
3.3	Embedded Systems Requirements	42
3.4	Fault Types	47
3.5	Component Failure Types	48
3.6	Real-Time Networks	51
3.7	Per-Stream Filtering and Policing Table	52
4.1	Example 802.1Qci Filter Configuration	74
6.1	Hormones for each possible task in the first case	97
6.2	Hormones for each possible task in the second case	97
6.3	Hormones for each possible task in the third case	98
6.4	Message Transmission Time	101
6.5	ADNA Timed Event examples.	103
7.1	Message Transmission Time	111

Listings

6.1	PE Reconfiguration State	104
A.1	TTOC simulator task dispatcher	127
A.2	TTOC simulator list scheduling algorithm	139
A.3	TTOC simulator systemc procedure	148
B.1	Co-simulation controller	150
C.1	OMNeT++ new environment functions	162

Abbreviations

ADNA Artificial DNA

AHS Artificial Hormone System

ALU Arithmetic Logic Unit

ASoC Autonomous System on Chip

ATMA Adaptive Time-Triggered Multi Core Architecture

AUTOSTAR AUTomotive Open System ARchitecture

AVB Audio-Video-Bridging

BMCA Best Master Clock Algorithm

BMTS Basic Message Transport Service

CAN Controller Area Network

CARLA CAR Learning to Act simulator

CLL Circular Linked List

CNC Centralized Network Configuration

CPU Central Processing Unit

DAG Directed Acyclic Graph

DES Discrete-Event Simulation

DFG Deutsche Forschungsgemeinschaft

DM Deadline Monotonic

DNA Deoxyribonucleic Acid

DP Deployment Phase

DVFS Dynamic Voltage and Frequency Scaling

EDF Earliest Deadline First

ET Event-Triggered

FCU Fault-Containment Unit

FEL Future Event List

FES Future Event Set

FPP Fixed Priority Preemptive

FSM Finite State Machine

FTU Fault-Tolerant Unit

GCL Gate Control List

gRPC g Remote Procedure Call

HUD Head-Up Display

IBM International Business Machines

ID Identifier

IEEE Institute of Electrical and Electronics Engineers

ILP Integer Linear Programming

IoT Internet of Things

LIN Local Interconnect Network

LLF Least Laxity First

MAC Media Access Control

MLF Minimal Laxity First

MOST Media Oriented Systems Transport

MTBF Mean-Time Between Failures

MTTF Mean-Time-To-Failure

MTTR Mean-Time to Repair

NED Network Description Files

NETCONF Network Configuration Protocol

NoC Network on Chip

OBU Onboard Unit

OC Organic Computing

OMT Optimisation Modulo Theories

PDSE Priority-based Dynamic Slot Extension

PE Processing Element

PID Proportional–Integral–Derivative

PSFP Per-Stream Filtering and Policing

PTP Precision Time Protocol

QoS Quality of Service

RMS Rate Monotonic Scheduling

SMT Satisfiability Modulo Theories

SoC System on Chip

SOS Slightly-Off-Specification

SuOC System under Observation and Control

TAS Time-Aware Shaper

TCP/IP Transmission Control Protocol/Internet Protocol

TMR Triple Modular Redundancy

TSN Time Sensitive Network

TT Time-Triggered

TTA Time-Triggered Architecture

TTOC Time-Triggered Organic Computing

TTP Time-Triggered Protocol

VLAN Virtual Local Area Network

WCET Worst Case Execution Time

WSN Wireless Sensor Networks

ABSTRACT

The complexity of embedded computing systems is growing due to applications with increasing functional and non-functional requirements, including performance, reliability, and energy efficiency. Organic Computing (OC) has emerged as a research area for handling complex distributed embedded computer systems and is inspired by the organization of biological entities. In biology, Deoxyribonucleic Acid (DNA) is responsible for the structure and organization of living organisms. The same concept can be used in embedded systems, where a single computer file, known as Artificial DNA (ADNA), contains information for the whole system (e.g., sensors, actuators). In addition, following the hormone system in mammals, an Artificial Hormone System (AHS) has been developed that has the ability to read the ADNA and self-organize the system by exchanging hormones. It provides improved adaptability, robustness, and reduces development efforts. However, organic computing has drawbacks in terms of determinism, composability, and dependability, which are key features for safety-critical applications. Distributed computing systems using time-triggered communication networks have these characteristics and therefore offer distinct advantages for safety-critical systems. By combining organic computing technologies, such as artificial DNA and hormone features, with time-triggered communications, we can make these systems more adaptive, safe, reliable, and suitable for safety-critical applications while also providing self-x properties.

Therefore, in this thesis, we present a Time-Triggered Organic Computing architecture (TTOC). In TTOC, computing nodes, also known as Processing Elements (PEs), exchange hormones with each other in order to perform the spatial allocation of application tasks. After a task is allocated, the chosen PE utilises a list-based scheduling algorithm that is designed specifically according to TTOC requirements to calculate the system schedule dynamically. The list-based algorithm calculates the temporal allocation of application tasks together with the spatial and temporal assignments of application messages. The task dispatcher is responsible for the execution of application tasks and messages as defined by the time-triggered schedule. In the case of a PE failure, the remaining active nodes will switch to a reconfiguration state where the execution of the affected task is halted. The system recomputes the application graph, and the AHS starts the process of hormone exchange to determine task allocation based on the newly computed graph. The system will return to a normal state when all the affected tasks from the PE failure have been reallocated, and new schedules are generated. To

validate our proposed solution, an environment consisting of network and computation simulations is constructed. In addition, a co-simulation controller has been developed to couple both simulations. The results obtained from the evaluation of the use cases show improved temporal predictability and fault containment of the system.

Chapter 1

INTRODUCTION

1.1 Motivation and Goal

Computer systems have become crucial in most of our daily work, and the most often used types of computers today are embedded ones. An embedded computer system can be described as a specialized computing system composed of a Central Processing Unit (CPU), memory, and input/output ports on a single chip. Their range of applications spans from simple systems in home appliances (e.g., wash machines, televisions) to safety-critical systems such as autonomous vehicles, medical devices, and avionics [1]. These large embedded computers with increasing complexity are becoming hard to maintain and control because of fast changes in the environment and the technology of the embedded systems. As a solution, autonomous computing systems have emerged due to their ability to handle these changes autonomously.

International Business Machines (IBM) defines systems as *autonomous* based on the so-called self-x properties that these systems exhibit [2]. Self-x properties are inspired by biological organisms and include self-organization, self-configuration, self-healing, and self-optimization. Organic computing is a paradigm for organizing distributed computer systems with a high degree of flexibility and self-healing, which is inspired by the concepts and principles of biological systems. It adapts the working principles of organic systems to manifest their self-organizing nature into complex embedded systems. OC may contain sensors and actuators for perceiving, manipulating, and adapting to the conditions of the environment [3]. The development of such systems can be divided into three domains [4]:

- The functional domain where the productive system resides.
- The organic and self-x infrastructure domain.
- The observer/controller domain.

A simple implementation of self-x techniques, such as self-organization in OC, is achieved by the observer/controller architecture. The observing part monitors the state

of the system, and the controller, based on the current state, influences the system whenever necessary to achieve the desired output. Another approach relies on the concept of so-called ADNA to build embedded systems [5] and AHS to organize them. ADNA models embedded applications by encoding and storing the structure as well as the organization of the system in a single computer file. In the ADNA file, basic elements function as building blocks for embedded systems. They represent functional elements like filters, sensors, arithmetic/logic units, or actuators, and embedded systems can be created by combining them (e.g., control loops). AHS [6] is a real-time OC middleware that realizes a self-organizing mechanism by exchanging hormones (small messages) to perform application task allocations based on suitability levels. The Eager value hormone determines the suitability level for each task in relation to the PE. A PE is more suitable for executing a task if the suitability level is higher. For example, a Proportional–Integral–Derivative (PID) controller task will perform better on a computing node that contains arithmetic features like an Arithmetic Logic Unit (ALU). The middleware builds these systems by analyzing the ADNA file during run-time. This makes complex embedded systems more adaptable and robust.

However, these OC systems are not suitable for safety-critical applications because they lack support for determinism and dependability. Predictability implies that given an initial state at some time instant m along with a set of future timed inputs, the future states can be predicted along with the timings of all future outputs so that the system behaves deterministically [7]. Deterministic systems simplify formal safety and reliability models and are therefore suitable for safety-critical applications. These characteristics are found in distributed embedded computer systems that are built using time-triggered communication networks. In time-triggered systems, control signals are generated at specific points in time of a synchronized global time base, and the communication of messages ensures predictability and resource adequacy through the scheduled transmission of messages. By combining self-x features of ADNA and AHS with time-triggered concepts, OC becomes more deterministic, reliable, and dependable since the outcome of the system is predictable. Therefore, a time-triggered organic computing architecture is proposed in this thesis. In the proposed TTOC architecture, all operations, such as application task execution and message communication, are predefined through table-based schedules that are dynamically generated at each computing node using a list scheduling algorithm.

To validate this new concept, we have developed a simulation environment for the TTOC architecture [8]. In this process, task execution is simulated based on the generated schedule, with a time-triggered dispatcher and application tasks, as a SystemC simulation. Message communication between computing nodes is simulated as a time-triggered network using a network simulator. OMNeT++ and its INET framework offer the necessary tools for this, where Time Sensitive Networking (TSN) is chosen due to its support for time-triggered features. The TSN network must adapt dynam-

ically to the schedule calculation of the simulated processing element. Therefore, the INET framework has been modified, where control and data planes are extended with an interface for dynamic reconfiguration. SystemC and OMNeT++ are two discrete event simulations that are coupled by a co-simulation interface in order to perform a comprehensive simulation of the TTOC middleware architecture.

1.2 Contributions

Time-triggered communication, when applied in embedded systems, offers fault tolerance, error detection, predictability, and redundancy. Since communication times are predetermined, the system operates with deterministic behavior. Each data transmission occurs at precisely scheduled intervals, ensuring predictable timing. This is essential for real-time systems, where strict requirements for low latency and precise timing are critical to maintaining reliable and accurate performance. This thesis proposes combining time-triggered concepts with organic computing technologies of ADNA and artificial hormones. This new middleware architecture is called time-triggered organic computing. The system contains features such as self-description, self-building, self-organization, self-healing, self-configuration, clock synchronization, and cycled communication based on predefined schedules. The following papers contributed to this thesis and were submitted to international conferences related to computer science, distributed systems, autonomous systems, or communication networks. In publication [1], the architecture of organic computing triggered by time in combination with a simulation tool, in which the tasks of an application are executed according to a predefined schedule, is presented. In addition, the newly developed list scheduling algorithm is introduced in the research paper [2]. It is tailored according to TTOC requirements, where the spatial allocation of application tasks is handled by AHS middleware and the algorithm calculates the temporal allocation. For the application messages, both spatial and temporal decisions are made. The simulation environment of TTOC includes a network simulator where TSN is simulated. TSN obtains the communication schedule dynamically via a message queue that runs on a Docker container. This work was presented in the research paper [3]. To coordinate the execution of these simulations in order to obtain a comprehensive simulation environment, a co-simulation framework is presented in [4].

Publications

- [1] M. Qosja, S. Meckel, and R. Obermaisser, “Simulator for time-triggered organic computing”, *Procedia Computer Science*, vol. 220, pp. 127–134, 2023.
- [2] —, “Time-triggered organic computing architecture for autonomous driving vehicles using list scheduling”, in *AmEC 2024–Automotive meets Electronics & Control; 14. GMM Symposium*, VDE, 2024, pp. 73–78.
- [3] M. Qosja, U. Raj, S. Meckel, and R. Obermaisser, “Dynamic tsn reconfiguration for time-triggered organic computing”, *Procedia Computer Science*, vol. 257, pp. 364–373, 2025.

- [4] M. Qosja, S. Meckel, and R. Obermaisser, “Discrete-event co-simulation interface for time-triggered organic computing”, in *2024 IEEE 12th International Conference on Intelligent Systems (IS)*, IEEE, 2024, pp. 1–7.

1.3 Organization

The remaining chapters of this thesis are organized as follows:

- In Chapter 2, a review of existing related works is done, which is focused on organic computing and time-triggered communication.
- In Chapter 3, a general overview of the basic concept of autonomous systems, self-x properties, OC, ADNA, and AHS are given. In addition, time-triggered concepts are explained along with real-time systems, fault tolerance, communication networks, and scheduling algorithms. The end of the chapter explains discrete-event simulations.
- Chapter 4 explains the proposed architecture of time-triggered organic computing.
- Chapter 5 gives details on the simulation environment that is constructed to simulate TTOC.
- Chapter 6 explains the use cases implemented to validate the newly suggested architecture.
- In Chapter 7, the evaluation of the implementation has been performed, and the results obtained from the experiments are analyzed.
- Chapter 8 gives conclusions about the thesis and future work.

Chapter 2

STATE-OF-THE-ART

In this chapter, the current state of the art for organic computing is given alongside the analysis of the research gap. At the end, our contribution is highlighted.

2.1 Organic Computing

Self-organizing systems have been part of research in computer science and autonomic computing. With the increase in demand for applications, computer systems are becoming more and more complex. Having complex systems that interact and self-organize can lead to systems becoming unmanageable and outputting undesired behaviors. The solution proposed to tackle the undesired emergent behavior is the use of principles observed in nature to construct computer systems that imitate organic entities. Therefore, in 2003, the German National Science Foundation (DFG) established the research field of organic computing to adapt working principles of adaptive self-organizing systems into complex embedded systems [9]. The initial research work began on defining such systems. In publications [10], [11], [12], [13], [14], and [15] the vision of organic computing has been laid down along with a variety of research topics. Proposed topics include adaptability, reconfigurability, and self-organization of such systems. The work in [16], [17], [18], [19], [20] continued laying down the principles of OC into real-world examples.

The architectural designs for the OC were inspired by looking into the examples in nature. Looking at the colonies of ants and bees, [21] tries to bring their self-organizational behavior into OC. A basic OC architecture is based on the observer/-controller proposed in [22], which aims for a centralized or distributed self-organization system that allows acceptable controls in case of unexpected behaviors. Another observer/controller architecture for OC system implementations is presented in [23], where the architecture is tested on a traffic light controller. In robotics, there is research to establish different organic computing self-x properties. Configuring and Adapting Robot (OSCAR) is introduced in [24] to test robot walking behaviors in different terrains. In [25] an Organic Robotic Control Architecture (ORCA) is presented with the aim

of developing a modular and self-organizing component-based software architecture for autonomous mobile robots. Another work [26], which is similar to ORCA, where each agent in a robotic swarm implements a multi-layered observer/controller architecture. This design allows for local and in unison, also in global optimisation of the swarm agents' behaviours.

Furthermore, research has also addressed middleware for self-x features such as self-configuration, self-optimization, and self-healing. In ubiquitous computing, a middleware based on organic computing principles was developed [27]. It is based on the observer/controller architecture with an additional automated planner that can combine multiple self-x features into one component. An organic real-time middleware is presented in [28]. This middleware is based on an AHS for task allocation [29] where the middleware can configure itself by allocating tasks to the most suitable processing element. In addition, artificial DNA builds embedded distributed systems inspired by biological computations. As DNA contains genetic instructions for the development, functioning, and growth of organisms, ADNA contains the structure and organization of the computer system [5]. The ADNA file is stored in each processing element, and the AHS processes the ADNA file at run-time for building the application. Significant research has been conducted to introduce self-organization and OC properties in the automotive domain.

An alternative middleware approach [30] dynamically configures automotive embedded systems by providing transparency and flexible platform-independent support for portability. Another research that proposes an autonomic middleware for automotive embedded systems that exhibits high flexibility and automatic runtime reconfigurations is presented in [31]. OC will handle the increasing complexity of embedded systems in the autonomous vehicle sector. The concept of ADNA and AHS can be successfully applied to an environment consisting of low-performance automotive microcontrollers interconnected by a classical Controller Area Network (CAN) bus [32]. The dynamic concepts of ADNA and AHS can also be used in AUTomotive Open System ARchitecture (AUTOSTAR) as shown in publication [33]. In addition, organic computing has also been used to improve the dependability of automotives as described in [34]. A vehicle simulator that evaluates a control unit for automotives based on OC was developed at [35]. The OC methods were implemented and evaluated in the CAR Learning to Act simulator (CARLA) with the purpose of increasing the reliability of vehicle functions [36]. In [37], the authors evaluate OC capabilities in a case study of a typical vehicle infotainment system based on the Media Oriented Systems Transport (MOST) technology. OC concepts have also been transferred into building drones. ADNA computer files contain the description of a highly dependable drone swarm provided by self-x properties [38].

Research has also been made to build Systems on Chip (SoC) based on organic computing characteristics. Autonomous SoC (ASoC) is an architecture presented in [39]

that is established on OC self-correction properties. In addition, for the SoC, a Network on Chip (NoC) is developed based on the OC requirements [40]. OC principles are transferred to different technologies. For the traffic management, a system based on the observation/control of OC has been implemented in [41], [42]. Self-organizing sensors and actuators have been developed in [43].

2.2 Time-Triggered Communication

Dependable distributed real-time embedded systems can be built utilizing the Time-Triggered Architecture (TTA). The global time base specifies the interface between computing processing elements, performs error-checking, agreement protocols, and guarantees the real-time requirements [44]. The architecture has been integrated into different safety-critical systems like automotive, railway, and aerospace. In [45], TTA is proposed in the transportation systems. In automotive systems, the Time-Triggered Protocol (TTP), was developed by the Vienna University of Technology and the Daimler-Benz Research [46]. In addition, TTA principles are used in modeling a (SoC) [47]. As a middleware in [48], where a time-triggered message-triggered object structuring scheme was developed. TTA design has inspired and has been adapted also in many communication networks. Time-triggered Ethernet has been presented in [49]. Time control of messages has also been implemented for CAN in [50], FlexRay [51]. TTA served as a foundation for the TSN and its standards [52].

TSN is composed of Ethernet standards with the goal of improving determinism and packet delivery. Its aim is to provide high availability and low predictable transmission latency over different real-time applications (e.g., audio/video, automotive, industrial control) [53]. In TT communication networks, computing automatically appropriate preruntime schedules remains a challenge for this architecture [54]. For TSN, the challenge relies on computing table schedules that have coordinated transmission times for all TSN streams as well as Gate Control Lists (GCL) of all TSN bridges in order to meet the request real-time requirements [55].

Scheduling in TSN defines computing GCL entries and appropriate cycle times. Different scheduling algorithms that plan the GCL entries have been used. In [56], the authors have defined the scheduling constraints and then computed the table schedules utilising Satisfiability Modulo Theories (SMT) and Optimisation Modulo Theories (OMT) solvers. Authors at [57] have proposed an Integer Linear Programming (ILP) formulation to tackle the scheduling problem of TT flows such that all frames are schedulable and to minimize worst-case end-to-end delay of the Audio Video Bridging (AVB). They are also determining the routing of TT and AVB streams. A genetic algorithm was developed in [58] that combines routing and scheduling constraints to generate static global schedules using joint constraints in a single step. Another genetic algorithm has been used by [59] to optimize schedules in TSN. TSN traffic shaping mechanisms and network configuration problems are analysed by [60] in order to provide the desired Quality of Service (QoS). Earliest Deadline First-based scheduling in TSN to provide support for event-driven real-time traffic [61]. A protection band that allows rapid transmission of emergency event traffic without increasing the effect on the scheduled traffic is developed at [62]. This work was evaluated in the OMNeT++ network simulator.

2.3 Research Gap

Embedded systems composed of safety-critical applications (e.g., autonomous vehicles, aviation, railways) must provide ultra-high reliability with 10^{-9} failures per hour and strict real-time performance. These systems are also called ultra-dependable because failure to provide services can lead to catastrophic events. Key features include determinism and dependability. Determinism refers to the ability of the system to be predictable by calculating its output based on the given input. It is important to guarantee that application tasks will be executed at a specified time because missing a deadline leads to system failure. Dependability includes safety, availability, fault tolerance, reliability, and maintainability.

Analyzing related work, it is noticeable that efforts have been made to make OC safe and reliable, which would make it suitable for ultra-dependable systems [63]–[65]. In [66], a new framework is proposed to adapt formal compositional analysis techniques for realizing self-x properties in embedded systems in relation to real-time properties such as latency constraints or buffer sizes. Publications [67]–[69] analyze the safety of self-adaptive systems utilizing formal methods. The authors show how to model adaptive systems using formal methods in order to verify functional correctness under adaptation. OC systems can be modeled using SCADE [70], which is widely used in safety-critical applications. In addition, a technique for formal specification and compositional verification of self-adaptive systems is presented in [71], [72]. This technique separates the self-x and functional behaviors. OC phases and productive phases are also split in [73]. This separation allows for a better description of how organic functionalities can be specified and implemented. In [74], a set of dependency measures is selected based on stochastic measures to detect mutual influences in OC. A fault injection framework for OC ADNA/AHS-based systems has been developed at [75]. This framework analyzes the effects of different fault cases that may occur in the AHS. A fault may be, for example, an incorrect hormone value, which can lead to wrong application task allocation. The proposed solution is the integration of a stress hormone within AHS, focusing on fault diagnosis [76]. A self-configuration algorithm which includes a fault handling mechanism was developed at [77]. It enables the system to provide services even in the presence of faults.

Efforts have also been made to introduce self-adaptation in the time-triggered domain. In [78], authors have described that self-adaptation in time-triggered systems derives its inspiration from efficient energy utilization. Devices powered by batteries usually have a limited supply and hence must be energy efficient. Dynamic Voltage and Frequency Scaling (DVFS) along with clock gating are popular choices for energy-efficient computing. Adaptation in time-triggered systems must ensure that no new faults are introduced in the system due to the newly calculated schedule. The duration of adaptation, which includes calculation of the new schedule and its transmission to all

other processing elements, must take place within a known fixed time interval. At the same time, it is crucial that all the nodes in the system switch to the newly calculated task schedule at the same time to avoid consistency problems where some PEs execute the old schedule, and others run the new one. This will also lead to non-deterministic behaviour where one can no longer predict the system's state due to different schedules being executed on different PEs.

To tackle these challenges, the Adaptive Time-Triggered Multi-Core Architecture (ATMA) was developed. This new architecture introduces adaptation capabilities in time-triggered systems. The architecture precomputes different schedules corresponding to different faults that can arise in the system. This helps in avoiding runtime overhead since the schedules are precomputed, the resources can be used to execute the critical tasks as per the new schedule, and there will be no need to dedicate resources at run time for new schedule computation. The advanced calculation of the schedules also helps in maintaining synchronous operation and preventing deadlocks in the system. Another benefit is that timing constraints will not be violated as the schedules are pre-computed. Following the same principles, different approaches for the meta-scheduling are followed [79]–[82]. They include a genetic algorithm or artificial intelligence for the calculation of schedules.

Fault detection and recovery are other important aspects of adaptation. Adaptation is mostly needed in a system when a fault occurs. The system should recognize the fault and invoke fault recovery mechanisms to ensure quality of service is maintained. In [83], the authors have developed a resource management system for mixed criticality applications whose main objective is to avoid the occurrence of faults in the system. The system can be reconfigured by allowing it to dynamically change between schedules at runtime based on varying resource requests. A resource monitor provides information on the resource requests, and the system makes a switch between schedules based on the information provided.

To construct an adaptive system composed of time-triggered (TT) concepts, the reconfiguration of the communication network must be considered. This relates to the dynamic reconfiguration of existing schedules. In [84], for subsystems, there were local schedules defined, which were then integrated into the global one. This approach reduces complexity by applying divide and conquer principles. For TSN networks, reconfiguration includes removing old streams and inserting new streams under guaranteed timeliness [85]. Adding new streams to an existing schedule can be complicated for two reasons. First, the transmission queues may be occupied from the previous streams, and second, the transmission offset must change. In [86], the authors have analyzed different heuristic dynamic scheduling for the TSN. A heuristic algorithm that determines transmission gate entries during runtime in order to satisfy the deadlines [87]. This approach minimizes the queue usage to handle non-critical stream traffic. The self-configuration property has been transferred to the real-time networks. In [88], the

authors have proposed a configuration agent that monitors the network and modifies the configuration to adapt it to the changes that have occurred.

A similar approach has been followed by [89], where a central entity monitors the application running to detect changes in the topology. This central entity utilizes Network Configuration Protocol (NETCONF) to reconfigure TSN devices in case of faults. After the reconfiguration, TSN streams can be resumed. In addition, the dynamic scheduling proposed in [90] combines the score value, where each value is dedicated to a single path as a measure for the schedule and incoming flow in multiple redundant paths at runtime. It is important to ensure high reliability for the safety-critical streams. The dynamic scheduling is improved by taking into consideration the source node in the stream path [91]. A Priority-based Dynamic Slot Extension (PDSE) is developed at [92] for handling the high-criticality streams coming from the application tasks. Algorithms that incrementally modify TSN based on the Time-Aware Shaper (TAS), which introduces network configuration during runtime [93]. Scheduling time-critical traffic poses a challenge in Wireless TSN. Dynamic traffic classification can give faster access to the high-priority flows in the wireless medium [94].

2.4 Requirements / Challenges

Organic computing concepts and time-triggered communication have fundamental contradictions between them. OC self-x properties do not go down well with time-triggered systems. This is due to numerous reasons, such as the ability of OC to self-organize, which makes them more flexible and independent of the engineer/programmer. On the other hand, the nature of TT systems is rigid. They operate on fixed, predetermined schedules and are therefore predictable, leading to little room for flexibility. To reconfigure such systems, dynamic schedules are needed. The reconfiguration in time triggered systems will require a new schedule calculation and ensure that the new schedule is made available to all the nodes in the system. Failing to do so will lead to inconsistency in the system, where different nodes and switches will operate on different schedules. Furthermore, the reconfiguration must be done in a way that no task misses its deadline. This is crucial in safety-critical applications such as avionics and autonomous vehicles, where if a task misses its deadline, it could lead to catastrophic events. However, when combining these two technologies, one must take into consideration:

1. The self-x properties of OC.
2. Schedule calculation and injection.
3. Timing requirements of the application.

Therefore, there is still a research gap in integrating organic computing into safety-critical applications, especially in those composed of time-triggered communication. This gap is mostly related to OC being deterministic and to self-x properties combined with table schedules. In the current state-of-the-art OC has no knowledge of the system's future output; they monitor the behavior and control the execution of application tasks in the operating system's threads. This is done to fulfill the ultra-dependable systems' hard tasks deadlines. Our approach is different; we introduce time-triggered concepts in OC to make the system predictable and deterministic. TTOC allows the system to be flexible while being under a degree of control. The description of the system will be encoded in the ADNA file, and the AHS middleware will perform spatial allocation of application tasks. Each PE will contain its own schedule, and the OC operations will have fixed slots. Hence, PEs will calculate schedules dynamically via the list algorithm. Overall, scheduling in TTOC must hold:

- A fixed known time for adaptation.
- All nodes of the system must switch to the same schedule at the same time without being inconsistent.
- No compromise on the original objective of the system.
- Nodes that are not a part of the adaptation process should remain unaffected.

2.5 Thesis contribution

Contributions of the thesis include:

- Incorporating the time-triggered concepts and organic computing into a new architecture TTOC.
- Design and implementation of a scheduling algorithm to calculate and fill the table schedules.
- Runtime reconfiguration of time-triggered schedules based on OC hormones.
- Dynamic reconfiguration of TSN in OMNeT++ network simulator.
- Building a simulation environment for the new architecture.
- Modification of the time-triggered network TSN in the network simulator called OMNeT++ to support dynamic schedule injection.
- Designing a communication controller for co-simulating the simulators within the simulation environment .

Chapter 3

BASIC CONCEPTS AND TECHNOLOGIES

3.1 Organic Computing

Recent advances in technology have introduced new challenges in distributed computer systems, mainly related to the increase in complexity that is approaching the limit of human capabilities. To address these challenges, we need systems that can manage themselves dynamically (self-organisation). Examples of these entities can be found in the biological world, such as in human bodies, where the autonomic nervous system self-manages many of our vital functions (e.g., heart rate, temperature) separately from the brain [95]. Organic computing is one of the research fields based on these principles that aims to enable complex autonomous systems to self-organize without external control mechanisms. The term organic refers to a computing system composed of actuators and sensors that has the ability to adapt to the changing environment during system runtime. Sensors have the capacity to provide information about the environment, while the actuators can manipulate it. Different from other autonomous computing systems, OC does not intend to create fully autonomous systems whose behaviour can be unpredictable, but rather a controlled self-organisation. Systems that have an output that cannot be anticipated or controlled can lead to catastrophic consequences both to the environment and humans. On the other hand, organic computing must allow a certain degree of freedom needed by the self-x properties. Therefore, each OC system must contain two parts:

- One part that realizes the self-x mechanism.
- One that handles basic functionalities (e.g., execution of application tasks).

An initiative for organic computing was taken up by the German National Science Foundation DFG under the Priority Program 1183 that focused on three phases, which included:

1. Phase one relates to understanding and analyzing the concept of self-organisation in order to develop the basic concepts needed to design organic computing systems.
2. A second stage where organic computing technologies are equipped with self-x properties.
3. In the last stage, organic computing knowledge was used for developing applications.

Self-x Properties of Organic Systems

IBM defines systems as *autonomous* based on the so-called self-x properties that they exhibit [2]. The autonomy of organic computing systems relies on the term self because the x features (e.g., self-organisation) represent the degree of freedom of the system. The level of autonomy can be described as follows:

- No autonomy at all, control is in the hands of engineers/programmers.
- Intermediate autonomy where a combination of self-x properties with external control is present.
- Fully autonomous, where no external control is present (As stated, full autonomy is not desired due to unpredictable outcomes).

Some of the organic computing self-x properties include features like:

- Self-Organisation: The organic computing systems need to adapt their structure and functionalities if there are significant changes in the environment. Part of these changes can also be other elements of the system that must be modified to achieve stability again.
- Self-Configuration: Also known as self-adaptation, gives the system the ability to modify certain parameters.
- Self-Stabilizing: One of the challenges of OC is to perform continuous adaptation in case of environment changes, to achieve a stable behaviour.
- Self-Protecting: In case of attacks, the system should protect itself but also offer protection to the higher system it belongs to.
- Self-Healing: In case of different failures, the system should detect, diagnose, and repair itself. This limits the interference from outside engineers.
- Self-Describing: It refers to the ability of organic computing to describe the structure and organization of the embedded system in a single computer file that is stored in each computing node.

- **Self-Building:** The system is built dynamically, meaning that all the information needed (e.g., application tasks, communication) is obtained from a single computer file.

3.1.1 Observer/Controller Architecture

As previously stated, OC responds to changes in the environment by adapting its internal organization and parameters. The system becomes more flexible and robust due to this adaptive behaviour. One of the emerging architectures based on these concepts is the Observer/Controller [96], as shown in Figure 3.1. The architecture is centralised, but it can also be fully distributed. For large and complex systems, it is difficult for one Observer/Controller component to handle the increasing complexity. The design rep-

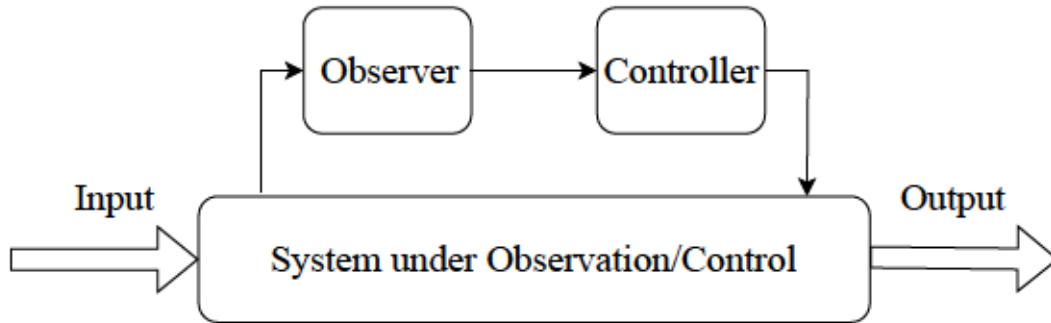


Figure 3.1: Central OC Observer/Controller architecture

resents a feedback mechanism that observes and controls the system in an autonomous manner. The main components are the following:

1. System under Observation and Control (SuOC)

SuOC is the lower layer of the architecture, and it is separated from the higher level (e.g., self-x mechanism). It represents the part of the OC system that is under monitoring and performs specific computations as designed by the user. In case of failure of the self-x mechanism, the lower layer can continue to operate. However, SuOC has to meet some application requirements for it to be compatible with the Observer/Controller design. The environment and the behaviour should be measurable by the Observer as well as the output of its computation. In addition, SuOC needs to be composed of parameters that allow a dynamic modification/adaptation that will be performed by the Controller.

2. Observer

In the OC system based on the architecture of Observer/Controller, the Observer's duty is to check on SuOC state, utilizing sensors, to analyze it, and to send this data to the Controller. The Observer's aim is to understand the SuOC state phenomenon and to foresee the future outcomes. This is performed by

an observation model which selects the necessary attributes that are observable, chooses the suitable tool for analysis and prediction of the output.

3. Controller

After the Observer provides information about SuOC state, the Controller modifies SuOC parameters to achieve the desired organic computing system behaviour. As can be seen in Figure 3.1, a feedback mechanism is created. The Controller's manipulation of SuOC behaviour has three possibilities: a) to manipulate parameters in order to achieve the desired behaviour, b) to end SuOC behaviour which is not wanted, and c) to build a system where in the future no undesired behaviours will emerge.

3.1.2 Artificial DNA

Another organic computing technology that follows the same biological principles is the artificial DNA. It is based on the concepts of biological DNA, where the DNA carries the genetic information for the growth and functioning of all organisms. In the same approach, we can construct embedded systems by putting the overall description of the system (structural and organizational) in an executable computer file (ADNA file) located in each processing core [97]. In this manner, the system can be self-describing

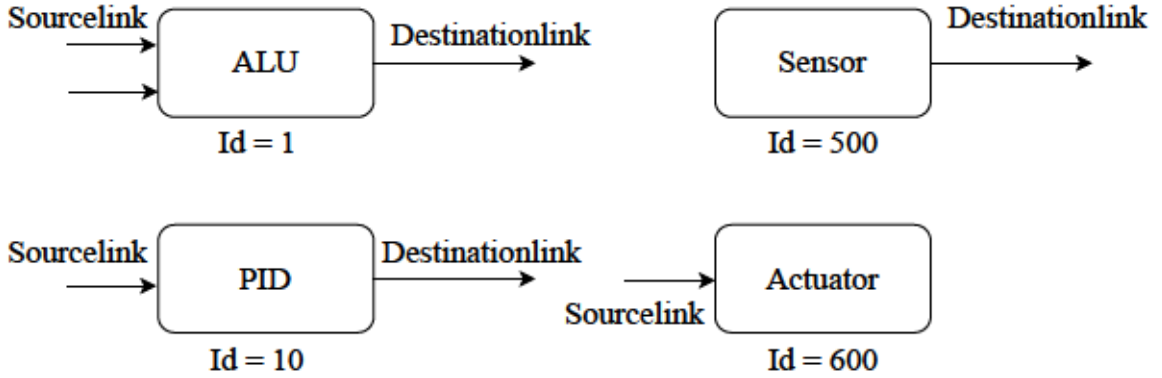


Figure 3.2: Basic element

since all the necessary functionalities (e.g., task structures, messages) will be extracted from the ADNA file.

If a sufficient set of standardized basic elements with unique Ids is available, an embedded system will no longer be programmed, but composed by connecting and parametrizing these elements. Table 3.1 gives an extract of such a set. Overall, hundreds of these elements should be adequate to compose many kinds of embedded real-time systems. Each line of ADNA corresponds to a basic element and holds an ID, plus the connections to other basic elements. The *Sourcelink* denotes a reactive link

ADNA		
ID	Destinationlink	Parameter
10	1:2.1	PID Controller
30	1:2.2	FIR Filter
70	1:2.1	Constant
500	1:2.1	Temperature Sensor
602	1:2.1	Motor

Table 3.1: ADNA description of an Embedded System

responding to incoming requests, and the *Destinationlink* is an active link for sending requests. Figure 3.2 gives details about basic element construction. ADNA relies on defining functional elements such as filters, sensors, or actuators as basic elements distinguished by an ID. From a set of these building blocks, numerous embedded system models can be designed by combining these elements (e.g., control loops).

3.1.3 Artificial Hormone System (AHS)

With the increasing complexity, we need systems that are able to organize and configure autonomously. In distributed systems, middleware is the layer that hides the details of network programming from the application and manages the collaboration between tasks. That is why middleware is convenient to realize the self-x properties of OC. In [28], a real-time organic computing middleware is developed for realizing spatial allocation of application tasks. This new design, called, AHS follows the principles of the hormone system in animals. Similar to hormones in biology that are chemical messages to produce certain effects, AHS utilizes artificial messages to perform self-x properties. In addition, AHS has the purpose of reading the ADNA computer file and building the system functionalities. In this way, the system is self-built during run-time. Other self-x properties include:

1. **Self-Organising** because the task allocation is decided by AHS itself.
2. **Self-Configuration** since for allocation, the conditions of the computing node are taken into consideration.
3. **Self-Healing** in case of task or node failures, the system reallocates the affected application tasks to the remaining processing units.
4. **Self-Optimization** is achieved because tasks are allocated to the most suitable processing element.

5. **Decentralized** due to zero influence from the outside world on the allocation decision.
6. **Real-Time** capabilities due to upper time bounds for self-configuration.

After the system functionalities are created, AHS starts the procedure to allocate application tasks. Each task contains three main hormone types: *Suppressors* for lowering suitability, *Accelerators* for increasing suitability, and *Eager values* for determining the suitability level of a task in relation to the PE. This procedure is called the hormone loop, and it is executed in every processing element. It will sum up all the accelerators, suppressors, and local eager values received and compare them with the received eager values, as seen in Figure 3.3. AHS will allocate the tasks to specific processing elements

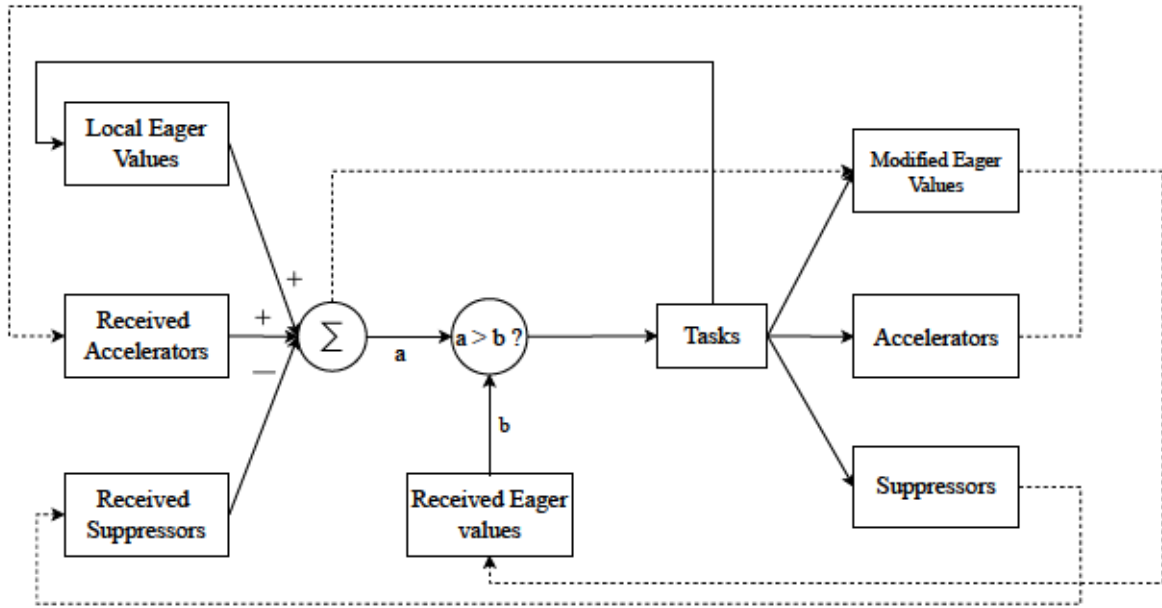


Figure 3.3: Hormone Loop of AHS

based on their suitability, configuring in this way the system (self-configuring). In the case of a control loop, a task from the PID controller will have higher suitability on a PE that performs better the arithmetic calculations. Another feature of AHS relies on task distribution to minimize communication distances (self-optimizing). This procedure is repeated in every PE for all the application tasks in the system. The advantages of ADNA are based on the properties of self-x, such as self-organisation, self-building, self-optimization, and self-healing. In ADNA, the system is self-organized because the task distribution is done internally by the exchange of hormones between nodes, taking into consideration the suitability of tasks, but also the load of a node. In this case, the system becomes more optimized since PEs are prevented from high load. Furthermore, the whole system is built using simple artificial DNA files, and in case of different faults like PE failure, the system is reconfigured, and all the tasks are migrated to other processing elements.

3.2 Embedded Real-Time Systems

An embedded computer is a dedicated computing system integrated within large electronic systems to perform specific tasks [98]. They are composed of a computing unit, input/output, and sensors/actuators. Embedded systems have become part of our daily lives; their range of applications spans from home appliances to industrial automation, to safety-critical transportation systems such as avionics or autonomous driving vehicles. Safety-critical real-time applications must perform computation and deliver the outcomes in a specific period of time called a deadline. Such applications are also known as hard real-time systems. A computer system whose behaviour depends on the logical results of its computation and on the instant that these computations are made, is called a real-time system [99]. Time is modeled as a directed line from the past to the future, and the sequence of outputs in time represents the behaviour of the system. The changes of state in such systems occur as a function of physical time. A real-time system is composed of three parts, as is shown in Figure 3.4. First, the real-time computer that is responsible for all computations, the controlled object, and the human operator. The operator and the controlled object are known as the environment of the real-time computer system.

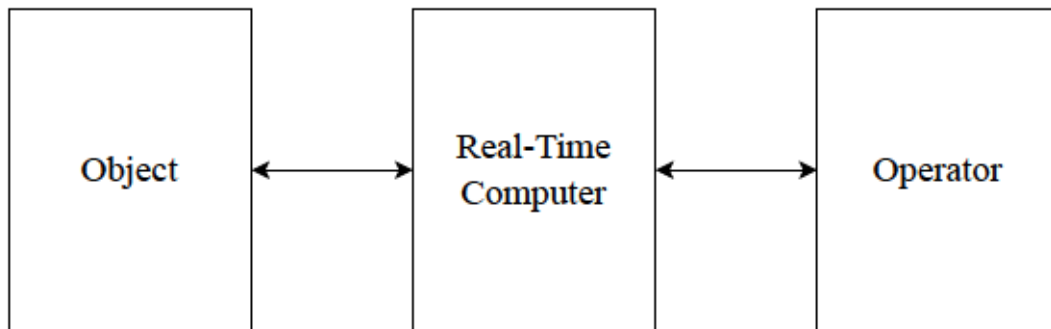


Figure 3.4: Real-Time System

Real-Time Applications

A real-time application is composed of an embedded computer system and a controlled object. The computer system interacts with the controlled object using sensors and actuators. The interactions include certain transformations performed by the sensor or actuator on the physical variable of the controlled object or digital computations. This system is called a cyber-physical or embedded real-time system. As stated, real-time systems must deliver output of computations within a certain period. There are three types of deadlines [100]:

- **Soft** when some delays are allowed in meeting the deadline and missing the output period does not have a big impact on the application or the environment.

- **Firm** when the result has no utility anymore after the deadline is missed.
- **Hard** when the application must produce the result within the deadline. In this case, missing the deadline could cause a catastrophe.

In the case of safety-critical applications, the computer system must meet at least one hard deadline. In this thesis, we are focusing on embedded real-time systems that execute safety-critical applications.

Scheduling Algorithms

In real-time systems, to ensure that all packages will meet their deadlines, real-time traffic scheduling must be performed. The scheduling algorithm determines which tasks are going to be processed by the scheduling system. The real-time scheduling system has three main components: a scheduler, a clock, and hardware elements. The scheduler uses a scheduling algorithm for the system in two forms: preemptive or non-preemptive. In the preemptive form, if a task with a lower priority is blocked by one with a higher priority, the process with lower priority will be terminated, allowing the execution of the higher one. In the non-preemptive case, the process does not terminate, but it waits until the execution time is complete. There are two types of scheduling algorithms, static and dynamic [101], as can be seen from Figure 3.5. In the static case, scheduling

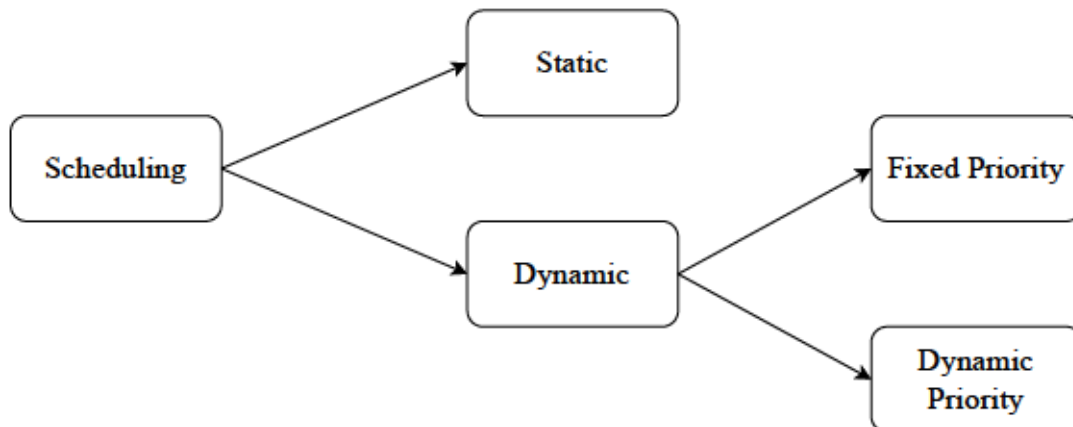


Figure 3.5: Scheduling hierarchy

is done before the program starts, and the system has a list that includes the times of each job that needs to be executed. In dynamic scheduling, everything is done during the runtime of the system, and it is divided into two classes: fixed priority and dynamic priority. The notion of a clock in a real-time system is very important for real-time traffic scheduling as the data packets have deadlines. For each task in the scheduling system, deadlines and priorities are determined by the scheduling algorithm. In the hardware elements, tasks are allocated, and they are processed in real time. On the

other hand, scheduling increases reliability by ensuring that tasks are executed in a way that optimizes the system resources. The reliability is also increased by using redundancy, where the same message will be sent by multiple nodes, increasing the probability of a successful transmission. In addition, scheduling makes it possible to introduce redundancy by transmitting the same information packet over several time slots in order to increase the chance for the packet to reach its destination correctly and before its deadline [102]. Table 3.2 displays some of the most used scheduling algorithms for real-time systems.

Real-Time Scheduling Algorithms		
Algorithm	Type	Description
Rate Monotonic Scheduling (RMS)	Fixed Priority	Shorter period higher priority
Earliest Deadline First (EDF)	Dynamic Priority	Tasks prioritized based on earliest deadline
Least Laxity First (LLF)	Dynamic Priority	Priority given to the earliest slack time
Fixed Priority Preemptive (FPP)	Fixed Priority	Priority of tasks fixed offline
Deadline Monotonic (DM)	Fixed Priority	Shorter relative deadlines higher priority
Minimal Laxity First (MLF)	Dynamic Priority	Task with smallest remaining slack is executed
Slack Stealing	Dynamic Priority	Lower priority task receives spare time
List Scheduling	Mixed Priority	Tasks sorted based on their dependencies

Table 3.2: Real-Time Scheduling Algorithms

List Scheduling

List scheduling is a process used to schedule tasks represented as a Directed Acyclic Graph (DAG). For example, there is a DAG consisting of six application tasks and their Worst Case Execution Times (WCET), alongside message sizes in Figure 3.6. The tasks are arranged based on their depth in the graph and the time required for their execution. Each task is represented by a node and the dependencies between them are represented by edges in the graph. The scheduling algorithm determines the order in which the tasks should be executed and aims to assign each task to an available processing element [103]. It takes into consideration data dependencies, timing constraints, system model, critical path, scheduling priorities, and optimization of factors, such as load balancing,

together with minimizing idle time. In this thesis, an extension of the list scheduling is

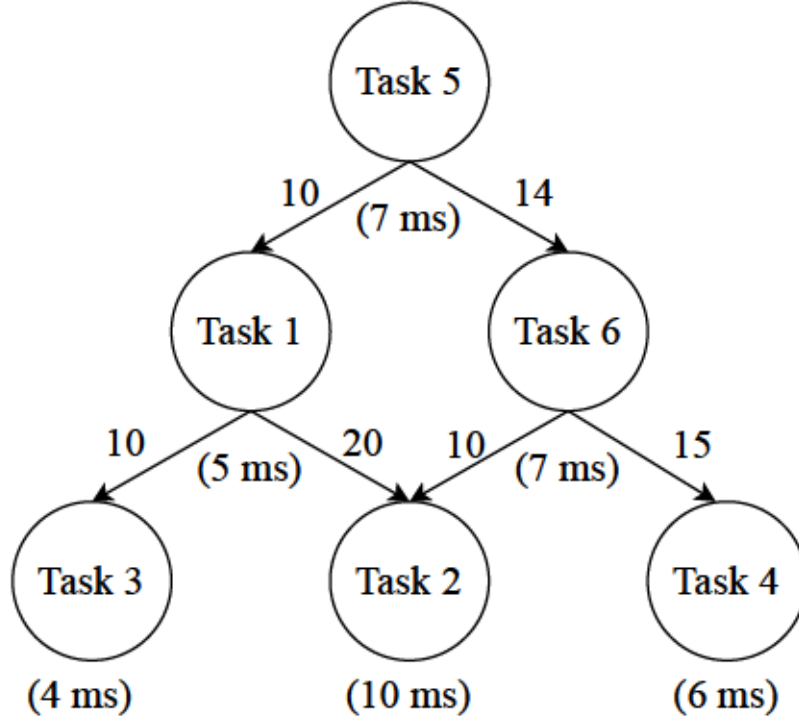


Figure 3.6: Example of DAG graph

introduced. The algorithm calculates dynamically temporal allocations of application tasks, as well as allocations of messages. A detailed explanation of the algorithm is given in section 4.3.1.

3.2.1 Requirements

To construct embedded real-time systems, requirements must be taken into consideration, as shown in Table 3.3. Functional requirements monitor and control how the system performs. Data collection observes the real-time entities of the system, where an entity represents a state variable. In a computer system, each observation of an entity is given as a real-time image. Direct digital control calculates the variable for the actuator in order to directly control the object, while a man-machine interface communicates to the operator the current state of the system. Real-time systems that contain control loops have strict temporal requirements. The computer delay is the time that processing elements wait before continuing program execution. Meanwhile, the difference between the maximum and the minimum delay is called jitter of computer delay [104]. The time in between the observation of the real-time entity and the

Embedded Systems Requirements		
Functional	Temporal	Dependability
Data Collection	Minimal Latency Jitter	Safety
Direct Digital Control	Minimal Error-Detection Latency	Reliability
Man-Machine Interaction		Security
		Maintainability
		Availability

Table 3.3: Embedded Systems Requirements

controlled object response of computer activity is called dead time. Latency jitter occurs when the time interval of the events diverges from the expected one. Providing a predictable and consistent timing, minimizing the latency jitter is very important [105]. In safety-critical applications, the detection of errors within the control system must be in a short time and with a high probability. The response of the computer system to the error is possible if the error detection latency is in the same order of magnitude as the sampling period.

The dependability of a computer system relates to the quality of service that it can deliver during an interval of time. Its characteristics include:

- **Safety**

Safety of a real-time system is reliability with reference to critical failures [106]. Two types of critical failures exist: a) malign, where the failure is critical to the system, and b) benign, when the failure is noncritical. Most of the safety-critical applications must have a failure rate in the order of 10^{-9} failures/hour, which is considered ultra-high reliability.

- **Reliability**

The probability that the system provides the intended services from the time it was operational until a given time t is denoted as the reliability $R(t)$ of that system [107]. The function to calculate the reliability of the system in a constant failure rate per hour λ is shown in equation 3.1.

$$R(t) = \exp(-\lambda(t - t_0)) \quad (3.1)$$

Mean-Time-To-Failure (MTTF) $1/\lambda$ is the inverse of the failure rate.

- **Security**

It represents the ability of a system to protect services against unwanted access.

- **Maintainability**

In case of noncritical failures, when the system must be repaired, the time required for the system to become fully operational is called maintainability. This time is measured in Mean-Time to Repair (MTTR).

- **Availability**

We can measure the alternation of correct and incorrect service of a system. It is called availability, and it is measured as a fraction of time that the system delivers a correct service. The relation between availability, maintainability and reliability is given in equation 3.2 where the sum of MTTF and MTTR is known as Mean Time Between Failures (MTBF).

$$A = MTTF / (MTTF + MTTR) \quad (3.2)$$

3.2.2 Classification

In addition to the first classification of real-time systems, which we explained as soft and hard, more classifications can be performed based on application characteristics. These classifications include: fail-safe versus fail-operational; guaranteed timeliness against best effort; resource adequate over resource inadequate; and event-triggered versus time-triggered.

Resource Adequate and Resource Inadequate

Safety-critical applications that provide services within a certain deadline are based on resource adequacy. This means that for every fault scenario and task load, there are sufficient resources available. For non-safety-critical applications, resource inadequacy is commonly used since it is more economically feasible.

Guaranteed Timeliness and Best Effort

Guaranteed timeliness relates to the concept of designing a system that offers a guaranteed response, where analysis and planning are needed.

If such an analytic guarantee cannot be given, we speak of a best-effort design. Best-effort systems do not require a rigorous specification of the load and fault-hypothesis. The design proceeds according to the principle of best possible effort taken, and the sufficiency of the design is established during the test and the integration phases. It is difficult to establish that a best-effort design operates correctly in a rare-event scenario. At present, many non-safety-critical real-time systems are designed according to the best-effort paradigm.

Fail-Safe and Fail-Operational

For safety-critical applications, if a safe state can be accessed in case of a failure, the system is denoted as a fail-safe. For such systems, the error-detection must be high. The operations of the computer systems are monitored by an external device, which typically receives periodic information. In case this information is not provided within a specified time, the controlled object enters a safe state. In some applications, a safe state cannot be identified for the controlled object. These systems must provide services and stay operational in case of a failure (fail-operational).

Time-Triggered and Event-Triggered

In computer systems, the execution of a task or the transmission of a message is caused by an event. This event is a trigger signal. In real-time applications, two trigger mechanisms are present, Event-Triggered (ET) and (TT). Computer systems that are built upon event-triggered control can perform actions based on all events that occur and that are different from the event of a clock tick. The interrupt mechanism is responsible for notifying the processing units that events must be executed. The scheduling strategy for ET systems is dynamic. On the other hand, in TT systems, time is in control of the task execution and message transmission [108]. Each application task or message communication starts periodically at a tick of the clock. The interrupt mechanism used in TT is the periodic real-time clock interrupt. If the architecture of the embedded real-time system is distributed, computing nodes synchronize their clocks to form a global time base. Each control event in the distributed system occurs at the tick of the global time base.

To implement a dependable real-time application that is fault-tolerant and tolerates errors, its functions must be distributed. A distributed real-time computer system is composed of a set of computing nodes interconnected via a real-time communication network.

3.2.3 Distributed Architecture

The term distributed refers to computing elements (nodes) which are software plus hardware and that have the ability to operate independently and in coordination with each other [109]. To the application or to the user, these systems appear as a single entity. The coordination means that computing elements have to collaborate, and this is achieved via message communication that happens over a communication network, as can be seen in Figure 3.7. Types of applications in distributed systems include client-server, peer-to-peer, real-time, Internet of Things (IoT), and distributed databases. Key characteristics of distributed application systems are:

- Concurrency

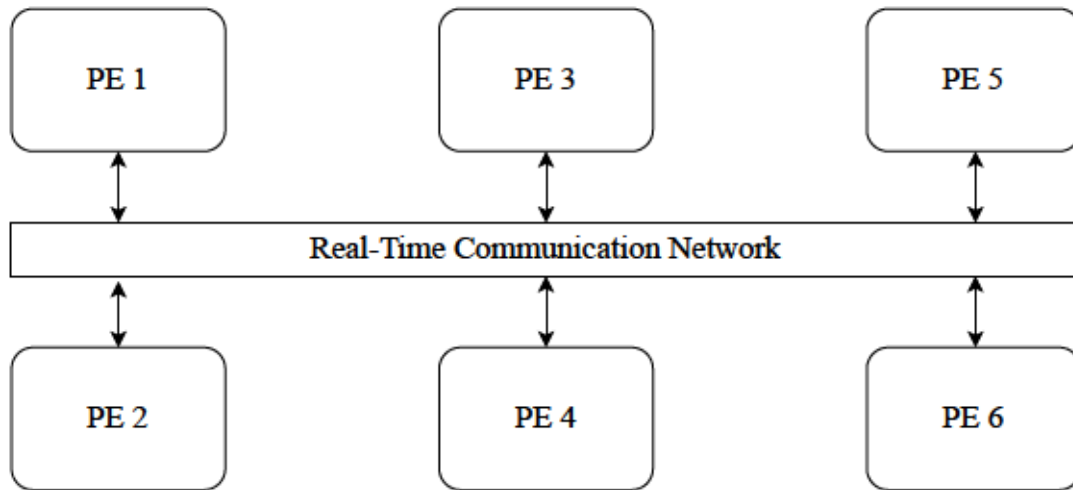


Figure 3.7: Distributed Computer System

It refers to the ability of nodes to perform task execution in parallel, thereby improving the performance and scalability of the system.

- **Fault Tolerance**
Distributed systems should be robust against failures. Fault tolerance ensures that the system is operational in case of a node failure.
- **Scalability**
For improving the performance or to handle more operations, it is possible to add more nodes to the system.
- **Autonomy**
The nodes in the distributed system can operate independently.
- **Transparency**
Component distribution in the system is hidden from the user.

3.2.4 Global Time Base

In the behaviour of many real-time computer systems, events are processed in the temporal order, which can be ensured by a global time base and the time stamps of the events. The time stamp is obtained by assigning the state of the node at the time after the event has occurred. These specific points in time that are identified by a value of the global time base are called instants. They are used to establish the temporal order of the system. A duration is described as a section in the timeline which lies between two different instants [110]. Events occur simultaneously if they occur in the same instant. The causal order of two events helps to determine their order. It is important to know

the exact temporal order of the events in order to identify which one was the primary one.

- if event a1 occurs after a2 it cannot be the cause of event a2
- if event a1 occurs before a2, it is possible to be the cause of event a2

Clock

The clock is a particular device used to measure time using a counter with an oscillation mechanism that generates an event that increases the counter. Each of these periodic events is denoted as a microtick of the clock. Granularity describes the duration between two microticks that are consecutive.

$$drift_i^k = \frac{z(microtick_{i+1}^k) - z(microtick_i^k)}{n^k} \quad (3.3)$$

Equation 3.3 taken from [99] describes the clock drift where the drift of the physical clock k along microtick i and i+1 is the frequency ratio between clock k and the reference clock at the instant of microtick i. Good clocks have a drift rate close to 1, while a perfect clock will have the drift rate of 1. Equation 3.4 describes the notion of drift rate p_i^k .

$$p_i^k = \left| \frac{z(microtick_{i+1}^k) - z(microtick_i^k)}{n^k} - 1 \right| \quad (3.4)$$

Two clocks j and k, where the granularity is the same, have an offset as defined in Equation 3.5. The offset measures the time difference of microticks between two clocks, measured in the number of microticks of the reference clock.

$$offset_i^{jk} = |z(microtick_i^j) - z(microtick_i^k)| \quad (3.5)$$

In a number n of clocks, the maximum offset between two clocks is denoted as precision π at microtick i, as can be seen from Equation 3.6. The accuracy of a clock k is the maximum offset of this clock compared to a reference clock in a duration of interest.

$$\pi_i = max offset_i^{jk} \quad (3.6)$$

The offset of clock k with respect to the reference clock z at microtick i is called the accuracy $_i^k$. The maximum offset over all microticks i that is of interest is called the accuracy ^{k} of clock k. The accuracy denotes the maximum offset of a given clock from the external time reference during a duration of interest. The clocks in the distributed system must be synchronized to guarantee that global ticks occur with a certain precision, despite the local nodes' drift rate. In distributed real-time applications, a global time base is crucial for the operation of the system.

If all events in the set of events that will occur (e.g., message communication) can occur at any instant, this timeline is called dense. In case some events are not allowed to occur in some intervals of duration, then this is called sparse time.

3.2.5 Dependability

Computer systems are designed to provide dependable services. The basic concepts of dependability are: fault, error, and failure. If the behaviour of the system is not the same as intended, then the system has failed [111]. A failure occurs when the system's state enters an unwanted state due to an error. Any incorrect state in a component is called an error, which happens due to faults that occur in the internal or external part of the real-time system. A component is a Fault-Containment Unit (FCU) when the faults impact only that particular component. Table 3.4 shows the categories of faults that can affect the system. They are classified into two domains, spatial and temporal. Transient and permanent faults are based on their duration and time of occurrence.

Faults	
Time	Space
Transient	Hardware
Permanent	Software
	Environment
	Data

Table 3.4: Fault Types

Transient faults occur in a short interval and then disappear, while the permanent ones remain in the system until they are removed.

Errors in the system are activated in case a performed computation accesses it. Two types of software errors are Bohrbugs and Heisenbugs. Bohrbugs can be reproduced if an input pattern that contains the error data is given, while a Heisenbug changes its behaviour when being examined [112]. Errors are discovered when the results of the computations diverge from the intended design. The time between the start of an error and its detection is the error-detection latency, while error-detection coverage is the probability that the error will be detected. The deviation of the service from the intended behaviour at a particular instant is denoted as a failure. In Table 3.5, component failure types are given. Domain failures can be value and temporal. Value failures occur when the value presented is incorrect, while the temporal ones occur when the timing criteria are not met. Frequency failures are related to the rate at which these failures occur or if they are repeated over a particular period of time. Permanent failures happen only once, while transient ones are repeated. If the failing

Failures			
Domain	Frequency	View	Severity
Value	Transient	Consistent	Benign
Temporal	Permanent	Inconsistent	Malign

Table 3.5: Component Failure Types

behaviour is observed by all users, the failure is consistent, while if it is seen differently, it is called inconsistent. The severity of a failure can be benign or malignant.

Fault-Tolerance

The ability of a computer system to operate in the presence of a fault is called fault-tolerance [113]. A fault-tolerant computer system is based on a fault hypothesis. The hypothesis determines the faults that will be tolerated by the system. These faults are divided into normal and rare. Normal faults are covered by the hypothesis, while rare faults are outside of it. If a normal fault occurs, the fault-tolerance mechanism will bring the system to a correct state, while for the rare ones, the system state is not covered by the mechanism. The hypothesis will specify the FCUs that impact the reliability of the system. FCUs are grouped together to form fault-tolerant units (FTU). The FTU masks the failure of a single FCU. Triple Modular Redundancy (TMR) is one of the fault-masking methods. TMR masks failures by using triplication, the system executes three identical modules concurrently, and the correct output is decided by a voting mechanism [114]. Types of failures that can arise in a real-time system are:

- **Fail-Stop** represents a failure when a computing node does not produce outputs because its execution is stopped. This failure is easily detectable from system components.
- **Crash failures** happen when a system component (e.g., computing unit, sensor) suddenly halts operating. For the system, this failure is critical because it can lead to malfunctioning or catastrophic results.
- If an application task or message is not handled by the component, an **Omission failure** has occurred.
- Faulty components can take the identity of another component and endanger the real-time system security and stability. This failure is called **Masquerading**.
- In **Slightly-off-Specification (SOS)**, the system operates slightly outside the limits, but the results produced are not correct. This type of failure becomes problematic due to potential inconsistencies with the passage of time because small errors accumulate and lead to system malfunction.

- If the timing requirements are not met, even though the produced result may be correct in the value domain, a **Timing failure** has happened. Timing failures are critical for the system since they can lead to catastrophic failures.
- A **Byzantine failure** occurs when a system component provides incorrect, inconsistent, or malicious output. This failure is more difficult to detect since the current state of the component can produce misleading information.
- In case of **Babbling Idiot failure**, it is a subtype of Byzantine failure that occurs when messages are sent at incorrect times or excessive rates even though the message may be correct. The communication channel is flooded with messages, and the system becomes unstable.

3.3 Real-Time Communication

3.3.1 Communication Network

In a distributed system, computing elements exchange messages via a communication network [115]. A communication network is defined as the set of protocols together with processing elements that are interconnected between them and allow flow of messages. For a communication network to be real-time, it should fulfill these requirements: a) a global time base should be present, b) low latency and minimal jitter, c) temporal error containment, and d) fast error detection at the receiver. Messages are transported using the Basic Message Transport Service (BMTS) with a specified reliability and latency. Three types of communication networks are used in real-time applications: real-time network, field bus, and backbone network. For safety-critical applications, the real-time network is more suitable due to high reliability and guaranteed data delivery.

3.3.2 Real-Time Network

In a real-time application that requires strict timing constraints for data transmission, message exchange should be predictable and with a minimum delay [116]. A dependable real-time network covers these requirements:

- Predictable and reliable message transmission.
- Low latency and minimal jitter.
- Clock synchronization and global time base.
- Support for fault-tolerance.
- Failed node detection.

In dependable real-time networks, there is no central control because the failure of a single unit should not lead to a failure of the whole network. For example, a babbling idiot failure disrupts the communication between nodes, and the failed node can be blocked from communication with the distributed control. In the network, processing elements exchange periodic and non-periodic messages. Three types of messages are offered by BMTS:

- **Event-triggered messages** are transmitted when specific events occur at the sender. This event can be, for example, an interrupt signal or a task completion. The sender and receiver are equipped with queues for storing the messages. BMTS will read the queue and deliver the message. At the receiver side, before transmitting, BMTS performs error checking to control if the message was corrupted during transmission. If the message was corrupted, BMTS will drop it.

- **Time-triggered messages** are different from event-based, and they occur at regular intervals. This interval is defined by the communication schedule, which is expressed in a cyclic model of time. Each message contains a period and a phase. The phase defines the point in time when the message has to be sent, while the period defines the repetition time. Time-triggered messages are transmitted with minimum jitter since the communication system has knowledge about the schedule.
- **Rate-constrained messages** differ from both event and time-triggered messages. The transmission of the message is performed with a predefined maximum rate. This is done for two reasons: a) not to surpass the bandwidth and b) not to exceed the processing capacity of the system. Sender and receiver agree on a minimum guaranteed bandwidth before the communication starts.

In Table 3.6, some examples of real-time networks based on events, time, and a combination of both triggering mechanisms are shown. In this thesis, time-sensitive networking is chosen due to time-triggered characteristics.

Real-Time Networks		
Time-Triggered	Event-Triggered	ET & TT
Time-Triggered Protocol (TTP)	Ethernet	TTEthernet
TSN	Controller Area Network (CAN)	TTCAN
	Local Interconnect Network (LIN)	FlexRay
	Wireless Sensor Networks (WSN)	PROFINET
		TSN

Table 3.6: Real-Time Networks

3.3.3 Time Sensitive Network

Time-sensitive networking is a group of standards defined by the Institute of Electrical and Electronics Engineers (IEEE) (802.1) that transmits data using an Ethernet network in a time-sensitive manner. It offers deterministic real-time communication as well as guaranteed data transport with low latency, low delay, and low data loss for critical time applications [52]. Resources for critical traffic flow are reserved by TSN and utilize shaping methods, thereby achieving zero congestion loss. In addition, TSN offers

protection from malfunctioning, attacks, and bandwidth infringement. These features are realized by implementing different protocols. Essential TSN protocols include:

- **IEEE 802.1AS** is a time synchronization protocol developed based on Precision Time Protocol (PTP) from IEEE 1588. It provides precision in the range of nanoseconds and selects the master clock using Best Master Clock Algorithm (BMCA).
- **IEEE 802.1Qci** is a TSN protocol that provides flow management. It is known as Per-Stream Filtering and Policing (PSFP). Utilizing filtering, gating, and policing mechanisms, this protocol establishes specific traffic flows that can be delivered. Each stream is identified by a Unique Identifier (ID) denoted as streamID, which is composed of Media Access Control (MAC) address plus Virtual Local Area Network (VLAN) identifier. Main functions of PSFP include: a) identifying and classifying incoming flows, b) deciding whether to accept or block the stream based on timing criteria, c) forwarding to the correct gate, and d) monitoring bandwidth. Table 3.7 shows an example of how these functions are realized. TSN PEs and switches contain tables where all the incoming streams are mapped, as well as the correct time for sending or receiving and forwarding ports. If the streamID matches the one in the table and the time is correct, the stream will be forwarded to that specific port.

Stream Identifier		
StreamID	Time (seconds)	Port
0x001B190000010001	0.0125	1
0x001B190000010002	0.02234	2
0x001B190000010003	0.0442	3
0x001B190000010004	0.0033	4

Table 3.7: Per-Stream Filtering and Policing Table

- **IEEE 802.1Qbv** is the protocol responsible for scheduling and traffic shaping. It establishes a time-aware shaper, which is fundamental for a highly reliable communication network with low latency and determinism. TAS ensures that the packets are transmitted at predefined times, utilizing cyclic schedules for determining when communication must occur. The traffic queues are controlled by gate control lists, which have the power to decide if the transmission gates should be open or closed. GCL of TAS can be seen in Figure 3.8. In addition, 802.1Qbv protects the network from the transmission of low-priority traffic before the time-critical one. In this way, TSN can allow for both traffic types to coexist.

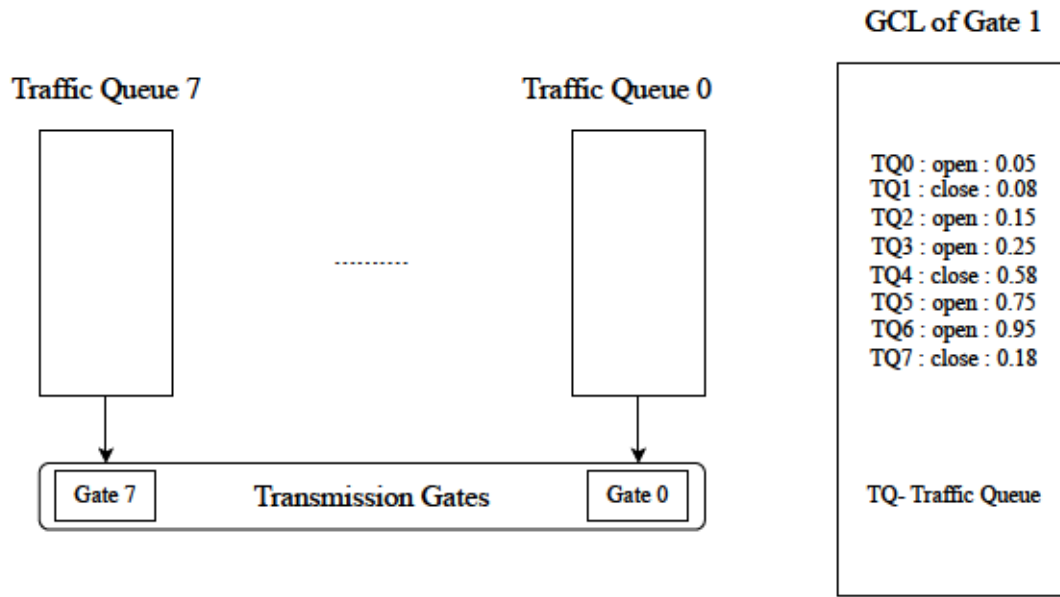


Figure 3.8: Gate Control List of TAS

- **Cut-Through Switching** is a TSN mechanism for sending packets before they are fully received. It reduces latency and delay, much needed for real-time traffic communication.
- **IEEE 802.1Qbu** is the TSN protocol for frame preemption. The main task relies on high-priority streams to interrupt those with lower priority, thus reducing latency and boosting network efficiency. After high-priority streams are sent, lower ones resume transmission.
- **IEEE 802.1CB** is the TSN frame replication and elimination protocol responsible for implementing stream redundancy in the network. Stream redundancy guarantees reliability and availability for time-critical traffic by masking link failures as well as packet losses. Packets coming from a sender are duplicated and transmitted using multiple separate routes. At the receiver side, the original packet is reconstructed, and the duplicated streams are dropped.

3.4 Discrete-Event Simulations

A system is discrete if the state changes occur in jumps [117]. Based on this approach, Discrete-Event Simulations (DES) can be constructed where all the operations of the systems are modeled as time sequence events. The state of the system changes when the events are executed, and each event is carried out at a particular instant of time. In between events, the state is not modified, and the simulation time jumps directly to the time of the next event. This is known as next-event progression time. The main

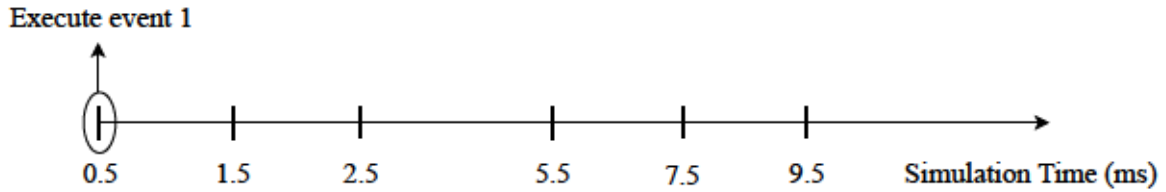


Figure 3.9: Discrete-event simulation event list

components of DES are:

- **State** is defined as the set of variables that represent the properties of the system that is under observation. The state of the system changes when events occur.
- **Clock** is important to implement the simulation time. As stated, time jumps from one event to the other event time.
- Another important property is the event list. The simulation groups the start times of all events in a list. This list is known as Future Event List (FEL) or as Future Event Set (FES). Figure 3.9 and 3.10 show a FES as well as the time hops.

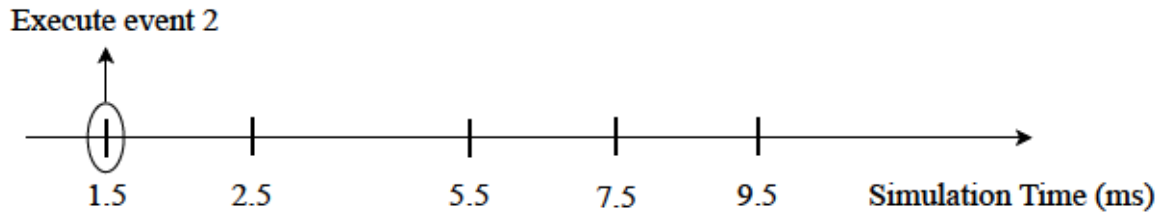


Figure 3.10: Example of discrete-event simulation event list

In this thesis, two discrete event simulations have been utilized in chapter 5, SystemC and OMNeT++.

3.4.1 SystemC

SystemC is a simulation language written in C++ that provides discrete-event simulation [118]. It is also used to model electronic systems, both in hardware and software. The building block of SystemC is the `SC_MODULE` (C++ class) that represents a component in real-time systems. Like in C++, modules must contain constructors, and this is realized by using `SC_CTOR`. The most important SystemC modules include: ports, signals, channels, interfaces, events, processes, and data types. `SC_EVENT` is the method used for process instance and synchronization. SystemC offers a concurrent execution of simulation processes. To achieve parallelism, `sc_wait()` function calls must be used that suspend the process thread, thus allowing the execution of other threads. For all the processes, the simulation time remains the same. A simulation process can be:

1. **SC_THREAD** is an execution thread that consumes simulation time. It allows the usage of wait calls.
2. **SC_METHOD** is not a thread and therefore consumes no simulation time. Wait functions can not be used.
3. **SC_CTHREAD** is a `SC_THREAD` that is sensitive to the clock edge event.

3.4.2 OMNeT++

OMNeT++ is an object-oriented framework for discrete-event network simulation [119]. It consists of simple modules written in C++ that communicate via messages. Simple modules can be grouped to form more complex modules called compounds. A network itself is a compound module. OMNeT++ uses Network Description Files (NED) to represent simple and compound modules. The parameters of modules (simple and compound) are configured using a configuration file called the INI file. Figure 3.11 gives an overview of OMNeT++'s modular architecture. The user interfaces available are Cmdenv and Qtenv. The ENVIR module is a library containing all the code shared by all the user interfaces, and SIM is the module containing the simulation kernel and class libraries. All network components, like simple modules, channels, messages, compound modules, and the corresponding C++ objects, can be found in the Model Component Library. A simulation can only run network models if all components are linked. The model to be simulated is located in the module called Executing Model. ENVIR is the module that has full control over the simulation. It determines the model to be used and contains the simulation loop in which events are executed. Furthermore, it invokes the event scheduler and handles any errors that occur during the simulation. The simulation concept of OMNeT++ is based on discrete-event simulation. As previously stated in Section 3.4, a discrete-event system is defined as a system in which state

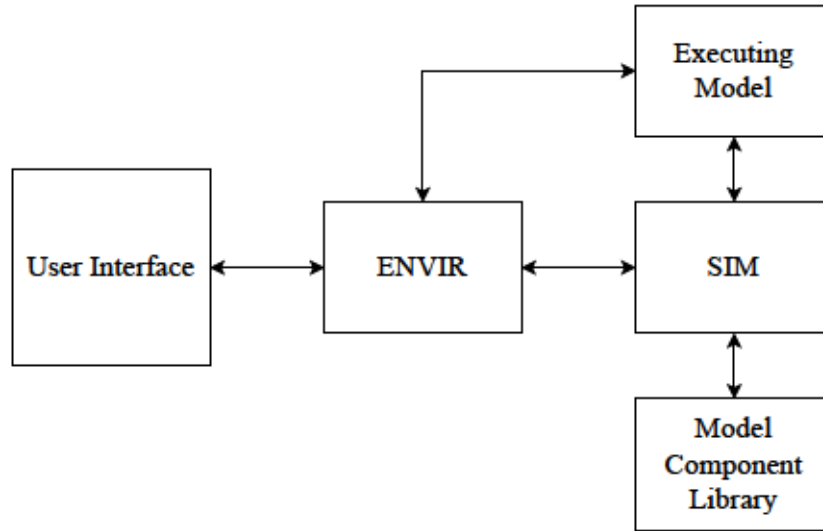


Figure 3.11: OMNet++ simulations architecture [120]

changes (events) occur at specific points in time, and events take no time to occur. It is assumed that nothing of interest happens between two successive events, i.e., there is no change of state in the system between events. In OMNeT++, events are represented as messages and stored in the future event set [121]. The event loop is responsible for executing and processing all events from the FES. Causality is maintained by a timestamp, which ensures that no current event can have an effect on previous events.

Chapter 4

TIME-TRIGGERED ORGANIC COMPUTING ARCHITECTURE

4.1 System Architecture

Embedded systems that are built based on organic computing have the ability to manage themselves utilizing the self-x properties. Using self-x, they can configure themselves, detect and recover from different faults, defend against attacks, and improve performance. The artificial hormone system and artificial DNA are used in this research. AHS is a decentralized, self-organizing, self-healing, self-optimizing, self-configuring OC middleware that performs spatial allocation of application tasks to processing elements. The AHS middleware can configure a distributed system by autonomously selecting a task allocation—that is, identifying the most suitable processing element for each task. It can then adapt this allocation to optimize system performance under changing environmental conditions or to recover from processing element or task failures. For effective self-healing, it is particularly crucial that these adaptive capabilities are decentralized to prevent single points of failure. The structure and organization of the embedded system are defined by an artificial DNA, which is compact enough to be stored in every processing element. All necessary information—such as task structure, inter-task cooperation, communication patterns, and core suitability—can be derived from this ADNA. As a result, the system becomes self-constructing, guided entirely by its artificial DNA.

On the other hand, systems that rely on time-triggered communication consist of processing elements where each PE includes a communication controller and a host computer. The communication controller manages the communication protocol for each PE and synchronization with the global time base, while the host computer runs the operating system and application software. The global time is necessary for time-triggered operation, which supports consistency. This is achieved when events across all processing elements are processed in a consistent order. By basing operations on a global time base, all PEs follow the same sequence of events, introducing determinism

to the system. This predictability makes the system deterministic. Reliability can be improved with active redundancy, which needs determinism for exact voting. The global time base is also used for error detection in the communication protocols. For instance, in the case of a "babbling idiot" failure, where a processing element sends messages at incorrect times, the issue is addressed by predefined communication slots for each node. If communication occurs outside these designated times, bus guardians block the messages. In addition, as can be seen from the architecture picture, at the operating system layer, each processing element will contain a task dispatcher and a configuration. Task dispatcher has the ability to read the system schedule and execute the predefined application task or message injection. The configuration represents the time-triggered schedule that is deployed in the communication controller. All operations, such as application task execution and message communication, are predefined through table-based schedules.

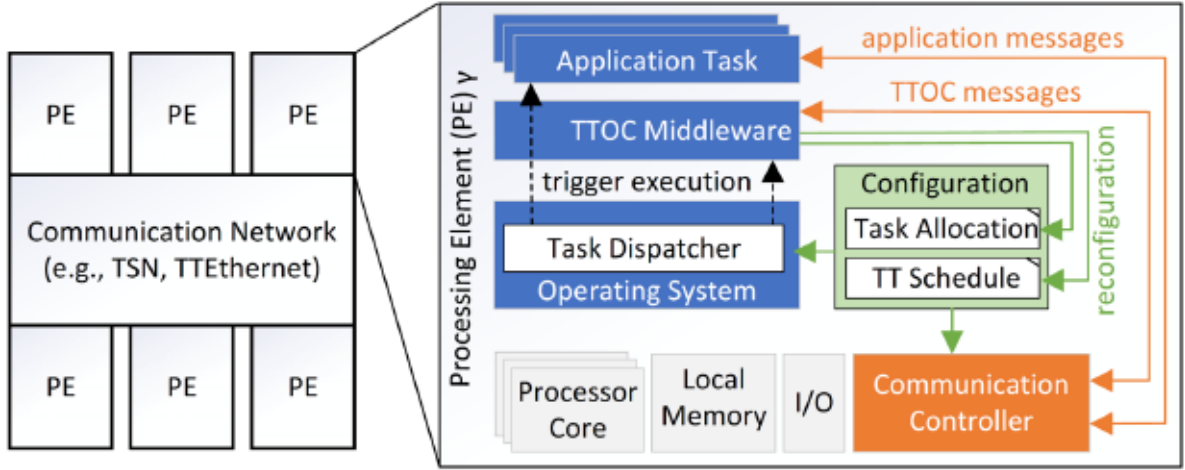


Figure 4.1: Inside view of TTOC processing element

The research focus of this thesis is to integrate two confronting ideas, self-x properties of an autonomous system, such as organic computing, with time-triggered execution of application tasks and message communication. A new architecture for distributed real-time systems composed of organic computing technologies of ADNA and AHS together with time-triggered communication is presented in Figure 4.1. Each processing element in the TTOC system is characterized by two operational states: a) normal and b) reconfiguration. Normal state represents the system working as intended, while in reconfiguration, the system is recovering from the occurrence of a failure. In addition, the self-x properties of organic computing (e.g., self-organizing, self-configuring, self-healing) are present and combined with schedule-based communication. The structure and organization of the system are described in the ADNA computer file, which is stored inside each processing element. AHS is responsible for reading the ADNA file, building the system's application functionalities, and performing spatial allocation of application tasks (self-organization). AHS also handles self-reconfiguration. In case

of a PE failure, the spatially scheduled task must switch its allocation. However, in a time-triggered system, all system operations are based on the schedule tables that define exactly when a message must be transmitted or a task must be executed. The scheduling decisions are as follows:

- Allocation of application tasks to processing elements
- Application task execution times
- Message injection times
- Messages paths

Since the spatial allocation of application tasks is performed by AHS, a new scheduling system is needed. In this approach, the execution of the hormone loop is integrated in the communication schedule, where each processing element will exchange hormones based on predefined time slots. Initially, the AHS middleware determines the allocation of tasks (of the application) in the system at run-time, without execution times, thus allowing for the desired flexibility that saves hardware cost. After the spatial allocation of the application task is performed, the list scheduling algorithm in TTOC then dynamically calculates the tasks' execution times along with the message injection times and the paths for the application messages, according to the given task assignments. Each PE calculates its own schedule dynamically, utilizing a heuristic list-based scheduling algorithm. The list algorithm is tailored based on TTOC requirements; it receives the spatial allocation from AHS and calculates the temporal slot for the application task. Furthermore, it decides the path and the injection time of the system messages (application and AHS). Any time-triggered communication network is suitable for TTOC (e.g., TSN, TTEthernet). In this work, the communication channel is considered to be TSN. Due to dynamic schedule generation and application tasks reconfiguration, the message path and time must be updated. New control messages are needed for the TSN switches to allow forwarding of TSN streams at defined times. A detailed explanation is given in the sections below.

4.2 System State

The self-x capabilities of AHS (e.g., self-organization, self-configuration, self-healing) allow the organic computing system to manage itself in case of failure scenarios or environment changes. These scenarios include PE failure, task failure, and PE task respawning due to the increase or decrease in hormone values. In TTOC, the computational model, also known as Finite State Machine (FSM) for the distributed embedded system, is defined as a 5-tuple:

$$FSM = (S, \Sigma, \delta, s_0, F)$$

where:

- $S = \{\text{Normal}, \text{Reconfiguration}\}$ is the set of states
- $\Sigma = \{\text{FaultDetected}, \text{ReconfigComplete}\}$ is the set of input events
- $\delta: S \times \Sigma \rightarrow S$ is the state transition function, defined as:

$$\delta(\text{Normal}, \text{FaultDetected}) = \text{Reconfiguration}$$

$$\delta(\text{Reconfiguration}, \text{ReconfigComplete}) = \text{Normal}$$

- $s_0 = \text{Normal}$ is the initial state
- $F = \{\text{Normal}\}$ is the set of final (stable) states

The operation of the TTOC system is defined in two states: a) normal and b) reconfigured, as it can be seen from Figure 4.2.

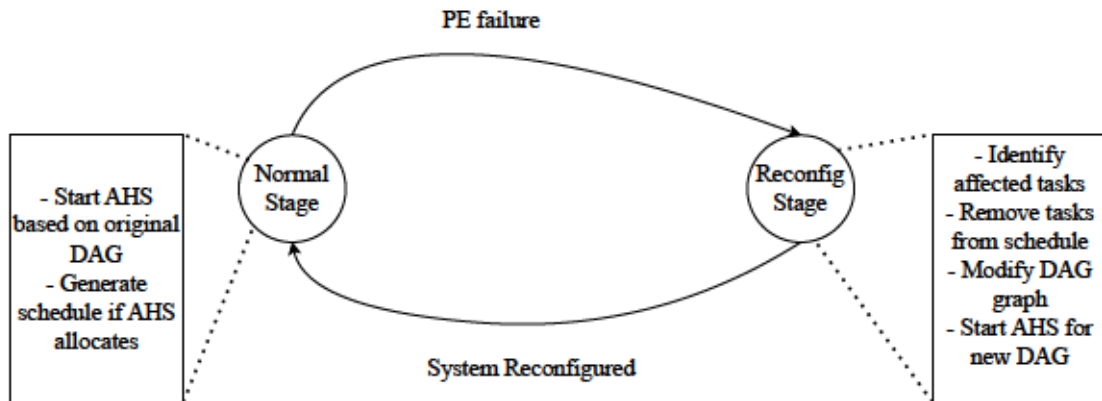


Figure 4.2: Processing element state

4.2.1 Normal State

The normal state is the initial state of TTOC. After the ADNA file has been decoded and all the system components are built, the normal state begins functioning by allocating application tasks to PEs based on the DAG. Overall, the normal state functionalities are:

- All PEs function as per the designed real-time schedules and application logic.
- Communication between nodes follows predefined protocols (e.g., TTEthernet, TSN).
- Control tasks, sensing, and actuation are executed predictably within timing constraints.
- The system operates in a steady-state, providing expected functionality and performance.

4.2.2 Reconfiguration State

In case of a PE failure, the remaining processing elements will switch into a reconfiguration state. In this state, each PE will identify the affected application tasks and will remove them from the schedule. The task allocation process within the AHS hormone loop will start reallocating these affected tasks to the remaining PEs. When all the tasks are scheduled, each PE returns to the normal state. With new schedules computed dynamically, the communication network must also be adapted to new configurations. For example, if a TSN network is chosen as the communication medium, its GCL and stream identifiers for each switch have to be updated for the next period before the new schedule is executed. Overall, the reconfiguration state functionalities are:

1. This state is entered when a disruption occurs, such as:
 - Processing element failure
 - Communication failure
2. The system initiates a reconfiguration strategy
 - Detection and removal of affected application tasks from the schedule
 - Task rescheduling procedures
 - Network routing updates

Algorithm 1 Handle Task Reconfiguration

```
1: function FAILEDTASKS(failed_tasks)
2:   Initialize: reconfigured_msg  $\leftarrow \{\}$ , new_tasks  $\leftarrow \{\}$ , starttasks  $\leftarrow \{\}$ 
3:   msg_  $\leftarrow$  get messages from configuration
4:   adj  $\leftarrow$  empty, inDegree  $\leftarrow$  empty
5:   for all task_id in failed_tasks do
6:     task  $\leftarrow$  FINDTASK(task_id)
7:     TASKSRESCHEDULED(task.id)
8:   end for
9:   for all rescheduled_id in reconfiguredTasks_ do
10:    task  $\leftarrow$  FINDTASK(rescheduled_id)
11:    Clear task's parents and children
12:    Reset task.scheduled, start_time, end_time
13:    Append task to new_tasks
14:   end for
15:   TASKRECONFIGURED(reconfiguredTasks_)
16:   for all task in new_tasks do
17:     noparent  $\leftarrow$  true
18:     for all message in msg_ do
19:       if message.receiver = task.id and TASKSEXIST(message.sender) then
20:         noparent  $\leftarrow$  false
21:         break
22:       end if
23:     end for
24:     if noparent then
25:       Append task.send_msg to reconfigured_msg
26:       Append task to starttasks
27:     end if
28:   end for
29:   Sort starttasks by descending priority
30:   for all msg in reconfigured_msg do
31:     sender  $\leftarrow$  FINDTASK(msg.sender)
32:     receiver  $\leftarrow$  FINDTASK(msg.receiver)
33:     Append receiver to sender.children
34:     Append sender to receiver.parent
35:     ADDEDGE(msg.sender, msg.receiver)
36:   end for
37:   adj  $\leftarrow$  GETTASKSCONNECTIONS
38:   inDegree  $\leftarrow$  GETTASKSDEGREE
39: end function
```

Algorithms 1 and 2 have been developed to identify and reconfigure the affected application tasks. They ensure systems resiliency by rebuilding the task dependency graph and rescheduling tasks in an appropriate order. The algorithms follows these steps:

1. First, it initializes key data structures:
 - `reconfigured_msg` – stores inter-task messages involved in the reconfiguration.
 - `new_tasks` – holds tasks that require rescheduling.
 - `starttasks` – list of root tasks (those without any parent dependencies).
 - `adj` – an adjacency list representing the task dependency graph.
 - `inDegree` – a list that tracks the number of incoming edges (dependencies) for each task.
2. It iterates over each failed task in order to:
 - Finds the corresponding task using a helper function.
 - Marks the job as rescheduled using `tasks_rescheduled()`.
3. Prepares tasks for reconfiguration
 - The parent and child dependencies are cleared.
 - All scheduling and timing information is reset.
 - The task is added to the `new_tasks` list.
4. The algorithms identify tasks that have no remaining parents by analyzing the message graph. If a task has no incoming messages from existing tasks, it is considered a starting point for rescheduling and added to both:
 - `starttasks`, and
 - `reconfigured_msg` (for tracking communications).
5. Sorting the tasks by priority.
6. Rebuilding task dependency graph.
7. Topological sorting of affected tasks:
 - Each task in `starttasks` is processed, added to the final sorted list, and removed from the queue.
 - The in-degree of each child task is reduced.

- If any child's in-degree becomes zero, it is added to `starttasks`.
- The list is re-sorted by priority at each iteration.

8. Updating tasks states:

- The corresponding application-level task is found.
- Its internal state is reset (e.g., scheduling flags, offered status).
- It is added to the `ReconfiguredTaskList` for future scheduling.

The algorithms provide a robust mechanism to dynamically recover from PE failures in distributed systems. By selectively clearing, reordering, and reconfiguring only the affected tasks:

- System uptime and task execution consistency are preserved.
- Task dependencies are respected via graph analysis.
- Priorities are incorporated into the rescheduling process.

This approach enables fault-tolerant execution while minimizing computational and scheduling overhead.

Algorithm 2 Handle Task Reconfiguration

```
1: function FAILEDTASKS(failed_tasks)
2:   while starttasks is not empty do
3:     task  $\leftarrow$  starttasks[0]
4:     Append task to reconfiguredtasks_
5:     Remove task from starttasks
6:     for all neighbor in adj[job.id] do
7:       inDegree[neighbor] -= 1
8:       if inDegree[neighbor] = 0 then
9:         Append FINDJOB(neighbor) to startjobs
10:      end if
11:    end for
12:    Sort starttasks by descending priority
13:  end while
14:  for all task in reconfiguredtasks_ do
15:    for all task in GETTASKLIST do
16:      if task.id = job.id then
17:        app_task  $\leftarrow$  FINDAPPTASK(task.id)
18:        task.WipeActive()
19:        task.ClearOffered()
20:        task.ClearOfferTime()
21:        app_task.set_scheduled(FALSE)
22:        Append app_task to ReconfiguredTaskList
23:      end if
24:    end for
25:  end for
26: end function
```

4.3 Time-Triggered Schedule

In TTOC, all communication that will happen between one PE and another must be determined by a time-triggered schedule. The scheduling decision is distributed, meaning that every processing element calculates its own schedule, and there is no central module that contains information about the global schedule. The scheduling problem for the time-triggered communication is divided into two parts: a) the timing model of the system, that models the period and the phase, and b) the algorithm which ensures all the timing criteria are met.

4.3.1 Timing Model

To address one of the challenges of this research, which is the integration of self-x properties with time-triggered communication, a new schedule system has been developed. In this system, the execution of the hormone loop of organic computing has become part of the schedule. In the classical design, the scheduler organizes:

- Allocation of application tasks to processing elements
- Application task execution times
- Message injection times
- Messages paths

In the newly proposed design, this process is split and performed incrementally. Initially, the AHS middleware determines the allocation of application tasks in the system at run-time, without execution times, thus allowing for the desired flexibility that saves hardware cost. After the spatial allocation of the application task is performed, the list scheduling algorithm in TTOC dynamically calculates the tasks' execution times along with the message injection times and paths for the application messages, according to the given task assignments.

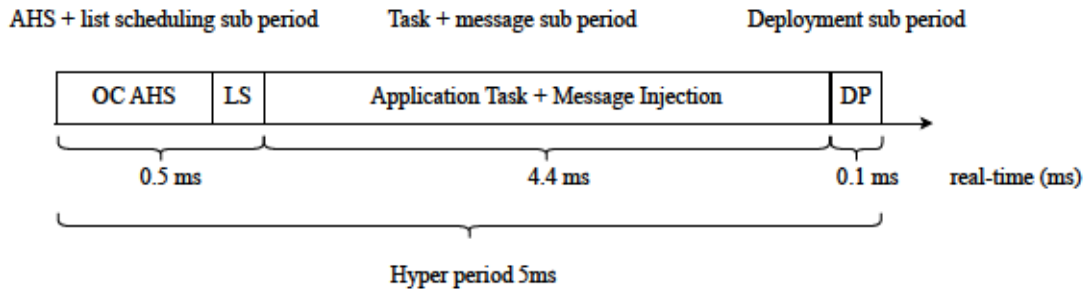


Figure 4.3: Example of System Timing Model

The hormone loop of the AHS middleware that runs on each PE will contain a predefined time slot within the schedule period. Our proposed time-triggered schedule relies on a 5 millisecond hyperperiod in which all task execution inside each PE and communication between PEs are realized. The period represents the full schedule cycle as it can be seen from Figure 4.3. It is divided into 3 timing intervals, where the first interval is reserved for the scheduling phase, which is carried out by AHS and the list scheduling algorithm. This interval is 0.5 milliseconds. The second one has a reserved slot of 4.4 ms and is used for the execution of application tasks plus message exchange. The last sub-interval is called the Deployment Phase (DP), in which the results of the scheduling decision are updated for the next period of the schedule.

4.3.2 Scheduling Algorithm

As stated, in TTOC, the system schedule is computed dynamically at each PE. The dynamic scheduling algorithm that has been taken into consideration for TTOC is the list scheduling algorithm. List scheduling is a process used to schedule tasks represented as a directed acyclic graph. In the example in Figure 4.4, there is a DAG consisting of

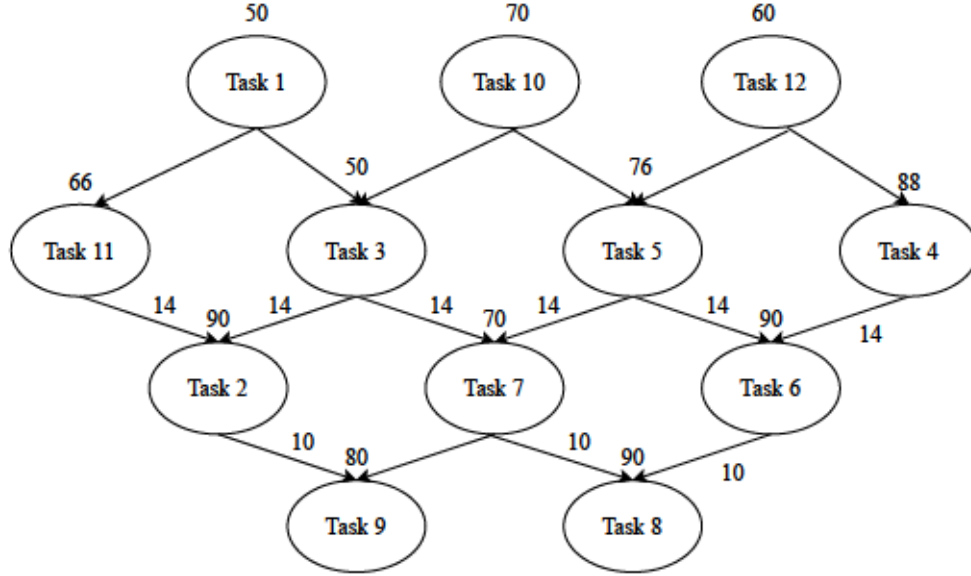


Figure 4.4: DAG graph

twelve application tasks and their worst-case execution times (wcet) alongside message size. The tasks are arranged based on their depth in the graph and the time required for their execution. Each task is represented by a node, and dependencies between them are represented by edges in the graph. The scheduling algorithm determines the order in which the tasks should be executed and aims to assign each task to an available processing element. It takes into consideration the data dependencies, timing

constraints, system model, critical path, scheduling priorities, and optimizes for factors such as load balancing, together with minimizing idle time. Since the spatial allocation is taken care of by TTOC middleware, the AHS will start processing the hormone loop tasks based on their execution order. For that reason, it is important to analyze and set the execution order of tasks. In our above DAG graph, task two is data dependent from task one and task three because the messages that arrive from them are needed for its computations. In this case, task two must wait for the messages coming from task one and task three before it can start executing. Task one and task three are denoted as the immediate parents of task two.

Algorithm 3 List Scheduling Algorithm

```

1: function LISTSCHEDULING(task_id, pe_id)
2:   task  $\leftarrow$  find_task(task_id)
3:   appTask  $\leftarrow$  FindApplicationTask(task_id)
4:   task.runs_on  $\leftarrow$  pe_id
5:   appTask.setPEId(pe_id)
6:   task.wcet  $\leftarrow$  task.wcet.fullspeed
7:   taskHasSlot[task.id]  $\leftarrow$  False
8:   for each parent in the task.parents do
9:     for each message from parent to task do
10:      if parent.runs_on == task.runs_on then
11:        schedule message locally (no route)
12:        inject at current start time
13:      else
14:        compress message size
15:        route  $\leftarrow$  shortest path from parent to task
16:        injectionTime  $\leftarrow$  find earliest via PE DRT
17:        lock route and port
18:        schedule inter-node message
19:        update task.start_time based on message arrival
20:      end if
21:    end for
22:  end for
23: end function

```

In addition, the critical path in the DAG is calculated by taking into consideration the computation time (WCET) of the task and the communication times (message size) on that particular path. Hence, we start processing tasks from the most critical path of the DAG. The bottom-level has been used to determine the node levels and scheduling priority. After the tasks are ordered, the AHS can start the hormone loop to determine the suitability levels of the task and assign it to the most suitable PE. In case that an allocation decision is made, the PE will activate the list algorithm to generate the temporal allocation of the application tasks, the spatial and temporal allocation of messages. The Algorithms 3,4 describe the procedures' functionality. As input, the

algorithm takes the application task that has to be scheduled, and it follows these steps:

1. Task and Metadata Initialization

- Locate the task and its metadata.
- Assign it to the specified PE and record its worst-case execution time (WCET).
- Mark the task as not yet scheduled.

2. Handle Parent Communication

- For each parent task, check all outgoing messages to this task.
- If the parent and the task are on the same PE:
 - Schedule the message locally with no routing.
- Otherwise:
 - Compress the message.
 - Determine a route between the parent and the task using the routing algorithm.
 - Compute the injection time using the DRT (Direct Routing Table).
 - Lock routes and schedule the message through links and ports.
 - Update the task's earliest valid start time based on message arrival.

3. Task Slot Allocation

- If the task is not yet assigned to a time slot:
 - Align the task start time with the system time.
 - For periodic tasks:
 - * Use system time to assign task start time if there are no parents.
 - * Use the latest message arrival time from parents, otherwise, to assign the task start time.
 - Lock the time slot on the assigned PE.
 - Update the application's schedule table.
 - Set flags to mark the task as scheduled.
 - Record the end time of the task.

4. Child Task Preparation

- For each outgoing message from this task:
 - Locate the child task.
 - Register this task as one of its parents.

5. Period Update and Completion

- Update the global scheduling period.
- Return True to indicate successful scheduling.

6. Alternate Scheduling Path (If Already Scheduled)

- Recalculate the earliest start time:
 - Use system time if the task has no parents.
 - Use the maximum end time of all scheduled parents otherwise.
- Check for conflicts with existing scheduled tasks.
- Lock a new time slot and update metadata accordingly.

Due to the TTOC characteristics of scheduling one task per period, it is not possible to schedule the outgoing messages from a task when the receiving node is not yet scheduled. For all tasks except the ones without parents, before scheduling the temporal allocation, the spatial and temporal scheduling of incoming messages takes place. Step 6 is related to reconfiguration cases, where tasks are already scheduled in other PEs. This is important in the reconfiguration state, where allocated tasks from a failed PE must be rescheduled. Application tasks and messages have fixed time slots where they can be allocated, which is from the end of the scheduling phase, 0.5 ms to 4.9 ms before the deployment phase (recall Figure 4.3). This means that all application tasks and messages will need to be scheduled in the period, otherwise the AHS will not allocate application tasks to PEs, and the whole system will malfunction. For this reason, in the algorithm, it is constantly checked if the start time, or the start time plus the width of the application tasks, does not overlap with the interval. In addition, the PE can also be in an idle stage; in this case, it waits for the next application task to be processed. Figure 4.5 shows a schedule table with the start and finish execution times of the tasks and idle stages.

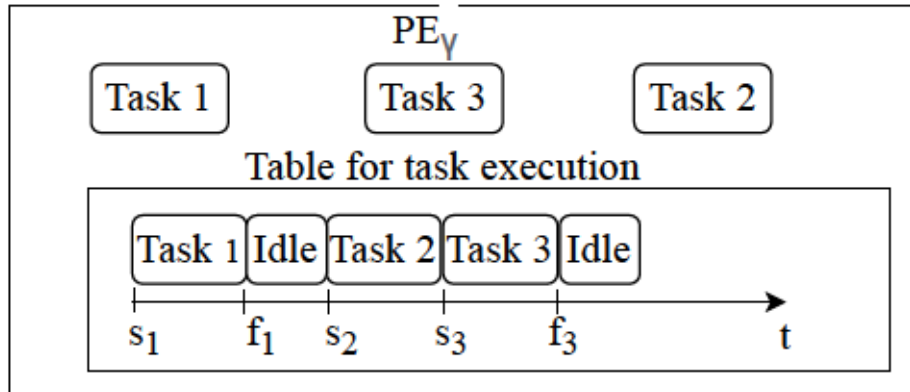


Figure 4.5: Schedule table for task execution

Algorithm 4 List Scheduling Algorithm

```
1: function LISTSCHEDULING(task_id, pe_id)
2:   if not taskHasSlot[task.id] then
3:     if schedulePeriod == 0 then
4:       if task.start_time  $\leq$  getScheduleTime() then
5:         task.start_time  $\leftarrow$  getScheduleTime()
6:       end if
7:     else
8:       if task has no parents and task.start_time  $\leq$  getScheduleTime() then
9:         task.start_time  $\leftarrow$  getScheduleTime()
10:      end if
11:    end if
12:    lock PE time for [start_time, start_time + wcet]
13:    set appTask times and update flags
14:    m_start_time  $\leftarrow$  task.end_time
15:    for each message to child do
16:      child  $\leftarrow$  find_task(message.receiver)
17:      child.parents.push_back(task)
18:    end for
19:    taskHasSlot[task.id]  $\leftarrow$  True
20:  end if
21:  Add appTask to the scheduled list
22:  schedulePeriod += getSchedulePeriod()
23:  return True
24:
25:  if task has no parents then
26:    earliest  $\leftarrow$  getScheduleTime()
27:  else
28:    earliest  $\leftarrow$  max(parent.end_time for scheduled parents)
29:  end if
30:  for each scheduled task, update the earliest if overlap
31:  lock PE time for [earliest, earliest + wcet]
32:  set appTask times and update flags
33:  m_start_time  $\leftarrow$  task.end_time
34:  Add appTask to the scheduled list
35:  schedulePeriod += getSchedulePeriod()
36:  return True
37: end function
```

4.4 Communication Controller

The communication controller is responsible for managing all time-triggered message transmissions and receptions between processing elements over the communication network. It operates based on a globally synchronized clock, enforcing transmission times according to the time-triggered schedule. This schedule dictates who can send at what time, ensuring that messages are transmitted at predefined time intervals without collisions. Core responsibilities of the communication controller are:

- Clock synchronization.
- Maintaining synchronization with bounded precision with other PEs using a global time base.
- Receives or transmits synchronization messages to ensure all controllers operate on the same timeline.
- Message transmission and reception at preallocated time slots.
- Deterministic behavior.
- Adheres to the schedule, regardless of application-level activity.

In the TTOC system, the controller also supports runtime updates to communication parameters (e.g., transmission windows, priority settings), enabling on-the-fly reconfiguration of stream behavior without violating timing guarantees. Additionally, it implements TSN features such as 802.1Qbv TAS for traffic gating, 802.1Qci for ingress policing, and 802.1AS for time synchronization. This ensures low-latency, deterministic communication even in mixed-criticality environments.

4.5 Task Dispatcher

The Task Dispatcher serves as the real-time execution engine within the PE, triggering application task execution in accordance with the time-triggered schedule. It interfaces closely with the operating system and the TTOC middleware to ensure that tasks are dispatched precisely at their assigned time slots. This guarantees deterministic behavior and avoids timing conflicts, which is essential in safety-critical applications. When a reconfiguration is applied (e.g., due to a mode switch or a fault recovery), the dispatcher updates its internal execution table, adjusting the task activations accordingly. Its tight integration with the middleware and the configuration module allows the system to remain predictable and correct even under dynamic conditions. Algorithm 5 describes the execution logic implemented in the dispatcher. Initially, the first task of the processing element schedule will be obtained. The algorithm runs infinitely, first waiting for the designated start time and then invoking the task and proceeding to the next one.

Algorithm 5 Task Dispatcher

```
1: Current task = task 1
2: while true do
3:   wait(task.time)
4:   invoke(task)
5:   goes to the next task
6: end while
```

4.6 Time Sensitive Network

The time-sensitive communication network forms a deterministic backbone for distributed real-time systems. It ensures bounded latency, minimal jitter, and reliable delivery of time-critical data by using a suite of IEEE 802.1 standards. In a dynamically reconfigurable setting, the TSN network not only guarantees time-deterministic communication but also supports online adaptation of traffic schedules and flow rules without disrupting ongoing system operation.

Table 4.1: Example 802.1Qci Filter Configuration

Filter ID	Match (Stream ID)	Rate (bps)	Burst (bytes)	Action
101	MAC + VLAN ID	2 Mbps	512	Allow
102	MAC + VLAN ID	1 Mbps	256	Drop excess
103	IP + Port	unlimited	N/A	Block

At the core of this flexibility lies the Centralized Network Configuration (CNC) entity, which computes and distributes gate control lists for 802.1Qbv time-aware shaper and Ingress Policing rules for 802.1Qci as seen in Table 4.1. These configurations define precise transmission time windows and stream-specific filtering behavior. Runtime reconfiguration is made possible through NETCONF/RESTCONF protocols, allowing the CNC to update switch configurations while preserving synchronization via 802.1AS. When a reconfiguration is triggered (e.g., due to a task reallocation, system mode switch, or fault recovery), the CNC issues updated GCLs and stream reservation parameters. The switches then apply these updates at a deterministic point in time (e.g., based on AdminBaseTime), ensuring a glitch-free transition. This approach enables the network to adapt to dynamic workloads or operational contexts while maintaining hard real-time guarantees.

In TTOC, the system schedule will be calculated by the list algorithm dynamically. That is why the message path and times for the TSN processing elements and switches will be inserted during runtime. In case of system reconfiguration, all the entries for the TAS will be deleted. After the new schedule is calculated, new control messages will be sent to the TSN switches to allow forwarding of TSN streams at the new defined times.

Chapter 5

SIMULATION ENVIRONMENT FOR TIME-TRIGGERED ORGANIC COMPUTING

5.1 Simulation Architecture

To validate the proposed TTOC architecture, a simulation environment has been constructed. Figure 5.1 gives a complete view of the simulation system. It is composed of a simulator based on SystemC for replicating a processing element of TTOC, of a TSN network running in OMNeT++, and of a cosimulation interface that controls the execution order. As stated in previous chapters, both SystemC and OMNeT++ are discrete-event simulations. Therefore, any progression of time is based on jumping into the next event time. SystemC provides the simulation of the computational resources (e.g., the node computers and their software) and integrates OC with time-triggered communication. The simulator is responsible for starting the AHS system and reading the ADNA file. An initial schedule is created that is based on the architecture timing model. The SystemC thread starts executing operations according to this predefined schedule. These operations include the AHS hormone loop that is divided into multiple smaller tasks, the schedule task in case an allocation decision is made, and application tasks. SystemC cannot send application messages from one PE to another because it lacks a communication network.

Therefore, to simulate message communication between processing elements in a distributed system based on time-triggered organic computing, we model a TSN network running in INET/OMNeT++ that operates according to a communication schedule that is calculated by TTOC during run-time. However, the TSN implementation in INET does not support dynamic loading of the network communication schedule. The configuration of GCLs for TSN switches, as well as message generation time for processing elements, is performed statically. Everything is done offline and given as input in the configuration file of the OMNeT++ simulation. Furthermore, the messages that

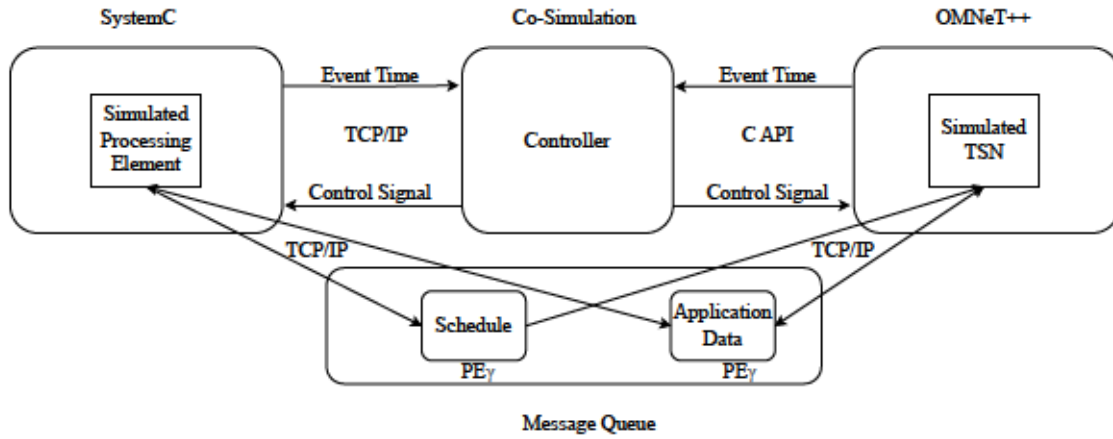


Figure 5.1: Time-Triggered Organic Computing Simulation Architecture

are sent in the network do not contain application data, and their path cannot be modified during runtime. That is why the TSN implementation in OMNeT++ has been adapted to include dynamic reconfiguration of schedules and streams, and for application packets, additional fields are added that store the data that will be transmitted by the message.

The approach chosen is to use message queues for storing the schedule and application data generated by the SystemC simulator. ZeroMQ (also known as ZMQ) is an asynchronous messaging library designed for use in distributed or concurrent applications, which provides message queues different from traditional message-oriented middleware. To insert these changes, a new module for the TSN network has been developed. This module is called Reconfiguration periodically reads ZMQ, constructs events in OMNeT++ for message generation/reception, modifies GCL and layer 2 services for stream direction. To coordinate the execution of the SystemC simulator with the network in OMNeT++ is essential to ensure accurate behavior and output from both simulations. This integration of two distinct simulations into a unified system is achieved through a co-simulation interface that synchronizes the execution of the simulations. As shown, the SystemC simulator connects to the co-simulation controller through the TCP/IP network protocol. In this setup, it provides the timing of the next event, while the co-simulation controller sends a control signal that acts as a trigger, prompting the simulator to execute the specified event.

Similar modifications are also performed for OMNeT++ but instead of TCP/IP, C API calls and functions have been used. A new user interface based on Cmdenv and functions is added to OMNeT++ with the purpose of controlling the execution process of the FES. The time of the next event is sent to the co-simulation controller, and a trigger signal is receiver whether to execute that event or not. The following sections give insight into each specific component of the simulation environment.

5.2 Simulated PE

As a starting point for the simulation of the computational resources of the TTOC system, the simulator DNASIM was used, which combines ADNA and AHS. DNASIM was originally developed by the University of Frankfurt for testing the creation of an application model from ADNA computer files and testing the AHS middleware functionality of temporal allocation of these tasks.

5.2.1 Old Simulated PE

The DNASIM simulator was built in a Windows environment, and it contained a user interface where the configuration files are loaded. User interface serves also for modifying system parameters during runtime and for visualizing the system behaviour. Two configuration files can be loaded. The first one, denoted as hsm configuration, contains a description of processing elements, possible tasks for each PE, their relationships, hormone values, and timed events. Timed events create possible faults that can occur in the PE with the purpose of analyzing how the self-x properties of AHS react. Five timed event types are:

- **Kill PE** simulates the crash failure of the processing element.
- **Kill Task** removes the allocation of the task from the PE.
- **Respawn** simulates reallocation of the task to the processing element.
- **Monitoring Accelerator** modifies the values of the accelerator.
- **Monitoring Suppressor** modifies the values of the suppressor.

In the second case, the ADNA file is loaded alongside two configuration files. The first file defines the basic ADNA classes and the basic processing element classes compatible with the real DNA-AHS system. The second file defines PE and their resource names, and the last one gives the basic element descriptions. The Simulator is programmed using the C++ programming language and uses C++ objects for describing application tasks or timed events and C++ object lists for storing them in a container. The value reception list is an object list where the received values of suppressors, accelerators, and eagervalue are stored in the PE. The simulator does not have a communication network; instead, the communication is done via shared memory. The communication is realized by creating a receive value object for the destination PE and storing it in the respective value reception list. For the simulation to start, windows threads are created with PEs as a pointer, where each thread will execute the hormone loop procedure. Figure 5.2 shows two windows of threads that model the simulation of two processing elements. Threads run in parallel, where they will send suppressors, accelerators, and

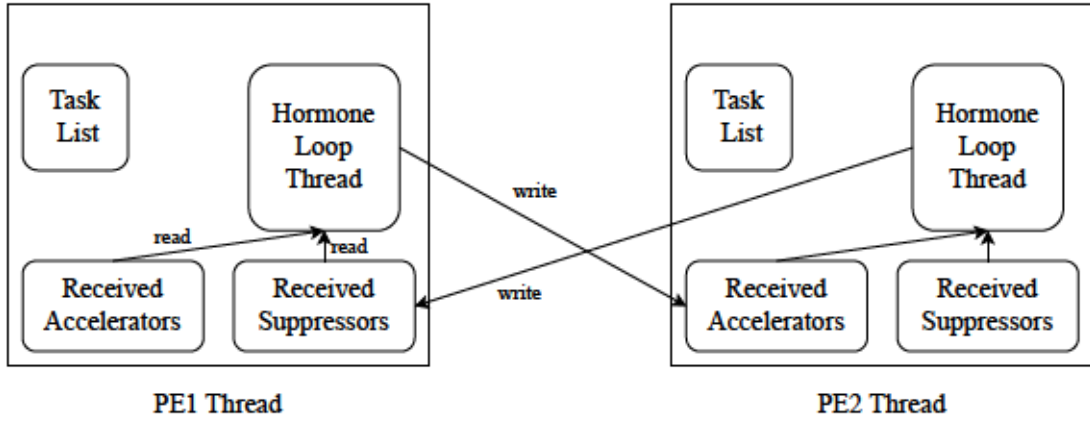


Figure 5.2: DNASIM Architecture

eager values to other PEs. There is a waiting period of $2 \cdot \text{HormoneLoopPeriod}$ for the synchronization of threads to ensure each one has received all necessary hormones before entering the task allocation stage. The tasks do not contain real-world application functionality, and therefore, message communication is not present. The hormone loop decides about one task allocation in each cycle.

5.2.2 New Simulated PE

The simulated PE for the TTOC simulation environment is based on the design of the mentioned DNASIM for the OC functionalities (e.g., task allocation), but it differs in the temporal control. The user interface and all the Windows threads are removed since they are not discrete event therefore they are not deterministic. That is why they are replaced by SystemC threads, where each thread simulates the working principles of TTOC processing element. The simulator operates according to a schedule as it is described in TTOC, which is calculated dynamically in each PE using the list scheduling algorithm. Application task execution and message transmissions (hormones and application tasks) have a fixed time slot. The task dispatcher is responsible for reading the schedule and executing the specified operation. In addition, application tasks are extended to model custom applications. The execution of SystemC simulation is controlled by the co-simulation controller via TCP/IP protocol.

Time-Triggered Schedule

Simulated processing elements operate based on a time-triggered schedule as defined by the system timing model. The time-triggered schedule is implemented using a Circular Linked List (CLL) with each entry of the CLL containing the time parameter and the task to be executed, as shown in Figure 5.3.

Initially, the CLL will contain organic computing tasks, which are constructed by

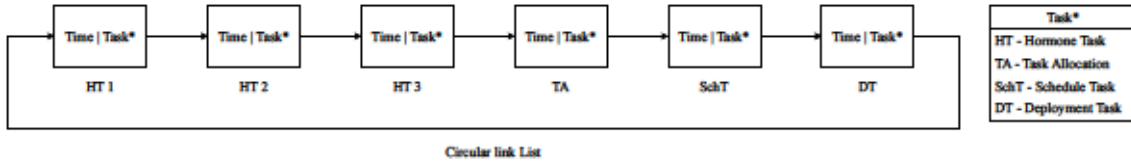


Figure 5.3: Circular Linked List

splitting the hormone loop into four smaller tasks in a predefined order and scheduling deployment.

- **Hormone Task1** where each PE will send accelerator and suppressor values to other PEs.
- **Hormone Task2** where PEs will calculate their current eager value and send it to the others.
- **Hormone Task3** for sending organ accelerators.
- **Task Decision** determines the spatial allocation of the application task that has the highest priority.
- **Deployment Task** will update the schedule for the next period.

If the hormone loop decides to allocate an application task to a processing element in the task decision phase, the schedule will be updated, as it can be seen from Figure 5.4.

- **Schedule Task** for activating the list algorithm if the hormone loop decides to allocate the application tasks.
- **Application Task** for executing real tasks.

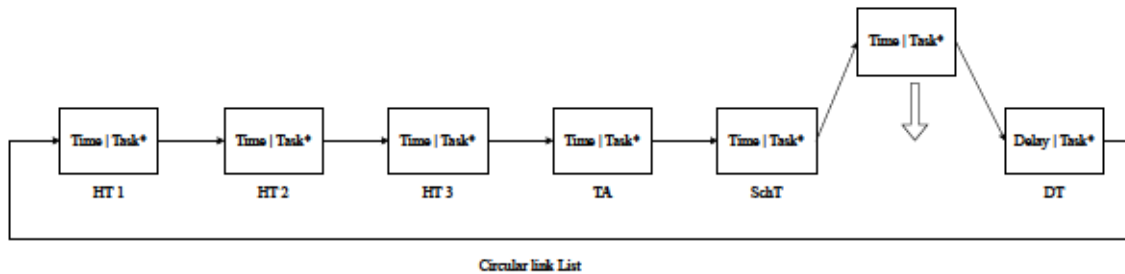


Figure 5.4: Updated Circular Linked List

The hormone loop is split because SC_THREAD differs from Windows threads, as they run sequentially and not concurrently. However, the desired parallelism can be

achieved using SystemC wait calls that suspend the thread, thus allowing other threads to get executed. This is important since all the threads of processing elements need to be executed before each PE enters the task decision phase. If the hormone loop makes a decision for a task in the absence of hormones from other PEs, the allocation of tasks would be wrong. Figure 5.5 demonstrates the parallel execution flow of three PEs.

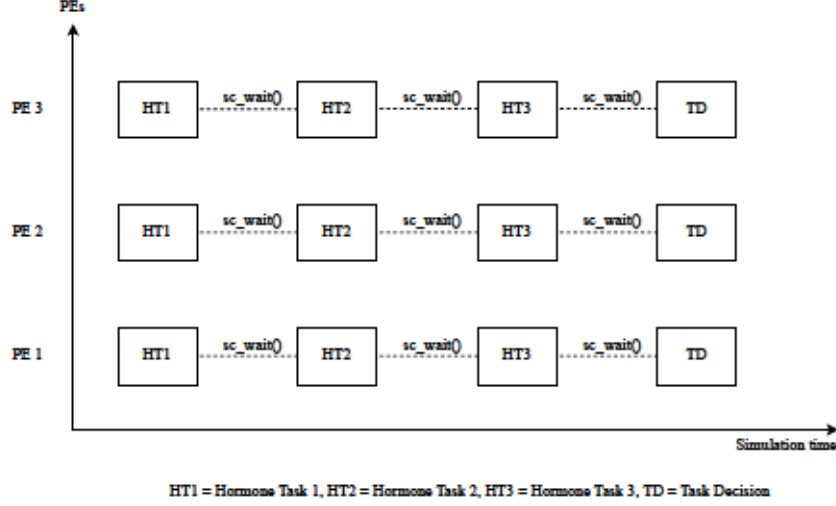


Figure 5.5: Execution flow of three processing elements

List Scheduling Implementation

As mentioned, the scheduling algorithm must be computed at each processing element dynamically. If the task decision phase of organic computing allocates a task to a particular PE, the list scheduling algorithm will be activated. The Algorithm 6 described below takes as input the task to be scheduled and the allocated PE. Before the algorithm starts execution, the application tasks have been ordered based on their dependencies and the depth of the DAG. The AHS hormone loop will allocate one application task per period. For this reason, messages that the parent application task has to send cannot be scheduled since the spatial allocation of the receiving PE is not yet performed. That is why the first step carried out is to check if there are messages to be scheduled, and then proceed with the temporal allocation of the application task. If there are no parent tasks or if the scheduling of messages has finished, the algorithm continues with temporal allocation. If the task has not been assigned a slot, the scheduler will assign the next free slot.

In case a processing element has failed, the application tasks that were allocated will be rescheduled along with the children of these tasks. The state of the PE switches to Reconfiguration, and the AHS will start the allocation process for the emerged sub-graph. The list algorithm must keep in mind the dependencies between tasks. The fully implemented code of the algorithm can be found in section C.1.

Algorithm 6 TTOC List algorithm for simulation

Ensure: T as task to be scheduled

Ensure: PE

```
for All tasks parents do
  for All Parent messages do
    if If message receivers is same as T then
      if Tasks run on same PE then
        Calculate message injection time
        Calculate message arrival time
        Update schedule table of PE
      end if
    else
      Calculate message injection time
      Calculate message path
      Calculate message arrival time
      Update schedule table of PE
    end if
  end for
end for
if Task is not scheduled then
  Task start time == PE time
  if Task is the first one to be scheduled then
    Task schedule generation
    Update TDMA table of PE
  end if
  if Task start time < hyper period + TTOC period then
    start time = hyper period + TTOC period
    Task schedule generation
    Update TDMA table of PE
  end if
  if Task start time + task wcet > hyper period + TTOC period then
    start time = 2 * hyperperiod + TTOCperiod
    Task schedule generation
    Update TDMA table of PE
  end if
else
  Task start time == PE time
  if Task is the first one to be scheduled then
    Task schedule generation
    Update TDMA table of PE
  end if
  if Task start time < hyper period + TTOC period then
    start time = hyper period + TTOC period
    Task schedule generation
    Update TDMA table of PE
  end if
  if Task start time + task wcet > hyper period + TTOC period then
    start time = 2 * hyperperiod + TTOCperiod
    Task schedule generation
    Update TDMA table of PE
  end if
end if
```

Task Dispatcher

The dispatcher was developed as a SystemC module composed of SC_THREAD that is in control of reading the CLL schedule and invokes the specific tasks with each entry of the CLL containing a time parameter and the task to be executed, as shown in Figure 5.6. The dispatcher algorithm is explained in Section 4.5, and the procedure developed is shown in the listings below. Since SystemC threads are not able to run in parallel, we have to utilize `sc.wait()` in order to achieve parallelism.

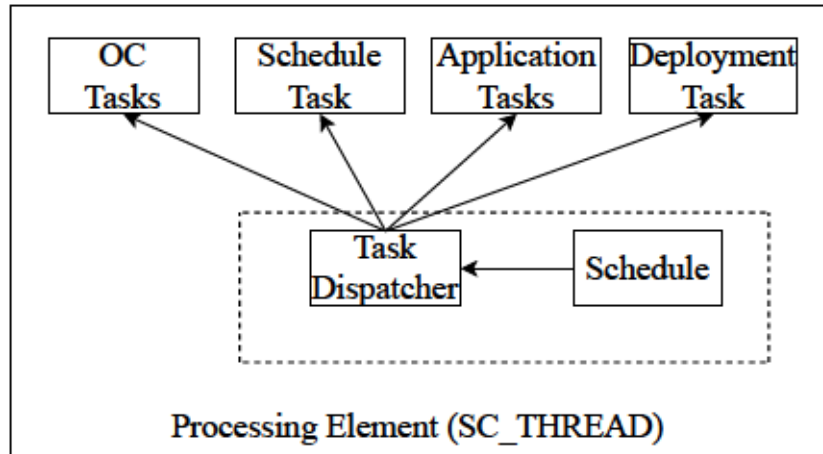


Figure 5.6: Inside view of TTOC processing element

5.2.3 SystemC Thread

As illustrated in Figure 5.1, the SystemC simulator is linked to the co-simulation controller via the TCP/IP network protocol, where it provides the time of the next event, and the co-simulation controller sends the control signal. The control signal is a trigger command from the controller to the simulation to initiate the execution of the event. Consequently, the computation logic of the TTOC simulator must be modified. As stated in Section 5.2, SystemC threads are employed in the TTOC simulator. In SystemC, the method `sc.start()` is responsible for the simulation phase. It comprises three distinct phases. The first is the elaboration, during which the execution of all the statements preceding the `sc.start()` is carried out. This is important to construct key data structures required for the simulation. The second phase is the execution of the threads until the final events in the scheduler. The last one is the post-processing, during which the data structures are destroyed, and the simulation is finished.

The manner in which `sc.start()` is invoked determines the subsequent course of the simulation. If it is called once and without arguments (the argument usually being the simulation time), the scheduler runs until the end of the simulation. With the

Algorithm 7 TTOC SystemC procedure

```
Ensure: time = 0
Ensure: connected = false
Ensure: execute = true
  Start TCP connection
  while true do
    if execute then
      if connected then
        Execute
        Obtain next event time
        Send time
        Wait co-simulation response
      end if
    else
      Wait co-simulation response
    end if
  end while
  Close TCP connection
```

integration of the TCP/IP protocol and the triggered execution of events, the simulation logic changes. Algorithm 7 outlines the newly developed procedure. First, the TCP/IP connection is established, and the procedure commences. The time of the next event is retrieved from the SystemC scheduler and sent to the co-simulation interface. If the response of the co-simulation is a Boolean value of 1, execution is triggered via `sc_start()`, and the time that was sent is passed as an argument. If the response from the controller is a value of 0, the SystemC thread halts until the interface sends a value of 1. If the TCP/IP connection is not established or if there are no further events to be processed, the socket will be closed and the procedure will conclude.

5.3 Simulation of Communication Network

Similar modifications that are performed to control the simulation of SystemC should also be done for OMNeT++. The difference is that instead of the TCP/IP protocol, C API calls and functions have been used. To recall Section 3.4.2, ENVIR is the module that controls the simulation. It needs a runnable ENVIR which can be instantiated as a user interface. For this reason, we have constructed a user interface based on Cmdenv and new functions that are added to OMNeT++ source files. In the previous approach, after the ENVIR had been built, the system automatically loaded the network description files and the configuration file and executed the events of the FES. This process is modified and divided into multiple functions representing different stages of the simulation. These stages include Setup_Omnet, Build_Network, and Execute_Simulation. Setup_Omnet creates a user interface that loads the configuration file, network description, and constructs an active simulation. Build_Network handles the creation of C objects of the simulation (e.g., simple, compound modules), and Execute_Simulation triggers the execution of events. In the new design, only one event is executed by the function call, like in TTOC, where events were executed only after the TCP/IP signal. In addition, to obtain the time of the event in the OMNeT++ scheduler, a new function is created.

5.3.1 Dynamic TSN simulation

OMNeT++ is an object-oriented framework designed for discrete-event network simulation. It consists of simple modules written in C++ that communicate through messages. These simple modules can be grouped together to create more complex structures known as compound modules. A network itself is classified as a compound module. OMNeT++ utilizes network description files to define both simple and compound modules. The parameters for these modules, whether simple or compound, are configured using a configuration file called the INI file. TSN, an IEEE 802.1 standard, enables time-sensitive data transmission within Ethernet networks. Key features include time synchronization (IEEE 802.1AS), stream redundancy, cut-through switching, per-stream filtering and policing (IEEE 802.1Qci), scheduling and traffic shaping (IEEE 802.1Qbv), frame preemption, frame replication and elimination (IEEE 802.1CB), as well as automated network configuration for fault tolerance and gate scheduling are supported in OMNeT++ via the INET framework [122]. The INET framework includes three network devices configured to implement TSN features: TSN PE, TSN Clock, and TSN Switch. Other modules like GateScheduleConfigurator or NetworkConfigurator operate offline and modify parameters of the three above mentioned devices, like transmission gate time or network IP addresses. That is why we have developed a new module called ReconfigurationConfigurator, as shown in Figure 5.7, that operates online in a predefined period that can be inserted in the parameters, and it modifies network nodes'

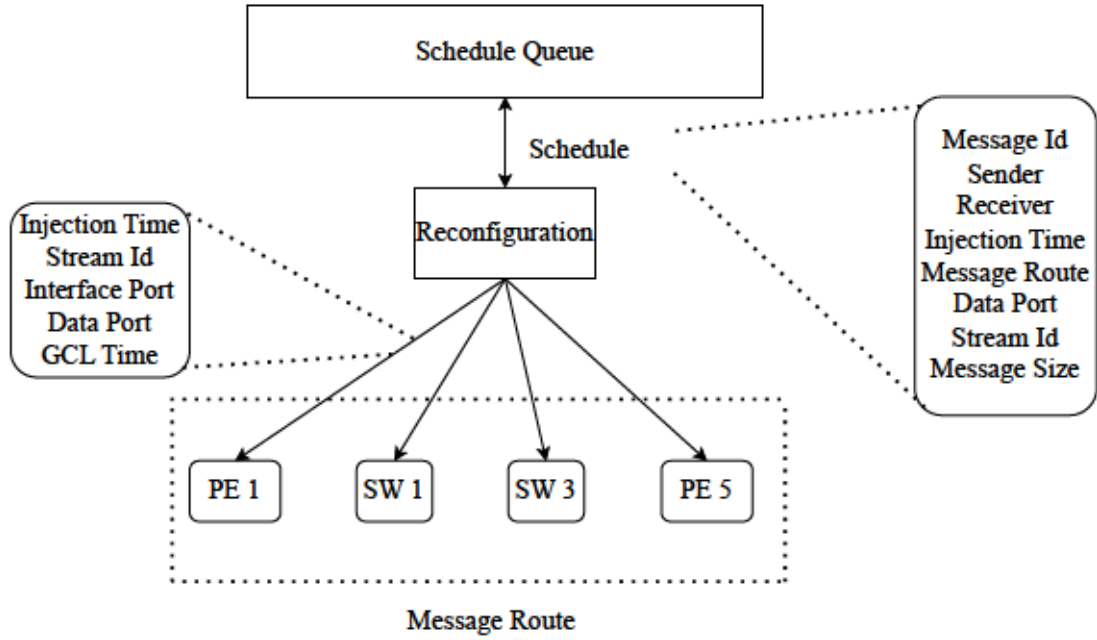


Figure 5.7: Reconfiguration Module in INET/OMNeT++

properties. Dynamic modifications of TSN include:

- Message construction
- Gate Control List for switches and nodes of the message route.
- Layer 2 services like stream handling and packet forwarding.

5.3.2 Message Construction

First let's take a look at the current design of the network nodes and their message construction. In the INET design, PEs utilize a generic application interface called IApp. Most examples are based on User Datagram Protocol (UDP) or Transmission Control Protocol (TCP), which do not allow any modification in the message construction (e.g., creation, send) time events with parameters inserted in the configuration statically. These events are then repeated periodically. Furthermore, in these messages, the application data is missing. In our approach, layer 3 of the communication network is unsuitable to use due to the missing receiver MAC address and Internet Protocol (IP) address. As mentioned above, IP addresses or destination addresses must be given before the simulation executes. That is why we have created a new TSN processing element that uses layer 2. We call this processing element TTOCTSN PE as shown in Figure 5.8. It is based on removing the layer 3 services and connecting the traffic generator directly with the layer 2. The Logical Link Control (LLC) has been used for

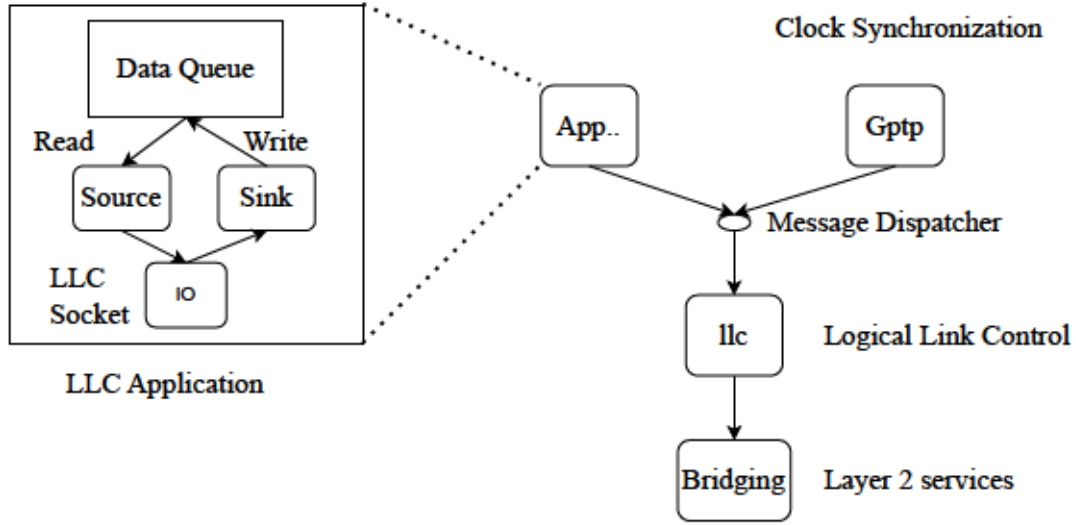


Figure 5.8: TTOC TSN network device in INET/OMNeT++

passing the application message from the traffic source to the link layer. For the sending and receiving packets, new source and sink files are created. The traffic generation must be bounded by the TTOC schedule for the creation and send time, but also for the destination address of the receiver.

In our simulation system, messages should be constructed during runtime after obtaining the communication schedule from the schedule queue. Each network node inside the reconfiguration module communicates with the queue and requests if there is a message present for that node. The structure of the message that is obtained is shown in Figure 5.7. The Reconfiguration's algorithm starts modifying the network nodes following the message route, with the first element of the route being the sender node. Injection time, streamId, data port, and message size are crucial for the creation of new events. Basically, we are telling the node to create a message sending event at this injection time, and this new message will be created by obtaining the data from the queue using the data port. StreamId and message size define other properties that are important for stream handling and the calculation of arrival time at the next switch.

5.3.3 Gate Control List

In the TSN network, computing nodes have predefined time slots for communication that are calculated from the schedule. GCL is responsible for determining the traffic (e.g., video, best-effort, time-triggered) that is allowed to be transmitted at a specific point in time within a certain cycle. The INET framework uses Periodic Gates as GCL for indicating the length of time when a specific entry will be active. As stated, in our simulation system, the communication schedule is obtained during runtime; the events that open or close the Periodic Gate cannot be initialized offline. Therefore, the module

is modified so that from the reconfiguration module, new events will be constructed that will open or close the switch transmission gate for the particular interface.

5.3.4 Layer 2 services

The layer 2 services in INET are implemented using Bridginglayer module. This module is responsible for packet forwarding, stream handling, interface selection, and virtual local area network handling. Bridging itself contains submodules like processingDelayLayer, directionReverser, streamIdentifier, streamRelay, streamFilter, streamCoder, interfaceRelay, and vlanPolicy. Modules in INET exchange information via tags, which are supplementary data added to the application packet. For example, incoming streams contain the streamInd tag, while in outgoing packet streamReq is present. Application packets identified by the stream name are replaced by the streamId and the tags are replaced by the streamTable as is shown in Figure 5.9.

The reconfiguration module obtains information about the outgoing interface port of that streamId and fills the table of the network node. StreamId is constructed from the source address plus Vlan. The reconfiguration algorithm utilizes the INET TSN NetworkConfigurator module to obtain switch or node connection links. If the end node of this link matches the next message route node id, the interface port id of the connection will be inserted in the StreamTable. When the message arrives at the Bridging Layer submodule, if the streamId matches the one in the table, it will be processed and forwarded to the corresponding interface port.

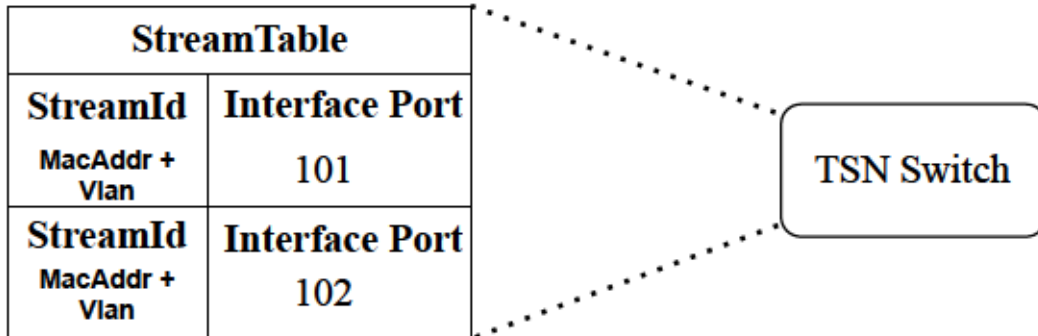


Figure 5.9: StreamTable of the TSN Switch

5.4 Simulation Controller

As illustrated in Figure 5.1, the SystemC simulator is linked to the co-simulation controller via the TCP/IP network protocol, where it provides the time of the next event, and the co-simulation controller sends the control signal. The control signal is a trigger command from the controller to the simulation that initiates the execution of the event. Consequently, the computation logic of the simulated PE must be modified. Figure 5.10 demonstrates the co-simulation computational logic.

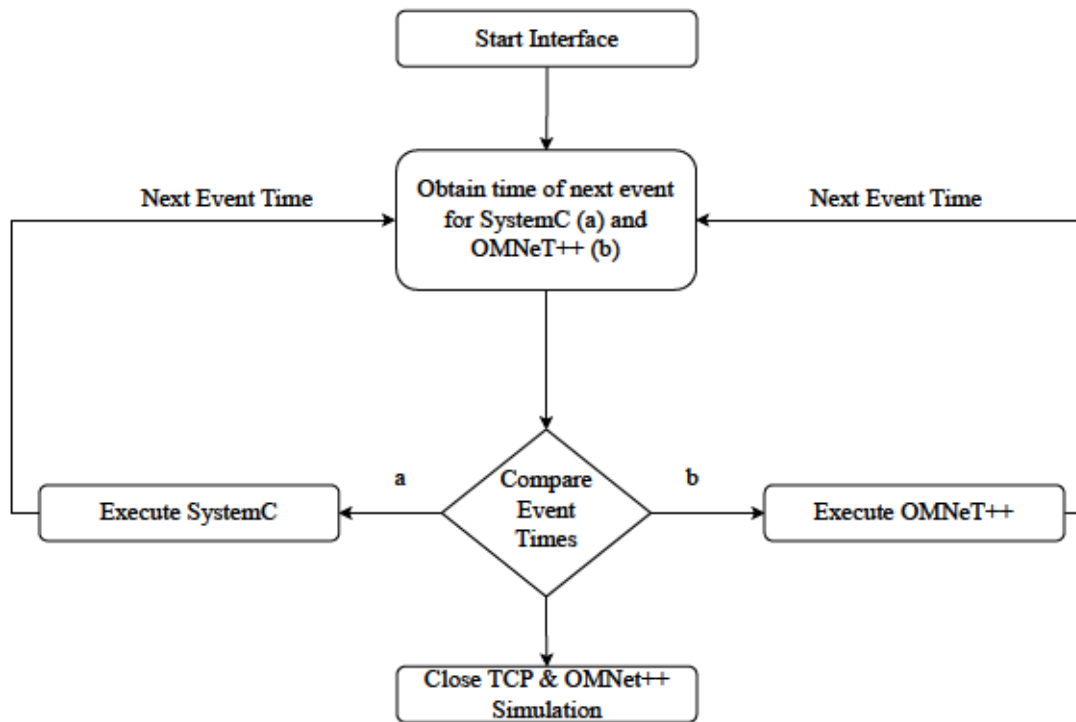


Figure 5.10: Co-simulation logic

5.5 Control and Data Plane

The data communication layer represents a critical component of the time-triggered organic computing simulation, facilitating a seamless exchange of application and configuration data among its various components. This layer ensures that tasks within the application and middleware layers can communicate effectively over the simulated network in OMNeT++. By utilizing a centralized Message Queue (MSQ), the data communication layer maintains data consistency and simplifies the management of communication between distributed tasks. Tasks in the application and middleware layers communicate over the simulated network in OMNeT++ by writing to and reading from the MSQ. Unlike the high-performance Remote Procedure Call (gRPC) [123] from Google, which employs Protocol Buffers for binary data interchange and is optimized for synchronous communication, we utilize JSON [124] for message encoding and data serialization due to its simplicity and human readability. Moreover, since the tasks that need to communicate are not required to run concurrently, our design relies exclusively on asynchronous communication through the MSQ. This approach aligns with the principles of asynchronous communication in distributed systems, where components interact without tight synchronization [125].

5.5.1 Message Queue (MSQ) System Design

The choice of a centralized MSQ is driven by the need for efficient data management and consistency across the TTOC simulator. A centralized approach simplifies the architecture by providing a single point for data exchange, reducing the complexity associated with multiple communication channels. This centralization ensures that all tasks interact with the same data repository, streamlining synchronization and coordination among tasks. A single instance of the MSQ serves all requests over the Transmission Control Protocol and stores data in memory. Figure 5.11 shows the MSQ architecture, where each task can have its own multiple named *send* and *receive* queues, where messages to be sent or received can be enqueued or dequeued, respectively. Further, each task can have multiple *read* and *write* sampling ports. Writing to this port overwrites previous data, and reading from the *read* sampling port returns a copy of the data.

Queues within the MSQ are created dynamically upon receiving an enqueue request, while read operations on non-existent or empty queues return an *empty* response, ensuring robust error handling. The MSQ enforces that all messages adhere to predefined JSON schemas. These schemas define the structure and data types expected in messages, ensuring consistency and facilitating validation. During the enqueue operation, the MSQ validates each message against its schema. If a message does not conform, it is rejected, and an error response is returned to the sending task. This validation prevents malformed or incorrect data from entering the communication system. Thus, the MSQ acts as an error boundary by isolating task failures. If a task produces invalid

messages or fails during execution, the MSQ prevents these issues from affecting other tasks by not forwarding erroneous data.

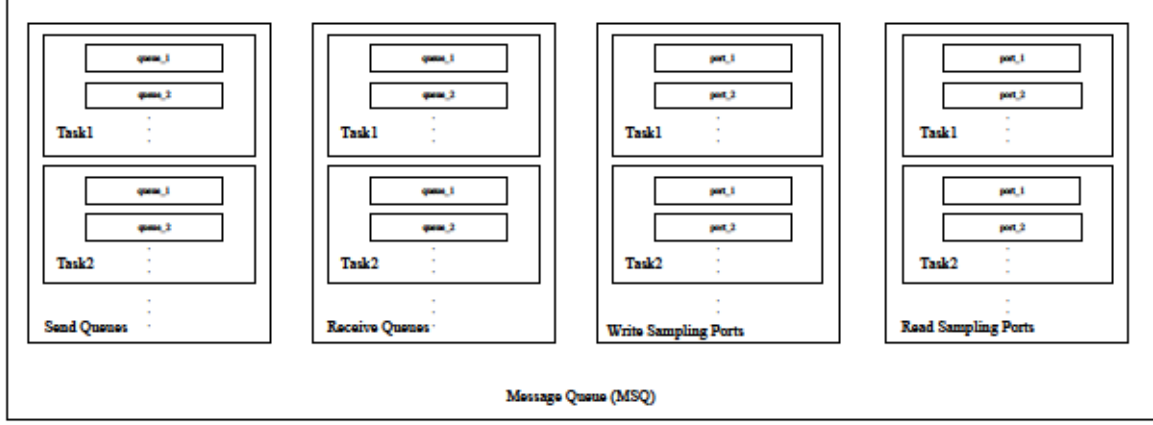


Figure 5.11: Message Queue Architecture

5.5.2 Message Flow Between Tasks

In the TTOC simulator, tasks do not need to run concurrently due to the time-triggered nature of the system. The global time base ensures that tasks execute at their designated times, and the MSQ facilitates communication without requiring simultaneous task execution. The communication mechanism in the TTOC simulator revolves around the MSQ. When a task needs to send a message, it writes the message to its designated *send* queue within the MSQ. OMNeT++ then reads this message from the *send* queue and simulates its transmission over the network. Upon reaching the recipient, the message is written to the recipient task's *receive* queue, from which the task can read and process it. Only OMNeT++ reads and writes to various tasks' queues; all other tasks interact solely with their respective queues. Tasks interact with the MSQ through their pre-configured ports. Write-configured ports allow tasks to enqueue messages into their *send* queues, while read-configured ports enable tasks to dequeue messages from their *receive* queues. That enables asynchronous inter-task communication, allowing tasks to operate without requiring concurrent execution.

OMNeT++ periodically checks the *send* queues of tasks for new messages. When a message is detected, OMNeT++ initiates the simulation of its transmission based on the schedule provided by the TTOC scheduler. The communication paths are defined by the TTOC scheduler and represent the routes that messages take through the simulated network. OMNeT++ uses these paths to simulate network conditions, including propagation delays and potential network congestion, providing a realistic model of message transmission. After simulating the transmission, OMNeT++ writes the message to the recipient task's *receive* queue in the MSQ. The recipient task can then read the message at its scheduled time. The synchronization between tasks is managed by

the TTOC scheduler and the MSQ. By adhering to the global time base, tasks can rely on messages being available in their *receive* queues when needed, eliminating the need for complex synchronization mechanisms.

Chapter 6

EVALUATION

6.1 Evaluation Goals

In this section, we are evaluating the proposed TTOC architecture with experiments that are conducted in the simulation environment. The goal of these use cases is to measure the system performance in the time-triggered domain as well as the self-x properties of organic computing. They address key features of TTOC like:

- Reliability: Does TTOC provide correct output over a specific period despite the occurrence of faults?
- Scalability: Does TTOC performance degrade in cases where the system size is increased?
- Fault Tolerance: What is the response of TTOC in case of a failure (babbling idiot), and how does it protect the system?
- Predictability: How consistent and deterministic is TTOC?
- Jitter: What is the variability of the system response time?
- Self-Configuration: How are the self-x properties functioning when mixed with scheduled communication?
- Dynamic Reconfiguration: How is the system reacting when application tasks are migrating, and how are the new schedules computed? How long does it take for the system to reconfigure, and how many hormone messages are needed?

To address the questions stated in the evaluation goals, multiple use cases have been evaluated using SystemC and OMNeT++ simulators in the simulation environment. They include application scenarios with various PEs and application messages that simulate different real-world scenarios. This will help us determine the scalability of TTOC and how the system's schedule calculation reacts to an increasing workload. In addition, in the simulation environment, we can test the TTOC when a PE has failed

and stopes working. This will help to understand the reconfiguration and reallocation of application tasks. In the OMNeT++ network simulator, we are able to test processing element malfunctioning. The babbling idiot use case will simulate the reaction of the system when a particular PE has failed silently or is sending untimed messages. Before executing these use cases, the simulation environment was tested to ensure a correct simulation behavior. The output of these experiments will help define potential benefits of integrating OC with scheduled communication as well as highlighting areas in which improvements are necessary.

6.2 Simulation Environment

For testing the simulation environment, we are focused on the simulated processing elements in the SystemC simulator, the message communication in TSN, and a simulation controller.

6.2.1 Simulated Processing Element

In the SystemC simulation, we have tested the scheduling algorithm alongside the integration of self-x properties with the time-triggered execution of application tasks. The range of application models varies to examples having up to forty tasks and a total of fifty messages. All the results, from the ordering of tasks, spatial and temporal allocations, and execution times of the dispatcher, are dumped into a log file.

6.2.2 OMNeT++ Network Simulator

In the network simulation, the focus has been to test the updated simulator regarding the newly implemented TSN features along with the extended functions of OMNeT++. These include:

1. New ENVIR module that connects with a co-simulation controller.
2. Reconfiguration Module.
3. New TSN processing element.
4. New TSN switch.
5. Layer 2 services like stream handling and packet forwarding.

6.2.3 Simulation Controller

In the co-simulation interface, we evaluate the controller functionality by testing if the interface obtains the execution times of both simulators and if it correctly sends the trigger signal for executing the events. Algorithm 8 specifies the procedure that is built for testing the co-simulation interface.

The first steps include declaring two boolean values (PEtimeflag, OMNeTtimeflag) as true, enabling the TCP connection, and activating the OMNeT++ simulation. After obtaining the next event time for both simulators, we compare the results to determine which event to execute. The event with the earliest time must be executed. In the case of SystemC execution, the OMNeTtimeflag becomes false as the value previously obtained remains the same, and a trigger signal is sent via TCP with the value 1. If OMNeT++ time is the earliest, PEtimeflag becomes false, and the OMNeT++ event

Algorithm 8 Controller procedure

Ensure: PEtimeflag = true

Ensure: OMNeTtimeflag = true

Start TCP connection

Start OMNeT++ simulation

while true **do**

if PEtimeflag **then**

 Obtain SystemC event time

end if

if OMNeTtimeflag **then**

 Obtain OMNeT event time

end if

 Compare event times (SystemC time and OMNeT time)

if PE **then**

 PEtimeflag = true

 OMNeTtimeflag = false

 Execute PE = 1

end if

if OMNeT **then**

 PEtimeflag = false

 OMNeTtimeflag = true

 Execute OMNeT = 1

 Execute PE = 0

end if

end while

Close TCP connection

Close OMNeT++ simulation

is executed. In addition, a TCP trigger with value 0 is sent to notify SystemC that the OMNeT++ simulation is running.

6.3 Scalability

When designing embedded systems, engineers should determine their scalability. Scalability relates to the ability of the system to handle growing size and complexity. We focused on the system performance as the number of application tasks increases. In TTOC, the artificial hormone system, when combined with cyclic schedules, allocates one application task per cycle. The increase in the workload increases the number of application tasks. Therefore, it will take more cycles for processing elements in TTOC to complete their schedule. In the simulation environment, we can test the scalability of the TTOC by utilizing different application models as examples of real-world scenarios. These scenarios include:

- In the first case, ADNA describes a system that contains three PEs and one switch, with each PE having a set of seven possible tasks. Each task holds hormone values of eager values, suppressors, accelerators, and load suppressors as shown

Table 6.1: Hormones for each possible task in the first case

Hormone type	Hormone value
Eager values	5
Suppressors	40
Load Suppressor	1
Accelerators	3
Offer Accelerator	2

in Table 6.1. All the results of system simulation (e.g., task allocations, number of cycles) are dumped into a log file.

- In the second case, ADNA represents a closed loop system which contains six processing elements and three switches with a total number of 25 possible application tasks per PE. The application model represents a closed control loop

Table 6.2: Hormones for each possible task in the second case

Hormone type	Hormone value
Eager values	10
Suppressors	80
Load Suppressor	2
Accelerators	6
Offer Accelerator	4

with resource names of sensors/actors like angle, speed, motor, temperature, and heating. In Table 6.3, the values of hormones are provided.

- In the third case, ADNA represents a vehicle network which contains fourteen processing elements and six switches with an application model of fifty tasks. TSN nodes include rear camera, front camera, wheel, engine actuator, lidar, steering, display, Onboard Unit (OBU), and Head-Up Display (HUD). In Table 6.3, the values of hormones used in this test case are provided. The values of the hormones were determined based on previous examples developed by University of Frankfurt.

Table 6.3: Hormones for each possible task in the third case

Hormone type	Hormone value
Eager values	20
Suppressors	160
Load Suppressor	4
Accelerators	12
Offer Accelerator	8

6.4 Fault Tolerance

In safety-relevant applications, the system must continue to provide a correct service despite the occurrence of faults. To evaluate the behavior of TTOC in case of a failure and how it protects the system, we have tested one of the most severe failure modes that occur, which is the babbling idiot failure.

6.4.1 Babbling Idiot Failure

In a babbling idiot failure, the processing element has not just failed, but it sends untimely corrupted messages to the communication network. This leads to disruption of communication also for the working processing elements. For the babbling idiot failure, we have designed a physical model and an application model similar to examples used in the automotive industry. The simulated processing elements from the SystemC simulator will start executing an initial schedule, exchange hormones for allocating application tasks, and calculate the schedule of the PEs. TSN processing elements in the network simulator will receive the communication schedules, but based on the physical model, we assume that one random PE is corrupt and it sends messages outside its predefined time slot for communication. For this reason, we have considered two scenarios.

1. In the first scenario, processing element 4 has been randomly chosen, and it has one application, as it can be seen from Figure 6.1. Application from the PE will send untimely messages every 400 microseconds. The gate control lists in the TSN network should block the messages from application 1 because they are outside the schedule.

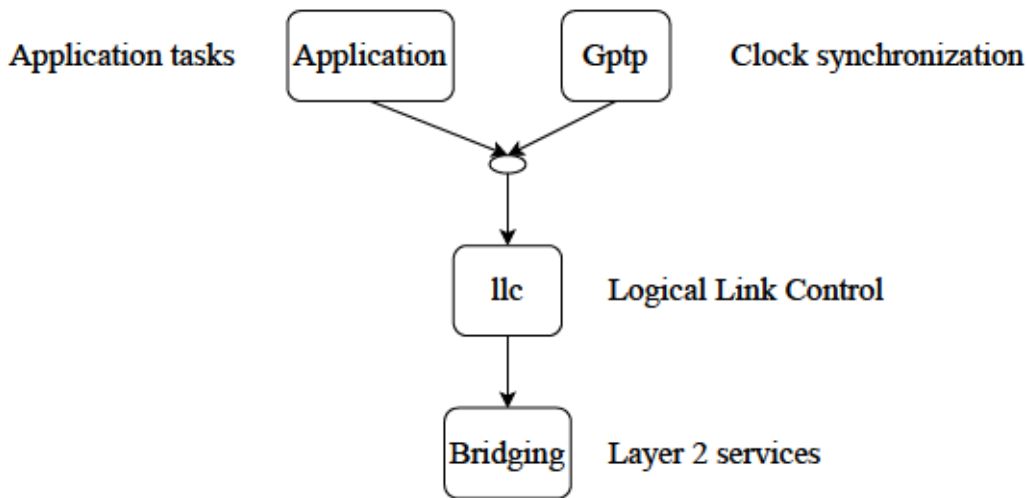


Figure 6.1: TSN PE in TTOC with one application

2. In the second scenario, a new application is added in the processing element as shown in Figure 6.2. The TSN PE 5 has been randomly chosen. Application 1 will read the schedule from the queue, create events, and send messages based on the TTOC schedule. On the other hand, application 2 will send untimely messages every 300 microseconds. The gate control lists in the TSN network should block the messages from application 2 because they are outside the schedule, and only messages from application 1 should proceed.

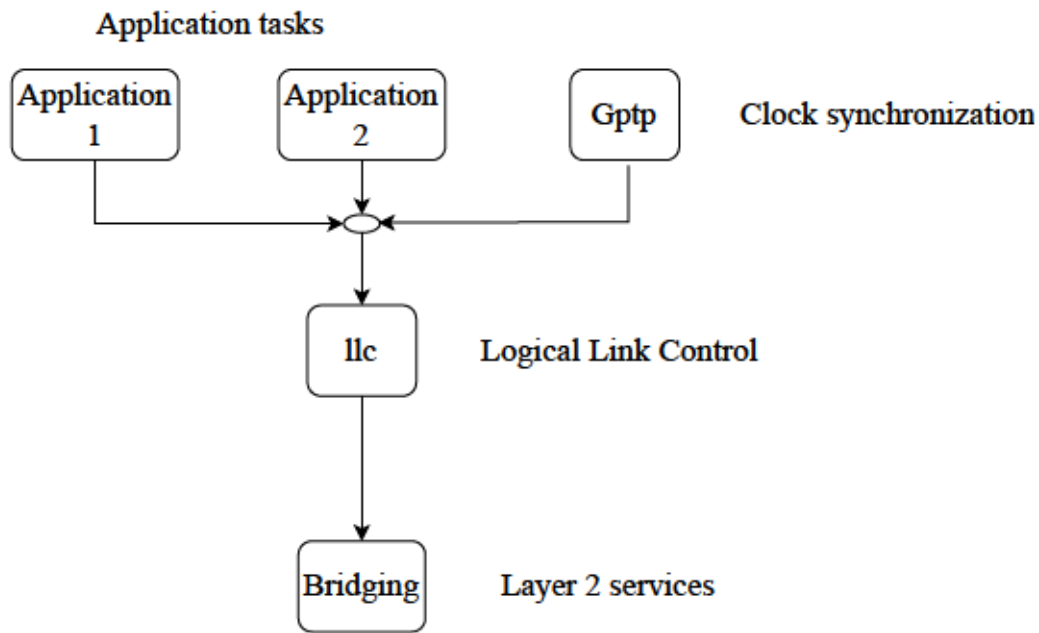


Figure 6.2: TSN PE in TTOC with two applications

6.5 Predictability

In real-time applications, predictability refers to the ability to bound maximum execution and communication times and jitter. Delay is the time needed for the message to arrive at the receiving processing element, and jitter refers to the variability of this delay. In time-sensitive networks, delay can cause:

- Communication failure due to the message not arriving in the assigned time slot.
- Reliability risks, especially in safety-critical systems like automotive, where timing errors can cause wrong control decisions.
- Delays can make the system less deterministic because of the clock drift caused by the message delay.
- Jitter can make the system less predictable.

Therefore, we have tested the message delay in TTOC.

6.5.1 Delay

In this use case, measurements of the TTOC delay values were performed on the network simulator. The application model is constructed of six processing elements and three TSN switches. The focus is on measuring and comparing the delay of the application messages. Therefore, we have taken two examples. First, we have utilized a time-triggered TSN network where the transmission time is controlled. Sec-

Table 6.4: Message Transmission Time

PE	Normal Message
1	0,0005 (s)
2	0,00025 (s)
3	0,000125 (s)
4	0,00025 (s)
5	0,0005 (s)
6	0,00025 (s)

ondly, we have modified TSN so that the transmission gates are opened during the whole simulation and have added another application that transmits messages without a periodic schedule. In OMNeT++, we can make the transmission gates open directly in the configuration file by writing the following command: `.ProcessingElement.eth[i].macLayer.queue.transmissionGate[*].initiallyOpen = true` In the configuration file, the generation of messages from the non-time-triggered applications

can be inserted as a time interval. In our example, we have inserted a random injection time for these applications. In Table 6.4, the transmission times of the messages in the system are shown.

6.6 Dynamic Reconfiguration

The self-organization and self-configuration capabilities of the AHS are tested in various scenarios. These scenarios include PE failure, task failure, PE task respawn, and

Table 6.5: ADNA Timed Event examples.

Timestamp	PE	Type
7	1	PE Failure
30	2	Respawn Tasks
50	3	Task Failure
150	4	Accelerator Value
200	5	Suppressor Value
500	6	Suppressor Value

increasing or decreasing hormone values. When a PE fails, its tasks are reallocated by the system. This functionality is evaluated by triggering timed events to simulate processing element failures. In AHS, a timestamp represents one organic computing cycle. In this use case, the failure of a PE 2 is simulated at the timestamp 7 as shown in Table 6.5. In the listing below, we have implemented the identification of a PE failure and its scheduled application tasks. The tasks are stored in a vector, and the remaining PEs will enter the reconfiguration stage. The affected application tasks will now serve as a new DAG graph and will be fed to the AHS hormone loop. The AHS and the scheduling algorithm will reallocate these tasks and create new schedule tables.

```

if (pProcessingElement->IsKilled())
{
    std::vector<int> failed_tasks;
    for (pos = pProcessingElement->GetAppTasksScheduled()->
        GetHeadPosition(); pos != NULL;)
    {
        pAppTask = (CApplicationTask*)pProcessingElement->
            GetAppTasksScheduled()->GetNext(pos);
        failed_tasks.push_back(pAppTask->GetId());
    }

    for (pos = pProcessingElement->GetOwnerList()->GetHeadPosition
        (); pos != NULL; )
    {
        CProcessingElement* PE = (CProcessingElement*)
            pProcessingElement->GetOwnerList()->GetNext(pos);
        if (PE->GetGridPosX() == pProcessingElement->GetGridPosX()
            && PE->GetGridPosY() == pProcessingElement->
            GetGridPosY())
        {
            continue;
        }
        else
        {
            PE->GetPeEvents()->RemoveHead();
            CPeEvent* reconfig_event = new CPeEvent(
                PE_RECONFIGURATION_STATE, PE_RECONFIGURATION_STATE)
                ;
            PE->GetPeEvents()->AddSorted(reconfig_event);
            PE->setReconfiguration(true);
            PE->FailedTasks(failed_tasks);
        }
    }

    break;
}

```

Listing 6.1: PE Reconfiguration State

Chapter 7

Result Analysis

In the evaluation part, the designed co-simulation controller, the combination of self-x properties with time-triggered concepts in the simulated processing elements, and message communication in the network simulator are tested. Furthermore, we have conducted experiments for the scalability, predictability, fault tolerance, and dynamic reconfiguration of TTOC. The physical and application models are described in ADNA files. In OMNeT++, the network description file is written based on the same physical models as in the TTOC simulator. As previously stated, the selected communication network is TSN, and the results of each component of the system (controller, SystemC, OMNeT++) are logged.

7.1 Simulation Environment

7.1.1 Simulated Processing Element

As stated in Section 6.2, all the results from the SystemC simulation are dumped into a log file. The log file contains information about the hormone exchange of AHS, allocation decisions, and task scheduling. It also gives insights about the task dispatcher module and the execution times of all operations inside the simulated processing element.

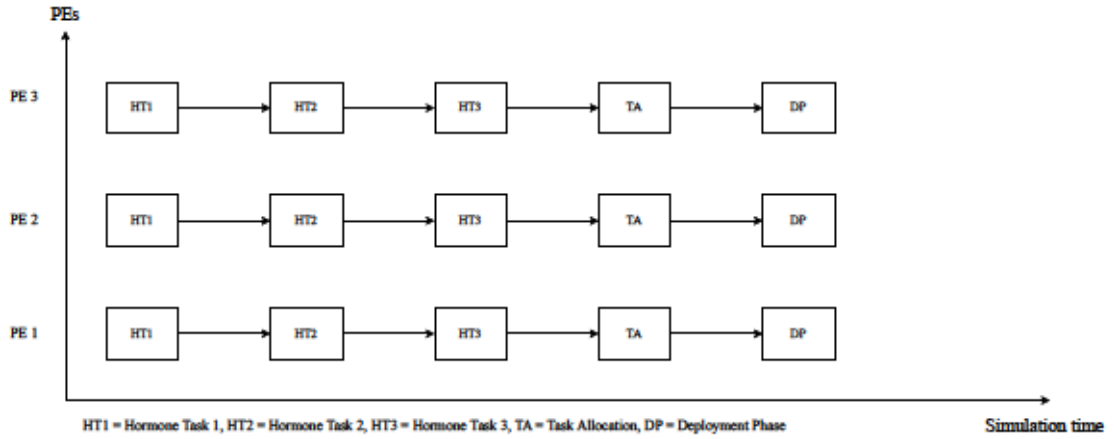


Figure 7.1: Execution flow in SystemC simulator

Picture 7.1 displays the execution flow of three processing elements in the first clock cycle. As it can be seen, initially, hormone tasks are executed, and then the task allocation phase follows. After the allocation decision is made, the deployment phase will update the schedule for the next clock cycle.

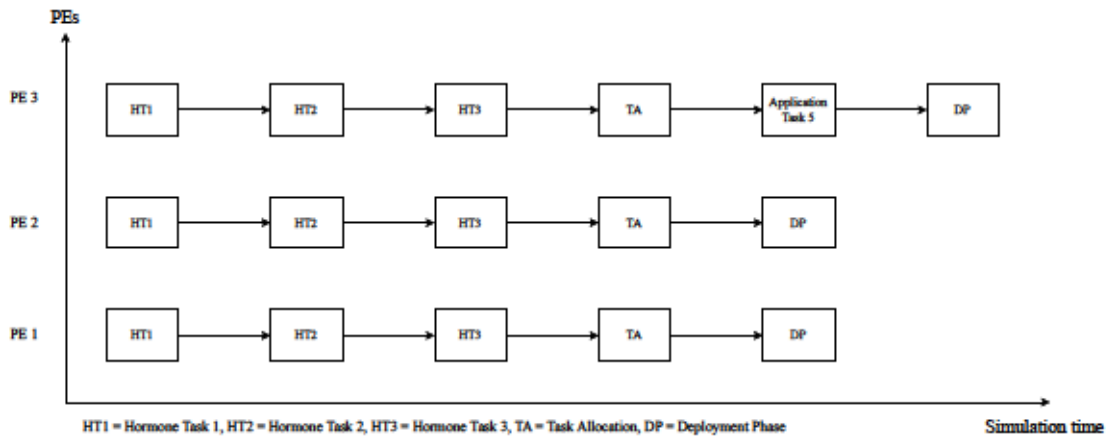


Figure 7.2: Execution flow in SystemC simulator

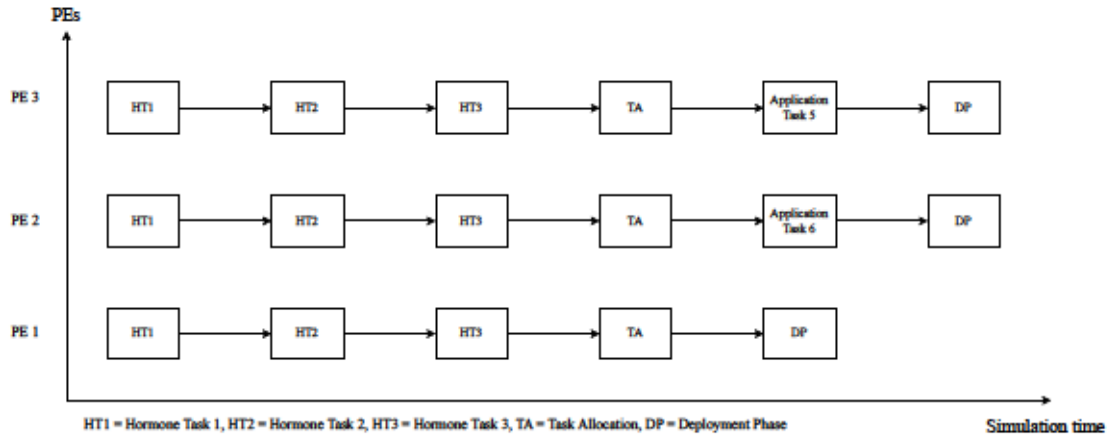


Figure 7.3: Execution flow in SystemC simulator

The artificial hormone system has allocated application task five to processing element three. The list-based scheduling algorithm calculates the temporal allocation of the application tasks. The time-triggered schedule has been updated in the deployment phase, and it now includes the execution of application tasks, as it can be seen in Picture 7.2.

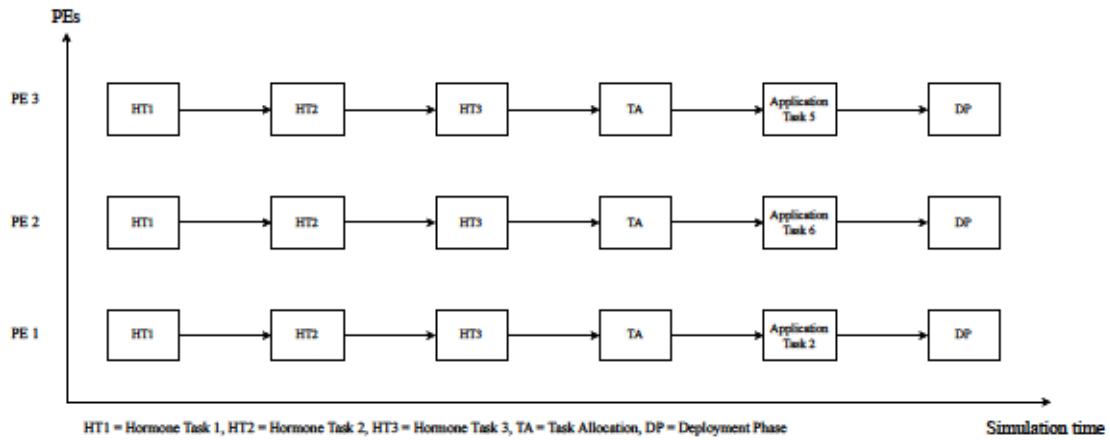


Figure 7.4: Execution flow in SystemC simulator

In addition, Pictures 7.3, 7.4 display the schedule after application tasks six and three have been allocated and inserted in the scheduling system of the PEs. This process follows the same logic for the remaining tasks, which are waiting to be allocated.

7.1.2 TSN Simulation

As previously stated, the TSN implementation in OMNeT++, as well as the network simulator functionalities, have been modified. From the results that are dumped in the log file, we can evaluate that:

- The new module in OMNeT++ functions as intended, and it allows for the simulation to be initialized from the co-simulation interface.
- Both message injection and forwarding information are loaded from the message queue by the reconfiguration module.
- TSN streams include stream identifiers.
- The TSN switch reads the identifier and forwards the packets to the respective ports as described in the forwarding table.
- TSN executes operations controlled by the given command from the controller.

7.1.3 Simulation Controller

An automated cross-checking of the co-simulation, the SystemC, and OMNeT++ log files is performed. In our analyses, we run various scenarios of ADNA files (e.g., different simulated physical and application models). The outcomes show that the co-simulation interface obtains correct event times and that both simulations perform the event executions from the trigger command of the co-simulation.

7.2 Scalability

To answer the question of how the TTOC performance reacts in cases where the system size increases, we tested different scenarios. Our focus lies on the reaction of the time-triggered schedule to the increasing number of application tasks. Analysing the log file, we can draw conclusions for each test case:

1. For the first case, scheduling all application tasks took a total of 8 schedule cycles. In total, it took 40 ms for the scheduling of whole tasks to be completed.
2. In the second case, scheduling all application tasks took 25 schedule cycles. In total, it took 125 ms for the scheduling of whole tasks to be completed.
3. In the last example, scheduling all application tasks took a total of 50 schedule cycles. For this scenario, it took 250 ms for the scheduling of all tasks to be completed.

In TTOC, the makespan of the system relates to the total number of tasks in the application model. This is due to the scheduling decision, which allocates one application task per hormone cycle dynamically. On the other hand, increasing the number of tasks leads to an increase in makespan but does not cause delays. In this system, the relationship between makespan and resource availability shows that performance scales effectively, maintaining reasonable execution times even as task complexity increases. This demonstrates that the system's scheduling and resource management are well-optimized for scalable operation.

7.3 Fault Tolerance

For the babbling idiot failure, we have performed experiments in the simulation environment and on the network simulator OMNeT++. Let us analyze the log file in order to evaluate the two selected scenarios.

1. In the first test case, the application reads the schedule from the message queue, creates message injection events, and transmits the messages based on the calculated communication schedule. The generated application messages are injected according to the calculated time plus the random message, which is periodic every 400 microseconds. All the messages have passed the first filter, the stream identifier, since the id in the stream tables matches the one inside the message. In the second filter, which is the transmission gate, the random messages every 400 microseconds have been dropped. They are dropped since there is no entry at the forwarding table that matches the time of this particular message.
2. In the second case, Application 1 reads the schedule from the message queue, creates message injection events, and transmits the messages based on the calculated communication schedule. Application 2 generates random messages, which are again periodic every 300 microseconds. Different from the first case, in this example, we can see from the log file that the application messages from Application 2 have not passed the stream identifier filter and have been blocked and dropped immediately. All the messages from Application 1 have passed the stream filter and the transmission gate and are forwarded to the correct ports.

From these experiments, we can demonstrate that in TTOC, the communication network is protected from the babbling idiot failure. The effects of this PE failure do not have an impact on the transmission of the messages from the processing elements that have not failed.

7.4 Predictability

In the third use case, measurements of the delay were performed on two different scenarios in the simulation environment. Table 7.1 shows the transmission time of the TT messages calculated and executed in the performed experiments.

Table 7.1: Message Transmission Time

PE	TT Message
1	0.001350 (s)
2	0.002500 (s)
3	0.08350 (s)
4	0.1250 (s)
5	0.15220 (s)
6	0.3350 (s)

- In the first use case, observing the log file, there is no collision between all the messages due to the filter on the transmission gate, which forwards only the messages that reside in the forwarding table. Therefore, there is a near constant delay on the receiving processing elements because all the nodes and switches abide by the time-triggered schedule.
- For the second case, where the transmission gates are opened, there is a collision between messages. This collision can be seen from Table 7.1 and the table of non-TT messages from Section 6.5. OMNeT++ uses a propagation delay, which is a static value that can be defined by the user for potential network congestion, providing a realistic model of message transmission. This delay can be calculated using the following Equation 7.1:

$$T_{\text{total}} = T_{\text{tx}} + T_{\text{prop}} = \frac{L}{R} + d \quad (7.1)$$

where, L = packet size (in bits), R = transmission data rate (in bps) and d = channel propagation delay (in seconds) In our examples, $L = 300$, $R = 100$ Mbps and $d = 5 \times 10^{-8}$ s. Based on the Equation 7.2 this yields:

$$\boxed{T_{\text{total}} = 3.05 \times 10^{-6} \text{ s} = 3.05 \text{ } \mu\text{s}} \quad (7.2)$$

Overall, TTOC offers better protection to the system against delays in the communication channel due to the predefined transmission times.

7.5 Dynamic Reconfiguration

The AHS self-organization and self-configuration capabilities under different failure and recovery scenarios are evaluated in the context where AHS is integrated into the time-triggered communication schedule. Analysing the log file, we observe that at the selected

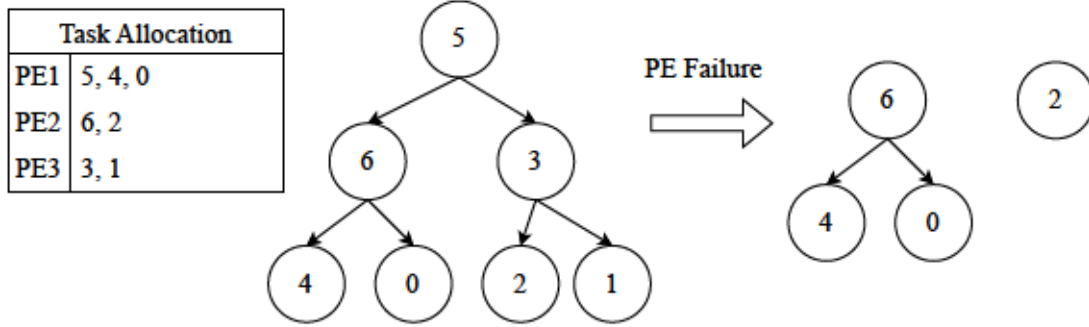


Figure 7.5: Application tasks affected by PE failure

time, the processing element is removed, and after this failure, the new graph with the affected tasks is constructed, as it is shown in Figure 7.5. The application tasks are allocated to active nodes, and new schedule tables are generated. The OC self-configuration property is maintained. The effects of TTOC reconfiguration can also be noticed in the TSN network that is performing the message communication and in the message queues that store the schedule for each PE. In this scenario, if a TTOC processing element is malfunctioning, there should not be messages sent or received in the simulation. Analysing the log file of OMNeT++, it revealed that processing element 2 has not received any schedule from the message queue. Furthermore, there was no message sent or received by the same PE in the network simulator. This demonstrates that PE 2 has failed and has stopped working.

Looking at the results of these experiments, we can show that in TTOC, the integration of AHS with the TT schedule has been successful. This new architecture maintains and exhibits the benefits of both systems.

Chapter 8

CONCLUSION AND FUTURE WORK

This chapter summarizes the content and results of this thesis. Furthermore, it describes the methods and techniques to improve the proposed architecture.

8.1 Conclusion

The Time-Triggered Organic Computing architecture improves the reliability, safety, and determinism of distributed embedded computer systems by combining the self-x properties of ADNA and AHS with time-triggered concepts. In the SystemC simulator, simulated processing elements following the initial schedule perform application task allocation based on AHS hormone exchange. After the spatial allocation, the list scheduling algorithm calculates the remaining schedule dynamically. For time-triggered message communication, a simulation environment based on OMNeT++ simulates a TSN network that supports these features. A co-simulation controller synchronizes the execution of the SystemC and OMNeT++ simulators.

Various scenarios were tested, including different physical and application models described in ADNA files. Our design implements the ADNA features like self-describing and self-building with the pre-defined schedule execution of the tasks, with the purpose of making embedded systems composed of OC more reliable, deterministic, and safe. These use cases include measurements of delay levels, scalability, dynamic reconfiguration, as well as injecting untimely messages from a faulty node in the network (e.g., babbling idiot failure). The simulation results show that the functionality of the middleware AHS remains the same even though the working principle is different. Furthermore, the TTOC architecture improves the temporal predictability of the system by having all the tasks executed at particular points in time. Simulation results showed that after the occurrence of a PE failure, the system entered the reconfiguration stage and the affected application tasks were rescheduled. In addition, TSN simulations demonstrated that messages were created during runtime and transmitted as defined

by the communication schedule.

8.2 Future Work

In this section, we provide possible future work to further extend TTOC. Two focus points are: scheduling of AHS messages and application model examples. The list algorithm performs the spatial and temporal allocation of application messages. It can be extended to include the scheduling of hormone communication, which is lacking at the moment. In addition, the simulation environment can involve and integrate an autonomous driving vehicle simulator (e.g., CARLA). The CARLA simulator will provide real-world application tasks, further testing, and validation of the TTOC architecture.

References

- [1] A. Malinowski and H. Yu, “Comparison of embedded system design for industrial applications”, *IEEE transactions on industrial informatics*, vol. 7, no. 2, pp. 244–254, 2011.
- [2] J. O. Kephart and D. M. Chess, “The vision of autonomic computing”, *Computer*, vol. 36, no. 1, pp. 41–50, 2003.
- [3] S. Tomforde, B. Sick, and C. Müller-Schloer, “Organic computing in the spotlight”, *arXiv preprint arXiv:1701.08125*, 2017.
- [4] H. Seebach, F. Ortmeier, and W. Reif, “Design and construction of organic computing systems”, in *2007 IEEE Congress on Evolutionary Computation*, IEEE, 2007, pp. 4215–4221.
- [5] U. Brinkschulte, “An artificial dna for self-describing and self-building embedded real-time systems”, *Concurrency and Computation: Practice and Experience*, vol. 28, no. 14, pp. 3711–3729, 2016.
- [6] U. Brinkschulte, M. Pacher, and A. Von Renteln, “Towards an artificial hormone system for self-organizing real-time task allocation”, in *IFIP International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, Springer, 2007, pp. 339–347.
- [7] R. Obermaisser, *Time-triggered communication*. CRC Press, 2018.
- [8] M. Qosja, S. Meckel, and R. Obermaisser, “Simulator for time-triggered organic computing”, *Procedia Computer Science*, vol. 220, pp. 127–134, 2023.
- [9] *Dfg schwerpunktprogramm 1183 organic computing*, last accessed: 2024-02-15. [Online]. Available: <https://gepris.dfg.de/gepris/projekt/5472210>.
- [10] C. Müller-Schloer, C. von der Malsburg, and R. P. Würt, “Organic computing”, *Informatik-Spektrum*, vol. 27, no. 4, pp. 332–336, 2004.
- [11] C. Müller-Schloer and B. Sick, “Emergence in organic computing systems: Discussion of a controversial concept”, in *International Conference on Autonomic and Trusted Computing*, Springer, 2006, pp. 1–16.
- [12] R. P. Würtz, “Organic computing methods for face recognition (methoden des organic computing zur gesichtserkennung)”, *it-Information Technology*, vol. 47, no. 4, pp. 207–211, 2005.

- [13] H. Schmeck, "Organic computing-a new vision for distributed embedded systems", in *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, IEEE, 2005, pp. 201–203.
- [14] A. Agarwal and B. Harrod, "Organic computing", *Self Aware Computing Concepts Paper*, <http://www.cag.csail.mit.edu/raw/documents/Agarwal-Harroddorganic-2006.pdf>, 2006.
- [15] C. Müller-Schloer, "Organic computing: On the feasibility of controlled emergence", in *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2004, pp. 2–5.
- [16] R. P. Würtz, *Organic computing*. Springer Science & Business Media, 2008.
- [17] C. Müller-Schloer and H. Schmeck, "Organic computing: A grand challenge for mastering complex systems", 2010.
- [18] C. Müller-Schloer, H. Schmeck, and T. Ungerer, *Organic computing—a paradigm shift for complex systems*. Springer Science & Business Media, 2011.
- [19] C. Müller-Schloer and S. Tomforde, *Organic Computing-Technical Systems for Survival in the Real World*. Springer, 2017, vol. 1.
- [20] J.-P. Steghöfer and W. Reif, "Die guten, die bösen und die vertrauenswürdigen—vertrauen im organic computing", *Informatik-Spektrum*, vol. 35, no. 2, pp. 119–131, 2012.
- [21] D. Merkle, M. Middendorf, and A. Scheidler, "Organic computing and swarm intelligence", in *Swarm Intelligence: Introduction and Applications*, Springer, 2008, pp. 253–281.
- [22] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, and H. Schmeck, "Towards a generic observer/controller architecture for organic computing", in *INFORMATIK 2006 – Informatik für Menschen, Band 1*, C. Hochberger and R. Liskowsky, Eds., Bonn: Gesellschaft für Informatik e.V., 2006, pp. 112–119.
- [23] J. Branke, M. Mnif, C. Müller-Schloer, H. Prothmann, U. Richter, F. Rochner, and H. Schmeck, "Organic computing—addressing complexity by controlled self-organization", in *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, IEEE, 2006, pp. 185–191.
- [24] B. Jakimovski, M. Litza, F. Msch, and A. e. Sayed Auf, "Development of an organic computing architecture for robot control", in *INFORMATIK 2006 – Informatik für Menschen, Band 1*, C. Hochberger and R. Liskowsky, Eds., Bonn: Gesellschaft für Informatik e.V., 2006, pp. 145–152.

- [25] F. Mösch, M. Litza, A. El Sayed Auf, E. Maehle, K. E. Großpietsch, and W. Brockmann, “Orca – towards an organic robotic control architecture”, in *Self-Organizing Systems*, H. de Meer and J. P. G. Sterbenz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 251–253, ISBN: 978-3-540-37669-9.
- [26] S. von Mammen, S. Tomforde, J. Höhner, P. Lehner, L. Förchner, A. Hiemer, M. Nicola, and P. Blickling, “Ocbotics: An organic computing approach to collaborative robotic swarms”, in *2014 IEEE Symposium on Swarm Intelligence*, IEEE, 2014, pp. 1–8.
- [27] M. Roth, J. Schmitt, R. Kiefhaber, F. Kluge, and T. Ungerer, “Organic computing middleware for ubiquitous environments”, in *Organic Computing—A Paradigm Shift for Complex Systems*, Springer, 2011, pp. 339–351.
- [28] U. Brinkschulte, M. Pacher, A. von Renteln, and B. Betting, “Organic real-time middleware”, in *Self-Organization in Embedded Real-Time Systems*. Springer, 2012, pp. 179–208.
- [29] U. Brinkschulte, M. Pacher, and A. v. Renteln, “An artificial hormone system for self-organizing real-time task allocation in organic middleware”, in *Organic Computing*, Springer, 2009, pp. 261–283.
- [30] R. Anthony, P. Ward, D. Chen, A. Rettberg, J. Hawthorne, M. Pelc, and M. Törngren, “A middleware approach to dynamically configurable automotive embedded systems”, May 2010. DOI: 10.4108/ICST.ISVCS2008.3551.
- [31] R. Anthony, D. Chen, M. Törngren, D. Scholle, M. Sanfridson, A. Rettberg, T. Naseer, M. Persson, and L. Feng, “Autonomic middleware for automotive embedded systems”, *Autonomic Communication*, pp. 169–210, 2009.
- [32] U. Brinkschulte and F. Fastnacht, “Applying the concept of artificial dna and hormone system to a low-performance automotive environment”, in *International Conference on Architecture of Computing Systems*, Springer, 2019, pp. 87–99.
- [33] U. Brinkschulte, E. Hutter, and F. Fastnacht, “Adapting the concept of artificial dna and hormone system to a classical autosar environment”, in *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, IEEE, 2019, pp. 35–42.
- [34] T. Kisselbach, S. Meckel, M. Pacher, U. Brinkschulte, and R. Obermaisser, “Organic computing to improve the dependability of an automotive environment”, in *International Conference on Architecture of Computing Systems*, Springer, 2022, pp. 211–225.
- [35] M. Brinkschulte, “Development of a vehicle simulator for the evaluation of a novel organic control unit concept”, in *INFORMATIK 2020*, Gesellschaft für Informatik, Bonn, 2021, pp. 883–890.

- [36] T. Kisselbach, P. Wörner, M. Pacher, and U. Brinkschulte, “An organic computing approach for carla simulator”, in *International Conference on Architecture of Computing Systems*, Springer, 2024, pp. 142–157.
- [37] G. Weiss, M. Zeller, D. Eilers, and R. Knorr, “Towards self-organization in automotive embedded systems”, in *International Conference on Autonomic and Trusted Computing*, Springer, 2009, pp. 32–46.
- [38] V. S. U. O. Computing, “Towards dependable unmanned aerial vehicle swarms using organic computing”, *Architecture of Computing Systems*, p. 311,
- [39] A. Bouajila, J. Zeppenfeld, W. Stechele, A. Herkersdorf, A. Bernauer, O. Bringmann, and W. Rosenstiel, “Organic computing at the system on chip level”, in *2006 IFIP International Conference on Very Large Scale Integration*, IEEE, 2006, pp. 338–341.
- [40] C. Schuck, S. Lamparth, and J. Becker, “Artnoc - a novel multi-functional router architecture for organic computing”, in *2007 International Conference on Field Programmable Logic and Applications*, 2007, pp. 371–376. DOI: 10.1109/FPL.2007.4380674.
- [41] M. Sommer, S. Tomforde, and J. Hähner, “An organic computing approach to resilient traffic management”, in *Autonomic Road Transport Support Systems*, Springer, 2016, pp. 113–130.
- [42] C. Gershenson and D. A. Rosenblueth, “Self-organizing traffic lights at multiple-street intersections”, *Complexity*, vol. 17, no. 4, pp. 23–39, 2012.
- [43] F. Dressler, *Self-organization in sensor and actor networks*. John Wiley & Sons, 2008.
- [44] H. Kopetz and G. Bauer, “The time-triggered architecture”, *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.
- [45] G. Heiner and T. Thurner, “Time-triggered architecture for safety-related distributed real-time systems in transportation systems”, in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 98CB36224)*, IEEE, 1998, pp. 402–407.
- [46] B. Hedenetz, “A development framework for ultra-dependable automotive systems based on a time-triggered architecture”, in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*, IEEE, 1998, pp. 358–367.
- [47] R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz, “The time-triggered system-on-a-chip architecture”, in *2008 IEEE International Symposium on Industrial Electronics*, IEEE, 2008, pp. 1941–1947.

- [48] K. Kim, M. Ishida, and J. Liu, "An efficient middleware architecture supporting time-triggered message-triggered objects and an nt-based implementation", in *Proceedings 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)(Cat. No. 99-61702)*, IEEE, 1999, pp. 54–63.
- [49] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer, "The time-triggered ethernet (tte) design", in *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, IEEE, 2005, pp. 22–33.
- [50] T. Fuehrer, B. Mueller, F. Hartwich, and R. Hugel, "Time triggered can (ttcan)", *SAE transactions*, pp. 143–149, 2001.
- [51] R. Makowitz and C. Temple, "Flexray-a communication network for automotive control systems", in *2006 IEEE International Workshop on Factory Communication Systems*, IEEE, vol. 27, 2006, pp. 207–212.
- [52] J. Farkas, L. L. Bello, and C. Gunther, "Time-sensitive networking standards", *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 20–21, 2018.
- [53] J. L. Messenger, "Time-sensitive networking: An introduction", *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 29–33, 2018.
- [54] K. Schild and J. Würtz, "Scheduling of time-triggered real-time systems", *Constraints*, vol. 5, no. 4, pp. 335–357, 2000.
- [55] T. Stüber, L. Osswald, S. Lindner, and M. Menth, "A survey of scheduling algorithms for the time-aware shaper in time-sensitive networking (tsn)", *Ieee Access*, vol. 11, pp. 61 192–61 233, 2023.
- [56] S. S. Craciunas, R. S. Oliver, M. Chmehk, and W. Steiner, "Scheduling real-time communication in iee 802.1 qbv time sensitive networks", in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, 2016, pp. 183–192.
- [57] P. Pop, M. L. Raagaard, S. S. Craciunas, and W. Steiner, "Design optimisation of cyber-physical distributed systems using iee time-sensitive networks", *IET Cyber-Physical Systems: Theory & Applications*, vol. 1, no. 1, pp. 86–94, 2016.
- [58] M. Pahlevan and R. Obermaisser, "Genetic algorithm for scheduling time-triggered traffic in time-sensitive networks", in *2018 IEEE 23rd international conference on emerging technologies and factory automation (ETFA)*, IEEE, vol. 1, 2018, pp. 337–344.
- [59] M. Yao, J. Liu, J. Du, D. Yan, Y. Zhang, W. Liu, and A. M.-C. So, "A unified flow scheduling method for time sensitive networks", *Computer Networks*, vol. 233, p. 109 847, 2023.

- [60] H. Chahed and A. Kassler, "Tsn network scheduling—challenges and approaches", *Network*, vol. 3, no. 4, pp. 585–624, 2023.
- [61] G. Patti, L. L. Bello, and L. Leonardi, "Deadline-aware online scheduling of tsn flows for automotive applications", *IEEE Transactions on Industrial Informatics*, vol. 19, no. 4, pp. 5774–5784, 2022.
- [62] M. Kim, J. Min, D. Hyeon, and J. Paek, "Tas scheduling for real-time forwarding of emergency event traffic in tsn", in *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, IEEE, 2020, pp. 1111–1113.
- [63] J.-P. Steghöfer, R. Kiefhaber, K. Leichtenstern, Y. Bernard, L. Klejnowski, W. Reif, T. Ungerer, E. André, J. Hähner, and C. Müller-Schloer, "Trustworthy organic computing systems: Challenges and perspectives", in *International Conference on Autonomic and Trusted Computing*, Springer, 2010, pp. 62–76.
- [64] H. Schmeck, C. Müller-Schloer, E. Çakar, M. Mnif, and U. Richter, "Adaptivity and self-organization in organic computing systems", *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 5, no. 3, pp. 1–32, 2010.
- [65] S. Tomforde, S. Rudolph, K. Bellman, and R. Würtz, "An organic computing perspective on self-improving system interweaving at runtime", in *2016 IEEE International Conference on Autonomic Computing (ICAC)*, IEEE, 2016, pp. 276–284.
- [66] S. Stein, A. Hamann, and R. Ernst, "Real-time property verification in organic computing systems", in *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, IEEE, 2006, pp. 192–197.
- [67] M. Güdemann, F. Nafz, W. Reif, and H. Seebach, "Towards safe and secure organic computing applications", 2006.
- [68] M. Güdemann, F. Ortmeier, and W. Reif, "Formal modeling and verification of systems with self-x properties", in *International Conference on Autonomic and Trusted Computing*, Springer, 2006, pp. 38–47.
- [69] —, "Safety and dependability analysis of self-adaptive systems", in *second international symposium on leveraging applications of formal methods, verification and validation (isola 2006)*, IEEE, 2006, pp. 177–184.
- [70] M. Gudemann, A. Angerer, F. Ortmeier, and W. Reif, "Modeling of self-adaptive systems with scade", in *2007 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2007, pp. 2922–2925.

- [71] F. Nafz, F. Ortmeier, H. Seebach, J.-P. Steghöfer, and W. Reif, "A universal self-organization mechanism for role-based organic computing systems", in *International Conference on Autonomic and Trusted Computing*, Springer, 2009, pp. 17–31.
- [72] F. Nafz, H. Seebach, J.-P. Steghöfer, S. Bäuml, and W. Reif, "A formal framework for compositional verification of organic computing systems", in *International Conference on Autonomic and Trusted Computing*, Springer, 2010, pp. 17–31.
- [73] M. Güdemann, F. Nafz, F. Ortmeier, H. Seebach, and W. Reif, "A specification and construction paradigm for organic computing systems", in *2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, IEEE, 2008, pp. 233–242.
- [74] S. Rudolph, R. Hihn, S. Tomforde, and J. Hähner, "Comparison of dependency measures for the detection of mutual influences in organic computing systems", in *International Conference on Architecture of Computing Systems*, Springer, 2016, pp. 334–347.
- [75] S. Nahar, S. Meckel, and R. Obermaisser, "Fault injection framework for organic computing architecture", in *Proceedings of SAI Intelligent Systems Conference*, Springer, 2023, pp. 701–717.
- [76] S. Nahar, U. Raj, S. Meckel, and R. Obermaisser, "Application level fault diagnosis in organic computing using stress hormone propagation", in *2024 8th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, IEEE, 2024, pp. 1–6.
- [77] N. Msadek, R. Kiefhaber, and T. Ungerer, "A trustworthy, fault-tolerant and scalable self-configuration algorithm for organic computing systems", *Journal of Systems Architecture*, vol. 61, no. 10, pp. 511–519, 2015, Special section on Architecture of Computing Systems edited by Editors: Wolfgang Karl, Erik Maehle, Kay Römer, Eduardo Tovar, Martin Danek Special section on Testing, Prototyping, and Debugging of Multi-Core Architectures edited by Editors: Frank Hannig Andreas Herkersdorf Special section on Embedded Vision Architectures and Applications edited by Editors: Christophe Bobda, Walter Stechele, Ali Ahmadiania and Miaoqing Huang, ISSN: 1383-7621. DOI: <https://doi.org/10.1016/j.sysarc.2015.07.012>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S138376211500082X>.
- [78] R. Obermaisser, H. Ahmadian, A. Maleki, Y. Bebawy, A. Lenz, and B. Sorkhpour, "Adaptive time-triggered multi-core architecture", *Designs*, vol. 3, no. 1, p. 7, 2019.

- [79] P. Muoka, D. Onwuchekwa, and R. Obermaisser, "Adaptive scheduling for time-triggered network-on-chip-based multi-core architecture using genetic algorithm", *Electronics*, vol. 11, no. 1, p. 49, 2021.
- [80] D. Onwuchekwa, O. Hekal, and R. Obermaisser, "Scheduling the exchange of context information for time-triggered adaptive systems", *Algorithms*, vol. 18, no. 8, p. 456, 2025.
- [81] C. Lua, D. Onwuchekwa, and R. Obermaisser, "Ai-based scheduling for adaptive time-triggered networks", in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, IEEE, 2022, pp. 1–7.
- [82] S. Alshaer, C. Lua, P. Muoka, D. Onwuchekwa, and R. Obermaisser, "Graph neural networks based meta-scheduling in adaptive time-triggered systems", in *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2022, pp. 1–6.
- [83] H. Ahmadian, F. Nekouei, and R. Obermaisser, "Fault recovery and adaptation in time-triggered networks-on-chips for mixed-criticality systems", in *2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, IEEE, 2017, pp. 1–8.
- [84] F. Sagstetter, S. Andalam, P. Waszecki, M. Lukasiewicz, H. Stähle, S. Chakraborty, and A. Knoll, "Schedule integration framework for time-triggered automotive architectures", in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.
- [85] P. Pedreiras and L. Almeida, "The flexible time-triggered (ftt) paradigm: An approach to qos management in distributed real-time systems", in *Proceedings International Parallel and Distributed Processing Symposium*, IEEE, 2003, 9–pp.
- [86] A. A. Syed, S. Ayaz, T. Leinmüller, and M. Chandra, "Dynamic scheduling and routing for tsn based in-vehicle networks", in *2021 IEEE International Conference on Communications Workshops (ICC Workshops)*, IEEE, 2021, pp. 1–6.
- [87] M. L. Raagaard, P. Pop, M. Gutiérrez, and W. Steiner, "Runtime reconfiguration of time-sensitive networking (tsn) schedules for fog computing", in *2017 IEEE Fog World Congress (FWC)*, IEEE, 2017, pp. 1–6.
- [88] M. Gutiérrez, A. Ademaj, W. Steiner, R. Dobrin, and S. Punnekkat, "Self-configuration of ieee 802.1 tsn networks", in *2017 22nd IEEE international conference on emerging technologies and factory automation (ETFA)*, IEEE, 2017, pp. 1–8.

- [89] M. Pahlevan, J. Schmeck, and R. Obermaisser, "Evaluation of tsn dynamic configuration model for safety-critical applications", in *2019 ieee intl conf on parallel & distributed processing with applications, big data & cloud computing, sustainable computing & communications, social computing & networking (ispa/bdcloud/socialcom/sustaincom)*, IEEE, 2019, pp. 566–571.
- [90] A. A. Syed, S. Ayaz, T. Leinmüller, and M. Chandra, "Fault-tolerant dynamic scheduling and routing for tsn based in-vehicle networks", in *2021 IEEE Vehicular Networking Conference (VNC)*, IEEE, 2021, pp. 72–75.
- [91] H. Chen, M. Liu, J. Huang, Z. Zheng, W. Huang, and Y. Xiao, "A heuristic-based dynamic scheduling and routing method for industrial tsn networks", in *2023 IEEE 10th International Conference on Cyber Security and Cloud Computing (CSCloud)/2023 IEEE 9th International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, IEEE, 2023, pp. 440–445.
- [92] H. Li, T. Zhang, and K. Zhu, "Dynamic slot extension based high-criticality tasks scheduling in tsn-based dmcs", *IEEE Internet of Things Journal*, 2025.
- [93] C. Gärtner, A. Rizk, B. Koldehofe, R. Guillaume, R. Kundel, and R. Steinmetz, "Fast incremental reconfiguration of dynamic time-sensitive networks at runtime", *Computer Networks*, vol. 224, p. 109 606, 2023.
- [94] J. Haxhibeqiri, X. Jiao, P. A. Campos, I. Moerman, and J. Hoebeke, "To update or not: Dynamic traffic classification for high priority traffic in wireless tsn", in *2023 IEEE 19th International Conference on Factory Communication Systems (WFCS)*, IEEE, 2023, pp. 1–8.
- [95] H. Tianfield and R. Unland, "Towards autonomic computing systems", *Engineering Applications of Artificial Intelligence*, vol. 17, no. 7, pp. 689–699, 2004, Autonomic Computing Systems, ISSN: 0952-1976. DOI: <https://doi.org/10.1016/j.engappai.2004.08.029>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0952197604001137>.
- [96] S. Tomforde, H. Prothmann, J. Branke, J. Hähner, M. Mnif, C. Müller-Schloer, U. Richter, and H. Schmeck, "Observation and control of organic systems", *Organic computing—a paradigm shift for complex systems*, pp. 325–338, 2011.
- [97] U. Brinkschulte, "Technical report: Artificial dna—a concept for self-building embedded systems", *arXiv preprint arXiv:1707.07617*, 2017.
- [98] J. Wang, *Real-time embedded systems*. John Wiley & Sons, 2017.
- [99] H. Kopetz, *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.
- [100] G. Bernat, A. Burns, and A. Liamosi, "Weakly hard real-time systems", *IEEE transactions on Computers*, vol. 50, no. 4, pp. 308–321, 2001.

- [101] X. Evers, W. Csg, R. B. CSG, I. Herschberg, D. Epema, and J. De Jongh, "A literature study on scheduling in distributed systems", *Delft University of Technology*, 1992.
- [102] M. Kashef and N. Moayeri, "Real-time scheduling for wireless networks with random deadlines", in *2017 IEEE 13th International Workshop on Factory Communication Systems (WFCS)*, IEEE, 2017, pp. 1–9.
- [103] H. Gholami and R. Zakerian, "A list-based heuristic algorithm for static task scheduling in heterogeneous distributed computing systems", in *2020 6th international conference on web research (ICWR)*, IEEE, 2020, pp. 21–26.
- [104] P. Marti, J. M. Fuertes, G. Fohler, and K. Ramamritham, "Jitter compensation for real-time control systems", in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*, IEEE, 2001, pp. 39–48.
- [105] J. J. G. Garcia and M. G. Harbour, "Minimizing the effects of jitter in distributed hard real-time systems", *Journal of systems architecture*, vol. 42, no. 6–7, pp. 431–447, 1996.
- [106] J. C. Knight, "Safety critical systems: Challenges and directions", in *Proceedings of the 24th international conference on software engineering*, 2002, pp. 547–550.
- [107] M. Rausand, *Reliability of safety-critical systems: theory and applications*. John Wiley & Sons, 2014.
- [108] R. Obermaisser, *Event-triggered and time-triggered control paradigms*. Springer Science & Business Media, 2004, vol. 22.
- [109] M. Van Steen and A. S. Tanenbaum, *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.
- [110] H. Kopetz and W. Ochsenreiter, "Clock synchronization in distributed real-time systems", *IEEE transactions on computers*, vol. 100, no. 8, pp. 933–940, 2009.
- [111] F. Reghenzani, Z. Guo, and W. Fornaciari, "Software fault tolerance in real-time systems: Identifying the future research questions", *ACM Computing Surveys*, vol. 55, no. 14s, pp. 1–30, 2023.
- [112] M. Grottke and K. S. Trivedi, "A classification of software faults", *Journal of Reliability Engineering Association of Japan*, vol. 27, no. 7, pp. 425–438, 2005.
- [113] A. Kumar, R. S. Yadav, and A. J. Ranvijay, "Fault tolerance in real time distributed system", *International Journal on Computer Science and Engineering*, vol. 3, no. 2, pp. 933–939, 2011.
- [114] T. Arifeen, A. S. Hassan, and J.-A. Lee, "Approximate triple modular redundancy: A survey", *IEEE Access*, vol. 8, pp. 139 851–139 867, 2020.
- [115] P. Verissimo and L. Rodrigues, *Distributed systems for system architects*. Springer Science & Business Media, 2012, vol. 1.

- [116] N. G. Nayak, F. Dürr, and K. Rothermel, “Time-sensitive software-defined network (tssdn) for real-time applications”, in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, 2016, pp. 193–202.
- [117] U. W. Pooch and J. A. Wall, *Discrete event simulation: a practical approach*. CRC press, 2024.
- [118] P. R. Panda, “Systemc: A modeling platform supporting multiple design abstractions”, in *Proceedings of the 14th international symposium on Systems synthesis*, 2001, pp. 75–80.
- [119] A. Varga, “Omnet++”, in *Modeling and tools for network simulation*, Springer, 2010, pp. 35–59.
- [120] *Omnet++ homepage*, last accessed: 2024-05-24. [Online]. Available: <https://doc.omnetpp.org/omnetpp/manual/>.
- [121] —, “A practical introduction to the omnet++ simulation framework”, in *Recent advances in network simulation: the OMNeT++ environment and its ecosystem*, Springer, 2019, pp. 3–51.
- [122] L. Mészáros, A. Varga, and M. Kirsche, “Inet framework”, *Recent Advances in Network Simulation: The OMNeT++ Environment and its Ecosystem*, pp. 55–106, 2019.
- [123] G. LLC, *Grpc: A high performance, open-source universal rpc framework*, <https://grpc.io/>, Version 1.57.0, 2024.
- [124] T. Bray, *The javascript object notation (json) data interchange format*, <https://www.rfc-editor.org/rfc/rfc8259>, RFC 8259, 2017.
- [125] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th. Addison-Wesley, 2011.

Appendix A

TTOC Simulation

```
void task_dispatcher::Initialize_Thread()
{
    // variable declaration
    char          str[LINE_LENGTH];
    CHormoneTask* pHTasks;
    CApplicationTask* pAppTask;
    CApplicationTaskList* pAppTaskList;
    Schedule_Node * node;
    int operation;
    double delay;
    bool type = false;
    CProcessingElement* pProcessingElement;
    CPeEvent* pEvent;
    POSITION pos, pos2;
    double period = 0;

    // Take the object of Processing Element to be runned by SystemC
    thread
    for (int i = 0; i < nr_pe; i++)
    {

        pos = pe_list.GetHeadPosition();
        pProcessingElement = (CProcessingElement*)pe_list.GetNext(pos)
        ;
        std::cout << "Thread_of_PE_" << pProcessingElement->
            GetGridPosX() << "_" << pProcessingElement->GetGridPosY()
            << std::endl;
        pe_list.RemoveHead();
        nr_pe--;
        sc_core::wait(0, SC_US);
    }
    // Obtain the OC tasks
    pHTasks = (CHormoneTask*)pProcessingElement->GetHormoneTasks()->
        GetHead();
}
```

```

// Execution loop
while (true)
{
    // get head of the list of the PE events
    if (pProcessingElement->GetPeEvents()->IsEmpty()) break;
    pEvent = (CPeEvent*)pProcessingElement->GetPeEvents()->GetHead
        ();
    if (pEvent == NULL) break; // No state

    // This condition is to simulate cases when PE has failed.
    // Utilizing Timed Events, a PE is failed at specific time
    if (pProcessingElement->IsKilled())
    {
        // In case PE has failed identify affected tasks and their
        // children
        std::vector<int> failed_tasks;
        for (pos = pProcessingElement->GetAppTasksScheduled()->
            GetHeadPosition(); pos != NULL;)
        {
            pAppTask = (CApplicationTask*)pProcessingElement->
                GetAppTasksScheduled()->GetNext(pos);
            failed_tasks.push_back(pAppTask->GetId());
        }
        //CProcessingElementList* m_OwnerListPtr = pProcessingElement
        //->GetOwnerList();
        for (pos = pProcessingElement->GetOwnerList()->GetHeadPosition
            (); pos != NULL; )
        {
            CProcessingElement* PE = (CProcessingElement*)
                pProcessingElement->GetOwnerList()->GetNext(pos);
            if (PE->GetGridPosX() == pProcessingElement->GetGridPosX()
                && PE->GetGridPosY() == pProcessingElement->
                GetGridPosY())
            {
                continue;
            }
            else
            {
                // Switch the other PE to reconfiguration state, and
                // update schedule.
                PE->GetPeEvents()->RemoveHead();
                CPeEvent* reconfig_event = new CPeEvent(
                    PE_RECONFIGURATION_STATE, PE_RECONFIGURATION_STATE)
                    ;
                PE->GetPeEvents()->AddSorted(reconfig_event);
                PE->setReconfiguration(true);
            }
        }
    }
}

```

```

        PE->FailedTasks(failed_tasks);
    }

}

break;

// Update period
if (sc_core::sc_time_stamp().to_double() == period)
{
    period += pProcessingElement->getSchedulePeriod();// 100
    previous
}

switch (pEvent->GetEvent())
{
    // Reconfiruation stage
    case PE_RECONFIGURATION_STATE: if (pProcessingElement->
        getReconfiguration())
    {
        // Obtain the schedule of reconfiguration state
        Schedule_Node* reconfiguration_event =
            pProcessingElement->GetReconfigurationScheduleList
            ()->returnHead();
        operation = reconfiguration_event->id;
        delay = reconfiguration_event->end_time -
            reconfiguration_event->start_time;

        if (operation == Calendar_HT1 && sc_core::
            sc_time_stamp().to_double() ==
            reconfiguration_event->start_time)
        {
            // Output results in log file
            if (theProperties.GetLogLevel() >= 3)
            {
                sprintf(str, LOG_HT1, pProcessingElement->
                    GetGridPosX(), pProcessingElement->
                    GetGridPosY(), sc_core::sc_time_stamp().
                    to_double());
                pProcessingElement->GetLogFile()->WriteLine(
                    str, TRUE);
            }
            pHTasks->HT1(pProcessingElement);
            pProcessingElement->
                GetReconfigurationScheduleList()->delete_head
                ();
        }
    }
}

```

```

        // Execute HT1 and wait the other threads to
        // execute their HT1
        sc_core::wait(delay, SC_US);

    }
    else if (operation == Calendar_HT2 && sc_core::
        sc_time_stamp().to_double() ==
        reconfiguration_event->start_time)
    {
        // Output results in log file
        if (theProperties.GetLogLevel() >= 3)
        {
            sprintf(str, LOG_HT2, pProcessingElement->
                GetGridPosX(), pProcessingElement->
                GetGridPosY(), sc_core::sc_time_stamp().
                to_double());
            pProcessingElement->GetLogFile()->WriteLine(
                str, TRUE);
        }
        pHTasks->HT2(pProcessingElement);
        pProcessingElement->
            GetReconfigurationScheduleList()->delete_head
            ();
        sc_core::wait(delay, SC_US);
    }
    else if (operation == Calendar_HT3 && sc_core::
        sc_time_stamp().to_double() ==
        reconfiguration_event->start_time)
    {
        // Output results in log file
        if (theProperties.GetLogLevel() >= 3)
        {
            sprintf(str, LOG_HT3, pProcessingElement->
                GetGridPosX(), pProcessingElement->
                GetGridPosY(), sc_core::sc_time_stamp().
                to_double());
            pProcessingElement->GetLogFile()->WriteLine(
                str, TRUE);
        }
        pHTasks->HT3(pProcessingElement);
        pProcessingElement->
            GetReconfigurationScheduleList()->delete_head
            ();
        sc_core::wait(delay, SC_US);
    }
    // TRA -> Task ReAllocation
    else if (operation == Calendar_TRA && sc_core::
        sc_time_stamp().to_double() ==

```

```

reconfiguration_event->start_time)
{
    // Output results in log file
    if (theProperties.GetLogLevel() >= 3)
    {
        sprintf(str, LOG_TA, pProcessingElement->
            GetGridPosX(), pProcessingElement->
            GetGridPosY(), sc_core::sc_time_stamp().
            to_double());
        pProcessingElement->GetLogFile()->WriteLine(
            str, TRUE);
    }
    // Make type 1 in case allocation is successful
    type = pHTasks->ReAllocateTask(pProcessingElement
        );
    pProcessingElement->
        GetReconfigurationScheduleList()->delete_head
        ();
    sc_core::wait(delay, SC_US);
}
else if (operation == Task_Schedule && sc_core::
    sc_time_stamp().to_double() ==
    reconfiguration_event->start_time)
{
    if (type == 1)
    {
        // Activate List Scheduling Algorithm
        pProcessingElement->list_scheduling(
            pProcessingElement->scheduled_task(),
            pProcessingElement->GetPEId());
        // Wait till all the other threads finish
        sc_core::wait(delay, SC_US);
        pProcessingElement->
            GetReconfigurationScheduleList()->
            delete_head();
    }
    else
    {
        // Wait till all the other threads finish
        sc_core::wait(delay, SC_US);
        pProcessingElement->
            GetReconfigurationScheduleList()->
            delete_head();
    }
}
else if (operation == ApplicationTask && sc_core::
    sc_time_stamp().to_double() ==

```

```

reconfiguration_event->start_time)
{
    // Execute Application tasks
    pAppTask = (CApplicationTask*)pProcessingElement
        ->GetAppTasks()->FindTask(
            reconfiguration_event->job_id);
    // Output results in log file
    if (theProperties.GetLogLevel() >= 3)
    {
        sprintf(str, LOG_TASKEVENT, pAppTask->GetId()
            , pProcessingElement->GetGridPosX(),
            pProcessingElement->GetGridPosY(), sc_core
                ::sc_time_stamp().to_double());
        pProcessingElement->GetLogFile()->WriteLine(
            str, TRUE);
    }
    pAppTask->Execute(pProcessingElement);
    pProcessingElement->
        GetReconfigurationScheduleList()->delete_head
            ();
    // Wait till all the other threads finish
    sc_core::wait(delay, SC_US);
}
else if (operation == Deployment_Task && sc_core::
    sc_time_stamp().to_double() ==
    reconfiguration_event->start_time)
{
    // Create the new Reconfiguration schedule for
    next period

    node = new Schedule_Node(Calendar_HT1, period,
        period + pProcessingElement->getHT1(), 0, 0,
        0);
    pProcessingElement->
        GetReconfigurationScheduleList()->append(node)
        ;
    node = new Schedule_Node(Calendar_HT2, period +
        pProcessingElement->getHT1(), period +
        pProcessingElement->getHT2(), 0, 0, 0);
    pProcessingElement->
        GetReconfigurationScheduleList()->append(node)
        ;
    node = new Schedule_Node(Calendar_HT3, period +
        pProcessingElement->getHT2(), period +
        pProcessingElement->getHT3(), 0, 0, 0);
    pProcessingElement->
        GetReconfigurationScheduleList()->append(node)

```

```

;
node = new Schedule_Node(Calendar_RTA, period +
    pProcessingElement->getHT3(), period +
    pProcessingElement->getRTA(), 0, 0, 0);
pProcessingElement->
    GetReconfigurationScheduleList()->append(node)
;
node = new Schedule_Node(Task_Schedule, period +
    pProcessingElement->getRTA(), period +
    pProcessingElement->getSchedule_Time(), 0, 0,
    0);
pProcessingElement->
    GetReconfigurationScheduleList()->append(node)
;
for (pos2 = pProcessingElement->
    GetAppTasksScheduled()->GetHeadPosition();
    pos2 != NULL;)
{
    pAppTask = (CApplicationTask*)
        pProcessingElement->GetAppTasksScheduled()
        ->GetNext(pos2);
    node = new Schedule_Node(ApplicationTask,
        period + pAppTask->get_start_time(),
        period + pAppTask->get_finish_time(),
        pAppTask->GetId(), pAppTask->GetNodeId(),
        0);
    pProcessingElement->
        GetReconfigurationScheduleList()->append(
        node);
}
// New deployment node for next period
node = new Schedule_Node(Deployment_Task, period
    + pProcessingElement->getDeployment_time(),
    period + pProcessingElement->
    getDeployment_time()+ pProcessingElement->
    getHT1(), 0, 0, 0);
pProcessingElement->
    GetReconfigurationScheduleList()->append(node)
;
pProcessingElement->
    GetReconfigurationScheduleList()->delete_head
    ();

// Check if all task that failed have been
// reallocated and switch state
if (pProcessingElement->GetReconfiguredTaskList()
    ->IsEmpty())

```

```

        {
            pProcessingElement->setReconfiguration(false)
            ;
            pProcessingElement->GetPeEvents()->RemoveHead
            ();
            // Add new event for the normal execution
            CPeEvent* new_event = new CPeEvent(
                PE_NORMAL_STATE, PE_NORMAL_STATE);
            pProcessingElement->GetPeEvents()->AddSorted(
                new_event);
            Schedule_Node* reconfiguration_event =
                pProcessingElement->
                GetReconfigurationScheduleList()->
                returnHead();
            while (reconfiguration_event != nullptr) {
                pProcessingElement->GetScheduleList()->
                    append(reconfiguration_event);
                reconfiguration_event =
                    reconfiguration_event->next;        //
                    Move to the next node
            }
            break;
        }
        // Wait till all the other threads finish
        sc_core::wait(delay, SC_US);

    }
    else {

        sc_core::wait(reconfiguration_event->start_time -
            sc_core::sc_time_stamp().to_double(), SC_US);

    }
    // If all task are reallocated swith to normal state
    // Add new event for the normal execution
    CPeEvent* new_event = new CPeEvent(PE_NORMAL_STATE,
        PE_NORMAL_STATE);
    pProcessingElement->GetPeEvents()->AddSorted(
        new_event);
    break;
}
case PE_NORMAL_STATE:    if(pProcessingElement->
    getReconfiguration() != TRUE)
{
    // Obtain next node in the CLL

```

```

Schedule_Node* normal_event = pProcessingElement->
    GetScheduleList()->returnHead();
// obtain task id and execution time
operation = normal_event->id;
delay = normal_event->end_time - normal_event->start_time;
if (operation == Calendar_HT1 && sc_core::sc_time_stamp().
    to_double() == normal_event->start_time)
{
    // Output results in log file
    if (theProperties.GetLogLevel() >= 3)
    {
        sprintf(str, LOG_HT1, pProcessingElement->
            GetGridPosX(), pProcessingElement->GetGridPosY
            (), sc_core::sc_time_stamp().to_double());
        pProcessingElement->GetLogFile()->WriteLine(str,
            TRUE);
    }
    // Execute Hormone Task 1 and wait for other threads
    pHTasks->HT1(pProcessingElement);
    pProcessingElement->GetScheduleList()->delete_head();
    sc_core::wait(delay, SC_US);
}
else if (operation == Calendar_HT2 && sc_core::
    sc_time_stamp().to_double() == normal_event->start_time
    )
{
    // Output results in log file
    if (theProperties.GetLogLevel() >= 3)
    {
        sprintf(str, LOG_HT2, pProcessingElement->
            GetGridPosX(), pProcessingElement->GetGridPosY
            (), sc_core::sc_time_stamp().to_double());
        pProcessingElement->GetLogFile()->WriteLine(str,
            TRUE);
    }
    // Execute Hormone Task 2 and wait for other threads
    pHTasks->HT2(pProcessingElement);
    pProcessingElement->GetScheduleList()->delete_head();
    sc_core::wait(delay, SC_US);
}
else if (operation == Calendar_HT3 && sc_core::
    sc_time_stamp().to_double() == normal_event->start_time
    )
{
    // Output results in log file
    if (theProperties.GetLogLevel() >= 3)
    {

```

```

        sprintf(str, LOG_HT3, pProcessingElement->
            GetGridPosX(), pProcessingElement->GetGridPosY
            (), sc_core::sc_time_stamp().to_double());
        pProcessingElement->GetLogFile()->WriteLine(str,
            TRUE);
    }
    // Execute Hormone Task 3 and wait for other threads
    pHTasks->HT3(pProcessingElement);
    pProcessingElement->GetScheduleList()->delete_head();
    sc_core::wait(delay, SC_US);
}
// TA -> Task allocation
else if (operation == Calendar_TA && sc_core::
    sc_time_stamp().to_double() == normal_event->start_time
    )
{
    // Output results in log file
    if (theProperties.GetLogLevel() >= 3)
    {
        sprintf(str, LOG_TA, pProcessingElement->
            GetGridPosX(), pProcessingElement->GetGridPosY
            (), sc_core::sc_time_stamp().to_double());
        pProcessingElement->GetLogFile()->WriteLine(str,
            TRUE);
    }
    // Execute Task allocation and wait for other threads.
    If allocation is successful make type 1
    type = pHTasks->AllocateTask(pProcessingElement);
    pProcessingElement->GetScheduleList()->delete_head();
    sc_core::wait(delay, SC_US);

}
else if (operation == Task_Schedule && sc_core::
    sc_time_stamp().to_double() == normal_event->start_time
    )
{
    if (type == 1)
    {
        // Activate List scheduling algorithm
        pProcessingElement->list_scheduling(
            pProcessingElement->scheduled_task(),
            pProcessingElement->GetPEId());
        sc_core::wait(delay, SC_US);
        pProcessingElement->GetScheduleList()->delete_head
            ();
    }
}

```

```

else
{
    // Wait for other threads
    sc_core::wait(delay, SC_US);
    pProcessingElement->GetScheduleList()->delete_head
        ();
}
}
else if (operation == ApplicationTask && sc_core::
    sc_time_stamp().to_double() == normal_event->start_time
)
{
    // Obtain Application Tasks
    pAppTask = (CApplicationTask*)pProcessingElement->
        GetAppTasks()->FindTask(normal_event->job_id);
    // Output results in log file
    if (theProperties.GetLogLevel() >= 3)
    {
        sprintf(str, LOG_TASKEVENT, pAppTask->GetId(),
            pProcessingElement->GetGridPosX(),
            pProcessingElement->GetGridPosY(), sc_core::
                sc_time_stamp().to_double());
        pProcessingElement->GetLogFile()->WriteLine(str,
            TRUE);
    }
    // Execute Application Tasks
    pAppTask->Execute(pProcessingElement);
    pProcessingElement->GetScheduleList()->delete_head();
    sc_core::wait(delay, SC_US);
}
else if (operation == Deployment_Task && sc_core::
    sc_time_stamp().to_double() == normal_event->start_time
)
{
    // Update schedule for next period
    // Hormone Task will have predefined time slots and
    // application tasks have been scheduled
    node = new Schedule_Node(Calendar_HT1, period, period
        + pProcessingElement->getHT1(), 0, 0, 0);
    pProcessingElement->GetScheduleList()->append(node);
    node = new Schedule_Node(Calendar_HT2, period +
        pProcessingElement->getHLT1(), period +
        pProcessingElement->getHT2(), 0, 0, 0);
    pProcessingElement->GetScheduleList()->append(node);
    node = new Schedule_Node(Calendar_HT3, period +
        pProcessingElement->getHT2(), period +
        pProcessingElement->getHT3(), 0, 0, 0);
    pProcessingElement->GetScheduleList()->append(node);
}

```

```

node = new Schedule_Node(Calendar_TA, period +
    pProcessingElement->getHT3(), period +
    pProcessingElement->getTA(), 0, 0, 0);
pProcessingElement->GetScheduleList()->append(node);
node = new Schedule_Node(Task_Schedule, period +
    pProcessingElement->getTA(), period +
    pProcessingElement->getSchedule_Time(), 0, 0, 0);
pProcessingElement->GetScheduleList()->append(node);
// Iterate through all scheduled tasks
for (pos2 = pProcessingElement->GetAppTasksScheduled()
    ->GetHeadPosition(); pos2 != NULL;)
{
    pAppTask = (CApplicationTask*)pProcessingElement->
        GetAppTasksScheduled()->GetNext(pos2);
    node = new Schedule_Node(ApplicationTask, period +
        pAppTask->get_start_time(), period + pAppTask
        ->get_finish_time(), pAppTask->GetId(),
        pAppTask->GetNodeId(), 0);
    pProcessingElement->GetScheduleList()->append(node
    );
}
// Add Deployment Task start time 4.9ms and end time 5
ms
node = new Schedule_Node(Deployment_Task, period +
    pProcessingElement->getDeployment_time(), period +
    pProcessingElement->getDeployment_time()+
    pProcessingElement->getHT1(), 0, 0, 0);
pProcessingElement->GetScheduleList()->append(node);
pProcessingElement->GetScheduleList()->delete_head();
sc_core::wait(delay, SC_US);

}
else {
    // The thread should wait until the next event time
    sc_core::wait(normal_event->start_time - sc_core::
        sc_time_stamp().to_double(), SC_US);

}
// Add new event for the normal execution
CPeEvent* new_event = new CPeEvent(PE_NORMAL_STATE,
    PE_NORMAL_STATE);
pProcessingElement->GetPeEvents()->AddSorted(new_event);

break;

```

```

        }

        default:                break;

    }

}

}
}

```

Listing A.1: TTOC simulator task dispatcher

```

BOOL
CProcessingElement::list_scheduling(int task_id, int pe_id)
{
    // variable declarations
    char str[LINE_LENGTH]; //output log file character
    int port_id = 0; // specific port id for the message queue
    CProcessingElement* PE; // PE that receives the message
    vector<Route> routes; // all possible routes of the message
    CApplicationTask* pAppTask; // the application task object
    json message; // json message for the message queue
    POSITION pos; //iterating position for the list of objects

    // The task object is obtain
    TT_OC::Task& schedule_task = find_task(task_id);
    // Obtain the application tasks
    pAppTask = GetAppTasks()->FindTask(task_id);

    schedule_task.runs_on = pe_id; //task runs on this pe_id
    pAppTask->setPEId(schedule_task.runs_on); //application task runs
    on this pe_id
    // Worst case execution time
    schedule_task.wcet_with_compression = schedule_task.wcet_fullspeed
    ;
    task_has_slot[schedule_task.id] = false; // allocation still
    missing

    //Checking all parent tasks of the task that has to be scheduled
    for (auto j : schedule_task.parent)
    {
        // All messages that the parent task has to send
        for (auto i : j.send_msg)
        {
            //if the receiver of the messages is the task to be

```

```

        scheduled
    if (i.receiver == schedule_task.id)
    {
        //If the parent task and task to be scheduled run
        //on the same processing element
        if (j.runs_on == schedule_task.runs_on)
        {

            //m_start_time represents PE time
            i.injection_time = m_start_time;
            port_id = find_and_lock_port_pe(1, 16,
                message_ports, port_locks);
            // Route is empty since same PE
            Route route = {};
            // Message to be transmitted to queue
            json msg = to_json(i.id, i.size, m_id, m_id, i
                .injection_time, port_id, route);
            // Send message to the queue
            send_message_queue(GetPEId(), msg);
            // Output to log file
            if (theProperties.GetLogLevel() >= 3)
            {
                sprintf(str, LOG_Send_Message_SCHEDULE_PE,
                    i.id, i.injection_time);
                GetLogFile()->WriteLine(str, TRUE);
            }

        }
        //If the parent and child tasks run on different
        //computing nodes
    else
    {

        const TT_OC::Scheme& compressor =
            m_Configuration->scheme(0);
        i.compressed_size = ceil(i.size * compressor.
            wccr);

        // Find the routes of the message
        routes = find_msg_path(j.runs_on,
            schedule_task.runs_on, msg_routes_[j.
                runs_on]);
        // select the shortes
        Route route = routes[0];

        int hop_time = ceil(i.size * compressor.wccr);
        // Obtain the PE where the message will be sent
        //from
    }
}

```

```

PE = m_OwnerListPtr->
    FindProcessingElementNewID(j.runs_on);
double earliest = PE->DRT();
i.injection_time = PE->find_and_lock_pe(
    earliest, hop_time, route, msg_locks);
// Obtain the port id
port_id = PE->find_and_lock_port_pe(1, 16,
    message_ports, port_locks);
//Construct the message to be send to queue
json msg = to_json(i.id, i.size, PE->GetPEId()
    , m_id, i.injection_time, port_id, route);
// send the message
PE->send_message_queue(PE->GetPEId(), msg);
if (theProperties.GetLogLevel() >= 3)
{
    sprintf(str, LOG_Send_Message_SCHEDULE, i.
        id, i.injection_time);
    GetLogFile()->WriteLine(str, TRUE);
}
double arrival = 0;
int nr_hops = route.size() - 1;
// For calculating arrival time, TSN
    specification as defined
//in INET framework have been used
arrival = i.injection_time + (PE->
    getChannelDelay()*nr_hops) + (PE->
    getDataRate() / PE->getTransmissionRate() *
    nr_hops);
// Update start_time for both PEs
schedule_task.start_time = max(arrival,
    m_start_time);
m_start_time = arrival + 1;
if(theProperties.GetLogLevel() >= 3)
{
    sprintf(str, LOG_Receive_Message_SCHEDULE,
        i.id, arrival);
    GetLogFile()->WriteLine(str, TRUE);
}

// Update the processor ready time
m_start_time = schedule_task.start_time;
PE->set_DRT(i.injection_time);

}

```

```

    }

}

}

// Check if task has already an slot
if (!task_has_slot[schedule_task.id])
{
    // Assign task start time the one of PE which should be 0
    schedule_task.start_time = m_start_time;
    // Period 0 means the first task that has to be scheduled
    if(schedule_period == 0)
    {
        //Make start time after scheduling time slot
        if(schedule_task.start_time <= getSchedule_Time())
        { //Task should run after scheduling
            schedule_task.start_time = getSchedule_Time();
            schedule_task.end_time = schedule_task.start_time
                + schedule_task.wcet_with_compression;
            pAppTask->set_start_time(schedule_task.start_time)
                ;
            pAppTask->set_finish_time(schedule_task.start_time
                + schedule_task.wcet_with_compression);
        }
        // Lock PE start time
        schedule_task.start_time = find_and_lock_pe(
            schedule_task.start_time,
            schedule_task.wcet_with_compression,
            { pe_id,-1 },
            task_locks);
        task_has_slot[schedule_task.id] = true;
        // logfile output
        if (theProperties.GetLogLevel() >= 3)
        {
            sprintf(str, LOG_TASK_SCHEDULE, schedule_task.id,
                schedule_task.start_time, schedule_task.
                start_time + schedule_task.
                wcet_with_compression);
            GetLogFile()->WriteLine(str, TRUE);
        }
        // Make PE start time the end time of the scheduled
        task
        m_start_time = schedule_task.start_time +
            schedule_task.wcet_with_compression;
        for (auto i : schedule_task.send_msg)
        {
            schedule_task.scheduled = true;

```

```

        TT_OC::Task& child = find_task(i.receiver);
        child.parent.push_back(schedule_task);

    }
    // Task is scheduled
    schedule_task.scheduled = true;
    task_has_slot[schedule_task.id] = true;

}
//In case that in DAG multiple tasks have no parent
// If the task has no parents but period is not 0
else if (schedule_task.start_time <= getSchedule_Time())
{

    //Task should run after scheduling
    schedule_task.start_time = getSchedule_Time();
    pAppTask->set_start_time(getSchedule_Time());
    schedule_task.end_time = schedule_task.start_time +
        schedule_task.wcet_with_compression;
    pAppTask->set_finish_time(schedule_task.end_time);
    // lock start time
    schedule_task.start_time = find_and_lock_pe(
        schedule_task.start_time, schedule_task.
        wcet_with_compression,
    { pe_id,-1 },
    task_locks);
    // logfile output
    if (theProperties.GetLogLevel() >= 3)
    {
        sprintf(str, LOG_TASK_SCHEDULE, schedule_task.
            id, schedule_task.start_time, schedule_task.
            .start_time + schedule_task.
            wcet_with_compression);
        GetLogFile()->WriteLine(str, TRUE);
    }
    // Make PE start time the end time of the
    // scheduled task
    m_start_time = schedule_task.end_time;
    for (auto i : schedule_task.send_msg)
    {
        schedule_task.scheduled = true;
        TT_OC::Task& child = find_task(i.receiver);
        child.parent.push_back(schedule_task);

    }
    // Task has been given a slot
    schedule_task.scheduled = true;
    task_has_slot[schedule_task.id] = true;
}

```

```

    }
    // Allocate all other tasks
    else
    {
        pAppTask->set_start_time(schedule_task.start_time);
        schedule_task.end_time = schedule_task.start_time +
            schedule_task.wcet_with_compression;
        pAppTask->set_finish_time(schedule_task.end_time);

        //lock PE time
        schedule_task.start_time = find_and_lock_pe(
            schedule_task.start_time,
            schedule_task.wcet_with_compression,
            { pe_id,-1 },task_locks);

        // logfile output
        if(theProperties.GetLogLevel() >= 3)
        {
            sprintf(str, LOG_TASK_SCHEDULE, schedule_task.id,
                schedule_task.start_time, schedule_task.
                start_time + schedule_task.
                wcet_with_compression);
            GetLogFile()->WriteLine(str, TRUE);
        }
        //Update children
        for(auto i : schedule_task.send_msg)
        {
            schedule_task.scheduled = true;
            TT_OC::Task& child = find_task(i.receiver);
            child.parent.push_back(schedule_task);
        }
        // Make PE start time the end time of the scheduled
        task
        m_start_time = schedule_task.end_time;
        // Task assigned
        schedule_task.scheduled = true;
        task_has_slot[schedule_task.id] = true;

    }
}

// Add application task to the container of scheduled task
GetAppTasksScheduled()->AddTail(pAppTask);

// Go to next period
schedule_period += getSchedulePeriod();
return true; // Exit the function

```

```

else
{
    // parent task?
    if(schedule_task.parent.size() == 0)
    {
        double parent_earliest = 0.0, earliest = 0.0;
        parent_earliest = getSchedule_Time();
        // Find the earliest time from the affected tasks
        for(pos = GetAppTasksScheduled()->GetHeadPosition(); pos !=
            NULL;)
        {
            CApplicationTask* pAppTask2 = (CApplicationTask*)
                GetAppTasksScheduled()->GetNext(pos);
            TT_OC::Task& task = find_task(pAppTask2->GetId());
            if(parent_earliest + schedule_task.wcet_with_compression
                <= task.start_time)
            {
                earliest = parent_earliest;
            }
            if(parent_earliest + schedule_task.wcet_with_compression
                <= task.start_time)
            {
                earliest = parent_earliest;
            }
            else
            {
                earliest = parent_earliest + schedule_task.
                    wcet_with_compression;
            }
        }
    }
    // Make PE start time
    schedule_task.start_time = earliest;
    pAppTask->set_start_time(schedule_task.start_time);
    schedule_task.end_time = schedule_task.start_time +
        schedule_task.wcet_with_compression;
    pAppTask->set_finish_time(schedule_task.end_time);

    // Lock the start time
    schedule_task.start_time = find_and_lock_pe(schedule_task.
        start_time,
        schedule_task.wcet_with_compression,{ node_id,-1 },task_locks)
        ;
    task_has_slot[schedule_task.id] = true;
}

```

```

// logfile output
if(theProperties.GetLogLevel() >= 3)
{
    sprintf(str, LOG_TASK_RESCHEDULED, schedule_task.id,
        schedule_task.start_time, schedule_task.start_time +
        schedule_task.wcet_with_compression);
    GetLogFile()->WriteLine(str, TRUE);
}
//Update children
for(auto i : schedule_task.send_msg)
{
    schedule_task.scheduled = true;
    TT_OC::Task& child = find_task(i.receiver);
    child.parent.push_back(schedule_task);
}
// Make PE start time the end time of the scheduled task
m_start_time = schedule_task.end_time;
// Task assigned
schedule_task.scheduled = true;
task_has_slot[schedule_task.id] = true;
//Add application task to the container of scheduled task
GetAppTasksScheduled()->AddTail(pAppTask);
// Go to next period
schedule_period += getSchedulePeriod();
return true;//Exit function
}
else
{
    // Follow same procedure for finding PE start time
    double parent_earliest = 0.0, earliest = 0.0;
    for(auto parent : schedule_task.parent)
    {
        if(parent.scheduled)
        {
            if(parent_earliest <= parent.end_time)
            {
                parent_earliest = parent.end_time;
            }
        }
    }

    // for the scheduled tasks update time
    for(pos = GetAppTasksScheduled()->GetHeadPosition(); pos !=
        NULL;)
    {
        CApplicationTask* pAppTask2 = (CApplicationTask*)

```

```

        GetAppTasksScheduled()->GetNext(pos);
        TT_OC::Task& task = find_task(pAppTask2->GetId());
        if(parent_earliest <= task.end_time)
        {
            earliest = task.end_time;
        }
        else
        {
            earliest = parent_earliest + schedule_task.
                wcet_with_compression;
        }
    }
    // Make task start time the earliest
    schedule_task.start_time = earliest;
    pAppTask->set_start_time(schedule_task.start_time);
    schedule_task.end_time = schedule_task.start_time + schedule_task.
        wcet_with_compression;
    pAppTask->set_finish_time(schedule_task.end_time);
    // lock time
    schedule_task.start_time = find_and_lock_pe(schedule_task.
        start_time,
                                schedule_task.wcet_with_compression,
                                { pe_id,-1 },
                                task_locks);

    // logfile output
    if(theProperties.GetLogLevel() >= 3)
    {
        sprintf(str, LOG_TASK_RESCHEDULED, schedule_task.id,
            schedule_task.start_time, schedule_task.start_time +
            schedule_task.wcet_with_compression);
        GetLogFile()->WriteLine(str, TRUE);
    }
    //Update children of the tasks
    for(auto i : schedule_task.send_msg)
    {
        schedule_task.scheduled = true;
        TT_OC::Task& child = find_task(i.receiver);
        child.parent.push_back(schedule_task);
    }
    //Task has a slot
    schedule_task.scheduled = true;
    task_has_slot[schedule_task.id] = true;
    //Scheduling completed
    GetAppTasksScheduled()->AddTail(pAppTask);
    schedule_period += getSchedulePeriod();
    return true; // Exit the function with true value

```

```

    }
}

```

Listing A.2: TTOC simulator list scheduling algorithm

```

int sc_main(int argc, char* argv[])
{
    // Set the time resolution and default time to milliseconds
    sc_core::sc_time s;
    sc_core::sc_set_time_resolution(1, SC_MS);
    sc_core::sc_set_default_time_unit(1, SC_MS);
    // Variable declaration, time represent systemc next event time
    while
    // select_type for deciding which configuration to use
    double time = 0;
    bool select_type;
    cout <<"Enter the desired simulation: 0 for horsim and 1 for
        dnasim"<< endl;
    cin >> select_type;
    // Creating simulator for initialization of main objects
    Simulator* start = new Simulator(select_type);
    //Start of the TCP connection to the controller process
    start->Start_TCP();
    // Begin the loop for executing the systemc events
    while (true)
    {
        // Check if you are able to execute events
        if (start->Get_Execute())
        {
            // Check if there is connection established with the co-
            simulation
            if (!start->Get_Conn())
            {
                // Execute SystemC event for the specified time
                sc_start(time, SC_MS);
                // Obtain time of next event
                time = start->Get_DNA_Time(s);
                // Send next event time to co-simulation controller
                if (start->TCP_Send())
                {
                    puts("Sending Successful");
                }
                else
                {
                    puts("Sending failed");
                }
            }
        }
    }
}

```

```

    }
    // Obtain response from TCP connection
    if (start->TCP_Receive())
    {
        cout << "ExecuteUntil" << time << endl;
    }
    else
    {
        puts("Receiving failed");
        return 1;
    }
}
// If no connection possible close TCP
else
{
    puts("Connection not possible");
    start->TCP_Close();
}
}
// Wait for controller response
else {
    if (start->TCP_Receive())
    {
        cout << "Connection received" << endl;
    }
    else
    {
        puts("Connection not possible");
        start->TCP_Close();
    }
}
}
// Close TCP connection
start->TCP_Close();

return 0;
}

```

Listing A.3: TTOC simulator systemc procedure

Appendix B

Co-simulation Interface

```
#include <fstream>
#include <ostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "omn.h"
#include <set>
#include <Winsock2.h>
#pragma comment(lib,"ws2_32.lib")
#define PORT 8080

using namespace omnetpp;
using namespace omnetpp::common;
using namespace omnetpp::internal;
using namespace std;

Register_GlobalConfigOption(CFGID_LOAD_LIBS, "load-libs",
    CFG_FILENAMES, "", "A space-separated list of dynamic libraries to
    be loaded on startup. The libraries should be given without the '.
    dll' or '.so' suffix -- that will be automatically appended.");
Register_GlobalConfigOption(CFGID_CONFIGURATION_CLASS, "configuration-
class", CFG_STRING, "", "Part of the Envir plugin mechanism:
selects the class from which all configuration information will be
obtained. This option lets you replace omnetpp.ini with some other
implementation, e.g. database input. The simulation program still
has to bootstrap from an omnetpp.ini (which contains the
configuration-class setting). The class should implement the '
cConfigurationEx' interface.");
Register_GlobalConfigOption(CFGID_USER_INTERFACE, "user-interface",
    CFG_STRING, "", "Selects the user interface to be started. Known
good values are Cmdenv and Qtenv. This option is normally left
empty, as it is more convenient to specify the user interface via a
command-line option or the IDE's Run and Debug dialogs. New user
interfaces can be defined by subclassing 'cRunnableEnvir'");
```

```

// helper macro
#define CREATE_BY_CLASSNAME(var, classname, baseclass, description) \
    baseclass *var ## _tmp = (baseclass *)createOne(classname); \
    var = dynamic_cast<baseclass *>(var ## _tmp); \
    if (!var) \
        throw cRuntimeError("Class \"\%s\" is not subclassed from \"\%s\" #\n", \
            baseclass, (const char *)classname);

static void verifyIntTypes()
{
#define VERIFY(t,size) if (sizeof(t)!=size) {printf("INTERNAL_ERROR:\n\
sizeof(%s)!=%d, please check typedefs in include/inttypes.h, and\
report this bug!\n\n", #t, size); abort();}
    VERIFY(int8_t, 1);
    VERIFY(int16_t, 2);
    VERIFY(int32_t, 4);
    VERIFY(int64_t, 8);

    VERIFY(uint8_t, 1);
    VERIFY(uint16_t, 2);
    VERIFY(uint32_t, 4);
    VERIFY(uint64_t, 8);
#undef VERIFY

    char buf[32];
    int64_t a = 1, b = 2;
    sprintf(buf, "% PRIu64 \"% PRIu64\", a, b);
    if (strcmp(buf, "1 2") != 0) {
        printf("INTERNAL_ERROR: PRIu64 is incorrectly defined, please\
report this bug!\n\n");
        abort();
    }
}

static bool useStderr = false;
static std::ostream& err()
{
    std::ostream& err = useStderr ? std::cerr : std::cout;
    err << "\n<!-- Error: ";
    return err;
}

namespace omnetpp {
    namespace enviro {
        extern "C" ENVIRON_API int evMain(int argc, char* argv[]);
    } // namespace enviro
}

```

```

/*
 * Function to connect the controller to TTOC
 */
bool
Controller::connect_ttoc()
{
    printf("\nInitialising Winsock for TCP connection to TTOC...\n");
    if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("Failed. Error Code: %d", WSAGetLastError());
        return 0;
    }

    if ((server_socket = socket(AF_INET, SOCK_STREAM, 0)) ==
        INVALID_SOCKET)
    {
        printf("Could not create socket: %d", WSAGetLastError());
        return 0;
    }

    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(8888);

    //Bind
    if (bind(server_socket, (struct sockaddr*)&server, sizeof(server))
        == SOCKET_ERROR)
    {
        printf("Bind failed with error code: %d\n", WSAGetLastError());
        return 0;
    }
    puts("Bind done and Waiting for incoming connections..."); //Accept
    //and incoming connection
    listen(server_socket, 3);

    c = sizeof(struct sockaddr_in);
    new_socket = accept(server_socket, (struct sockaddr*)&client, &c);
    if (new_socket != INVALID_SOCKET)
    {
        puts("Connection accepted");
    }
    else
    {
        printf("accept failed with error code: %d", WSAGetLastError());
    }
}

```

```

        return 1;
    }
    return 0;
}

/*
 * Executing TTOC
 */
void
Controller::execute_ttoc(bool type)
{
    if ((send(new_socket, (char*)&type, sizeof(&type), 0) < 0))
    {
        puts("Sending failed");
    }
    else
    {
        puts("Sending successful");
    }
}

/*
 * Close TCP connection with TTOC
 */
void Controller::disconnect_ttoc()
{
    closesocket(new_socket);
    WSACleanup();
}

/*
 * Function to receive next event execution time of TTOC
 */
bool
Controller::obtain_ttoc_time()
{
    puts("Trying to get time of TTOC Sim");
    bool rcv;
    rcv = recv(new_socket, (char*)&ttoc_time, sizeof(&ttoc_time), 0);
    if (rcv)
    {
        printf("Received time of DNA Sim is: %f\n", ttoc_time);
        return 1;
    }
    else {
        printf("Error receiving value from client");
        return 0;
    }
}

```

```

    }
}

void
Controller::obtain_omnet_time()
{
    puts("_Trying_to_get_time_of_Omnet_Sim_");
    omnet_time = (getSimulation()->guessNextSimtime()).dbl();
}

int
Controller::setupOmnet(int argc, char* argv[])
{
    try {
        std::cout << "_Initializing_omnet_network_simulator_" << std::
            endl;
        // construct global lists
        CodeFragments::executeAll(CodeFragments::STARTUP);

        // verify definitions of int64_t, int32_t, etc.
        verifyIntTypes();

        // args
        ArgList args;
        args.parse(argc, argv, ARGSPEC);

        useStderr = !args.optionGiven('m');

        verbose = !args.optionGiven('s'); // "not silent"
        if (verbose) {
            std::cout << OMNETPP_PRODUCT "_Discrete_Event_Simulation_"
                (C)_1992-2022_Andras_Varga,_OpenSim_Ltd." << endl;
            std::cout << "Version:_ " OMNETPP_VERSION_STR ",_build:_ "
                OMNETPP_BUILDID ",_edition:_ " OMNETPP_EDITION << endl;
            std::cout << "See_the_license_for_distribution_terms_and_
                warranty_disclaimer" << endl;
            std::cout << endl;
        }

#ifdef _WIN32
        // All of this is here because we want to get a usable error
        // message if there is any trouble
        // loading the DLL-s (usually model code built as a shared

```

```

    library), or their linked dependencies.
// Unfortunately the only way to get that is a graphical
// dialog, and we don't want to get that
// when running in a terminal (especially when launching large
// simulation campaign batches).
// See: https://docs.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-seterrormode
{
    // This will be wrong if the app (envir) to start is
    // selected in the configuration
    // (which is by default SectionBasedConfiguration, reading
    // omnetpp.ini) and is not
    // overridden by a command-line argument. Additionally,
    // sometimes an envir might
    // be loaded from an extension library, and we don't have
    // those yet.
    const char *preliminaryAppName = args.optionValue('u'); //
        last -u argument
    // Look up the specified user interface or pick one from
    // those that we have.
    // We won't complain if the selected one is not found.
    // There is a second,
    // "real" lookup further down, after library loading,
    // which does it if needed.
    // Also not falling back to choosing the "best" because
    // the real lookup doesn't either.
    cOmnetAppRegistration *appReg = preliminaryAppName !=
        nullptr
        ? static_cast<cOmnetAppRegistration *>(omnetapps.
            getInstance()->lookup(preliminaryAppName))
        : cOmnetAppRegistration::chooseBest(); // might not
            find one either in an extreme case

    // This is not a very sophisticated check, but we don't
    // have a better one yet.
    if (appReg != nullptr && strcmp(appReg->getName(), "Qtenv"
        ) == 0)
        SetErrorMode(0);
}
#endif

//
// First, load the ini file(s). It might contain the name of
// the user interface
// to instantiate.
//
InifileReader *iniReader = new InifileReader();
const char *fname;

```

```

int inifilesRead = 0;
for (int k = 0; (fname = args.optionValue('f', k)) != nullptr;
    k++, inifilesRead++)
    iniReader->readFile(fname);
for (int k = 0; (fname = args.argument(k)) != nullptr; k++,
    inifilesRead++)
    iniReader->readFile(fname);
if (inifilesRead == 0) {
    fname = "omnetpp.ini";
    if (fileExists(fname))
        iniReader->readFile(fname);
    else if (!args.optionGiven('v') && !args.optionGiven('h')
        && !args.longOptionGiven("network"))
        throw cRuntimeError("Missing configuration: No ini files
            specified and no 'omnetpp.ini' in the current
            working directory (specify at least --network=<
            name> to suppress this message)");
}

// activate [General] section so that we can read global
// settings from it
bootConfig = new SectionBasedConfiguration();
bootConfig->setConfigurationReader(iniReader);
bootConfig->setCommandLineConfigOptions(args.getLongOptions(),
    getWorkingDir().c_str());

//
// Load all libraries specified on the command line ('-l'
// options),
// and in the configuration [General]/load-libs=.
// (The user interface library also might be among them.)
//
const char *libName;
for (int k = 0; (libName = args.optionValue('l', k)) !=
    nullptr; k++)
    loadExtensionLibrary(libName);
std::vector<std::string> libs = bootConfig->getAsFilenames(
    CFGID_LOAD_LIBS);
for (auto & lib : libs)
    loadExtensionLibrary(lib.c_str());

//
// Create custom configuration object, if needed.
//
std::string configClass = bootConfig->getAsString(
    CFGID_CONFIGURATION_CLASS);
if (configClass.empty()) {
    config = bootConfig;
}

```

```

}
else {
    // create custom configuration object
    CREATE_BY_CLASSNAME(config, configClass.c_str(),
        cConfigurationEx, "configuration");
    config->initializeFrom(bootConfig);
    delete bootConfig;
    bootConfig = nullptr;

    // load libs from this config as well
    std::vector<std::string> libs = config->getAsFileNames(
        CFGID_LOAD_LIBS);
    for (auto & lib : libs)
        loadExtensionLibrary(lib.c_str());
}

// validate the configuration, but make sure we don't report
// cmdenv-* keys
// as errors if Cmdenv is absent; same for other optional
// features.
std::string ignorableKeys;
if (omnetapps.getInstance()->lookup("Cmdenv") == nullptr)
    ignorableKeys += "_cmdenv-*";
if (omnetapps.getInstance()->lookup("Qtenv") == nullptr)
    ignorableKeys += "_qtenv-*";
#ifdef WITH_PARSIM
    ignorableKeys += "_parsim-*";
#endif
config->validate(ignorableKeys.c_str());

//
// Choose and set up user interface (EnvirBase subclass).
// Everything else
// will be done by the user interface class.
//

std::string appName = nulltoempty(args.optionValue('u')); //
    last -u argument
if (appName.empty())
    appName = config->getAsString(CFGID_USER_INTERFACE);

cOmnetAppRegistration *appReg = nullptr;
if (!appName.empty()) {
    // look up specified user interface
    appReg = static_cast<cOmnetAppRegistration *>(omnetapps.
        getInstance()->lookup(appName.c_str()));
    if (!appReg) {
        if (verbose) {

```

```

        std::cout << "User_interface_" << appName << "_\n"
        not_found(not_linked_in_or_loaded_dynamically)
        ." << endl << endl;
        std::cout << "Available_ones_are:" << endl;
        cRegistrationList *a = omnetapps.getInstance();
        for (int i = 0; i < a->size(); i++)
            std::cout << "  " << a->get(i)->getName() << "
            _:\n" << a->get(i)->str() << endl;
    }
    throw cRuntimeError("Could_not_start_user_interface_%s", appName.c_str());
}
}
else {
    // user interface not explicitly selected: pick one from
    // what we have
    appReg = cOmnetAppRegistration::chooseBest();
    if (!appReg)
        throw cRuntimeError("No_user_interface_(Cmdenv,_Qtenv,
        _etc.)_found");
}

//
// Create interface object.
//
if (verbose)
    std::cout << "Setting_up_" << appReg->getName() << "... "
    << endl << endl;
app = appReg->createOne();
}
catch (std::exception& e) {
    err() << e.what() << endl;
    if (app) {
        delete app;
        app = nullptr;
    }
    else {
        // normally, this is deleted by app
        delete bootConfig;
    }
}

//
// RUN
//
try {
    if (app) {
        // as we'll install app as the active env, take over the

```

```

        listeners added to the boot-time envir
for (auto l : getEnvir()->getLifecycleListeners())
    app->addLifecycleListener(l);

// install and run app
cosimulation = new cSimulation("simulation", app);
cSimulation::setActiveSimulation(cosimulation);
exitCode = app->runInterfaceSetup(argc, argv, config);

    }
    else {
        exitCode = 1;
    }
}
catch (std::exception& e) {
    err() << e.what() << endl;
    exitCode = 1;
}

return exitCode;
}

void
Controller::startController(int argc, char* argv[])
{
    bool ttocTimeFlag = true;
    bool omnetTimeFlag = true;
    bool dummyVar = false;

    int initError = 0;
    simPhase_t current = INITIALIZE;
    std::cout << "└─Inside─Start─Controller─" << endl;

    while (true) {
        switch (current) {
            case INITIALIZE: {
                std::cout << "└─Enter─Initialize─" << std::endl;
                initError = setupOmnet(argc, argv); // Start omnet
                initError += app->setRunNumber();

                if (initError != 0) break;
                initError = connect_ttoc();
                if (initError != 0) break;
                current = SETUP_NETWORK;
            }

```

```

}
case SETUP_NETWORK: {
    std::cout << "Enter setup network" << std::endl;
    if (!(app->run_network_ttoc()))
    {
        current = STOP_SIM;
    }
    current = SETUP_SIM;
}
case SETUP_SIM: {
    std::cout << "Enter setup sim" << std::endl;
    app->setupClock();
    current = OBTAIN_TIMES;
}
case OBTAIN_TIMES: {
    std::cout << "Enter obtaining time" << std::endl;
    if(ttocTimeFlag) {
        obtain_ttoc_time();
    }
    if(omnetTimeFlag) {
        puts("Trying to get time of Omnet Sim");
        omnet_time = (getSimulation()->guessNextSimtime()).dbl
            ();
        printf("time of omnet event is: %f\n", omnet_time)
            ;
    }
    current = EXECUTE_SIM;
}
case EXECUTE_SIM: {
    std::cout << "Enter Execute sim" << std::endl;
    if (ttoc_time < omnet_time)
    {
        puts("Executing TTOC");
        ttocTimeFlag = true;
        omnetTimeFlag = false;
        execute_ttoc(ttocTimeFlag);
        current = OBTAIN_TIMES;
    }
    else
    {
        puts("Executing Omnet");
        ttocTimeFlag = false;
        omnetTimeFlag = true;
        int simStatus = app->doSim();
        std::cout << "simstatus" << simStatus << std::endl;
        if (simStatus == 1) {
            if (app->finishSimulation(true)) //next

```

```

        iteration of simulation
        disconnect_ttoc();
    }
    else if (simStatus == 0) {
        app->finishSimulation(false);
        disconnect_ttoc();
    }
    else {
        //keep executing the current simulation
        execute_ttoc(ttocTimeFlag);
        current = OBTAIN_TIMES;
    }
}

}

}

}

}

Controller::Controller(int argc, char* argv[])
{
    startController(argc,argv);
}

int main(int argc, char *argv[])
{
    cStaticFlag dummy;

    Controller* cnt = new Controller(argc, argv);
    return 0;
}

```

Listing B.1: Co-simulation controller

Appendix C

OMNeT++ environment

```
bool EnvirBase::run_network_ttoc()
{
    bool retFlag = false;
    try {
        retFlag = buildNetwork();
    }
    catch (std::exception& e) {
        displayException(e);
    }
    return retFlag;
}

bool Cmdenv::buildNetwork()
{
    endRunRequired = false;
    bool networkSetup = true;

    int runNumber = runNumbers[runsTried];
    numErrors = 0;

    try {
        if (opt->verbose)
            out << "\nPreparing for running configuration " << opt->
                configName << ", run #" << runNumber << "..." << endl;

        cfg->activateConfig(opt->configName.c_str(), runNumber);
        readPerRunOptions();

        const char* iterVars = cfg->getVariable(CFGVAR_ITERATIONVARS);
        const char* runId = cfg->getVariable(CFGVAR_RUNID);
        const char* repetition = cfg->getVariable(CFGVAR_REPETITION);
        if (!opt->verbose)
            out << opt->configName << " run " << runNumber << ": " <<
```

```

        iterVars << ",_$_repetition=" << repetition << endl; //
        print before redirection; useful as progress indication
        from opp_runall

if (opt->redirectOutput) {
    processFileName(opt->outputFile);
if (opt->verbose)
    out << "Redirecting_output_to_file_" << opt->outputFile
        << "\"..." << endl;
startOutputRedirection(opt->outputFile.c_str());
if (opt->verbose)
    out << "\nRunning_configuration_" << opt->configName << ",
        _run_" << runNumber << "..." << endl;
}

if (opt->verbose) {
    if (iterVars && strlen(iterVars) > 0)
        out << "Scenario:_" << iterVars << ",_$_repetition=" <<
            repetition << endl;
    out << "Assigned_runID=" << runId << endl;
}

// find network
if (opt->networkName.empty())
    throw cRuntimeError("No_network_specified_(missing_or_
        empty_network=configuration_option)");
cModuleType* network = resolveNetwork(opt->networkName.c_str()
    );
ASSERT(network);

endRunRequired = true;

// set up network
if (opt->verbose)
    out << "Setting_up_network_" << opt->networkName.c_str()
        << "\"..." << endl;

setupNetwork(network);

// prepare for simulation run
if (opt->verbose)
    out << "Initializing..." << endl;

loggingEnabled = !opt->expressMode;
prepareForRun();

// run the simulation
if (opt->verbose)

```

```

        out << "\nRunning simulation..." << endl;

        // simulate() should only throw exception if error occurred
        and
        // finish() should not be called.
        notifyLifecycleListeners(LF_ON_SIMULATION_START);
    }
    catch (std::exception& e) {
        loggingEnabled = true;
        networkSetup = false;
        stoppedWithException(e);
        notifyLifecycleListeners(LF_ON_SIMULATION_ERROR);
        displayException(e);
    }

    return networkSetup;
}

/*
 * Cmdenv run function according to ttoc
 */
int Cmdenv::setRunNumber()
{
    // '-c' and '-r' option: configuration to activate, and run
    numbers to run.
    // Both command-line options take precedence over inifile settings
    .
    // (NOTE: inifile settings *already* got read at this point! as
    EnvirBase::setup()
    // invokes readOptions()).

    if (args->optionGiven('c')) // note: do not overwrite value from
        cmdenv-config-name option
        opt->configName = args->optionValue('c');
    if (opt->configName.empty())
        opt->configName = "General";

    if (args->optionGiven('r')) // note: do not overwrite value from
        cmdenv-runs-to-execute option!
        opt->runFilter = args->optionValue('r');

    try {
        runNumbers = resolveRunFilter(opt->configName.c_str(), opt->
            runFilter.c_str());
    }
    catch (std::exception& e) {
        displayException(e);
        exitCode = 1;
    }
}

```

```

    }

    numRuns = (int)runNumbers.size();
    return exitCode;
}

/*
 * Cmdenv simulate function of Omnet networks with ttoc principles
 */
int Cmdenv::doSim()
{
    std::cout << "␣running␣simulation␣" << std::endl;
    int simFinish = 2;
    // implement graceful exit when Ctrl-C is hit during simulation.
    // We want
    // to finish the current event, then normally exit via callFinish
    // () etc
    // so that simulation results are not lost.

    cSimulation* simulation = getSimulation();

    // The following macro was originally written as a lambda, but on
    // macOS it caused
    // the program to crash while writing to the 'out' stream after
    // returning from simulate(),
    // due to some spurious compiler bug which only manifested in MODE
    // =debug.
    // Converting 'auto finally = [&] { ... }' to a macro solved the
    // issue.

    try {
        if (!opt->expressMode) {
            cEvent* event = simulation->takeNextEvent();
            if (!event) {
                simFinish = 0;
                throw cTerminationException("Scheduler␣interrupted␣
                    while␣waiting");
            }
            // flush *between* printing event banner and event
            // processing, so that
            // if event processing crashes, it can be seen which event
            // it was
            if (opt->autoflush)
                out.flush();

            if (fakeGUI)
                fakeGUI->beforeEvent(event);
        }
    }
}

```

```

// execute event
simulation->executeEvent(event);

if (fakeGUI)
    fakeGUI->afterEvent();

// flush so that output from different modules don't get
// mixed
cLogProxy::flushLastLine();

checkTimeLimits();

if (sigintReceived) {
    simFinish = 0;
    throw cTerminationException("SIGINT_or_SIGTERM_received
                                ,_exiting");
}
else {
    cEvent* event = simulation->takeNextEvent();
    if (!event) {
        simFinish = 0;
        throw cTerminationException("Scheduler_interrupted_
                                    while_waiting");
    }

    speedometer.addEvent(simulation->getSimTime());

    // print event banner from time to time
    if ((simulation->getEventNumber() & 0xff) == 0 && elapsed(
        opt->statusFrequencyMs, last_update))
        doStatusUpdate(speedometer);

    if (fakeGUI)
        fakeGUI->beforeEvent(event);

    // execute event
    simulation->executeEvent(event);
    // flush so that output from different modules don't get
    // mixed
    cLogProxy::flushLastLine();
    if (fakeGUI)
        fakeGUI->afterEvent();

    checkTimeLimits(); // potential place to gain a few
                        // cycles

    if (sigintReceived) {

```

```

        simFinish = 0;
        throw cTerminationException("SIGINT_or_SIGTERM_received
        ,_exiting");
    }
}

}
catch (cTerminationException& e) {
    simFinish = 0; //for any other termination exceptions
    if(e.getErrorCode() == E_REALTIME || e.getErrorCode() ==
        E_CPUTIME) {
        simFinish = 1;
    }
    stoppedWithTerminationException(e);
    displayException(e);
}
catch (std::exception& e) {
    simFinish = 0;
    throw;
}

return simFinish;
}

bool Cmdenv::finishSimulation(bool normSimHalt)
{
    bool newSim = true;
#define FINALLY() { \
        if (opt->expressMode) \
            doStatusUpdate(speedometer); \
        loggingEnabled = true; \
        stopClock(); \
        deinstallSignalHandler(); \
    }

    int runNumber = runNumbers[runsTried];
    try {
        loggingEnabled = true;

        if (opt->verbose)
            out << "\nCalling_finish()_at_end_of_Run_" << runNumber
                << "..." << endl;

        getSimulation()->callFinish();
        cLogProxy::flushLastLine();
        checkFingerprint();

        notifyLifecycleListeners(LF_ON_SIMULATION_SUCCESS);
    }
}

```

```

        getSimulation()->deleteNetwork();

        if (endRunRequired) // send LF_ON_RUN_END notification
            notifyLifecycleListeners(LF_ON_RUN_END);
    }
    catch (std::exception& e) {
        loggingEnabled = true;
        numErrors++;
        stoppedWithException(e);
        notifyLifecycleListeners(LF_ON_SIMULATION_ERROR);
        displayException(e);
    }

    // stop redirecting into file
    stopOutputRedirection();

    if (numRuns > 1 && opt->verbose) {
        out << "\nRun statistics: total" << numRuns;
        out << ", successful" << (runsTried - numErrors) ;
        out << ", errors" << numErrors;
        out << ", pending" << (numRuns - runsTried);
        out << endl;
    }

    if ((runsTried < numRuns-1) && normSimHalt)
        runsTried++;
    else
        newSim = false;

    exitCode = numErrors > 0 ? 1 : sigintReceived ? 2 : 0;
    if (exitCode != 0 || opt->stopBatchOnError)
        newSim = false;

    FINALLY();
#undef FINALLY

    return newSim;
}

/*
 * Function created for ttoc execution of omnet++
 */
int EnvirBase::runInterfaceSetup(int argc, char* argv[],
    cConfiguration* configobject)
{
    opt = createOptions();

```

```

args = new ArgList();
args->parse(argc, argv, ARGSPEC);
opt->useStderr = !args->optionGiven('m');
opt->verbose = !args->optionGiven('s');
cfg = dynamic_cast<cConfigurationEx *>(configobject);
if (!cfg)
    throw cRuntimeError("Cannot cast configuration object %s to cConfigurationEx", configobject->getClassName());
if (cfg->getAsBool(CFGID_DEBUGGER_ATTACH_ON_STARTUP) &&
    detectDebugger() != DebuggerPresence::PRESENT)
    attachDebugger();

if (simulationRequired()) {
    if (!setup())
        exitCode += 1; //run(); // must not throw, because we want
                        //shutdown() always to be called
    //shutdown();
}
return exitCode;
}

void Cmdenv::setupClock()
{
    installSignalHandler();
    startClock();
    sigintReceived = false;

    if (opt->expressMode) {
        speedometer.start(getSimulation()->getSimTime());
        last_update = opp_get_monotonic_clock_usecs();
        doStatusUpdate(speedometer);
    }
}

```

Listing C.1: OMNeT++ new environment functions