RECONFIGURABLE EMBEDDED CONTROL SYSTEMS: PROBLEMS AND SOLUTIONS

By
Dr.rer.nat.Habil. Mohamed Khalgui

Martin Luther University, Germany

Research Manuscript for Habilitation Diploma in Computer Science

1. Reviewer: Prof.Dr. Hans-Michael Hanisch, Martin Luther University, Germany,

2. Reviewer: Prof.Dr. Georg Frey, Saarland University, Germany,

3. Reviewer: Prof.Dr. Wolf Zimmermann, Martin Luther University, Germany,

Day of the defense: Monday January 23rd 2012,

Table of Contents

Table of Contents vi

English Abstract x

German Abstract xi

English Keywords xii

German Keywords xiii

Acknowledgements xiv

Dedicate xv

1 General Introduction 1

2 Embedded Architectures: Overview on Hardware and Operating Systems 3
- 2.1 Embedded Hardware Components . . . 3
  - 2.1.1 Microcontrollers . . . 3
  - 2.1.2 Digital Signal Processors (DSP): . . . 4
  - 2.1.3 System on Chip (SoC): . . . 5
  - 2.1.4 Programmable Logic Controllers (PLC): . . . 6
- 2.2 Real-Time Embedded Operating Systems (RTOS) . . . 8
  - 2.2.1 QNX . . . 9
  - 2.2.2 RTLinux . . . 9
  - 2.2.3 VxWorks . . . 9
  - 2.2.4 Windows CE . . . 10
- 2.3 Known Embedded Software Solutions . . . 11
  - 2.3.1 Simple Control Loop . . . 12
  - 2.3.2 Interrupt Controlled System . . . 12
  - 2.3.3 Cooperative Multitasking . . . 12
  - 2.3.4 Preemptive Multitasking or Multi-Threading . . . 12
  - 2.3.5 Microkernels . . . 13
  - 2.3.6 Monolithic Kernels . . . 13
  - 2.3.7 Additional Software Components: . . . 13
- 2.4 Conclusion . . . 14

3 Embedded Systems: Overview on Software Components 15
- 3.1 Basic Concepts of Components . . . 15
- 3.2 Architecture Description Languages . . . 17
  - 3.2.1 Acme Language . . . 17
  - 3.2.2 Rapide Language . . . 17

3.2.3 Wright Language . . . 18
3.2.4 Aesop Language . . . 18
3.2.5 Metah Language . . . 18
3.2.6 Architecture Analysis and Design Language . . . 18
3.2.7 Unicon Language . . . 19
3.2.8 Darwin Language . . . 19
3.3 Component-based Technologies . . . 19
3.3.1 Industrial Technology IEC61131 . . . 19
3.3.2 Industrial Technology IEC61499 . . . 20
3.3.3 Port Based Object Technology . . . 21
3.3.4 Koala Technology . . . 23
3.3.5 Pecos Technology . . . 23
3.3.6 Rubus Technology . . . 24
3.4 Conclusion . . . 25

4 Formalisms for Modelling and Verification of Embedded Systems 26
4.1 Petri nets . . . 26
4.2 NCES Formalism: Extension of Petri Net . . . 27
4.3 Temporal Logic . . . 28
4.3.1 Computation Tree Logic . . . 28
4.3.2 Extended Computation Tree Logic . . . 29
4.3.3 Timed Computation Tree Logic . . . 30
4.4 Conclusion . . . 30

5 Problems and Contributions 31

6 Reconfigurable Component-based Embedded Systems: New Motivations 33
6.1 Plant Formalization . . . 34
6.2 Control Components . . . 34
6.2.1 Formalization . . . 35
6.2.2 Specification . . . 35
6.3 Reconfigurable Systems: State of the Art . . . 35
6.4 New Reconfiguration Semantic . . . 36
6.5 Industrial Case Studies: Reconfiguration of Benchmark Production Systems FESTO and EnAS . . . 37
6.5.1 FESTO Manufacturing System . . . 37
6.5.2 EnAS Manufacturing System . . . 38
6.6 Reconfiguration Forms . . . 43
6.7 Conclusion . . . 45

7 Safety Multi-Agent Reconfigurable Architectures: Modelling And Verification 47
7.1 Introduction . . . 47
7.2 State of the Art . . . 48
7.2.1 Model Checking . . . 48
7.2.2 Multi-Agent Systems . . . 49
7.3 Contribution: Multi-Agent Architecture for Reconfigurable Embedded Control Systems . . . 50
7.3.1 Reconfiguration in a Device . . . 51
7.3.2 Reconfiguration in a Distributed Architecture . . . 54
7.3.3 Implementation . . . 63
7.4 Contribution: NCES-based Modelling and SESA-Based Model Checking of Distributed Reconfigurable Embedded Systems . . . 64
7.4.1 NCES-Based Specification of a Reconfiguration Agent . . . 65
7.4.2 SESA-Based Model Checking in a Device . . . 66
7.4.3 SESA-Based Model Checking of the Coordination Agent . . . 69

7.5 Contribution: Hierarchical Verification of Control Components 73
7.5.1 Refinement-based Specification and Verification of a Network of Control Components 73
7.5.2 Generalization: Refinement-based Specification and Verification of a Reconfigurable System 79
7.6 Application: Implementation of Multi-Agent Reconfigurable IEC61499 Systems 81
7.6.1 Implementation of Reconfiguration Agents 83
7.6.2 Agent Interpreter 83
7.6.3 Reconfiguration Engine 85
7.6.4 Agent Converter 85
7.6.5 Implementation of The Coordination Agent 87
7.7 Conclusion 89
7.8 References of the Chapter's Contributions 90

8 Feasible Execution Models for Reconfigurable Real-Time Embedded Control Systems 91
8.1 Introduction 91
8.2 State of the Art 92
8.3 Temporal Properties of Reconfigurable Embedded Control Systems 93
8.4 Contribution: Formalization of Reconfigurable Embedded Control Systems 93
8.4.1 Action of Control Component 94
8.4.2 Trace of Control Component 95
8.5 Contribution: Verification and Assignment of Control Components to OS Tasks 95
8.5.1 Verification and Pre-scheduling of a Container 96
8.5.2 Verification of OS tasks for a Reconfiguration Scenario 101
8.6 Generalization: Verification and Assignment of Reconfigurable Embedded Control Systems 102
8.6.1 Algorithm 103
8.6.2 Discussion 104
8.7 conclusion 104
8.8 References of the Chapter's Contributions 105

9 Dynamic Low Power Reconfigurations of Real-Time Embedded Systems 106
9.1 Introduction 106
9.2 Related Works 107
9.2.1 Reconfigurations of Embedded Systems 107
9.2.2 Real-Time Scheduling 108
9.2.3 Low Power Scheduling 109
9.3 Formalization of Reconfigurable Real-Time Systems 110
9.4 Power Consumption of Reconfigurable Embedded Real-Time Systems . . . 111
9.5 Contribution: Agent-based Architecture for Low Power Reconfigurations of Embedded Systems 112
9.5.1 Modification of periods and deadlines 113
9.5.2 Modification of WCETs 113
9.5.3 Removal of Tasks 114
9.6 Experimentation 115
9.6.1 Implementation of the Reconfiguration Agent 115
9.6.2 Simulations 117
9.6.3 Analysis 119
9.6.4 Advantages of the First Solution 119
9.6.5 Application and Advantages of the Third Solution 120
9.7 Conclusion 121
9.8 References of the Chapter's Contributions 121

10 Simulation of Control Components implementing a Reconfiguration Scenario **126**
10.1 Introduction . . . 126
10.2 Industrial Case Study . . . 127
10.2.1 Presentation . . . 127
10.2.2 Problem . . . 128
10.2.3 Numerical Characterization . . . 128
10.3 State of the Art . . . 129
10.4 Contribution: Formalization of Control Components . . . 130
10.4.1 Control Modules . . . 130
10.4.2 Formalization . . . 131
10.4.3 Deadline Processing . . . 134
10.5 Contribution: Characterization of Critical Scenarios . . . 135
10.5.1 Blocking Problem . . . 135
10.5.2 Critical Scenarios . . . 136
10.6 Contribution: Optimization of the Simulation . . . 136
10.6.1 Architecture of Simulators . . . 136
10.6.2 Formalization . . . 137
10.6.3 Implementation . . . 140
10.7 Evaluation of the Performance . . . 142
10.8 Conclusion . . . 143
10.9 References of the Chapter's Contributions . . . 144

11 Conclusion **145**

Bibliography **147**

English Abstract

The postdoctoral thesis deals with reconfigurable embedded control systems following component based approaches. Assuming their behaviors to be adapted by automatic reconfigurations, we define components as software units such that a system is implemented by a network of components. We propose a multi-agent architecture to handle automatic reconfigurations where local scenarios are handled by reconfiguration agents in controllers, and a coordination agent is designed to coordinate between devices. We develop NCES-based optimal models for this architecture, use SESA to verify CTL properties, and address the generation of different real-time reconfigurable execution models for systems. Thereafter, an agent-based technique is engineered allowing low-power reconfigurations of embedded controllers. Finally, optimized simulations of systems are performed.

German Abstract

Die Habilitationsschrift beschftigt sich mit rekonfigurierbaren Embedded-Control-Systemen mittels Komponenten-basierten Anstzen. Ausgehend von einer automatischen Rekonfiguration von Systemverhalten, definieren wir Komponenten als Software-Einheiten, um ein System durch ein Netzwerk von Komponenten zu implementieren. Wir schlagen eine Multi-Agenten-Architektur fr die automatische Rekonfiguration vor, in der lokale Szenarien durch Rekonfigurationsagenten im Controller verarbeitet werden und ein Koordinationsagent die Abstimmung zwischen den Gerten bernimmt. Wir entwickeln NCES-basierte optimale Modelle fr diese Architektur, verwenden SESA zur berprfung von CTL Eigenschaften und behandeln die Erzeugung unterschiedlicher Echtzeit-rekonfigurierbarer Ausfhrungsmodelle fr die Systeme. Die Rekonfigurierung von Low-Power-Embedded-Controllern wird dann mithilfe einer Agenten-basierten Technik ermglicht. Schlielich wurden optimierte Systemsimulationen durchgefhrt.

English Keywords

Embedded Control System, Reconfiguration, Software Component, Multi-Agent Architecture, TNCES, Computation Tree Logic, Model Checking, Execution Model, Low-Power and Real-Time Scheduling, Simulation.

German Keywords

Embedded Control System, Rekonfiguration, Software-Komponente, Multi-Agenten- Architektur, TNCES, Computation Tree Logic, Model Checking, Execution Model, Low-Power-und Echtzeit-Scheduling, Simulation.

Acknowledgements

I would like to thank Prof. Hans-Michael Hanisch who helped and encouraged me to follow a post-doc research at Martin Luther University... supported me to have a fund from Humboldt Foundation.. and collaborated with me to apply for Habilitation Diploma at the same university. Today I want to say Thank You HAMI: Man and Professor.

I would like to thank Prof. Ludwig Staiger for useful services and rich collaboration, I am grateful also for the reviewers of this manuscript: Prof. Wolf Zimmermann and Prof. Georg Frey.. for their serious and useful reviewing tasks which helped me to improve the quality of this work...

I want to thank also all my colleagues at Martin Luther University.. in particular Prof. Zhiwu Li and also my colleagues at the research laboratory of Prof. Hanisch...

Finally, I want to thank Olfa: Wife and Scientific Collaborator... for stable support and encouragements...

Dedicate

To Olfa, my son Med Aziz and my daughter Nour El-Hayet.....

List of Figures

2.1 An 8-bit microcontroller that includes a CPU running at 12 MHz, 128 bytes of RAM, 2048 bytes of EPROM, and I/O in the same chip. 5
2.2 AMD Geode is an x86 compatible system-on-a-chip. 6
2.3 Microcontroller-based System-on-a-Chip. 7
2.4 An example of PLC. 8
2.5 RT-Linux Architecture. 10
2.6 Windows CE Architecture. 11
2.7 A Screenshot of Microsoft Windows CE 5.0. 11
2.8 Structures of Microkernels and Monolothic Kernels. 13

3.1 An IEC 61499 Function Block 21
3.2 The ECC of the FB *BeltFB* 21
3.3 A PBO module 22
3.4 Network of PBO modules 22
3.5 Network of PBO modules 23
3.6 Example of PBO Objects implementing a PID Algorithm 24
3.7 A PBO Component : *Speed Regulator* 24
3.8 A Rubus Component : *BrakeSystem* 25

4.1 A module of Net Condition/Event Systems 28

6.1 The FESTO modular production system 38
6.2 Functional operations of the FESTO system 39
6.3 EnAS-Demonstrator in Halle 40
6.4 Distribution of the EnAS stations 41
6.5 Functional operations of the system EnAS 41
6.6 Policy1: production of a tin with only one piece. 42
6.7 Policy2: production of tins with double pieces. 42
6.8 An IEC61499-based Design of FESTO 43
6.9 An IEC61499-based Design of EnAS. 44

6.10 Interactions between the Control Component CC_Jack1 and the plant sensors $S22$ and $S23$ before the activation of $act12$ 45

7.1 Multi-agent architecture of distributed reconfigurable embedded systems. . 50
7.2 Specification of the FESTO Agent by nested state machines 54
7.3 Specification of the EnAS Agent by nested state machines 55
7.4 The system's state machine of the FESTO Benchmark Production System: SSM(FESTO). 56
7.5 The system's state machine of the EnAS Benchmark Production System: SSM(EnAS). 57
7.6 A Coordination Matrix. 57
7.7 Coordination Matrices For the FESTO and EnAS Benchmark Production Systems. 59
7.8 Coordination between the FESTO and EnAS agents to optimize their productivities. 60
7.9 Coordination between the FESTO and EnAS agents when $Drill\_machine1$ is broken. 61
7.10 Evaluation of the multi-agent architecture by Varying the number of reconfigurations. 62
7.11 The main interface. 63
7.12 Reconfiguration Agent. 63
7.13 Coordination Agent. 64
7.14 First example of Communication Protocol. 65
7.15 Second example of Communication Protocol. 65
7.16 Design of the FESTO agent with the NCES formalism. 66
7.17 Design of the EnAS system with the NCES formalism. 67
7.18 Reachability graph of the first Production Policy. 68
7.19 Automatic Distributed Reconfigurations in FESTO and EnAS when the drilling machines $Drill\_machine1$ or $Drill\_machine2$ are broken at run-time. 71
7.20 Automatic Distributed Reconfigurations in FESTO and EnAS when hardware problems occurs at run-time. 72
7.21 Automatic Distributed Reconfigurations in FESTO and EnAS to regulate the whole system performance. 72
7.22 The abstract model $M^1_{\{2,1,2,1\};\{1,1,0,0\}}$ of FESTO and EnAS systems 75
7.23 The first Step of an automatic refinement process 76
7.24 The second Step of an automatic refinement process 77
7.25 The last Steps of an automatic refinement process 78

7.26 Refinement-based Verification of FESTO and EnAS Benchmark Production Systems . . . 80
7.27 (a) Number of states automatically generated by SESA for the refinement-based specification and verification of 7 different networks of Control Components. (b) Number of states generated by SESA for the specification and verification of 7 networks of Control Components without applying any refinement . . . 80
7.28 Automatic specification of feasible Control Components of FESTO and EnAS systems . . . 81
7.29 A reachability graph generated by SESA in Step7 when $Reconfiguration_{2,1,2,1}$ is applied in FESTO and $Reconfiguration_{1,1,0,0}$ is applied in EnAS . . . 82
7.30 A reachability graph generated by SESA in *Step 6* when $Reconfiguration_{1,1,1,1}$ is applied in FESTO and $Reconfiguration_{2,1,0,0}$ is applied in EnAS . . . 82
7.31 A reachability graph generated by SESA in *Step 6* when $Reconfiguration_{2,2,2,2}$ is applied in FESTO and $Reconfiguration_{1,1,0,0}$ is applied in EnAS . . . 83
7.32 Interaction between the Agent and the Embedded Control System. . . . 84
7.33 Function Blocks-based implementation of the agent interpreter in the EnAS system. . . . 84
7.34 Management Function Block". . . . 85
7.35 Management Services offered by the Function Block "Manager". . . . 86
7.36 Interactions between FESTO and EnAS agents . . . 88
7.37 The Manager behavior when the EnAS productivity has to be improved . . 89

8.1 Verification and Assignment of Reconfigurable Control Components to sets of Feasible OS Tasks . . . 92
8.2 Assumed distribution of FESTO's blocks on two resources . . . 94
8.3 Reachability graphs of $Container2$ for the high and medium production modes. . . . 98
8.4 A pre-scheduling portion of $container1$ and $container2$ for the Medium production mode. . . . 99
8.5 The OS tasks implementing the system in the Medium Production Mode. . 102
8.6 The OS tasks implementing the system in the High Production Mode. . . 103
8.7 Context switchings for the different production modes of the FESTO Manufacturing System. . . . 105

9.1 An example of periodic tasks . . . 109
9.2 Initial System Tasks . . . 117
9.3 Added Tasks . . . 117

9.4 Static Priorities . . . 118
9.5 Developed Software . . . 119
9.6 Low power reconfiguration after the addition of $AA1$ . . . 119
9.7 Low power reconfiguration after the addition of the first ten tasks . . . 120
9.8 Low power reconfiguration after the addition of the thirty new tasks . . . 120
9.9 Power save by modify periods . . . 121
9.10 Power save by modify WCETs . . . 122
9.11 Compare the number of removed task between two remove strategies . . . 122
9.12 Compare power save between two remove strategies . . . 123
9.13 New Configuration of the System (1) . . . 124
9.14 New Configuration of the System (2) . . . 125

10.1 A set of belts transporting pieces in the ITIA factory . . . 128
10.2 The ITIA footwear factory in Vigevano (Italy). . . . 129
10.3 Implementation of the footwear system in Vigevano (Italy). . . . 130
10.4 Formalization of a Control Component. . . . 131
10.5 Behavior of CFM and VPM modules. . . . 132
10.6 The modularity of the system of belts in Vigevano (Italy). . . . 133
10.7 Temporal behavior of a Control Component. . . . 134
10.8 Characterization of a blocking problem. . . . 136
10.9 Architecture of the simulators implemented in the Vigevano platform (Italy). 138
10.10Comparison between the proposed simulator and a simple simulator. . . . 142
10.11Evaluation of performance. . . . 143

Chapter 1

General Introduction

Embedded Systems exist in all aspects of our modern life due to many cases of their use. Nowadays, telecommunication systems employ numerous embedded systems from telephone switches to mobile phones at the end-user. Computer networking uses also dedicated embedded controllers-based routers and network bridges to route data. Consumer electronics include personal digital assistants (PDAs), mp3 players, mobile phones, videogame consoles, digital cameras, DVD players, GPS receivers, and printers. Many household devices, such as televisions, microwave ovens, washing machines and dishwashers, are including embedded electronics to provide flexibility, efficiency and features. Advanced Heating, Ventilating, and Air Conditioning (HVAC) systems use networked thermostats to more accurately and efficiently control temperature that can change by time of day and season. Home automation uses wired- and wireless-networking that can be used to control lights, climate, security, audio/visual, surveillance, etc., all of which use embedded controllers for sensing and controlling. Today, all the transportation systems from flight, maritime to automobiles, motorcycles and bikes increasingly use embedded electronics for control and supervision. New airplanes contain advanced avionics such as inertial guidance systems and GPS receivers that should have considerable safety requirements. Various electric motors brushless DC motors, induction motors and DC motors are using electric/electronic motor controllers. Automobiles, electric vehicles, and hybrid vehicles are increasingly using embedded systems to maximize efficiency and reduce pollution. Other automotive safety systems include anti-lock braking system (ABS), Electronic Stability Control (ESC/ESP), traction control (TCS) and automatic four-wheel drive. Today, medical equipment is continuing to advance with more embedded software solutions for vital signs monitoring, electronic stethoscopes for amplifying sounds, and various medical imaging (PET, SPECT, CT, MRI) for non-invasive internal inspections. In addition to commonly described embedded systems based on small computers, a new class of miniature wireless devices called motes are quickly gaining popularity as the field of wireless sensor networking rises. Wireless sensor networking, WSN, couples full wireless subsystems to sophisticated sensors, enabling people and companies to measure a myriad of things in the physical world and act on this information through monitoring and control controllers. These motes are completely self contained, and will typically run off a battery source for many years before the batteries need to be changed or

charged.

We are interested in this manuscript in reconfigurable embedded control architectures that should be automatically reconfigured in order to save their behaviors or sometimes improve their performance. Before presenting in Section 5 the problems and our contributions, we present in the next Section known hardware components as well as operating systems which are used in new generations of embedded architectures. We describe thereafter in Section 3 well-known Architecture Description Languages and Industrial Technologies used today for the development of software component-based embedded systems. We present in Section 4 the theoretical formalisms that will be used in our contributions. The goal behind these introductive chapters is to familiarize the reader with the environment of embedded technologies.

Chapter 2

Embedded Architectures: Overview on Hardware and Operating Systems

We present in this chapter some technical overviews on hardware architectures as well as operating systems of embedded controllers that we assume as parts of complete devices to control included physical parts. We present first of all the different possible hardware architectures that can compose these controllers, before describe thereafter different used operating systems of this technology and show also different used embedded software architectures [1].

2.1 Embedded Hardware Components

We present well-known hardware components that are usually selected and deployed by industrial developers of embedded systems.

2.1.1 Microcontrollers

These controllers are small self-contained computers based on single integrated circuits to be composed of (rich details are in http://www.microcontroller.com/): (a) Central Processing Unit - ranging from small and simple 4-bit processors to complex 32- or 64-bit processors, (b) Discrete input and output bits, allowing control or detection of the logic state of an individual package pin, (c) Serial input/output such as serial ports (UARTs), (d) Other serial communications interfaces like I^2C, (e) Serial Peripheral Interface and Controller Area Network for system interconnect, (f) Peripherals such as timers, event counters, PWM generators, and watchdog, (g) Volatile memory (RAM) for data storage ROM, EPROM,

[1] Some information and Figures are from www.wikipedia.org.

EEPROM or Flash memory for program and operating parameter storage, (h) Clock generator - often an oscillator for a quartz timing crystal, resonator or RC circuit, (i) Many include analog-to-digital converters, (j) In-circuit programming and debugging support.

In contrast to classic microprocessors used in high-performance or general purpose applications, most of current microcontrollers use four-bit words and operate at clock rate frequencies as low as 4 kHz, as this is adequate for many typical applications that enable low power consumptions (milliwatts or microwatts). They generally have the ability to retain functionality while waiting for an event such as a button press or other interrupt. In this case, the power consumption while sleeping may be just nanowatts. Microcontrollers are generally used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, and toys (Figure 2.1). By reducing the size and cost compared to a design that uses a separate microprocessor, memory, and input/output devices, microcontrollers make it economical to digitally control even more devices and processes. Nowadays, several dozen microcontroller architectures are exploited like (http://www.microcontroller.com/): (a) 68HC11,(b) 8051, (c) ARM processors (from many vendors) using ARM7 or Cortex-M3 cores, (d) STMicroelectronics STM8S (8-bit), ST10 (16-bit) and STM32 (32-bit), (e) Atmel AVR (8-bit), AVR32 (32-bit), and AT91SAM, (f) Freescale ColdFire (32-bit) and S08 (8-bit), (g) Hitachi H8, Hitachi SuperH, (h) Hyperstone E1/E2, (i) MIPS (32-bit PIC32), (j) NEC V850, (k) PIC (8-bit PIC16, PIC18, 16-bit dsPIC33 / PIC24), (l) PowerPC ISE, (m) PSoC, (n) Rabbit 2000, (o) Texas Instruments MSP430 (16-bit), C2000 (32-bit), and Stellaris (32-bit), (p) Toshiba TLCS-870, (q) Zilog eZ8, eZ80.

2.1.2 Digital Signal Processors (DSP):

They are specialized microprocessors with optimized architectures for fast digital signal processing that require large numbers of mathematical operations to be performed quickly and repetitively on sets of data[2]. Signals are constantly converted from analog to digital, manipulated digitally, and then converted again to analog form. Nowadays, many DSP applications have real-time constraints such that for the system to work, the DSP task must be completed within some fixed time. In addition, many microprocessors and operating systems are able to execute DSP algorithms successfully, but are not suitable for use in portable small devices such as mobile phones and PDAs because of power supply and space constraints. A specialized Digital Signal Processor is able to provide lower-cost solutions, with better performance, lower real-time constraints, and no requirements for specialized cooling or large batteries. The architecture of a digital signal processor is optimized specifically for digital signal processing. Digital signal processing is often deployed in specialized microprocessors such as the DSP56000, the TMS320, or the SHARC. These often process data using fixed-point arithmetic, although some versions are available which use floating point arithmetic and are more powerful. Finally, we note that known applications of DSP are speech compression and transmission in digital mobile phones, room

[2]More details are available in the website of Texas Instrument: http://www.ti.com.

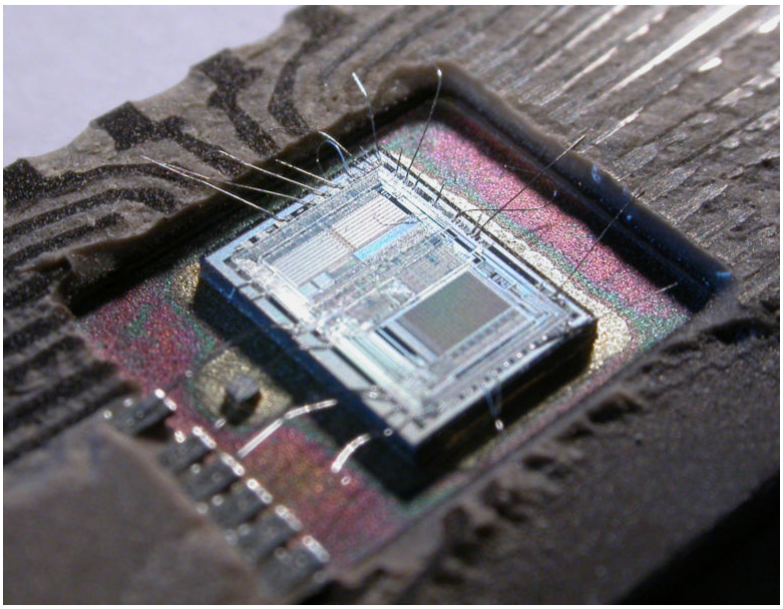

Figure 2.1: An 8-bit microcontroller that includes a CPU running at 12 MHz, 128 bytes of RAM, 2048 bytes of EPROM, and I/O in the same chip.

matching equalization of sound in Hifi and sound reinforcement applications, weather forecasting, economic forecasting, seismic data processing, analysis and control of industrial processes, computer-generated animations in movies, medical imaging such as CAT scans and MRI, MP3 compression, image manipulation, high fidelity loudspeaker crossovers and equalization, and audio effects for use with electric guitar amplifiers.

2.1.3 System on Chip (SoC):

This system deploys all components of a computer or other electronic systems into a single integrated circuit called a chip [1]. It may contain digital, analog, mixed-signal, and often radio-frequency functions all on a single chip substrate (Figure 2.2). Microcontrollers typically have under 100K of RAM (often just a few KBytes) and often assumed to be single-chip-systems; whereas the term SoC is typically used with more powerful processors which are capable to run complex software such as Windows or Linux, which need external memory chips (flash, RAM) to be useful, and which are used with various external peripherals. Systems-on-chips are largely used to reduce manufacturing costs and to enable smaller embedded systems. A typical SoC consists of: (a) One microcontroller (Figure 2.3), microprocessor or DSP core(s). Some SoCs (called multiprocessor System-on-Chip (MPSoC)) include more than one processor core, (b) Memory blocks including a selection of ROM, RAM, EEPROM and Flash, (c) Timing sources including oscillators and phase-locked loops,

Figure 2.2: AMD Geode is an x86 compatible system-on-a-chip.

(d) Peripherals including counter-timers, real-time timers and power-on reset generators, (e) External interfaces including industry standards such as USB, FireWire, Ethernet, USART, SPI, (f) Analog interfaces including ADCs and DACs, (g) Voltage regulators and power management circuits.

These blocks are connected by either a proprietary or industry-standard bus such as the AMBA bus. Finally, we note that SoCs can be fabricated by several technologies, including: (i) Full custom, (ii) Standard cell, (iii) FPGA.

2.1.4 Programmable Logic Controllers (PLC):

Being microprocessor-based devices, they have similar internal structures to many embedded controllers and computers[3]. They consist of CPU, Memories and I/O devices (Figure 2.4). These components are integral to these controllers. The main differences between PLCs and other microprocessor based devices are that PLC are units of rugged design for an industrial setting and are shielded for improved electrical noise immunity. Further they are modular, allowing easy replacement and addition of units. They support standardized I/O connections and signal levels and are designed for the ease of programming, to allow personnel unfamiliar with computer languages to program the PLCs in-plant. The CPU used in PLC is a standard CPU which is used in many other microprocessor controlled

[3]More details are available in the website of Plcopen: http://www.plcopen.org/

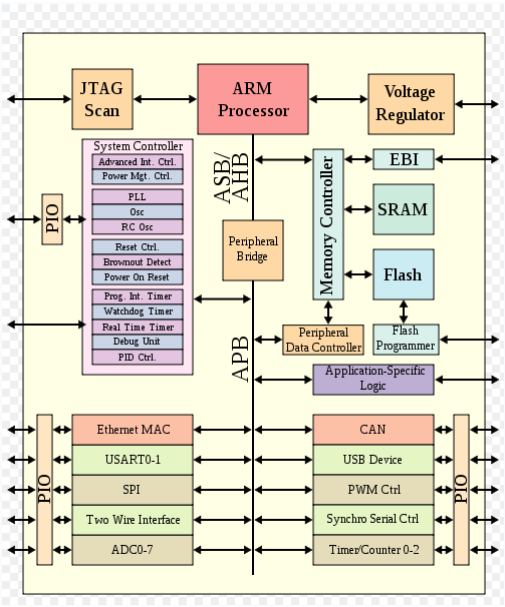

Figure 2.3: Microcontroller-based System-on-a-Chip.

systems. The choice of the CPU depends on the process to be controlled. Generally 8 or 16 bit CPUs fulfill the requirements adequately. Memory in PLC is divided into the program memory which is usually stored in EPROM/ROM, and the operating memory. The RAM memory is necessary for the operation of the program and the temporary storage of input and output data. Typical memory sizes of PLC systems are around 1kb for small PLCs, few kb for medium sizes and greater than 10-20 kb for larger PLC depending on the requirements. Many PLC would support easy memory upgrades. Input/Output units are the interfaces between the internal PLC and the external processes to be monitored and controlled. Small PLC units would have around 40 I/O connections with larger ones having more than 128 with either local or remote connections and extensive upgrade capabilities. Programming units are essential components of the PLC. Since they are used only in the development/testing stage of a PLC program, they are not permanently attached to the controller. The program in a ladder diagram or other form can be designed and usually tested before downloading to the PLC. The programming unit can be a dedicated device or a personal computer. It allows the graphical display of the program (ladder diagram). The unit, once connected to the PLC can download the program and allows for the real time monitoring of its operation to assist debugging. Once the program is found to operate as required the programming unit is disconnected from the PLC which continues the operation.

Figure 2.4: An example of PLC.

2.2 Real-Time Embedded Operating Systems (RTOS)

An embedded operating system is classically addressed for embedded computer architectures, and should be designed to be very compact and efficient, forsaking many functions that non-embedded computer operating systems provide, and which may not be used by the specialized applications they run. It is frequently a real-time operating system. A real-time operating system (RTOS) is an operating system (OS) intended for real-time applications, and offers programmers more control over process priorities. An application's process priority level may exceed that of a system process. Real-time operating systems minimize critical sections of system code, so that the application's interruption is nearly critical. A key characteristic of a real-time OS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task. A hard real-time operating system has less jitter (i.e. the variability) than a soft real-time operating system. A real-time OS that can usually or generally meet a deadline is a soft real-time OS, but if it can meet a deadline deterministically it is a hard real-time OS. A real-time OS has an advanced algorithm for scheduling where a scheduler flexibility enables a wider, computer-system orchestration of process priorities, but a real-time OS is more frequently dedicated to a narrow set of applications. Key factors in a real-time OS are minimal interrupt latency and minimal thread switching latency, but a real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time. Nowadays, the following real-time operating systems are the well-known and most widely deployed.

2.2.1 QNX

As a well-known microkernel-based OS, QNX is based on the idea of running most of the OS in the form of a number of small tasks, known as servers (rich details are available in http://www.qnx.com/). This differs from the more traditional monolithic kernel, in which the operating system is a single very large program composed of a huge number of "parts" with special abilities. In the case of QNX, the use of a microkernel allows users (developers) to turn off any functionality they do not require without having to change the OS itself; instead, those servers are simply not run. The system is quite small, with earlier versions fitting on a single floppy disk. QNX has been ported to a number of platforms and now runs on practically any modern CPU that is used in the embedded market. This includes the PowerPC, x86 family, MIPS, SH-4 and the closely related family of ARM, StrongARM and XScale CPUs.

2.2.2 RTLinux

It is a hard real-time RTOS microkernel that runs the entire Linux operating system (rich details are in http://www.rtlinuxfree.com/) as a fully preemptive process (Figure 2.5). Numerous versions of RT Linux are available, free or commercial. Two commonly available free RT Linux versions are: (i) Real-Time Application Interface (RTAI), developed by the Milan Polytechnical University and available at *www.aero.polimi.it/ rtai/*. (ii) RTL, developed by New Mexico Tech and now maintained by FSM Labs with a free version available at *www.rtlinux.org*. RTLinux was based on a lightweight virtual machine where the Linux Kernel was given a virtualized interrupt controller and timer, and all other hardware access was direct. From the point of view of the real-time component, the Linux Kernel is a thread. Interrupts needed for deterministic processing are processed by the real-time component, while other interrupts are forwarded to Linux, which runs at a lower priority than real-time threads. Linux drivers handle almost all I/O.

2.2.3 VxWorks

A popular real-time multi-tasking operating system for embedded microprocessor systems, designed by Wind River Systems of Alameda (rich details are available in www.windriver.com/). Like Unix and Linux, VxWorks is generally compliant with the IEEE's POSIX (Portable Operating System Interface) standard, version 1003.1b. The current release of VxWorks is version 5.4. VxWorks projects are usually developed in the Tornado 2 Integrated Development Environment (IDE) which provides for specifying a configuration (e.g., the libraries with which a project is linked), project builds, and code testing. VxWorks runs on many target processors including, but not limited to the following processors: Motorola PowerPC, 68K and CPU32 cores; MIPS; ARM; Intel X86 (386 and up) and i960. The key features of the current OS are: (a) Multitasking kernel with preemptive and round-robin scheduling and fast interrupt response, (b) Memory protection to isolate user applications from the

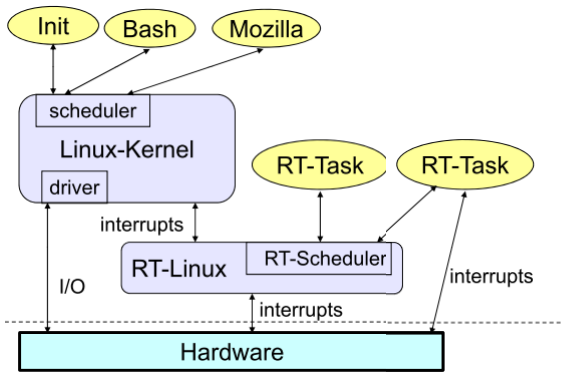

Figure 2.5: RT-Linux Architecture.

kernel, (c) symmetric multiprocessing (SMP) support, (d) Fast, flexible inter-process communication including Transparent Inter-Process Communication (TIPC), (e) Error handling framework, (f) Binary, counting, and mutual exclusion semaphores with priority inheritance, (g) Local and distributed message queues, (h) POSIX PSE52 certified conformance, (i) File system IPv6 networking stack, (j) VxSim simulator to simulate a VxWorks target for use as a prototyping and testing environment. Note that WindView provides advanced debugging tools for the simulator environment.

2.2.4 Windows CE

Also known officially as Windows Embedded Compact (rich details are available in www.windowsce.com/ Windows CE is an operating system developed by Microsoft for computers and embedded systems (Figure 2.6). Microsoft Windows CE's code base is separate from that of "industrial-strength" operating systems such as Windows 2000. CE was designed to run in memory- and power-constrained devices. It was also designed so that it could be quickly ported to new hardware architectures. The primary feature that differentiates Windows CE from competitors such as the Palm OS is that CE is a 32-bit, multi-threaded, multi-tasking operating system. Because Windows CE was designed to be portable to a wide range of processors, power management details differ from one device to the next. However, the CE API does support a set of power monitoring functions in order to allow applications to determine the remaining life of the battery, whether batteries are currently being used, and whether the batteries are currently being charged. The Windows CE API provides also access to a system object database. This database supports data compression, searching, sorting and synchronization with the desktop through the Microsoft ActiveSync services. Serial communications take place through a standard serial port on CE devices. The majority of the standard Win32 communication APIs have been ported to CE so it is very

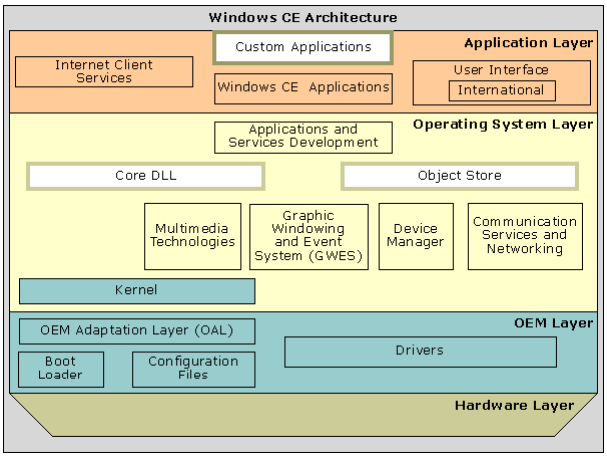

Figure 2.6: Windows CE Architecture.

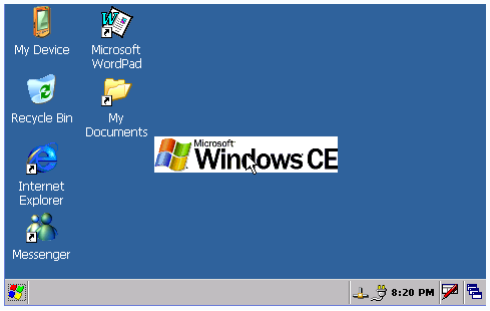

Figure 2.7: A Screenshot of Microsoft Windows CE 5.0.

likely that any communications code for Win32 can be quickly ported to a CE device with minor modifications. Windows CE devices can be expanded also to support standard Ethernet connections as well as wireless LAN connections. Finally, Windows CE devices can be expanded through their support of a CompactFlash slot. This expandability allows extra memory or other devices to be added quickly and inexpensively to a CE device (Figure 2.7).

2.3 Known Embedded Software Solutions

There are several different types of software architectures in common use for embedded systems.

2.3.1 Simple Control Loop

The software simply has a loop that calls subroutines, each of which manages a part of the hardware or software [72].

2.3.2 Interrupt Controlled System

Some embedded systems are predominantly interrupt controlled [40]. This means that tasks performed by the system are triggered by different kinds of events. An interrupt could be generated for example by a timer in a predefined frequency, or by a serial port controller receiving bytes. These kinds of systems are used if event handlers need a low latency and the event handlers are short and simple. Usually these kinds of systems run a simple task in a main loop also, but this task is not very sensitive to unexpected delays. Sometimes the interrupt handler will add longer tasks to a queue structure. Later, after the interrupt handler has finished, these tasks are executed by the main loop. This method brings the system close to a multitasking kernel with discrete processes.

2.3.3 Cooperative Multitasking

A form of multitasking where it is the responsibility of the currently running task to give up the processor to allow other tasks to run [49]. Cooperative multitasking requires the programmer to place calls at suitable points in his code to allow his task to be descheduled which is not always easy if there is no obvious top-level main loop or some routines run for a long time. If a task does not allow itself to be descheduled, then all the other tasks on the system will appear to "freeze" and will not respond to user action. The advantage of cooperative multitasking is that the programmer knows where the program will be descheduled and can make sure that this will not cause unwanted interaction with other processes.

2.3.4 Preemptive Multitasking or Multi-Threading

A type of multitasking where the scheduler can interrupt and suspend ("swap out") the currently running task in order to start or continue running ("swap in") another task [49]. The tasks under preemptive multitasking can be written as though they were the only task and the scheduler decides when to swap them. The scheduler must ensure that when swapping tasks, sufficient state is saved and restored that tasks do not interfere. The length of time for which a process runs is known as its "time slice" and may depend on the task's priority or its use of resources such as memory and I/O. We note OS/2, Unix use preemptive multitasking. This contrasts with cooperative multitasking where each task must include calls to allow it to be descheduled periodically.

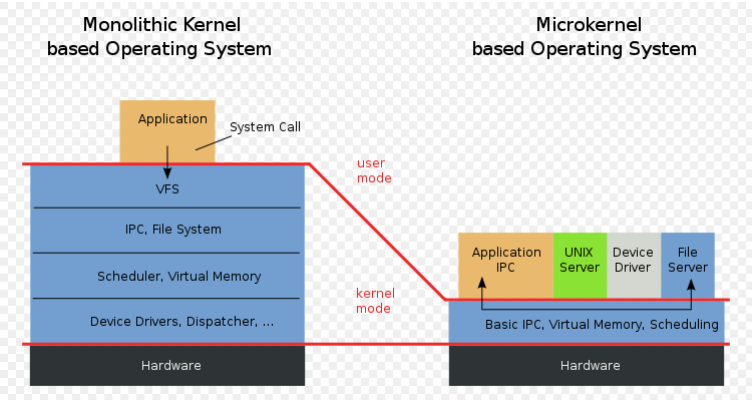

Figure 2.8: Structures of Microkernels and Monolothic Kernels.

2.3.5 Microkernels

A Microkernel provides the mechanisms needed to implement an operating system, such as low-level address space management, thread management, and inter-process communication [111, 41]. If the hardware provides multiple privilege levels, then the Microkernel is the only software executing at the most privileged level (generally referred to as supervisor or kernel mode). Actual operating system services, such as device drivers, protocol stacks, file systems and user interface code are contained in its user space.

2.3.6 Monolithic Kernels

A monolithic kernel (Figure 2.8) is a kernel architecture where the entire operating system is working in the kernel space and alone as supervisor mode [41]. The monolithic differs from other operating system architectures in that it defines alone a high-level virtual interface over computer hardware, with a set of primitives or system calls to implement all operating system services such as process management, concurrency, and memory management itself and one or more device drivers as modules. Common examples of monolithic kernels are Embedded Linux and Windows CE.

2.3.7 Additional Software Components:

in addition to the core operating system, many embedded systems have additional upper-layer software components. These components consist of networking protocol stacks like CAN, TCP/IP, FTP, HTTP, and HTTPS, and also included storage capabilities like FAT and flash memory management systems. If the embedded devices has audio and video capabilities, then the appropriate drivers and codecs will be present in the system. In

the case of the monolithic kernels, many of these software layers are included. In the RTOS category, the availability of the additional software components depends upon the commercial offering.

2.4 Conclusion

We cover in the first chapter the different well-known hardware technologies which are used in advanced embedded architectures. We present in particular four technologies: microcontrollers, digital signal processors, system on chip, and programmable logic controllers. We want to be independent of any one of them by assuming in the following of this manuscript a general hardware architecture to be composed of a processor, a memory and I-O interfaces. We cover also in this chapter the different well-known embedded operating systems where we detail QNX, RTLinux, VxWorks, and Windows CE. We want also in this research to be independent of any OS and any embedded software solutions, while assuming in the following that non-preemptive and preemptive scheduling policies are available to schedule tasks of embedded software.

Chapter 3

Embedded Systems: Overview on Software Components

The development of safe embedded systems is not a trivial activity because a failure can be critical for the safety of human beings (e.g. air and railway traffic control, nuclear plant control, aircraft and car control). They have classically to satisfy functional and temporal properties according to user requirements [15], but their time to market should be shorter than ever. To address all these important requirements, the component-based approach is studied in several academic research works and also in industrial projects to develop modular embedded systems. The general goal is to control the design complexity and to support the reusability of already developed components [39]. We present basic concepts of software components in this chapter, before describe thereafter well-known component-based architecture description languages (abbr, ADL) and industrial technologies [8].

3.1 Basic Concepts of Components

Nowadays, many definitions of software components are proposed in the component-based software development (CBSD). The best accepted and well-known definition is based on Szyperskis work [114]: *a component is a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently and is a subject to a third-party composition.* We follow in this book the definition of Szyperski by stressing the separation between the component's implementation and interface. Szyperski [114] tends to insist that components should be delivered in binary form, and that deployment and composition should be performed at run-time. To meet functional and real-time constraints in this manuscript, we assume off-line compositions of critical components. There are two basic prerequisites described in [8] and that enable components to be integrated and work together:

- A component model specifies the standards and conventions that components must follow to enable proper interactions,

- A component framework is the design-time and run-time infrastructure that manages resources for components and supports component's interactions.

There is an obvious correspondence between the conventions of a component model and the supporting mechanisms and services of a component framework. Component models and frameworks can be specified at different levels of abstraction as described in [8]:

- Some component models (e.g., COM) are specified on the level of the binary executable, and the framework consists of supporting OS services.
- Some component models (e.g., JavaBeans, CCM, or .Net) are specified on the level of byte- code.
- Some component models (e.g., Koala) are specified on the level of a programming language (such as C). The framework can contain glue code and possibly a runtime executive, which are bundled with the components before compilation.

We note also that a clear distinction between two perspectives of a component is defined in [8]:

- The component's implementation which is an executable realization to obey rules of the corresponding model. Depending on this model, component implementations are provided in binary form, byte code, compilable C code, etc.
- The component's interface summarizes the properties of the component that are externally visible to the other parts, and which can be used when designing the system. An interface may list the signatures of operations, in which case it can be used to check that components interact without causing type mismatches. An interface may contain additional information about the component's patterns of interaction with its environment or about extra-functional properties such as execution time; this allows more system properties to be determined when the system is first designed. An interface that, in addition to information about operation signatures, also specifies functional or extra-functional (for example temporal) properties is called a rich interface [8].

A contract is defined in the component-based approach as a specification of functional or extra-functional properties of a component, which are observable in its interface. A contract can be seen as specifying constraints on the interface of a component. Properties of components can be expressed in their contracts as defined in [8]. To structure the exposition into different types of component properties, we use the classification of contracts proposed in [21], where a contract hierarchy is defined consisting of four levels:

- **Level 1:** Syntactic interface, or signature (i.e. types, fields, methods, signals, ports etc., that constitute the interface),
- **Level 2:** Constraints on values of parameters and of persistent state variables, expressed, e.g., by pre- and post-conditions and invariants,

- **Level 3:** Synchronization between different services and method calls (e.g., expressed as constraints on their temporal ordering),
- **Level 4:** Extra-functional properties (in particular real-time attributes, performance, QoS (i.e. constraints on response times, throughput, etc.).

3.2 Architecture Description Languages

Architecture Description Languages (ADLs) have been developed as useful languages for expressing system's architectures as compositions of software modules and/or hardware objects. Typical concepts of ADLs are components, ports, connectors, etc. They can also describe various classes of component properties. Component properties expressed in a system description using an ADL should in principle be expressible in component interfaces. ADLs concentrate on the system's description, whose properties are the composition of properties visible in component interfaces. We present in the following some selected ADLs as described in [8].

3.2.1 Acme Language

Acme is a simple, generic software architecture description language (ADL) [8]. It is built on a core ontology of seven types of entities for the architectural representation: components, connectors, systems, ports, roles, representations, and rep-maps. Of the seven types, the most basic elements of any architectural description are components, connectors, and systems:

- **Components:** represent the primary computational elements and data stores of a system. Typical examples of components include such things as clients, servers, filters, objects, blackboards, and databases,
- **Connectors:** represent interactions among components. Computationally speaking, connectors mediate the communication and coordination activities among components. Examples include simple forms of interaction, such as pipes, procedure call, and event broadcast. But connectors may also represent more complex interactions, such as a client-server protocol or a SQL link between a database and an application.
- **Systems:** represent configurations of components and connectors.

3.2.2 Rapide Language

The language Rapide is generally used for building large-scale, distributed multi-language component-based systems [8]. This technology defines a component as a module implementing interfaces, and connectors as modules to connect sending and receiving interfaces.

Rapide is based upon a new generation of computer languages, called Executable Architecture Definition Languages (EADLs), and an innovative tool set supporting the use of EADLs in evolutionary development and rigorous analysis of large-scale systems.

3.2.3 Wright Language

The language Wright based on the classic concepts of components and connectors, provides formal basis for architectural specifications and analysis of software systems. Work on Wright has focused on the concept of explicit connector types, on the use of automated checking of architectural properties, and on the formalization of architectural styles. To further aid developers in the realization and exploitation of architectural abstractions, Wright defines a set of standard consistency and completeness checks that can be used to increase the designer's confidence in the design of a system. These checks are defined precisely in terms of Wright's underlying models, and can be checked by using a standard model checking technology [8].

3.2.4 Aesop Language

The Aesop language is used for the architectural design of component-based systems. It provides a generic toolkit and communication infrastructure that users can customize with architectural style descriptions, and a set of tools that they would like to use for architectural analysis. Example of Aesop tools include cycle detectors, type consistency verifiers, formal communication protocol analyzers, C-code generators, compilers, structured language editors, and rate-monotonic analysis tools [8]. An architectural style description includes items such as a vocabulary of design elements (components, connectors, and patterns) along with their associated semantics, global design rules, customized visualizations, and other information, if desired.

3.2.5 Metah Language

The language Metah provides means to express, analyze and implement architectures of embedded real-time software components [8]. This language works with architectural characteristics to predict overall system's behavior and permit rapid reconfigurations of components. Metah is a Computer Aided Software Engineering (CASE) tool set for architectural specifications, analysis, integrations and verifications of real-time embedded systems.

3.2.6 Architecture Analysis and Design Language

The Architecture Analysis and Design Language (AADL) is an architecture description language. AADL was first developed in the field of avionics, and was known formerly as the Avionics Architecture Description Language. It is derived from MetaH. AADL is used

to model the software and hardware architecture of an embedded, real-time system. Due to its emphasis on the embedded domain, AADL contains constructs for modelling both software and hardware components. This architecture model can then be used either as a design documentation, for analysis (such as schedulability and flow control) or for code generation [8].

3.2.7 Unicon Language

UniCon is an architectural description language whose focus is on supporting the variety of architectural parts and styles found in the real world and on constructing systems from their architecture descriptions. An architecture description in UniCon consists of a set of components and connectors. A component is a locus of data or computation, while a connector mediates the interaction among components. Each component has an interface that exports a set of players. These players engender the ways in which the component can interact with the outside world. Similarly, a connector's protocol exports a set of roles that engender the ways in which the connector can mediate interaction [8].

3.2.8 Darwin Language

Darwin is an Architecture Description Language (ADL) that can be used in a context of software engineering to describe the organization of a piece of software in terms of components, their interfaces, and the bindings between components. In comparison to others ADLs, such as Wright, the language does not provide the notion of connectors [8].

3.3 Component-based Technologies

Nowadays, rich component-based technologies have been proposed to develop embedded control systems. We present some of them which are well-known and well-used in Industry as described in [8].

3.3.1 Industrial Technology IEC61131

In the area of Industrial Automation, the programmable logic controllers are a widely used technology. However, for the last twenty years, the corresponding applications have been written in many different languages, resulting in inefficient work for technicians, maintenance personnel and system designers. For instance, there are numerous versions of the so-called ladder diagram language, and furthermore this language is poorly equipped with facilities such as: (a) Control over program execution, (b) Definition and manipulation of data structures, (c) Arithmetic operations or, (d) hierarchical program decomposition. These problems led to the constitution of a working group within the International Electrotechnical Commission IEC, with the aim to define a standard for the complete design of

programmable logic controllers. While previous efforts had been made before, IEC61131 has received worldwide international and industrial acceptance. The first document introducing the general concepts was published in 1992 and followed by the definition of equipment requirements and tests. The core of the standard is its third part, published in 1993, which describes the harmonization and coordination of the already existing programming languages. The technology IEC61131 is described as follows:

- Component types: an application is divided into a number of blocks,
- Supported languages: a block is written in any of the languages proposed in the standard. There are two textual languages (ST, IL) and three graphical languages (FBD, LD, SFC):
  - Function Block Diagram (FBD) is used for the description and regulation of signal and data flows through Function Blocks. It can nicely express the interconnection of control system algorithms and logic,
  - Structured Text (ST) is a high level textual language, with a Pascal-like syntax,
  - Instruction List (IL) is an assembler-like language, found in a wide range of PLCs,
  - Ladder Diagram (LD) is a graphical language based on the relay ladder logic, which allows the connection of previously defined blocks. For historical reasons, it is the most frequently used in actual PLC programs.
  - Sequential Function Chart (SFC) is used to combine in a structured way units defined with the four languages above. It mainly describes the sequential behavior of a control system and defines control sequences that are time- and event-driven. It can express both high-level and low-level parts of a program.

3.3.2 Industrial Technology IEC61499

We present the main concepts of the Component-based International Industrial Standard IEC61499 [50] which is an extension of the technology IEC 61131.3. According to this standard, a Function Block (FB) (figure 3.1) is a unit of software supporting functionalities of an application [73, 121, 84]. It is composed of an interface and an implementation such that the interface contains data/event inputs and outputs supporting interactions with the environment. Events are responsible for activations of the block while data contain valued information. The implementation of the block contains algorithms to execute when corresponding events occur. The selection of an algorithm to execute is performed by a state machine called Execution Control Chart (ECC) which is also responsible for sending output events at the end of the algorithm execution. The block $BELTFB$ shown in Figure 3.1 is a FB used to control conveyer belts. It is activated by the input events : $INIT$, OBJ_ARR, OBJ_LEFT and $STOPI$, and responds to the environment by the output events $INITO$, CNF, $MOVEF$ and $STOPO$. When the event OBJ_ARR occurs,

the state $OBJ\_ARRIVED$ is activated as shown in Figure 3.2 to execute the algorithm $Inc.Counter$. Once such execution finishes, the ECC sends the output event CNF and activates the states $MOVE$ **OR** $START$ depending on the value of the internal variable $Count$. In particular, when the output event CNF has to be sent, the block updates the corresponding output data $COUNT$ and $SPEED$. According to the Standard IEC61499, a control application is specified by a network of FBs where each event input (resp. output) of a block is linked to an event output (resp. input) by a channel and corresponds otherwise to a global input (resp. output). Data inputs and outputs follow the same rules. The architecture of the execution environment is well defined by a network of devices where each one is composed of one processing unit and interfaces (with sensors, actuators and the network). Moreover, it is characterized by logical execution unit(s) called resource(s). A resource defines *the important boundary existing between what is within the scope of the IEC61499 model and what is device (OS) and networks (communication protocols)* [73].

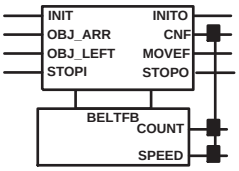

Figure 3.1: An IEC 61499 Function Block

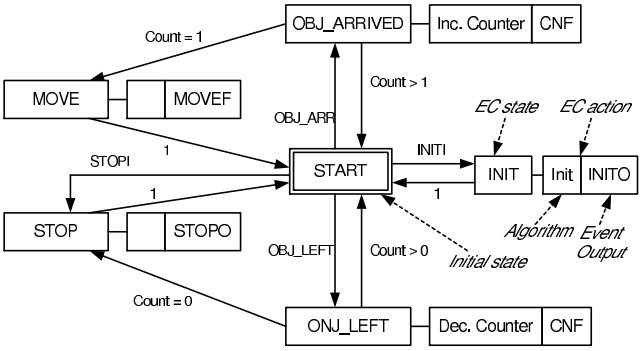

Figure 3.2: The ECC of the FB $BeltFB$

3.3.3 Port Based Object Technology

The technology Port-Based Objects combines the object-based design with port automaton design. A port-based object which is a software control module, is defined as an object

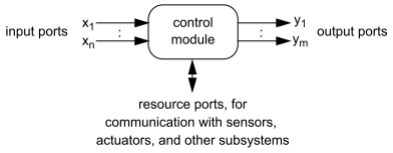

Figure 3.3: A PBO module

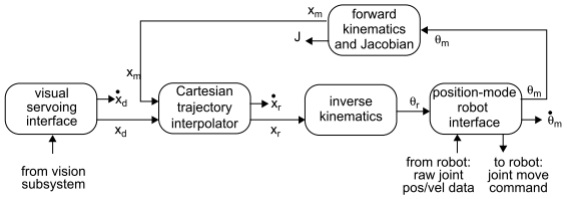

Figure 3.4: Network of PBO modules

that has various ports for real-time communications. Each module has a state and is characterized by its methods to be hidden from other objects. Only the ports of an object are visible to other objects. A simplified model of a port-based object is shown in Figure 3.3. Each module has zero or more input ports, zero or more output ports, and may have any number of resource ports. Input and output ports are used for communication between tasks in the same subsystem, while resource ports are used for external communication to the subsystem, such as with the physical environment, other subsystems, or a user interface. A link between two objects is created by connecting an output port of one module to a corresponding input port of another module. A configuration can be legal only if every input port in the system is connected to one, and only one, output port, but a single output may be used as input by multiple tasks (Figure 3.4).

A Port-Based Object can have two kinds of inputs: constant input that needs to be read during initialization (in-const), and variable input which must be read at the beginning of each control cycle (in-var) for periodic tasks, or at any start of event processing for aperiodic tasks. Similarly, a task can have output constants (out-const) or output variables (out-var). Both the constants and variables are transferred through the global state variable table. The input and output connections shown in the control module library in Figure 3.5 are all variables. An example of PBO objects implementing a PID algorithm is described in Figure 3.6. It uses three modules: the *joint position trajectory generator*, the *PID joint position controller*, and the *torque-mode robot interface*. We present also in Figure 3.7 *Speed Regulator* as a simple example of PBO components regulating the vehicle speed. The component *cyclic* periodically sends desired values to *Regulate* which regulates measured values from $Interface$.

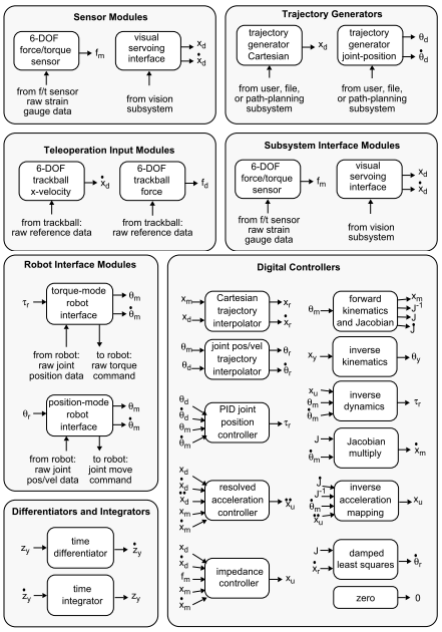

Figure 3.5: Network of PBO modules

3.3.4 Koala Technology

The technology Koala is developed and used at Philips [8]. It was designed to build software control units for consumer products such as televisions, video recorders, CD and DVD players, and recorders. A Koala component is a piece of code that can interact with its environment through explicit interfaces. As a consequence, a basic component has no dependencies to other components, and is characterized as follows:

- Component Implementation is a directory with a set of C and header files that may use each other in arbitrary ways, but communication with other components is routed only through header files generated by the Koala compiler,
- Component Interface: the directory also contains a component definition file, describing among other things the interfaces of the component.

3.3.5 Pecos Technology

The PECOS project aims to develop component-based embedded systems such as smart cell phones, PDAs, and industrial field devices [8]. It defines a component model as follows::

- Component's Interfaces are defined by Input-Output Ports, and connectors to connect compatible ports.

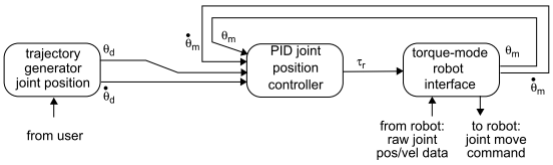

Figure 3.6: Example of PBO Objects implementing a PID Algorithm

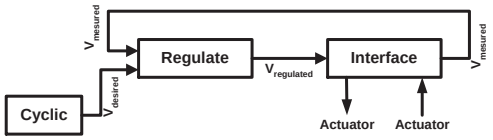

Figure 3.7: A PBO Component : *Speed Regulator*

- Component's Types: Active Components (with own thread), Passive Components (encapsulating behavior without threads), Event Components (triggered by events),
- Component's attributes can specify memory consumption, WCET, cycle time, and priority,

3.3.6 Rubus Technology

Rubus is a small Real Time Operating System, developed by Arcticus Systems (www.arcticus.se/) [8]. It is divided into a first part supporting time-triggered execution and a second supporting event-triggered execution. Time-triggered execution is to support hard real-time applications with a deterministic execution mechanism. To support component-based hard real-time systems, Arcticus Systems propose a component model and associated development tools for use with the Rubus operating system. A basic software component consists of a behavior, a persistent state, a set of in-ports/out-ports and an entry function which is its main functionality. A task provides the thread of execution for a component. The entry function takes as an argument a set of in-ports, the persistent state, and the reference to the out-ports. The attributes of a task are Task ID, Period, Release Time, Deadline, and WCET. In addition, precedence and mutual exclusion ordering between tasks can be specified. We present in Figure 3.8 *BrakeSystem* as a simple example of Rubus components to use in a vehicle. The component *BrakeLeftRight* allows to brake left or right by considering the pressure and also the speed of the vehicle. Detailed descriptions of this example are available in [32].

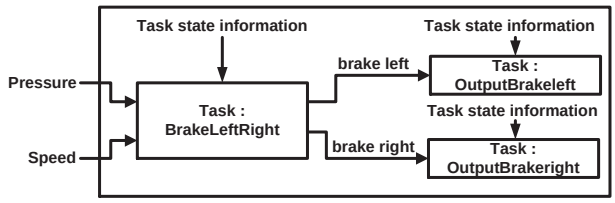

Figure 3.8: A Rubus Component : *BrakeSystem*

3.4 Conclusion

We present in this chapter the concept of component-based embedded control systems. A component is classically characterized by an implementation as well as a set of interfaces to interact with its environment. The implementation can be in a binary form, byte code, compilable C code, etc. Nowadays, several architecture description languages and component-based technologies have been proposed to develop systems. We describe in this research the well-known among them by detailing in particular their concepts of software components. Nevertheless, we want in the following to be independent of any technology and any language by defining a general concept of Control Components for the development of embedded control systems. The goal is to be able to reuse already developed components and exploit different rich libraries of ADL and Industrial Technologies for future developments of new embedded control systems.

Chapter 4

Formalisms for Modelling and Verification of Embedded Systems

A Finite State Machine is a model of computation consisting of a set of states, a start state, an input alphabet, and a transition function that maps input symbols and current states to next states. Computation begins in the start state with an input string. It changes to new states depending on the transition function. We present in this chapter the formalism Net Condition/Event Systems (NCES) which is an extension of Petri nets that belong to the family of finite state machines. We present thereafter the temporal logic "Computation Tree Logic" which is used for the specification of properties of NCES.

4.1 Petri nets

A Petri net is one of several mathematical modelling languages for the description of systems [98]. It consists of places, transitions, and directed arcs. Arcs run from a place to a transition or vice versa, never between places or between transitions. The places from which an arc runs to a transition are called the input places of the transition; the places to which arcs run from a transition are called the output places of the transition. Places may contain a natural number of tokens. A distribution of tokens over the places of a net is called a marking. A transition of a Petri net may fire whenever there is a token at the end of all input arcs; when it fires, it consumes these tokens, and places tokens at the end of all output arcs. A firing is atomic, i.e., a single non-interruptible step. The execution of Petri nets is non-deterministic: when multiple transitions are enabled at the same time, any one of them may fire. If a transition is enabled, it may fire, but it doesn't have to. Formally, a Petri net is defined by the following 3-tuple:

- S is a finite set of places,
- T is a finite set of transitions,
- S and T are disjoint, i.e. no object can be both a place and a transition,

- $W = (S \times T) \cup (T \times S) \longrightarrow \mathbb{N}$ is a multi set of arcs, i.e. it defines arcs and assigns to each arc a non-negative integer arc multiplicity; note that no arc may connect two places or two transitions.

The flow relation is the set of arcs: $F = \{(x, y)/W(x, y) > 0\}$. In many textbooks, arcs can only have multiplicity 1, and they often define Petri nets using F instead of W. A Petri net graph is a bipartite graph $(S \cup T, F)$ with node partitions S and T. The preset of a transition t is the set of its input places: ${}^\bullet t = \{s \in S/W(s, t) > 0\}$; its postset is the set of its output places: $t^\bullet = \{s \in S/W(t, s) > 0\}$. A marking of a Petri net (graph) is a multi set of its places, i.e., a mapping $M : S \longrightarrow \mathbb{N}$. We say the marking assigns to each place a number of tokens. A Marked Petri net is a 4-tuple (S, T, W, M_0), where:

- (S, T, W) is a Petri net graph,
- $M0$ is the initial marking.

4.2 NCES Formalism: Extension of Petri Net

The formalism of Net Condition/Event Systems (NCES) is an extension of Petri nets. It was introduced by Rausch and Hanisch in [97] and further developed through the last years, in particular in [42], according to which a NCES is a place-transition net formally represented by a tuple (figure 4.1):

$$NCES = (P, T, F, CN, EN, C^{in}, E^{in}, C^{out}, E^{out},$$

$$B_c, B_e, C_s, D_t, m_0) \text{ where,}$$

(i) P (resp, T) is a non-empty finite set of places (resp, transitions), (ii) F is a set of flow arcs, $F : (PXT) \bigcup (TXP)$, (iii) CN (resp, EN) is a set of condition (resp, event) arcs, $CN \subseteq (PXT)$ (resp, $EN \subseteq (TXT)$), (iv) C^{in} (resp, E^{in}) is a set of condition (resp, event) inputs, (v) C^{out} (resp, E^{out}) is a set of condition (resp, event) outputs, (vi) B_c (resp, B_e) is a set of condition (resp, event) input arcs in a NCES module, (vii) $B_c \subseteq (C^{in}XT)$ (resp, $B_e \subseteq (E^{in}XT)$), (viii) Cs (resp, Dt) is a set of condition (resp, event) output arcs, (ix) $Cs \subseteq (PXE^{out})$ (resp, $Dt \subseteq (TXE^{out})$), (x) $m_0 : P \to 0, 1$ is the initial marking.

The semantics of NCES are defined by the firing rules of transitions. There are several conditions to be fulfilled to enable a transition to fire. First, as it is in ordinary Petri nets, an enabled transition has to have a token concession. That means that all pre-places have to be marked with at least one token. In addition to the flow arcs from places, a transition in NCES may have incoming condition arcs from places and event arcs from other transitions. A transition is enabled by condition signals if all source places of the condition signals are marked by at least one token. The other type of influence on the firing can be described by event signals which come to the transition from some other transitions. Transitions having no incoming event arcs are called *spontaneous*, otherwise *forced*. A forced transition is enabled if it has token concession and it is enabled by condition and event signals [97].

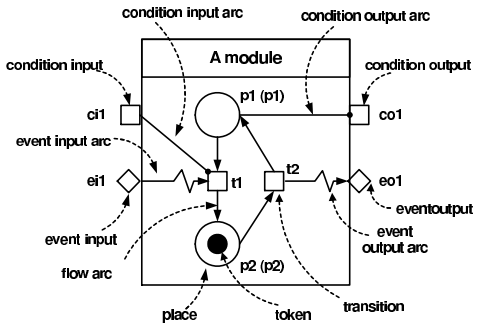

Figure 4.1: A module of Net Condition/Event Systems

On the other hand, the NCES formalism is enriched last years to consider time constraints applied to the input arcs of transitions: to every pre-arc of a transition, an interval [eft,lft] of natural numbers is attached with $0 \leq eft \leq w$ (w is a fixed integer). The interpretation is as follows, every place p bears a clock which is running iff the place is marked and switched off otherwise. All running clocks run at the same speed measuring the time the token status of its place has not been changed i.e. the clock on a marked place p shows the age of the youngest token on p. If a firing transition t is able to remove a token from the place p or adds a token to p then the clock of p is turned back to 0. In addition, a transition t is able to remove tokens from its pre-places (i.e. to fire) only if for any pre-place p of t the clock at place p shows a time $u(p)$ such that $eft(p,t) \leq u(p) \leq lft(p,t)$. Hence, the firing of transitions is restricted by the clock positions.

4.3 Temporal Logic

The "Computation Tree Logic" CTL offers facilities for the specification of properties to fulfill by the system behavior [99, 100]. In this section, we briefly present this logic, its extension "Extended Computation Tree Logic" (denoted by eCTL) and the "Timed Computation Tree Logic" (denoted by TCTL).

4.3.1 Computation Tree Logic

In CTL, all formulae specify behaviors of the system starting from an assigned state in which the formula is evaluated by taking paths (i.e. sequence of states) into account. The semantics of formulae is defined with respect to a reachability graph where states and paths are used for the evaluation. A reachability graph M consists of all global states that the system can reach from a given initial state. It is formally defined as a tuple $M = [Z, E]$ where,

- Z is a finite set of states,

- E is a finite set of transitions between states, i.e. a set of edges (z, z'), such that z, z' $\in Z$ and z' is reachable from z.

In CTL, paths play the key role in the definition and evaluation of formulae. By definition, a path starting in the state z_0 is a sequence of states $(z_i) = z_0 z_1 ...$ such that for all $j \geq 0$ it holds that there is an edge $(z_j, z_{j+1}) \in E$. In the following, we denote by (z_i) such path. The truth value of CTL formulae is evaluated with respect to a certain state of the reachability graph. Let $z_0 \in Z$ be a state of the reachability graph and φ a CTL formula, then the relation $\models$ for CTL formulae is defined inductively.

* *Basis:*

  ** $z_0 \models \varphi$ iff the formula φ holds in z_0,

  ** $z_0 \models true$ always holds,

  ** $z_0 \models false$ iff never holds,

* *Steps:*

  ** $z_0 \models EF\varphi$ iff there is a path (z_i) and $j \geq 0$ such that $z_j \models \varphi$,

  ** $z_0 \models AF\varphi$ iff for all paths (z_i) there exists $j \geq 0$ such that $z_j \models \varphi$,

  ** $z_0 \models AG\varphi$ iff for all paths (z_i) and for all $j \geq 0$ it holds $z_j \models \varphi$,

4.3.2 Extended Computation Tree Logic

In CTL, it is rather complicated to refer to information contained in certain transitions between states of a reachability graph. A solution is given in [99, 100] for this problem by proposing an extension of CTL called Extended Computation Tree Logic ECTL. A transition formula is introduced in ECTL to refer to a transition information contained in the edges of the reachability graph. Since it is wanted to refer not only to state information but also to steps between states, the structure of the reachability graph $M = [Z, E]$ is changed as follows:

- Z is a finite set of states,
- E is a finite set of transitions between states, i.e. a set of labeled edges (z,s,z'), such that $z, z' \in Z$ and z' is reachable from z by executing the step s.

Let $z_0 \in Z$ be a state of the reachability graph, τ a transition formula and φ an ECTL formula. The relation $\models$ for ECTL formulae is defined inductively:

- $z_0 \models E\tau X\varphi$: iff there exists a successor state z_1 such that there is an edge (z_0, s, z_1) $\in E$ where $(z_0, s, z_1) \models \tau$ and $z_1 \models \varphi$ holds,
- $z_0 \models A\tau X\varphi$: iff $z_1 \models \varphi$ holds for all successors states z_1 with an edge $(z_0, s, z_1) \in E$ such that $(z_0, s, z_1) \models \tau$ holds,

4.3.3 Timed Computation Tree Logic

TCTL is an extension of CTL to model qualitative temporal assertions together with time constraints. The extension essentially consists in attaching a time bound to the modalities and we note that a good survey can be found in [4]. For a reachability graph $M = [Z, E]$, the state delay D is defined as a mapping $D : Z \to N_0$ and for any state $z = [m, u]$ the number $D(z)$ is the number of time units which have to elapse at z before firing any transition from this state. For any path (z_i) and any state $z \in Z$ we put:

- $D[(z_i), z] = 0$, if $z_0 = z$,
- $D[(z_i), z] = D(z_0) + D(z_1) + ... + D(z_{k-1})$, if $z_k = z$ and $z_0, ..., z_{k-1} \neq z$,

With other words, $D[(z_i), z]$ is the number of time units after which the state z on the path (z_i) is reached the first time, i.e. the minimal time distance from z_0. Let $z_0 \in Z$ be a state of the reachability graph and φ a TCTL formula. The relation $\models$ for TCTL is defined as follows:

- $z_0 \models EF[l, h]\varphi$, iff there is a path (z_i) and a $j > 0$ such that $z_j \models \varphi$ and $l \leq D((z_i), z_j) \leq h$,
- $z_0 \models AF[l, h]\varphi$, iff for all paths (z_i), there is a $j > 0$ such that $z_j \models \varphi$ and $l \leq D((z_i, z_j) \leq h$,

4.4 Conclusion

We present in this chapter the formalism Net Condition/Event Systems (NCES) which is an extension of Petri nets, and which will be used for the modelling of reconfigurable embedded control systems in following sections. We present also the temporal logic Computation Tree Logic that we will use to describe functional and temporal properties of such systems. These choices are done regarding the rich experience of the host laboratory[1] in this domain.

[1]Research Laboratory on Automation Technology at Martin Luther University in Germany

Chapter 5

Problems and Contributions

We want in this monograph to propose new technical solutions for developments of reconfigurable embedded control systems following software component-based approaches. We are interested in automatic reconfigurations of autonomous and adaptive embedded controllers to be automatically changed at run-time.

The first question to ask in the following chapter is what Architecture Description Language or Industrial Technology should we choose to develop systems? the answer is difficult since each language or technology is rich, useful and well-used by known organizations and companies in the World (Chapter 4). A solution that we propose is to define a general concept of Control Components to be assumed as software units for controls of physical processes. A Control Component can be a Koala module, a PBO object, a Function Block or also any concept of components defined by an ADL. It is composed of an implementation supporting component's tasks and an interface for external interactions. We define a formalization of reconfigurable embedded control systems to be assumed as networks of Control Components with precedence constraints that we model according to the formalism Net Condition/Event systems (NCES). We define in the same chapter a new definition of reconfigurations that should be applied to bring the whole system at run-time to optimal and safe behaviors. We classify thereafter all possible reconfiguration scenarios to three forms dealing with the modification of the software architecture, the composition of components or the easy modification of data. This classification is applied to two Benchmark Production Systems FESTO and EnAS following the technology IEC61499 and available at Martin Luther University in Germany.

We are interested in Chapter 7 in the modelling and verification of component-based reconfigurable embedded control systems. The question is how can we model any form of reconfigurations ? how to perform correct and coherent distributed reconfigurations of different devices ? how can we apply a model checking while controlling the verification complexity ? finally an important question is how to implement distributed automatic reconfigurations ? We define in this chapter a multi-agent reconfigurable architecture where a Reconfiguration Agent is proposed for each device of the execution environment to handle local reconfigurations, and a Coordination Agent is defined to coordinate between devices

by using a well-defined communication protocol. the goal is to allow coherent and feasible distributed reconfigurations. We model these agents by NCES and apply the model checker SESA to check their feasibility. We use in this case the temporal logic "Computation Tree Logic" to describe functional and temporal properties of this architecture. Once agents checked, the next step to be addressed is the verification of the different networks of Control Components that can be executed after different reconfiguration scenarios. We define therefore a refinement-based approach that checks in step by step each network for the definition, modelling and verification of Control Components. The last section of this chapter deals with implementations of reconfiguration scenarios according to the technology IEC61499.

We are interested in Chapter 8 in feasible execution models of reconfigurable real-time embedded control systems. We mean by an execution model the set of feasible OS tasks encoding the system. We are not restricted in this research to a particular Operating System while assuming that preemptive and non-preemptive Earliest Deadline First policies are supported. We define an approach that constructs the different feasible sets of OS tasks implementing the system after different possible automatic reconfiguration scenarios. Only one set should be loaded in memory after the corresponding scenario. These tasks are verified by applying previous solid results in the theory of real-time scheduling.

We are interested in Chapter 9 in the low power scheduling of reconfigurable real-time embedded control systems. We assume synchronous real-time tasks starting at t=0 such that their deadlines are equal to their periods. After a particular reconfiguration scenario, the energy consumption should be stable or decreased in order to consider limitations in the embedded batteries. We define technical solutions allowing modifications of the system parameters if such consumption is increased after any scenario. In this case, we propose to change periods of some tasks, to change their execution times or to remove some of them. We present several simulations proving the benefits of our solutions.

Finally, we are interested in Chapter 10 in optimal simulations of reconfigurable real-time embedded control systems which is not an exhaustive approach. We limit our research to simulations of a network of components specifying the system after a particular scenario. We propose in this case to inject faults in order to bring their behaviors to well-defined critical executions. We define a master-slave architecture where the master decides new errors to be injected by the slaves which are located in different levels of the system hierarchical models. This contribution is applied to the footwear factory of the ITIA-CNR Institute in Italy where simulations and analysis are made.

Chapter 6

Reconfigurable Component-based Embedded Systems: New Motivations

Although each one of Architecture Description Languages and component-based technologies is rich and useful, we want in our research work to be independent of any one of them by defining the general concept of "Control Component" as an event-triggered software unit to be composed of an interface for external interactions and an implementation that provides control computations of physical processes [58]. A Control Component in this book can be a Function Block according to the Standard IEC61499, a Koala component according to the Koala technology, a software component according to an ADL..., etc. It checks and interprets evolutions of the environment (i.e. user new commands, evolution of corresponding physical processes or execution of other previous components) by reading data from sensors before possible reactions to activate corresponding actuators in the plant. We define the concept of Container to gather components sharing the control of same physical processes. A container defines a logic execution unit corresponding to time slots of the processing unit. It corresponds in the operational level to an OS task. We compose the plant of different physical processes under precedence constraints that define the production order. It is formalized by sets of sensors and actuators under precedence constraints to control corresponding physical processes.

A crucial criterion to consider for new generations of embedded systems is the automatic improvement of their performance at run-time. Indeed, these systems should dynamically and automatically improve their quality of service according to well-defined conditions. In addition, they should dynamically reduce the memory occupation and therefore the energy consumption in order to decrease the execution cost. They have also to dynamically reduce the number of running controllers or also the traffic on used communication networks. We define in this chapter a new reconfiguration semantic that we apply to Benchmark Production Systems available in the research laboratory of Prof.Dr. Hans-Michael Hanisch at Martin Luther University. To cover all possible reasons in industry, we define different

reconfiguration forms to change the architectures, structures or data of the whole system to safe and preferment behaviors.

6.1 Plant Formalization

We denote in the following by Sys the control system that controls, by reading data from sensors and activating corresponding actuators, the plant which is classically composed of a set of physical processes denoted by "$Plant$". Let $\alpha_{sensors}$ and $\alpha_{actuators}$ be respectively the set of sensors and actuators in the plant. For each sensor $sens \in \alpha_{sensors}$, we assume that any data reading by Sys is an event $ev = event(sens)$, and for each actuator $act \in \alpha_{actuators}$, we define a couple of events $(activ, cf) = activation(act)$ that corresponds to the activation of and the confirmation from act. Let $\phi_{sensors}$ be the set of events to occur when data are read from sensors of $\alpha_{sensors}$ and let $\phi_{actuators}$ be the set of couples of events when actuators of $\alpha_{actuators}$ are activated.

$$\phi_{sensors} = \{ev / \exists sens \in \alpha_{sensors}, ev = event(sens)\}$$

$$\phi_{actuators} = \{(activ, cf) / \exists act \in \alpha_{actuators}, (activ, cf) = activation(act)\}$$

We characterize each process $\varphi \in Plant$ by (i) a set denoted by $sensors(\varphi)$ of sensors that provide required data by Sys before the activation of φ; (ii) a set of actuators denoted by $actuators(\varphi)$ and activated by the system under well defined conditions. The control of the different physical processes of "$Plant$" should satisfy a partial order that we characterize as follows for each actuator $act \in \alpha_{actuators}$: (i) $prev(act)$: a set of actuators to be activated just before the activation of act, (ii) $follow(act)$: a set of actuator sets such that only one set should be activated between all sets in a particular execution scenario when the activation of act is done, (iii) $sensor(act)$: a set of sensors that provide required data by Sys before any activation of act. We denote in the following by $first(\alpha_{actuators})$ (resp. $last(\alpha_{actuators})$) the set of actuators with no predecessors in the plant: they are the first (resp. last) to be activated by the system.

6.2 Control Components

We define in this section the concept of "Control Components" for embedded control systems to be assumed in the following as networks of components with precedence constraints that allow controls of physical processes by reading and interpreting data from sensors before possible reactions and activations of corresponding actuators. To check the whole behavior of a control system when errors are assumed to occur at run-time, NCES-based models are proposed thereafter for these components.

6.2.1 Formalization

We define a Control Component CC as an event-triggered software unit of Sys to control physical processes of the plant. It is composed of an interface for external interactions with the environment (the plant or other Control Components), and an implementation which is a set of algorithms for interpretations of input data from sensors of $\alpha_{sensors}$ before possible activations of corresponding actuators of $\alpha_{actuators}$. The system Sys is assumed to be a set of Control Components with precedence constraints such that each component should start its control task of the plant when all its predecessors finish their executions. We define event flows to be exchanged between system components to order their executions according to their precedence constraints. We denote in the following by $\varphi(CC)$ the set of Control Components of Sys such that each component is characterized as follows:

$$CC = \{pred, succ, sens, act\}, \text{ where:}$$

(i) $pred(CC)$: the set of Control Components that should be executed before the activation of CC, (ii) $succ(CC)$: the set of Control Components to be activated when the execution of CC is well-finished, (iii) $sens(CC)$: the set of sensors that provide required data for CC, (iv) $act(CC)$: the set of actuators to be activated by CC. We assume a buffer in this research to store input events until their treatment by the components. To organize the distribution of Control Components in a same device controlling a subset of physical processes of *"Plant"*, we define the concept of logical containers. In the functional level, a container gathers components that share the control of same processes, and corresponds in the execution level to an OS task that can be under real-time constraints. Therefore, the Control Components of the system are distributed on several containers that should be assigned to OS tasks from the functional to operational architectures.

6.2.2 Specification

We specify the behavior of a Control Component CC by a NCES-based model that has only one initial place and is characterized by a set of traces such that each trace tr contains the following transitions: (i) $i(CC, sensors\_set)$: a transition of the CC model that allows data readings from a subset of sensors $sensors\_set \subseteq \alpha_{sensors}$, (ii) $a(CC, act\_set)$: a transition of the CC model that allows the activation of the subset $act\_set \subseteq \alpha_{actuators}$, (iii) $cf(CC, act\_set)$: a transition allowing a final confirmation from actuators of act_set once the corresponding physical processes finish their executions.

6.3 Reconfigurable Systems: State of the Art

Nowadays, rich research works have been proposed to develop reconfigurable embedded systems. The authors propose in [6] reusable Function Blocks to implement a broad range of embedded systems where each block is statically reconfigured without any re-programming.

This is accomplished by updating the supporting data structure, i.e. a state transition table, whereas the executable code remains unchanged and may be stored in permanent memory. The state transition table consists of multiple-output binary decision diagrams that represent the next-state mappings of various states and the associated control actions. The authors propose in [101] a complete methodology based on the human intervention to dynamically reconfigure control systems. They present in addition an interesting experimentation showing the dynamic change by users of a Function Block's algorithm without disturbing the whole system. The authors use in [116] Real-time-UML as a meta-model between design models and their implementation models to support dynamic user-based reconfigurations of control systems. The authors propose in [22] an agent-based reconfiguration approach to save the whole system when faults occur at run-time. Finally the authors propose in [2] an ontology-based agent to perform system reconfigurations that adapt changes in requirements and also in the environment. They are interested to study reconfigurations of control systems when hardware faults occur at run-time.

Although the applicability of these contributions in industry is clear and obvious, we believe in their limitation in particular cases (i.e. to resolve hardware faults or to add new functionalities like updates of algorithms in blocks) without studying all possible reasons to apply reconfigurations like improvements of the system's performance. They do not consider also particular reconfiguration techniques that we can probably apply at run-time like additions of data/event inputs/outputs in control systems.

6.4 New Reconfiguration Semantic

We are interested in this manuscript in dynamic reconfigurations of embedded systems (manual or automatic) that we define as follows.

Definition. *A dynamic reconfiguration is any change according to well-defined conditions in software as well as hardware components to lead the whole embedded system at run-time to better and safe behaviors.*

We mean in this definition by a change in software components any operation allowing the addition, removal or also update of components to improve the whole system's behavior. We mean also by a change in hardware components any operation allowing the addition, removal or also update of devices to be used in the execution environment. This new definition remains compatible with previous works on reconfigurations of systems. Indeed, as defined in [2], the reconfiguration is applied to save the system when hardware problems occur at run-time. In this case, we have to apply changes in software as well as hardware components to bring the whole architecture to optimal and safe behaviors. In addition, as defined in [101], the reconfiguration is manually applied to add new functionalities in the system. Therefore, it corresponds also to changes in software and hardware components in order to bring the whole architecture to optimal behaviors. Finally, a dynamic reconfiguration will be in our research work any automatic action that saves the system, enriches its behaviors or also improves its performance at run-time. To our knowledge, this definition covers all possible reconfiguration cases in industry.

6.5 Industrial Case Studies: Reconfiguration of Benchmark Production Systems FESTO and EnAS

We apply this new semantic of reconfiguration to two Benchmark Production Systems [1] following the Standard IEC61499: FESTO and EnAS available in the research laboratory of Prof.Dr. Hans-Michael Hanisch at Martin Luther University in Germany. For the sale of our contributions, we imagine new functionalities in these systems.

6.5.1 FESTO Manufacturing System

The FESTO Benchmark Production System is a well-documented demonstrator used by many universities for research and education purposes, and is used as a running example in the context of this chapter (Figure 6.1). FESTO is composed of three units: the Distribution, the Test and the Processing units. The Distribution unit is composed of a pneumatic feeder and a converter. It forwards cylindrical work pieces from a stack to the testing unit which is composed of the detector, the tester and the elevator. This unit performs checks on work pieces for height, material type and color. Work pieces that successfully pass this check are forwarded to the rotating disk of the processing unit, where the drilling of the work piece is performed. We assume in this research work two drilling machines $Drill\_machine1$ and $Drill\_machine2$ to drill pieces. The result of the drilling operation is next checked by the checking machine and the work piece is forwarded to another mechanical unit. We present in Figure 6.2 the sequence of functional operations in the system such that each operation needs required data from sensors to activate corresponding actuators.

For the sale of our contributions, we assume three production modes of FESTO according to the rate of input pieces denoted by *number_pieces* into the system (i.e. ejected by the feeder).

- **Case1: High production. If** $number\_pieces \geq Constant1$, **Then** the two drilling machines are used at the same time to accelerate the production. In this case, the Distribution and Testing units should forward two successive pieces to the rotating disc before starting the drilling with $Drill\_machine1$ **AND** $Drill\_machine2$. For this production mode, the periodicity of input pieces is $p = 11seconds$.
- **Case2: Medium production. If** $Constant2 \leq number\_pieces < Constant1$, **Then** we use $Drill\_machine1$ **OR** $Drill\_machine2$ to drill work pieces. For this production mode, the periodicity of input pieces is $p = 30seconds$.
- **Case3: Light production. If** $number\_pieces < Constant2$, **Then** only the drilling machine $Drill\_machine1$ is used. For this production mode, the periodicity of input pieces is $p = 50seconds$.

[1] Detailed descriptions are available in our website: http://aut.informatik.uni-halle.de.

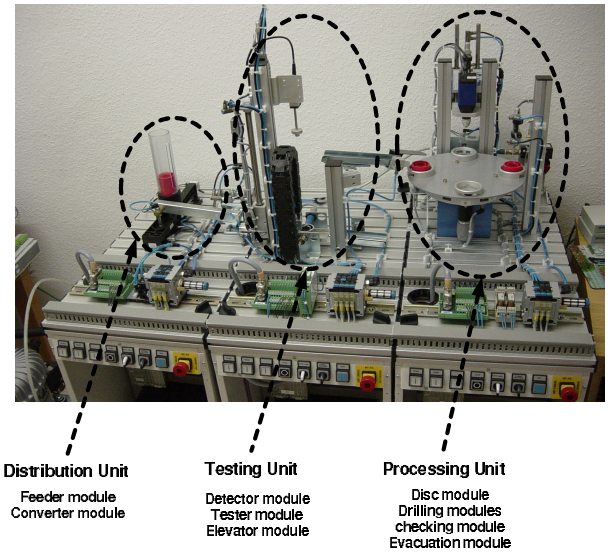

Figure 6.1: The FESTO modular production system

On the other hand, if one of the drilling machines is broken at run-time, then we have to only use the other one. In this case, we reduce the periodicity of input pieces to $p = 40seconds$. The system is completely stopped in the worst case if the two drilling machines are broken. According to the rate of input pieces, the dynamic reconfiguration is useful to:

- protect the whole system when hardware faults occur at run-time. Indeed, If $Drill\_machine1$ (resp, $Drill\_machine2$) is broken, then the drilling operation will be supported by $Drill\_machine2$ (resp, $Drill\_machine1$),
- improve the system productivity. Indeed, if the rate of input pieces is increased, then we improve the production from the Light to the Medium or from the Medium to the High mode.

This first example shows the new reconfiguration semantic in industry: we can change the system configuration to improve the performance even if there are no faults.

6.5.2 EnAS Manufacturing System

The Benchmark Production System EnAS was designed as a prototype to demonstrate energy-antarcic actuator/sensor systems. For the sale of this contribution, we assume that it has the following behavior: it transports pieces from the production system (i.e. FESTO system) into storing units (Figure 6.3). The pieces in EnAS shall be placed inside tins to

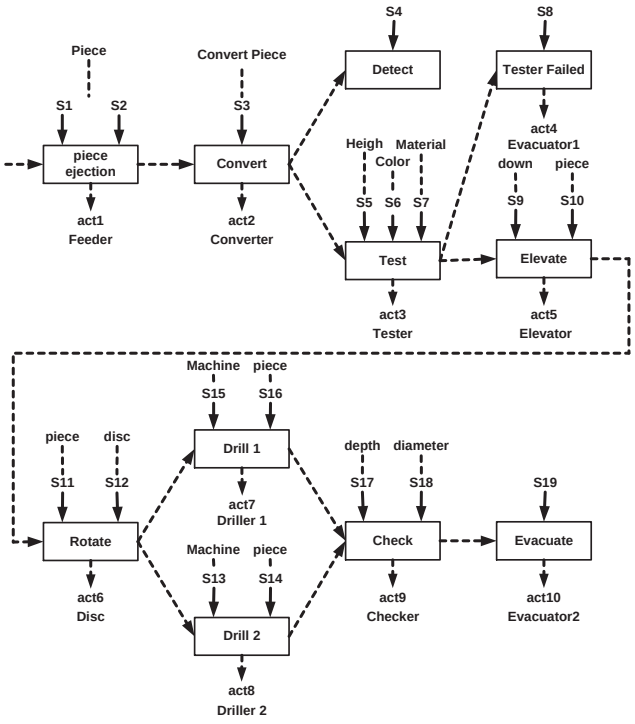

Figure 6.2: Functional operations of the FESTO system

close with caps afterwards. Two different production strategies can be applied : we place in each tin one or two pieces according to production rates of pieces, tins and caps. We denote respectively by nb_{pieces}, $nb_{tins+caps}$ the production number of pieces and tins (as well as caps) per hour and by $Threshold$ a variable (defined in user requirements) to choose the adequate production strategy.

The EnAS system is mainly composed of a belt, two Jack stations (J_1 and J_2) and two Gripper stations (G_1 and G_2) (Figure 8.4). The Jack stations place new produced pieces and close tins with caps, whereas the Gripper stations remove charged tins from the belt into the storing units. We present in Figure 6.9 the sequence of functional operations of EnAS such that each operation needs required data from sensors to activate corresponding actuators.

Initially, the belt moves a particular pallet containing a tin and a cap into the first Jack station J_1. According to production parameters, we distinguish two cases,

- **First production policy: If** ($nb_{pieces}/nb_{tins+caps} \leq Threshold$)**, Then** the Jack station J_1 places from the production station a new piece and closes the tin with the cap. In this case, the Gripper station G_1 removes the tin from the belt into the storing station St_1 (Figure 6.6).

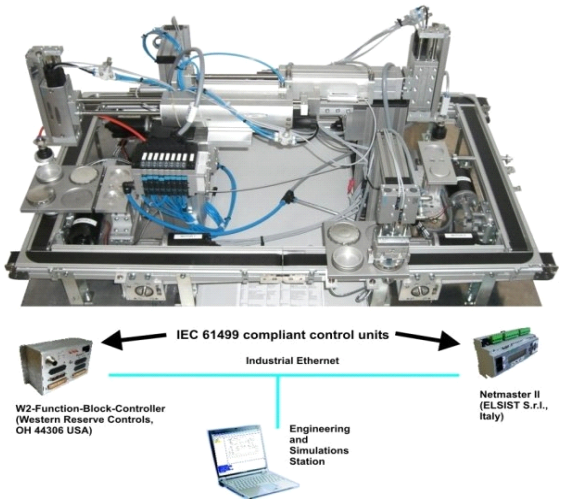

Figure 6.3: EnAS-Demonstrator in Halle

- **Second production policy: If** ($nb_{pieces}/nb_{tins+caps} > Threshold$)**, Then** the Jack station J_1 places just a piece in the tin which is moved thereafter into the second Jack station to place a second new piece. Once J_2 closes the tin with a cap, the belt moves the pallet into the Gripper station G_2 to remove the tin (with two pieces) into the second storing station St_2 (Figure 6.7).

For the sale of our contributions, we assume that each piece (resp, tin and cap) costs 0.4€ (resp, 0.6€). In addition, let us assume that $Threshold = 1.5$. According to production parameters, we have to apply the best production policy as follows,

- **If** $nb_{pieces}/nb_{tins+caps} = 180/100 > Threshold$, **Then**,
  - **If** we apply the first policy, **Then** we will charge 100 tins per hour that cost $100€/h$,
  - **Else If** we apply the second policy, **Then** we will only charge 90 tins per hour that cost $126€/h$ and the gain is 26%.
- **If** $nb_{pieces}/nb_{tins+caps} = 100/100 < Threshold$, **Then**,
  - **If** we apply the first policy, **Then** we will charge 100 tins per hour that cost $100€/h$,
  - **Else If** we apply the second policy, **Then** we will have 50 tins per hour that cost $70€/h$ and the loss is -30%.

According to production parameters, the dynamic reconfiguration of the transportation system is useful to:

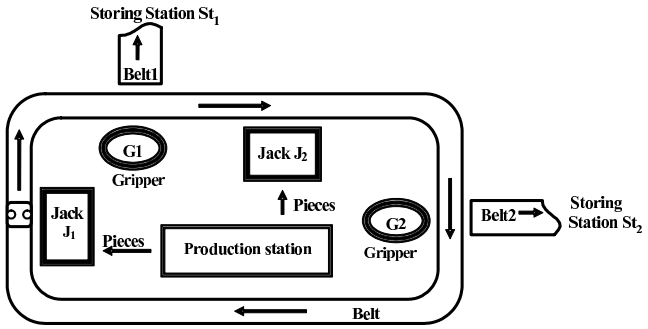

Figure 6.4: Distribution of the EnAS stations

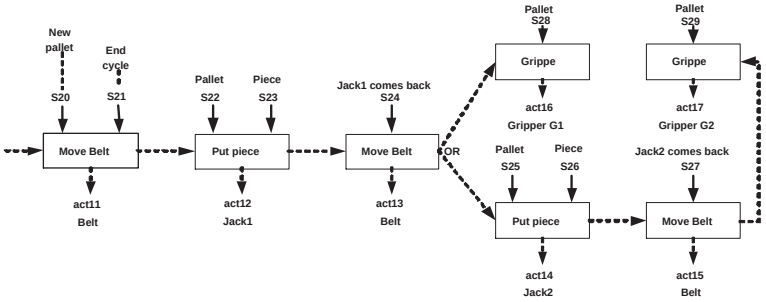

Figure 6.5: Functional operations of the system EnAS

- protect the system when hardware problems occur at run-time. For example, if the Gripper $G2$ is broken, then we have to follow the first production policy by placing only one piece in each tin.
- improve the production gain when ($nb_{pieces}/nb_{tins+caps} > Threshold$). In this case, we have to apply the second policy and have therefore to apply changes in the system's architecture and blocks to follow this policy.

In these Benchmark Production Systems FESTO and EnAS, the reconfiguration is not only applied to resolve hardware problems as proposed in [2, 22] but also to improve the system's performance by increasing the production gain or the number of produced pieces. This new semantic of reconfiguration will be a future issue in industry. We note in addition that 11 physical processes of $Plant$ are distinguished: $Feeder$, $Converter$, $Detector$, $Tester$, $Evacuator1$, $Elevator$, $Disc$, $Driller1$, $Driller2$, $Checker$, $Evacuator2$. The actuators $act6$ and $act7$ are characterized in particular as follows: $prev(act6) = \{act5\}$; $follow(act6) = \{\{act7\}, \{act8\}\}$; $sensor(act7) = \{S15, S16\}$. Before the activation of $act7$, Sys should know if a piece is available in $Drill\_machine1$ (i.e. information to be

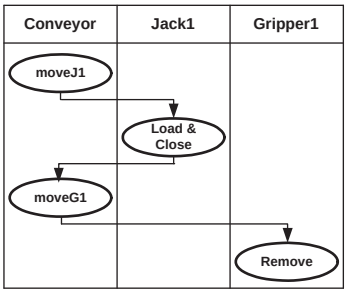

Figure 6.6: Policy1: production of a tin with only one piece.

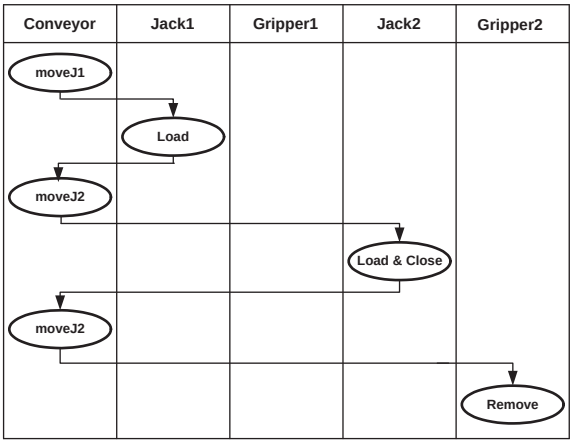

Figure 6.7: Policy2: production of tins with double pieces.

provided by the sensor $S16$) and also if this machine is ready (i.e. information provided by $S15$). Note that $act1$ (resp. $act10$) is the only actuator without predecessors (resp. successors) in FESTO: $act1 \in first(\alpha_{actuators})$ and $act10 \in last(\alpha_{actuators})$. The EnAS Benchmark Production System is composed of 7 physical processes. We characterize the actuator $act13$ corresponding to the Belt as follows: (i) $prev(act13) = \{act12\}$; (ii) $follow(act13) = \{\{act14\}, \{act16\}\}$, (iii) $sensor(act13) = \{S24\}$. Note that $act11$ (resp. $act16$ and $act17$) is (resp. are) the only actuator(s) without predecessors (resp. successors) in EnAS: (i) $act11 \in first(\alpha_{actuators})$, (ii) $act16 \in last(\alpha_{actuators})$, (iii) $act17 \in last(\alpha_{actuators})$.

In the Benchmark Production Systems FESTO and EnAS, the Control Components are Function Blocks according to the Standard IEC61499. We present respectively their assumed Function Blocks-based designs in Figure 6.8 and Figure 6.9. We present in particular the times to distribute, test, drill and finally check work pieces (In particular, the

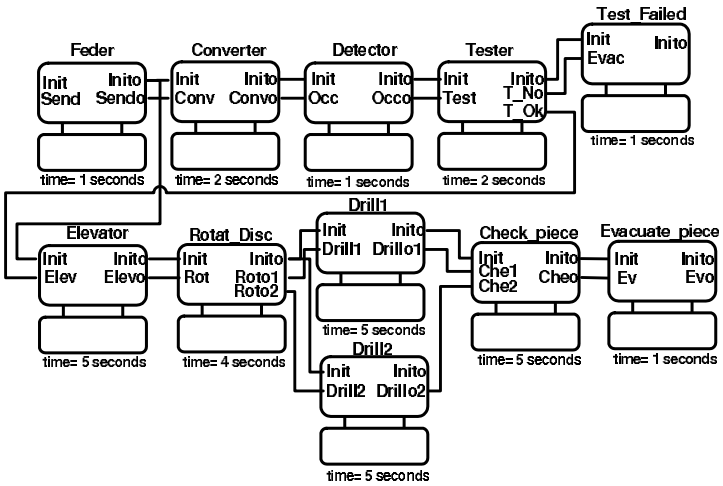

Figure 6.8: An IEC61499-based Design of FESTO

drilling operation takes 5 seconds before forwarding the piece into the checking machine). We show in Figure 6.10 a NCES-based model of the Control Component CC_Jack1 that controls in EnAS the Jack station J_1. This component should read required data from the sensors $S22$ (i.e. the pallet is in front of J_1) and $S23$ (i.e. a new piece is ready to be put in the tin) before a possible activation of the actuator $act12$.

6.6 Reconfiguration Forms

We propose in this section a classification of all possible reconfiguration forms of embedded control systems:

- **First Form.** It deals with the change of the application architecture that we consider as a composition of components. In this case, we have possibly to add, remove or also change the localization of components (from one to another device). This reconfiguration form requires to load new (or to unload old) blocks in (from) the memory

  Running example1. *In the FESTO manufacturing system, we distinguish two architectures:*

  - ***First Architecture (Light production).*** *We implement the system with the first architecture* ***If*** *we apply the Light production mode (i.e. number_pieces* $<$ *Constant2). In this case, the Function Block Drill2 is not loaded in the memory,*
  - ***Second Architecture.*** *We implement the system with the second architecture* ***If*** *we apply the High or also the Medium mode (i.e. number_pieces* $\geq$ *Constant2). In this case, we load in the memory the Function Blocks Drill1 and Drill2.*

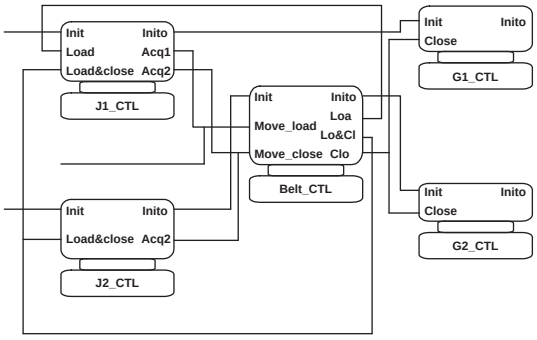

Figure 6.9: An IEC61499-based Design of EnAS.

Running example2. *In the EnAS manufacturing system, we distinguish two architectures:*

- *We implement the system with the first architecture* ***If*** *we follow the first production policy. In this case, we load in the memory the Function Blocks* $J1\_CTL$*,* $Belt\_CTL$ *and* $G1\_CTL$*,*
- *We implement the system with the second architecture* ***If*** *we follow the second production policy. In this case, we load in the memory the Function Blocks* $J1\_CTL$*,* $J2\_CTL$*,* $Belt\_CTL$ *and* $G2\_CTL$*,*

If *we follow the first production policy and* $nb_{pieces}/nb_{tins+caps}$ *becomes higher than* $Threshold$*,* ***Then*** *we should load the function block* $G2\_CTL$ *in the memory to follow the second production policy.*

- **Second form.** It deals with the reconfiguration of the application without changing its architecture (i.e. without loading or unloading components). In this case, we apply changes of the internal structure of components or of their composition.

  Running example1. *In the FESTO system, we distinguish for the second architecture the following cases:*

  - **High production. If** $number\_pieces \geq Constant1$, **Then** we should apply an automatic modification of the ECC of $Rotat\_Disc$ in order to use the two drilling machines $Drill\_machine1$ and $Drill\_machine2$,
  - **Medium production. If** $Constant2 \leq number\_pieces < Constant1$, **Then** we should apply a new modification of such ECC in order to use one of these machines.

  Running example2. *In the EnAS system, if we follow the second policy and the Jack station* $J2$ *is broken, then we should change the internal behavior (i.e. the ECC*

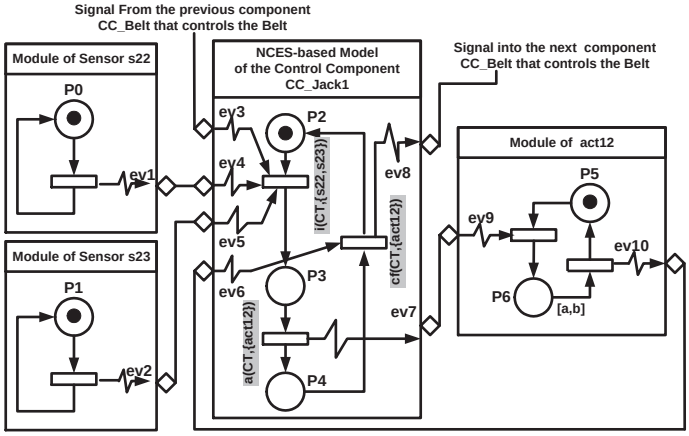

Figure 6.10: Interactions between the Control Component CC_Jack1 and the plant sensors $S22$ and $S23$ before the activation of $act12$

structure) of the block J1_CTL to close the tin with a cap once it contains only one piece. The tin will be moved directly to the Gripper G2. We do not change in this example the application architecture (i.e. loading or unloading blocks) but we just change the behavior of particular blocks.

- **Third form.** It simply deals with easy reconfigurations of application data (i.e. internal data of components or global data of the system).

  Running example1. *In the FESTO system,* ***If*** *we apply the Medium production mode (i.e. the second architecture),* ***Then*** *the production periodicity is 30 seconds, whereas* ***If*** *we apply in the same architecture the High mode* ***Then*** *the periodicity is 11 seconds.*

  Running example2. *In the EnAS system,* ***If*** *a hardware problem occurs at run-time,* ***Then*** *we have to change the value of Threshold to a number max_value. In this case, we will not be interested in the performance improvement but in the rescue of the whole system to guarantee a minimal level of productivity.*

6.7 Conclusion

We define in this chapter the concept of Control Components in order to be independent of any Architecture Description Language or industrial technology. Each component is a unit of software allowing controls of physical processes by reading required data from sensors before possible reactions to activate corresponding actuators. In the functional level, the system's components are assumed to be distributed on containers that we define as OS tasks

in the operational level. We propose in addition a new definition of reconfiguration scenarios to address new requirements in Industry: a dynamic reconfiguration is a technical run-time solution for systems to be optimal and safe. This definition is applied to two Benchmark Production Systems FESTO and EnAS available in the research laboratory of Prof. Hans-Michael Hanisch. Finally, we classify all reconfiguration scenarios into three forms. The first deals with the modification of software architectures, the second with compositions of components, and the last with easy updates of data. This classification covers all possible reconfiguration forms to dynamically bring systems to safe and optimal behaviors while satisfying user requirements and environment's evolutions.

Chapter 7

Safety Multi-Agent Reconfigurable Architectures: Modelling And Verification

7.1 Introduction

We propose in this chapter a distributed multi-agent architecture for automatic reconfigurations of embedded control systems. A Reconfiguration Agent *"RA"* is affected in this architecture to each device of the execution environment to handle local automatic reconfigurations, and a unique Coordination Agent *"CA"* is defined to manage distributed reconfigurations between devices because any uncontrolled automatic reconfiguration applied in a device can lead to critical problems or serious disturbances in others. We define the concept of "Coordination Matrix" to define for each distributed reconfiguration scenario the behavior of all concerned agents that should simultaneously react. According to the urgency of reconfiguration scenarios, we define priority levels for concurrent matrices allowing different reconfigurations of same devices. The Coordination Agent handles all matrices to coordinate between agents according to a well-defined communication protocol: when an agent applies in the corresponding device a reconfiguration scenario, the Coordination Agent should inform other concerned agents to react and to bring the whole distributed architecture into safe and optimal behaviors [59]. To check the whole distributed architecture, we model each reconfiguration agent by nested state machines (i.e. states can be other state machines) according to the formalism Net Condition-Event Systems (NCES), and use the well-expressive temporal logic "Computation Tree Logic" (denoted by CTL) as well as its extensions (eCTL and TCTL) to specify functional and temporal properties of agents and system's components that we verify by the model checker $SESA$. At this step, the internal behavior in each device is checked, but the coordination between agents should be verified in order to avoid any critical problems at run-time. We propose for each Coordination Matrix a NCES-based model and apply the model checker SESA to check if

all the system's agents react as described in user requirements to guarantee safe distributed reconfigurations [55].

The next step to be addressed is the specification and verification of different networks of Control Components that correspond to different reconfiguration scenarios to be automatically applied by each Reconfiguration Agent [62]. We want in this case to apply a model checking for automatic and manual verifications of functional properties of such networks [47]. This verification is difficult to do in complex cases like the assumed reconfigurable Benchmark Systems FESTO and EnAS. We propose therefore to apply a refinement-based technique that automatically models and checks in several steps each network of components. An abstract model of the network is defined in the first step according to the formalism NCES. It is automatically refined step by step thereafter to automatically generate in each step NCES-based models of Control Components to be automatically checked by the model checker SESA for the verification of deadlock properties. In addition, we manually verify functional properties described according to the temporal logic "Computation Tree Logic" (abbr. CTL) in order to check the correct behavior of the new generated components in each step [100]. If the refinement-based specification and verification of Control Components is feasible in different steps, Then the correctness of their network is deduced. The safety of the whole reconfigurable system is confirmed if all its networks of Control Components are correct.

Finally, If Control Components are assumed to be Function Blocks according to the Industrial Standard IEC61499, we propose in this chapter XML-based implementations for both coordination and reconfiguration agents that we tested to the benchmark production systems FESTO and EnAS [61].

7.2 State of the Art

We present well-known model checkers for verifications of functional and temporal properties, before describe thereafter some research studies on multi-agent systems.

7.2.1 Model Checking

Finite state machines (abbr. FSM) are widely used for the modelling of control flow in embedded systems and are amenable to formal analysis like model checking [28, 29, 30, 44, 120, 80]. Two kinds of computational tools have been developed in recent years for model checking: tools like KRONOS [33], UPPAAL [5], HyTech [43] and SESA [103] which compute sets of reachable states exactly and effectively, whereas emerging tools like CHECKMATE [27], d/dt [9] and level-sets [86] methods approximate sets of reachable states. Several research works have been proposed in recent years to control the verification complexity by applying hierarchical model checking for complex embedded systems. The authors propose in [3] an approach for verifications of hierarchical (i.e. nested) finite state machines whose states themselves can be other machines. The straightforward way to analyze a hierarchical

machine is to flatten and apply a model checking tool on the resulting ordinary FSM, but the authors show in this interesting research work that this flattening can be avoided by developing useful algorithms for verifications of hierarchical machines. We use SESA in our research work to validate multi-agent reconfigurable embedded systems, where each agent is specified by nested Net Condition-Event Systems.

7.2.2 Multi-Agent Systems

Several research works have been done in recent years in academia and also in industry to define inter-agents communication protocols for multi-agent systems. The authors focus in [91] on the communicative act between agents and define a general semantic framework for specifying a class of Agent Communication Language (ACLs) based on protocols. They introduce a small ACL denoted by sACL for different application domains and describe a development method to define an ACL for a particular application. The authors are interested in [81] in Distributed Constraint Satisfaction Problems (DisCSP) as an important area of research for multi-agent systems. The agents work together to solve problems that cannot be completely centralized due to security, dynamics, or complexity. The authors present an algorithm called asynchronous partial overlay (APO) for solving DisCSPs that is based on a mediated negotiation process. The same authors present in [83, 82] a cooperative negotiation protocol that solves a distributed resource allocation problem while conforming to soft real-time constraints in a dynamic environment. A fully automated and knowledge-based organization designer for multi-agent systems called KB-ORG is proposed in the same research activities [106]. Organization design is the process that accepts organizational goals, environmental expectations, performance requirements, role characterizations, as well as agent descriptions, and assigns roles to each agent. In [31], an agent-based power-aware sensor network called CNAS (Collaborative Network for Atmospheric Sensing) is proposed for ground-level atmospheric monitoring. The CNAS agents must have their radios turned off most of the time, as even listening consumes significant power. CNAS requires agent policies that can intelligently meet operational requirements while communicating only during intermittent, mutually established, communication windows. The authors describe in [87] the architecture and implementation of the security (X-Security) system, which implements authentification and secure communication among agents. The system uses certification authority (CA) and ensures full cooperation of secured agents and already existing (unsecured) ones. The authors propose augmenting in [20] the capabilities of current multi-agent systems to provide for the efficient transfer of low-level information, by allowing backchannels of communications between agents with flexible protocols in a carefully principled way.

In our research work, we are interested in the communication and collaboration between agents to guarantee safe and adequate distributed reconfigurations of embedded control systems. These communications are handled by a Coordination Agent that coordinates between agents according to environment's evolutions, user requirements and also priorities of reconfigurations.

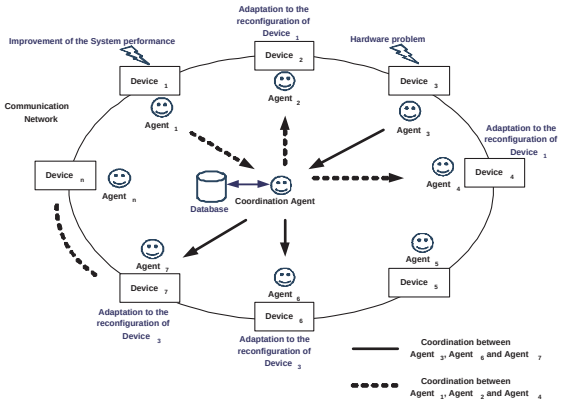

Figure 7.1: Multi-agent architecture of distributed reconfigurable embedded systems.

7.3 Contribution: Multi-Agent Architecture for Reconfigurable Embedded Control Systems

We define in this section a multi-agent architecture for distributed reconfigurable systems where a reconfiguration agent is assigned in this architecture to a device of the execution environment to handle automatic reconfigurations of Control Components. It is specified by nested state machines that support all reconfiguration forms. Nevertheless, the coordination between agents in this distributed architecture is extremely important because any uncontrolled automatic reconfiguration applied in a device can lead to critical problems, serious disturbances or also inadequate distributed behaviors in others. To guarantee safe distributed reconfigurations, we define the concept of *Coordination Matrix* that defines correct reconfiguration scenarios to be simultaneously applied in distributed devices, and define the concept of *Coordination Agent* that handles coordination matrices to coordinate between distributed agents. We propose in this section an inter-agents communication protocol to manage concurrent distributed reconfiguration scenarios in same devices (Figure 7.1).

Running Example. *In the Production Systems FESTO and EnAS where a reconfiguration agent is defined for each one of them, the reconfiguration of the first can lead to a reconfiguration of the second in order to guarantee a coherent production in the two platforms. This means:*

- ***If*** $Constant2 \leq number\_pieces$, ***Then*** *the FESTO Agent has to apply the Medium or the High Production Mode, and in this case the EnAS Agent has to improve the productivity by applying the Second Production Policy in order to put two pieces in each tin.*

- ***If*** *Constant*2 > *number_pieces,* ***Then*** *the FESTO Agent has to decrease the productivity by applying the Light mode (i.e. only Drill_machine*1 *is used), and in this case, the EnAS Agent has also to decrease to productivity by applying the First Production Policy in order to put only one piece in the tin according to user requirements.*

On the other hand, when a hardware problem occurs at run-time in a platform, a reconfiguration of the second is required as follows:

- ***If*** *one of the Jack stations J*1 *and J*2 *or the Gripper station G*2 *is broken in the EnAS Production System,* ***Then*** *the corresponding Agent has to decrease the productivity by applying the First Production mode, and in this case the FESTO Agent has also to follow the Light Production mode in order to guarantee a coherent behavior.*
- ***If*** *one of the drilling machines Drill_machine*1 *and Drill_machine*2 *is broken,* ***Then*** *the FESTO Agent has to decrease the productivity, and in this case the EnAS Agent has to follow the First Production Mode where only one piece is put in a tin.*

7.3.1 Reconfiguration in a Device

We define for each device of the execution environment a unique agent that checks the environment evolution and takes into account user requirements to apply automatic reconfiguration scenarios.

Architecture of the Reconfiguration Agent

We define the following units that belong to three hierarchical levels of the agent architecture:

- **First level:** (**Architecture Unit**) this unit checks the system behavior and changes its architecture (adds/removes Control Components) when particular conditions are satisfied.
- **Second level:** (**Control Unit**) for a particular *loaded* architecture, this unit checks the system behavior and: reconfigures compositions of components (i.e. changes the configuration of connections), or adds/removes event inputs/outputs, or reconfigures the internal behavior of components,
- **Third level:** (**Data Unit**) this unit updates data if particular conditions are satisfied.

We design the agent by nested state machines where the Architecture Unit is specified by an Architecture State Machine (denoted by ASM) in which each state corresponds to a particular architecture of the system. Therefore, each transition of the ASM corresponds to the load (or unload) of Control Components into (or from) the memory. We construct for each state S of the ASM a particular Control State Machine (denoted by CSM) in

the Control Unit. This state machine specifies all reconfiguration forms to possibly apply when the system's architecture corresponding to the state S is loaded (i.e. modification of compositions of components or of their internal behavior). Each transition of any CSM has to be fired if particular conditions are satisfied. Finally, the Data unit is specified also by Data State Machines (denoted by DSMs) where each one corresponds to a state of a CSM or the whole ASM.

Notation. we denote in the following by,

- n_{ASM} the number of states in the state machine ASM (i.e. the number of all possible software architectures to implement the system). ASM_i ($i \in [1, n_{ASM}]$) denotes a state of ASM to encode a particular architecture (i.e. particular network of components). This state corresponds to a particular state machine CSM that we denote by CSM_i ($i \in [1, n_{ASM}]$),
- n_{CSM_i} the number of states in CSM_i and let $CSM_{i,j}$ ($j \in [1, n_{CSM_i}]$) be a state of CSM_i,
- n_{DSM} the number of Data State Machines corresponding to all possible reconfiguration scenarios of the system. Each state $CSM_{i,j}$ ($j \in [1, n_{CSM_i}]$) is associated to a particular DSM state machine DSM_k ($k \in [1, n_{DSM}]$).
- n_{DSM_k} the number of states in DSM_k. $DSM_{k,h}$ ($h \in [1, n_{DSM_k}]$) denotes a state of the state machine DSM_k which can correspond to one of the following cases: (i) one or more states of a CSM state machine, (ii) more than one CSM state machine, (iii) all the ASM state machines.

The agent automatically applies at run-time different reconfiguration scenarios such that each one denoted by $Reconfiguration_{i,j,k,h}$ corresponds to a particular network of Control Components $Network_{i,j,k,h}$ as follows: (i) the architecture ASM_i is loaded in the memory, (ii) the control policy is fixed in the state $CSM_{i,j}$, (iii) the data configuration corresponding to the state $DSM_{k,h}$ is applied.

Running example. *We present in Figure 7.2 the nested state machines of the FESTO Agent. The ASM state machine is composed of two states $ASM1$ and $ASM2$ corresponding to the first (i.e. the Light Production Mode) and the second (the High and Medium modes) architectures. The state machines $CSM1$ and $CSM2$ correspond to the states $ASM1$ and $ASM2$. In $CSM2$ state machine, the states $CSM21$ and $CSM22$ correspond respectively to the High and the Medium Production Modes (where the second architecture is loaded). To fire a transition from $CSM21$ to $CSM22$, the value of number_pieces should be in $[Constant2, Constant1[$. We note that the states $CSM12$ and $CSM25$ correspond to the blocking problem where the two drilling machines are broken. Finally the state machines $DSM1$ and $DSM2$ correspond to the state machines $CSM1$ and $CSM2$. In particular, the state $DSM21$ encodes the production periodicity when we apply the High Production Mode (i.e. the state $CSM21$ of $CSM2$), and the state $DSM22$ encodes the production periodicity when we apply the Medium mode (i.e. $CSM22$ of $CSM2$). Finally, the state $DSM23$*

corresponds to $CSM23$ and $CSM24$ and encodes the production periodicity when one of the drilling machines is broken. We design the agent of the EnAS Benchmark Production System by nested state machines as depicted in Figure 7.3. The first level is specified by ASM where each state defines a particular architecture of the system. The state ASM_1 (resp. ASM_2) corresponds to the second (resp. first) policy where Control Components that control J_1, J_2 and G_2 (resp. only J_1 and G_1) are loaded in memory. We associate for each one of these states a CSM in the Control Unit. Finally, Data Unit is specified by DSM which defines the values that $Threshold$ takes under well-defined conditions. Note that if we follow the Second Production Policy (state ASM_1) and the gripper G_2 is broken, then we should change the policy and also the system architecture by loading the Control Component $G1\_CTL$ to remove pieces into $Belt1$. On the other hand, we associate in the second level for the state ASM_1 the CSM CSM_1 that defines the different reconfiguration forms to apply when the first architecture is loaded in the memory. In particular, If the state CSM_{11} is active and the Jack station J_1 is broken, Then we activate the state CSM_{12} in which the Jack station J_2 is running alone to place only one piece in the tin. In this case, the internal behavior of the block $Belt\_CTL$ should be changed (i.e. the tin should be transported directly to the station J_2). In the same way, If we follow the same policy in the state CSM_{11} and the Jack station J_2 is broken, Then we should activate the state CSM_{13} where the behavior of J_1 should be changed to place a piece in the tin that should be closed too (i.e. the behavior of the Control Component $J1\_CTL$ should be reconfigured). We finally specify in Data Unit a DSM where we change the value of $Threshold$ when Gripper G_1 is broken (we suppose as an example that we are not interested in the system performance when this Gripper is broken). By considering this hierarchical model of agents, we specify all possible reconfiguration scenarios that can be applied in embedded control systems: Add-Remove (first level) or Update the structure of Control Components (second level) or just Update data (third level).

System Behavior

The different reconfiguration scenarios applied by the agent define all possible behaviors of the system when well-fixed conditions are satisfied. We specify these behaviors by a unique System State Machine (denoted by SSM) in which each state corresponds to a particular Control Component.

Running example1. *We specify in Figure 7.16 the different behaviors of FESTO that we can follow to resolve hardware problems or to improve the system performance. The branch $Branch1$ specifies the system behavior when $Drill\_machine1$ or $Drill\_machine2$ is broken or also when the Medium Production Mode is applied, $Branch2$ defines the system behavior when the High Production Mode is applied, and $Branch3$ defines the behavior when the Light Production Mode is applied.*

Running example2. *We specify in Figure 7.5 the different behaviors of EnAS that we can follow to resolve hardware problems or to improve the system performance. In this example, we distinguish four traces encoding four types of different behaviors. The*

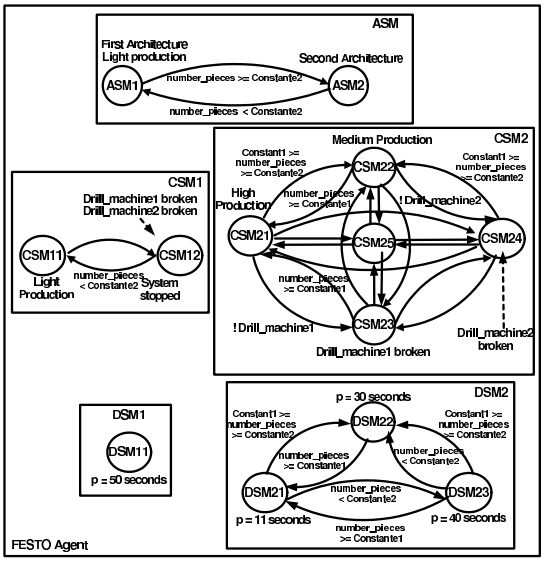

Figure 7.2: Specification of the FESTO Agent by nested state machines

trace trace1 *implements the system behavior when the Jack station* J_1 *is broken. The trace* trace2 *implements the system behavior to apply the second production policy. The trace* trace3 *implements the system behavior when the Jack station* J_2 *is broken. Finally the last scenario implements the system behavior when the Gripper* G_2 *is broken or when we have to apply the first production policy. Note finally that each state corresponds to a particular behavior of a system's component when the corresponding input event occurs.*

7.3.2 Reconfiguration in a Distributed Architecture

We are interested in automatic reconfigurations of Control Components to be distributed on networks of devices where the coordination between agents is important. We define in this section the concept of *Coordination Matrix* to define coherent reconfiguration scenarios in distributed devices. We propose in addition an architecture of multi-agent distributed reconfigurable systems where an inter-agents communication protocol is defined to guarantee safe behaviors after any distributed automatic reconfiguration.

Distributed Reconfigurations

Let Sys be a distributed reconfigurable system of n devices, and let Ag_1,..., Ag_n be n agents to handle automatic distributed reconfiguration scenarios of these devices. We denote in the following by $Reconfiguration^a_{i_a,j_a,k_a,h_a}$ a reconfiguration scenario applied by Ag_a ($a \in [1,n]$) as follows: (i) the corresponding state machine ASM is in the state ASM_{i_a}. Let

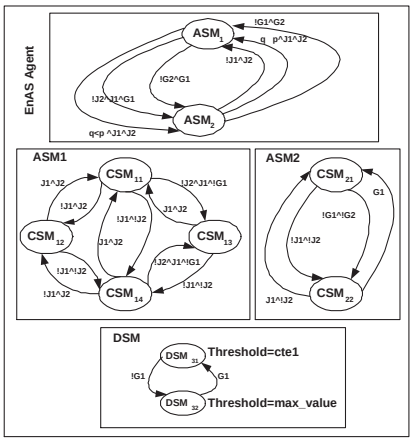

Figure 7.3: Specification of the EnAS Agent by nested state machines

$cond^a_{i_a}$ be the set of conditions to reach this state, (ii) the state machine CSM is in the state CSM_{i_a,j_a}. Let $cond^a_{j_a}$ be the set of conditions to reach this state, (iii) the state machine DSM is in the state DSM_{k_a,h_a}. Let $cond^a_{k_a,h_a}$ be the set of conditions to reach this state. To handle coherent distributed reconfigurations that guarantee safe behaviors of the whole system Sys, we define the concept of *Coordination Matrix* of size (n,4) that defines coherent scenarios to be simultaneously applied by different agents. Let CM be a such matrix that we characterize as follows: each line a ($a \in [1,n]$) corresponds to a reconfiguration scenario $Reconfiguration^a_{i_a,j_a,k_a,h_a}$ to be applied by Ag_a as follows:

$$CM[a,1] = i_a;\ CM[a,2] = j_a;\ CM[a,3] = k_a;\ CM[a,4] = h_a$$

According to this definition: If an agent Ag_a applies the reconfiguration scenario $Reconfiguration^a_{CM[a,1],CM[a,2],CM[a,3],CM[a,4]}$, Then each other agent Ag_b ($b \in [1,n]\backslash\{a\}$) has to apply the scenario $Reconfiguration^b_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]}$ (Figure 7.6). We denote in the following by *idle agent* each agent Ag_b ($b \in [1,n]$) which is not required to apply any reconfiguration when others perform scenarios defined in CM. In this case:

$$CM[b,1] = CM[b,2] = CM[b,3] = CM[b,4] = 0$$

$$cond^b_{CM[b,1]} = cond^b_{CM[b,2]} = cond^b_{CM[b,3],CM[b,4]} = True$$

We denote in addition by $\xi(Sys)$ the set of coordination matrices to be considered for the reconfiguration of the distributed embedded system Sys. Each Coordination Matrix CM is applied at run-time if for each agent Ag_a ($a \in [1,n]$) the following conditions are satisfied:

$$cond^a_{CM[a,1]} = cond^a_{CM[a,2]} = cond^a_{CM[a,3],CM[a,4]} = True$$

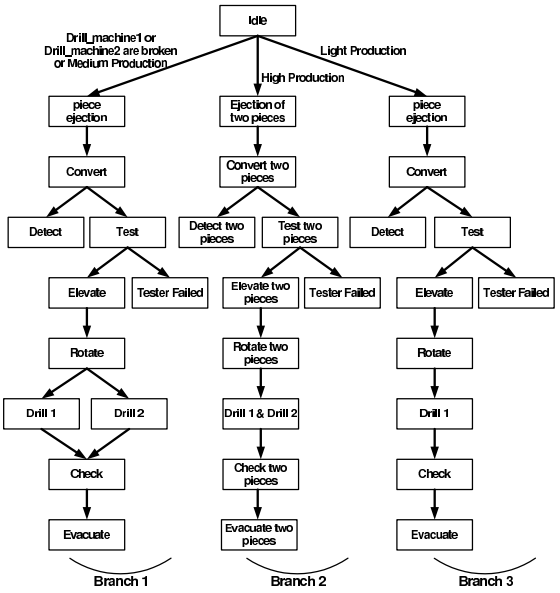

Figure 7.4: The system's state machine of the FESTO Benchmark Production System: SSM(FESTO).

On the other hand, we define *Concurrent Coordination Matrices*, CM_1 and CM_2 two matrices of $\xi(Sys)$ that allow different reconfigurations of a same agent Ag_b ($\forall b \in [1, n]$) as follows:

- $CM_j[b, i] \neq 0 \ \forall \ j \in \{1, 2\}$ and $i \in [1, 4]$. In this case, Ag_b should react when CM_1 or CM_2 is desired by the Coordination Agent,
- $CM_1[b, i] \neq CM_2[b, i] \ \forall i \in [1, 4]$. In this case, Ag_b should apply two different reconfiguration scenarios when CM_1 and CM_2 are simultaneously desired by the Coordination Agent.

In this case, the system behavior is not deterministic because the agent Ag_b should follow two different reconfiguration scenarios at the same time. To guarantee a deterministic behavior when Concurrent Coordination Matrices are required to be simultaneously applied, we define priority levels for them such that only the matrix with the highest priority level should be applied. We denote in the following by:

- $Concur(CM)$ the set of concurrent matrices of $CM \in \xi(Sys)$,
- $level(CM)$ the priority level of the matrix CM in the set $Concur(CM) \cup \{CM\}$.

In this case, $Concur(CM1) = \{CM2\}$ and If $CM1$ has the highest priority, Then the agent $Agent_b$ should apply $Reconfiguration^b{}_{CM1[b,1],CM[b,2],CM[b,3],CM[b,4]}$. The application of the second matrix CM_2 is rejected.

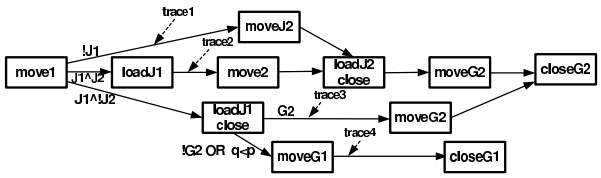

Figure 7.5: The system's state machine of the EnAS Benchmark Production System: SSM(EnAS).

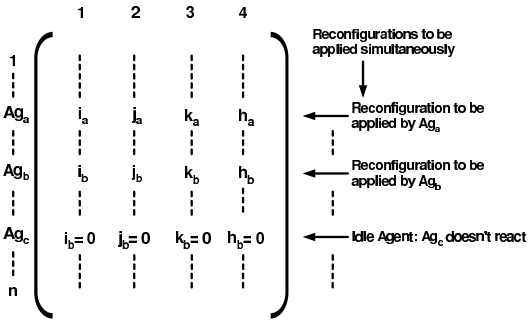

Figure 7.6: A Coordination Matrix.

Running Example. *In the Benchmark Production Systems FESTO and EnAS, we show in Figure 7.7 the Coordination Matrices to be applied in order to guarantee coherent distributed reconfigurations at run-time. According to Figures 7.2 and 7.3:*

- *the first matrix CM_1 is applied when the FESTO Agent applies the Light Production Mode (i.e. the states ASM_1, CSM_{11} and DSM_{11} are activated and $Reconfiguration_{1,1,1,1}$ is applied) and the EnAS Agent is required to decrease the productivity by applying the First Production Policy to put only one piece in each tin (i.e. the states ASM_2 and CSM_{21} are activated and $Reconfiguration_{2,1,0,0}$ is applied),*
- *the second matrix CM_2 is applied when the FESTO Agent applies the High Production Mode (i.e. the states ASM_2, CSM_{21} and DSM_{21} are activated and $Reconfiguration_{2,1,2,1}$ is applied) and the EnAS Agent is required to increase the productivity by applying the Second Production Mode to put two pieces into each tin (i.e. the states ASM_1 and CSM_{11} are activated and $Reconfiguration_{1,1,0,0}$ is applied),*
- *the third matrix CM_3 is applied when the FESTO Agent applies the Medium Production Mode (i.e. the states ASM_2, CSM_{22} and DSM_{22} are activated and $Reconfiguration_{2,2,2,2}$ is applied). In this case the EnAS System is required to apply the Second Production*

Policy (i.e. the states ASM_1 and CSM_{11} are activated and $Reconfiguration_{1,1,0,0}$ is applied),

- *the fourth matrix CM_4 is applied when the Jack station J_1 in the EnAS system is broken (i.e. the states ASM_1 and CSM_{12} are activated and $Reconfiguration_{1,2,0,0}$ is applied). In this case the FESTO system has to decrease the productivity by applying the Light Production Mode (i.e. the states ASM_1, CSM_{11} and DSM_{11} are activated and $Reconfiguration_{1,1,1,1}$ is applied),*
- *the matrix CM_5 is applied when the Jack station J_2 and the Gripper station G_1 are broken in the EnAS system (i.e. the states $ASM_1 and CSM_{13}$ are activated and $Reconfiguration_{1,3,0,0}$ is applied). In this case the FESTO system is required to decrease the productivity by applying the Light Production Mode,*
- *the matrix CM_1 is applied at run-time when the Gripper station G_2 is broken (i.e. the states ASM_2 and CSM_{21} are activated and $Reconfiguration_{2,1,0,0}$ is applied). In this case the FESTO agent has also to decrease the productivity by applying the Light Production Mode,*
- *the matrix CM_6 is applied when the Drilling machine Drill_machine1 is broken in FESTO (i.e. the states ASM_2, CSM_{23} and DSM_{23} are activated and $Reconfiguration_{2,3,2,3}$ is applied). In this case, EnAS is required to decrease the productivity by applying the First Production Mode (i.e. the states ASM_2 and CSM_{21} are activated and $Reconfiguration_{2,1,0,0}$ is applied),*
- *the matrix CM_7 is applied when the second drilling machine is broken at run-time. In this case the EnAS system is required also the decrease the productivity by applying the First Production Mode,*
- *finally, the matrix CM_8 is applied at run-time to stop the whole production when the two drilling machines Drill_machine1 and Drill_machine2 are broken. In this case the EnAS Agent has to reach the halt state (i.e. the states ASM_1 and CSM_{14} are activated and $Reconfiguration_{1,4,0,0}$ is applied).*

Coordination Between Distributed Agents

We guarantee a coherent behavior of the whole distributed architecture, by defining a *Coordination Agent* (denoted by $CA(\xi(Sys))$) that handles the Coordination Matrices of $\xi(Sys)$ in order to control the rest of agents (i.e. Ag_a, $a \in [1, n]$) as follows:

- When a particular agent Ag_a ($a \in [1, n]$) should apply a reconfiguration scenario $Reconfiguration^a_{i_a,j_a,k_a,h_a}$ (i.e. under well-defined conditions), it sends the following request to $CA(\xi(Sys))$ to obtain its authorization:

$$request(Ag_a, CA(\xi(Sys)), Reconfiguration^a_{i_a,j_a,k_a,h_a}).$$

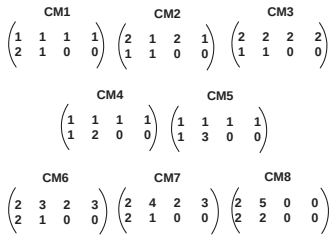

Figure 7.7: Coordination Matrices For the FESTO and EnAS Benchmark Production Systems.

- When $CA(\xi(Sys))$ receives this request that corresponds to a particular coordination matrix $CM \in \xi(Sys)$ and if CM has the highest priority between all matrices of $Concur(CM) \cup \{CM\}$, then $CA(\xi(Sys))$ informs the agents that have simultaneously to react with Ag_a as defined in CM. The following information is sent from $CA(\xi(Sys))$:

  For each Ag_b, $b \in [1, n] \setminus \{a\}$ and $CM[b, i] \neq 0$, $\forall\ i \in [1, 4]$:

$$reconfiguration(CA(\xi(Sys)), Ag_b, Reconfiguration^b_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]})$$

- According to well-defined conditions in the device of each Ag_b, the $CA(\xi(Sys))$ request can be accepted or refused by sending one of the following answers:

  - **If** $cond^b_{i_b} = cond^b_{j_b} = cond^b_{k_b,h_b} =$ True
    Then the following reply is sent from Ag_b to $CA(\xi(Sys))$:

$$possible\_reconfig(Ag_b, CA(\xi(Sys)), Reconfiguration^b_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]}).$$

  - **Else** the following reply is sent from Ag_b to $CA(\xi(Sys))$:

$$not\_possible\_reconfig(Ag_b, CA(\xi(Sys)), Reconfiguration^b_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]}).$$

- **If** $CA(\xi(Sys))$ receives positive answers from all agents, **Then** it authorizes the required reconfigurations in the concerned devices:

$$\textbf{For each } Ag_b,\ b \in [1, n] \text{ and } CM[b, i] \neq 0,\ \forall\ i \in [1, 4],\ apply(Reconfiguration^b_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]}) \text{ in } device_b.$$

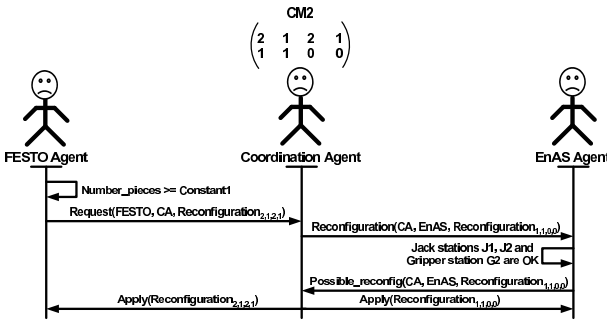

Figure 7.8: Coordination between the FESTO and EnAS agents to optimize their productivities.

Else If $CA(\xi(Sys))$ receives a negative answer from a particular agent, and **If** the reconfiguration scenario $Reconfiguration^{a}_{i_a,j_a,k_a,h_a}$ allows optimizations of the whole system behavior, **Then** $CA(\xi(Sys))$ refuses the request of Ag_a by sending the following reply:

$$refused\_reconfiguration(CA(\xi(Sys)), Ag_a,$$
$$Reconfiguration^{a}_{CM[a,1],CM[a,2],CM[a,3],CM[a,4]})).$$

Running example. *In the Benchmark Production Systems FESTO and EnAS, we show in Figure 7.8 the interactions between their agents when number_pieces* $\geq Constant1$. *In this case, the Coordination Agent uses the Matrix CM2 to coordinate between them in order to apply in FESTO the High Production Policy and in EnAS the Second Production Mode. We show in Figure 7.9 the coordination between these agents when Drill_machine*1 *is broken in FESTO. In this case, the Coordination Agent uses the Matrix CM6 to decrease the productivity in EnAS. We note that any reconfiguration scenario should be applied after the complete process of a piece in FESTO and EnAS.*

Evaluation of the Proposed Multi-Agent Architecture

We evaluate in this section the proposed multi-agent architecture for automatic reconfigurations of distributed embedded control systems by counting the maximum number of exchanged messages between agents after distributed reconfiguration scenarios. *We assume as a particular case that all Coordination Matrices are concurrent.* We denote by $number^{coordination}_{messages}$ such number when we use a Coordination Agent to coordinate between distributed devices of the execution environment, and by $number_{messages}$ when we do not apply any coordination. In this case, each agent should know priorities of matrices, and

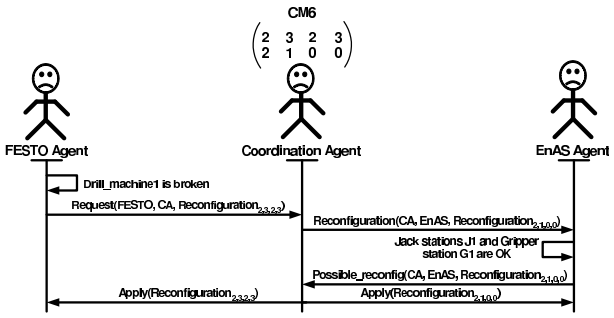

Figure 7.9: Coordination between the FESTO and EnAS agents when $Drill\_machine1$ is broken.

should inform all others before applying reconfiguration scenarios in the corresponding device. Let $number_{reconfigurations}$ be the number of desired reconfigurations by the distributed agents.

- **If** we apply the proposed approach where a Coordination Agent is applied to coordinate between devices such that the highest-priority reconfiguration scenario is applied at run-time, **Then** $number_{messages}^{coordination}$ is as follows:

$$number_{messages}^{coordination} = number_{reconfigurations} + 3 * (n-1) + 1$$

  Indeed, $number_{reconfigurations}$ among n agents desiring reconfigurations of corresponding devices send $number_{reconfigurations}$ messages to the Coordination Agent, but only the highest-priority message is accepted before a notification is sent to the rest (i.e. n-1) of agents. The Coordinator decides any scenario to be applied once answers are received from the distributed agents,

- **If** we apply an approach without any coordination where each agent should inform all others before applying reconfiguration scenarios, **Then** $number_{messages}$ is as follows:

$$number_{messages} = number_{reconfigurations} * 3 * (n-1)$$

  In this case, each agent desiring reconfigurations sends messages to all others before waiting their answers and deciding the next scenario to be applied.

The gain of our approach is then the decrease of these exchanged messages between distributed devices of the execution environment:

$$Gain = number_{messages}^{coordination} \; / \; number_{messages} = 1/(3*(\text{n-1})) + 1/number_{reconfigurations} + 1/(number_{reconfigurations} * 3 * (n-1))$$

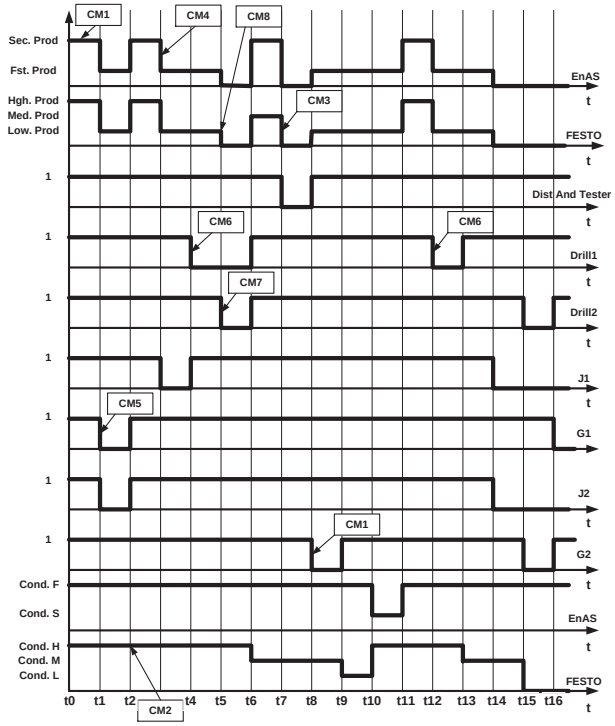

Figure 7.10: Evaluation of the multi-agent architecture by Varying the number of reconfigurations.

Application. If $n = 100$ and $number_{reconfigurations} = 100$, Then $Gain = 0,01$.

Running example. *We use in this research work the Simulink environment to simulate both FESTO and EnAS as well as their coordination when hardware faults occur in the plant or when the improvement of performance should be applied according to user requirements. We show in Figure 7.10 the behaviors of these benchmark platforms to be balanced between high, medium and light productions according to environment's evolutions. The figure shows also the different matrices to be used by the Coordination Agent to guarantee coherent distributed reconfigurations. In particular, the matrix* $CM2$ *is applied at* $t2$ *by the Coordination Agent to improve the productivity of FESTO and EnAS. The matrix* $CM4$ *is used at* $t3$ *to decrease this productivity under well-defined conditions. We show also in the same Figure how FESTO is stopped at* $t5$ *or at* $t7$ *when* $Drill1$ *and* $Drill2$ *or the Distribution and Tester units are broken.*

7.3.3 Implementation

We developed a complete tool ProtocolReconf by using Qt Creator 2.0.0 (for more information we refer to http://qt.nokia.com/products). We firstly present its different graphic interfaces before we show a simulation verifying the communication protocol. The tool ProtocolReconf offers the possibility to create the Reconfiguration and Coordination Agents (Figure 7.11) by introducing their parameters. For the Reconfiguration Agent (Figure 7.12), it is necessary to define the Data, Device and Reconfigurations. Each data must be defined by indicating its name and value, and each device is characterized also by its identifier and state (functional or broken). It is required to define the different scenarios that the Reconfiguration Agent can support so that when a modification occurs in the system, it should look for the convenient reconfiguration. For the Coordination Agent (Figure 7.13), it is necessary to define the set of Coordination Matrices and especially the current matrix to apply to the whole system. The communication between the different Reconfiguration Agents follows the specific defined protocol. To ensure a new reconfiguration, an agent sends a request to the Coordination Agent indicating the new reconfiguration to apply. Consequently, the coordinator searches the right Coordination Matrix and sends a request to the rest of concerned Reconfiguration Agents. After receiving all the feedbacks, the Coordination Agent decides to apply this new coordination matrix (if all reconfiguration agents accept this modification) or to cancel the corresponding reconfiguration scenario.

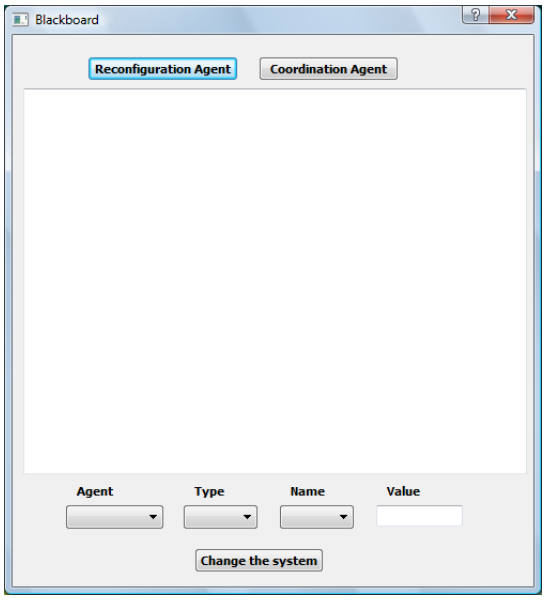

Figure 7.11: The main interface.

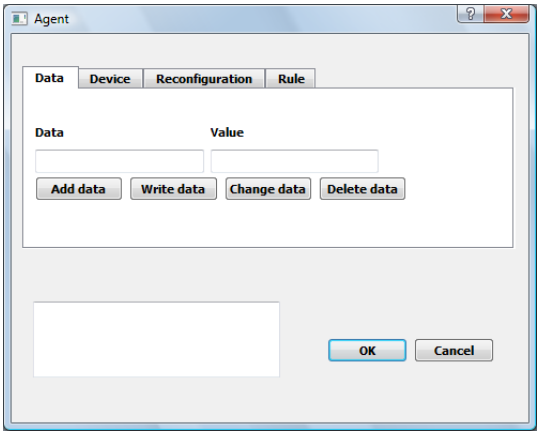

Figure 7.12: Reconfiguration Agent.

Running Example1. *In FESTO and EnAS (Figure 7.14), we assume that the matrix CM2 is applied i.e. the FESTO's Agent applies the High Production Mode and the EnASs Agent applies the Second Production Strategy. To verify the interaction between*

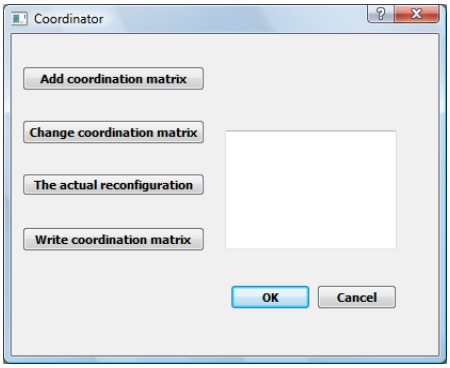

Figure 7.13: Coordination Agent.

these agents when a particular hardware problem occurs, we change the state of the device Driller1 which becomes broken. Consequently, the FESTO's Agent should decrease the production by sending a request to the Coordination Agent in order to look for the most convenient matrix which is CM6. The Coordination Agent sends a request to decrease the production in EnAS. The EnAS's Agent studies the feasibility of this new reconfiguration in order to accept the decrease of production. In this case, the Coordination Agent sends a final confirmation to officially apply this new Coordination Matrix.

Running Example2. *We assume that the matrix CM6 is applied i.e. the FESTO's Agent applies the Low Production Mode and the EnAS's Agent applies the First Production Strategy (Figure 7.15). When the state of the device Driller2 becomes broken, the FESTO's Agent should stop the production by sending a request to the Coordination Agent in order to halt the second agent. The Coordination Agent decides to apply the matrix CM8 and sends a request to stop the production in EnAS. The EnAS's Agent accepts this new reconfiguration. Consequently, the Coordination Agent sends a confirmation to stop the production in both EnAS and FESTO system.*

7.4 Contribution: NCES-based Modelling and SESA-Based Model Checking of Distributed Reconfigurable Embedded Systems

We specify each reconfiguration agent affected to a particular device of the execution environment by nested (i.e. hierarchical) Net Condition-Event Systems that support all reconfiguration forms, and specify also the different networks of Control Components that

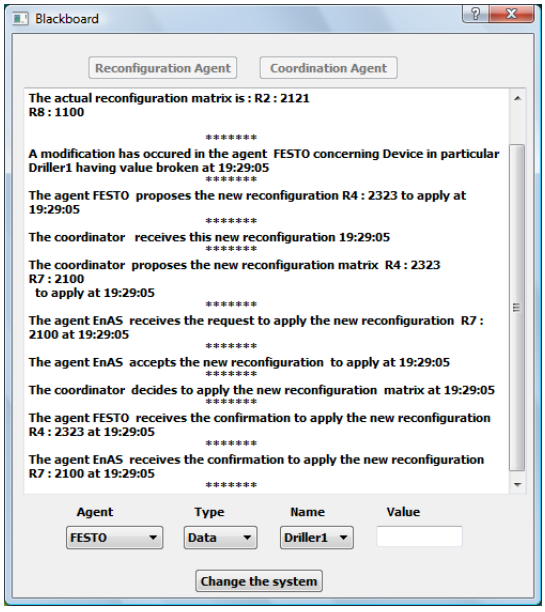

Figure 7.14: First example of Communication Protocol.

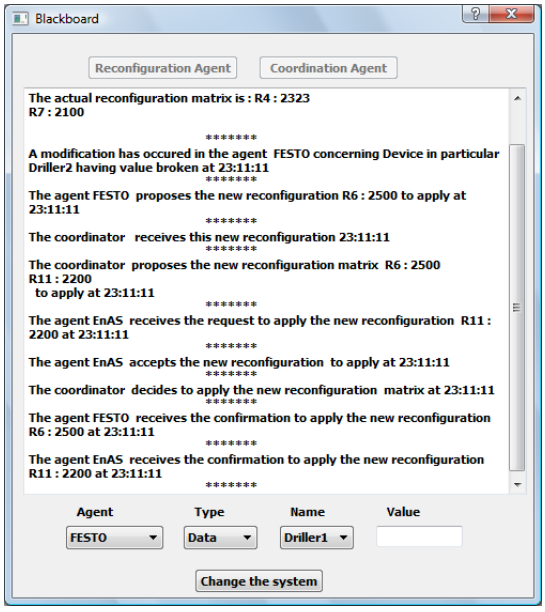

Figure 7.15: Second example of Communication Protocol.

implement this device in different possible reconfiguration scenarios by NCES-based models, before apply the model checker SESA to check eCTL-based functional and TCTL-based temporal properties described in user requirements. We specify in addition each Coordination Matrix by a NCES-based model and apply also SESA to check that any distributed reconfiguration scenario applied by agents brings the whole distributed architecture into safe and optimal behaviors.

7.4.1 NCES-Based Specification of a Reconfiguration Agent

We use in this research the formalism Net Condition/Event Systems that provides useful facilities to specify any synchronization between agents and Control Components. We use in particular event-condition signals from agents to activate traces of states in the SSM (i.e. a reconfiguration), and use also event signals to synchronize the state machines: ASM, CSM and DSM of each agent.

Running example1. *We show in Figure 7.16 the agent model in FESTO according to the formalism NCES. In the ASM state machine, the place* $PF1$ *defines the Light Production Mode and corresponds to the state machine* $CSM1(PF1)$*. The synchronization between these two machines is supported by an event signal* $ev1$*. On the other hand, the place* $PF2$ *defines the Medium or the High mode and corresponds to the state machine* $CSM(PF2)$ *(synchronization is supported by the event signal* $ev2$*). The place* $PF5$ *of* $CSM(P2)$ *defines the High Production Mode and corresponds to the place* $PF12$ *when the period* $p = 11seconds$ *is applied. The place* $PF3$ *of* $CSM(PF2)$ *defines the Medium Production Mode*

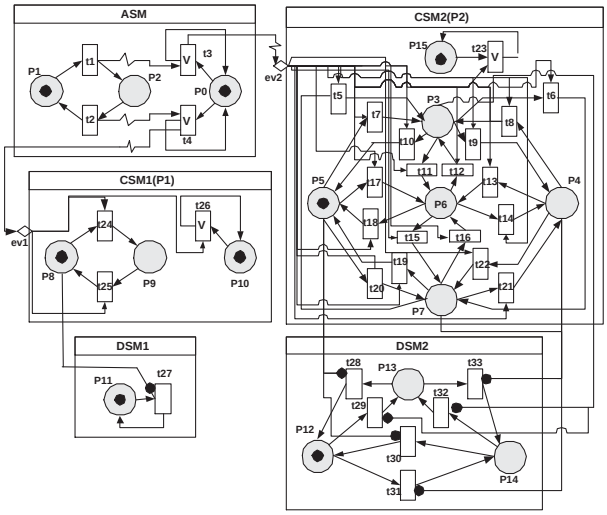

Figure 7.16: Design of the FESTO agent with the NCES formalism.

and corresponds to the place $PF13$ *when the period* $p = 30seconds$ *is applied.*

Running example2. *Figure 7.17 shows the models of EnAS according to the formalism NCES. We specify temporal intervals in the transitions of the system's model according to user requirements. When the Jack station* J_1 *is broken, the agent activates the place* PE_{12} *and sends a condition signal to activate the trace* trace1 *in the system. Note that the architecture and control state machines are communicating by event signals to synchronize the agent's behavior. Finally, the state "Well" represents a deadlock in the system when the Jack stations* J_1 *and* J_2 *are broken.*

7.4.2 SESA-Based Model Checking in a Device

We limit our research work to two types of extra-functional constraints namely: *Temporal* and *QoS* properties. We use the temporal logic CTL as well as its extensions to specify these properties and the model checker SESA to check the system feasibility in each device of the execution environment.

Running example. *In EnAS, we apply the model checker SESA to verify properties of its Control Components implementing the different reconfiguration scenarios. We generate with this tool different reachability graphs corresponding to these scenarios. We present in Figure 7.18 the graph that corresponds to the second Production Policy where the states* $State1$,...$State17$ *encode the behavior of the Agent as well as the system's components when* $nb_{pieces}/nb_{tins+caps} > Threshold$.

*** Verification of Functional Properties**

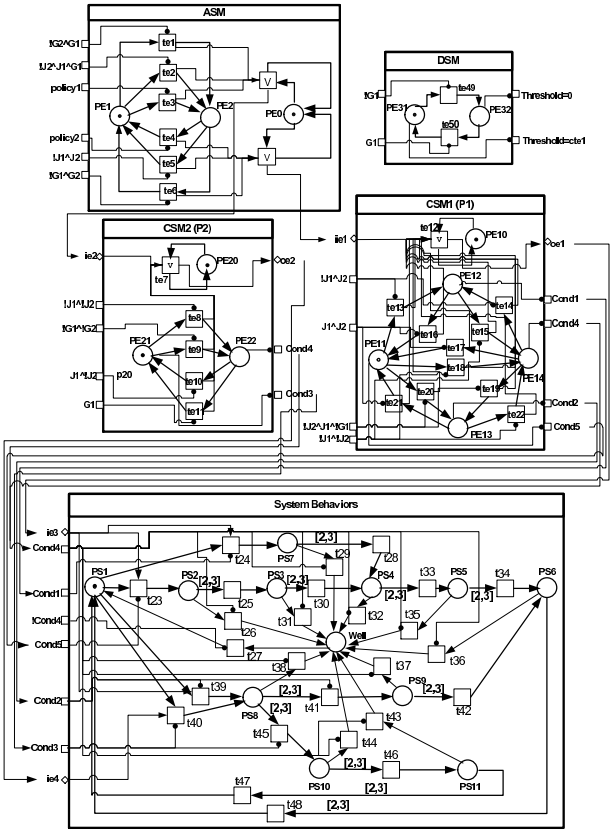

Figure 7.17: Design of the EnAS system with the NCES formalism.

In order to validate the whole architecture, we verify functional properties of state machines specifying the agent to prove the correct adaptation of the system to any change in its environment, and verify the functional correctness of the different networks of components.

Running example. *In EnAS, we check functional properties of the state machines encoding the Agent. In particular, we have to check if the system applies the second policy whereas the Gripper station G_2 is broken. We propose the following eCTL formula:*

$$z_0 \models AGAte1XPS2$$

This formula allows to check with SESA that whenever a state transition fulfilling te_1 is possible (e.g. G_2 is broken), this transition leads to a successor state in which PS_2 holds true in the reachability graph (e.g. we apply the second Production Policy). This formula is proven to be $False$.

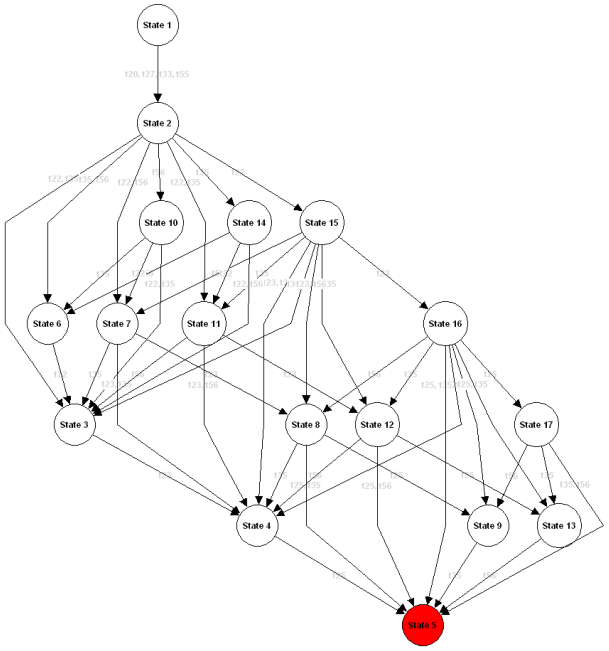

Figure 7.18: Reachability graph of the first Production Policy.

The following formula is proven to be True:

$$z_0 \models AGAte1XPS8$$

Indeed, whenever the Gripper G_2 is broken, the state PS_8 of SSM is activated. On the other hand, to check the behavioral order of the SSM when the Gripper station G_2 is broken (e.g. Load a piece and Close the tin in the Jack station J_1, then move the Belt to the Gripper station G_1 to remove the product to Belt1), we propose the following eCTL formula:

$$z_0 \models AGAt_{40}XAFEt_{45}XAFEt_{46}XTRUE$$

This formula is proven to be True with the model checker SESA.

*** Verification of Temporal Properties**

We verify TCTL-based temporal properties of the networks of components corresponding to the different reconfiguration scenarios as follows:

Running example. *In the EnAS platform, when the Gripper station G_2 is broken, we should check if the duration to activate the Gripper station G_1 does not exceed 5 time units. Therefore, we propose to apply the following formula:*

$$AF[2,5]PS_{11}$$

This formula is proven to be True with the Model checker SESA. To check if the Gripper station is reachable in 3 time units, we should verify the following formula:

$$EF[2,3]PS_{11}$$

It is proven to be False with the model checker.

*** Verification of QoS Properties**

The last property to be verified according to user requirements is the QoS where we check if the system provides the minimal accepted quality at run-time. We use in this case the eCTL logic to specify the QoS formulas.

Running example. *In the EnAS platform, we verify if the system provides the minimal accepted QoS. According to the value of $nb_{pieces}/nb_{tins+caps}$, we should verify if the system applies the best production policy. We propose to following eCTL formula,*

$$z_0 \models AGAte_3XPS11$$

Indeed, we should verify if $nb_{pieces}/nb_{tins+caps} \leq Threshold$ (e.g. the first Production Policy should be applied), then the fourth trace of SSM should be activated (e.g. the state PS_{11} should be activated). By applying the tool SESA, we find that this formula is $True$.

7.4.3 SESA-Based Model Checking of the Coordination Agent

The model checking of a distributed reconfigurable system is mandatory to check the reactivity of distributed agents when reconfiguration scenarios are applied in corresponding devices. We propose a NCES-based model for each Coordination Matrix to be handled by the Coordination Agent, and propose thereafter the verification of the whole system behavior by applying the model checker SESA and the temporal logic CTL.

*** NCES-Based Modelling of the Coordination Agent**

We model each Coordination Matrix $CM \in \xi(Sys)$ to be handled by the Coordination Agent $CA(\xi(Sys))$ by a NCES-based Coordination Model in which the conditions $cond^a_{i_a}$, $cond^a_{j_a}$ and $cond^a_{k_a,h_a}$ are verified for each non idle agent Ag_a ($a \in [1,n]$) (i.e. application of the reconfiguration scenario $Reconfiguration^a_{i_a,j_a,k_a,h_a}$) before an authorization is sent to all non idle agents in order to effectively apply corresponding reconfigurations.

Running Example1. *We show in Figure 7.19 the Coordination Module $Module(CM_{7,8})$ to be applied when the drilling machines $Drill\_machine1$ or $Drill\_machine2$ are broken (i.e. the states $PF4$ and $PF7$ of the $CSM1(PF1)$). In this case, the EnAS*

Agent should reduce the productivity by applying the First Production Mode (i.e. the state $PE2$ of $ASM(EnAS)$). We show in Figure 7.20 the module $Module(CM_{4,5})$ that defines the behavior of the Coordination Matrix when the Jack stations J_1, J_2 or the Gripper G_1 are broken. In this case, the FESTO Agent should reduce the productivity by applying the Light Production Mode (i.e. the state $PF1$ of $ASM(FESTO)$). We show in Figure 7.21 the module $Module(CM1)$ that defines the behavior of the Coordination Matrix when the Light Production Mode is applied by the FESTO Agent ($number\_pieces < Constant2$). In this case, the EnAS Agent should apply the First Production Mode in which only one piece is put in the tin. On the other hand, the module $Module(CM2,3)$ defines the behavior of the Coordination Matrix when the FESTO Agent should apply the High or the Medium modes. In this case the EnAS Agent should change the production strategy to the Second Production Mode where two pieces are put in the tin. To manage concurrent coordination matrices, the resolution of hardware problems is assumed to have a higher priority than any optimization of the system productivity. Therefore, the Coordination Matrix $CM_{7,8}$ ($CM_{4,5}$, resp) has higher priority than CM_1 ($CM_{2,3}$, resp). According to Figure 7.21, the matrix CM_1 ($CM_{2,3}$, resp) is applied if and only if the drilling machines $Drill\_machine1$ and $Drill\_machine2$ (the Jack stations J_1, J_2 and the Gripper station G_1) are not broken.

*** SESA-Based Verification of Distributed Reconfigurations**

We verify with the model checker SESA the behavior of the whole control system when distributed reconfigurations are applied by the Coordination Agent. Indeed, we have to check for each Coordination Matrix $CM \in \xi(Sys)$ that whenever an Agent Ag_a ($a \in [1,n]$) applies a reconfiguration scenario under well-defined conditions, the other non-idle agents have to react by applying required reconfigurations.

Running Example. *In the Benchmark Production Systems FESTO and EnAS, we apply the model checker SESA to verify distributed reconfiguration scenarios of NCES-based models describing agents. Our objective is to check that whenever one of these demonstrators improves or decreases the productivity, the other applies the same strategy. We have in addition to check that each one reacts when any hardware problem occurs in the other. This verification is mandatory in order to guarantee coherent behaviors of these complementary demonstrators. The model checker generates a reachability graph composed of 162 states for the NCES-based models of the considered agents. We specify the following functional properties according to the temporal logic CTL:*

- **Property1:** whenever the drilling machines $Drill\_machine1$ or $Drill\_machine2$ are broken in the Benchmark System FESTO (i.e. the states $PF7$ or $PF4$ are reached), the EnAS system should therefore decrease the productivity (i.e. the state $PE2$ is reached). The following formulas are proven to be true by the model checker SESA:

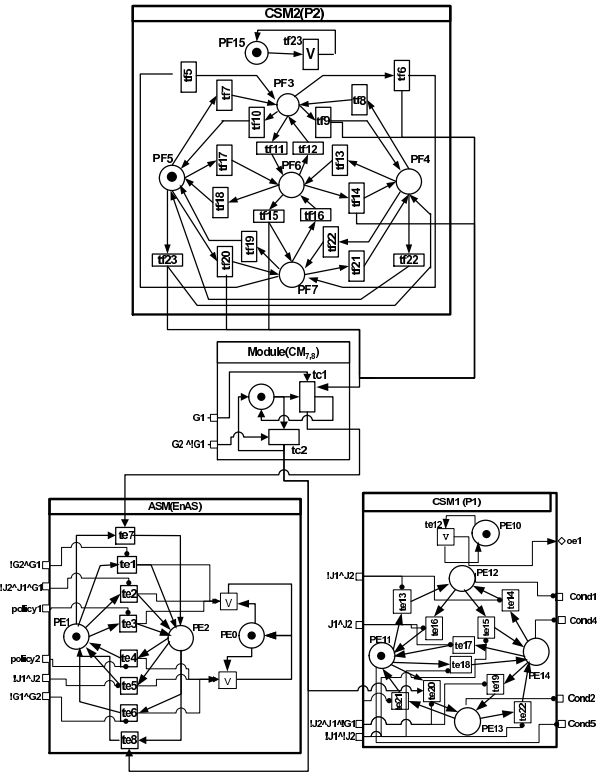

Figure 7.19: Automatic Distributed Reconfigurations in FESTO and EnAS when the drilling machines $Drill\_machine1$ or $Drill\_machine2$ are broken at run-time.

Formula1: $z_0 \models AGAte7XAGAtc1XPF7$

Formula2: $z_0 \models AGAte7XAGAtc1XPF4$

- **Property2:** whenever the Jack stations $J1$ and $J2$ or the Gripper station $G1$ are broken in the Benchmark System EnAS. the FESTO system should react by decreasing the productivity to the Light Production Mode. The following formulas are proven to be true by the model checker SESA:

Formula2 (J1 is broken): $z_0 \models AGAtf2XAGAtc3XPF12$

Formula3 (J2 and G1 are broken): $z_0 \models AGAtf2XAGAtc3XPF13$

- **Property3:** If the condition $number\_pieces \geq Constant2$ is satisfied and the system FESTO improves in this case the productivity to the Medium or the High modes (i.e. the place $PF2$ is reached), then the EnAS system should improve also the productivity by applying the second Production Policy where two pieces

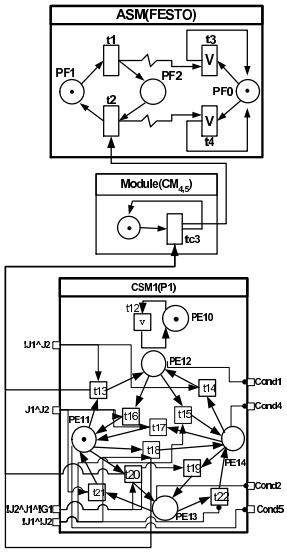

Figure 7.20: Automatic Distributed Reconfigurations in FESTO and EnAS when hardware problems occurs at run-time.

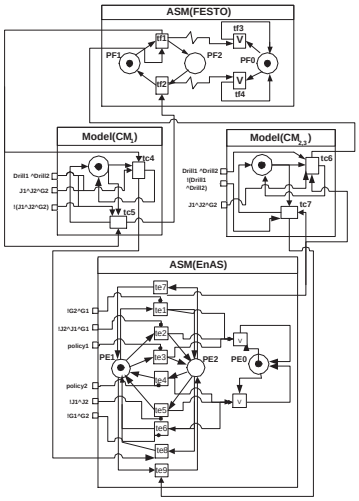

Figure 7.21: Automatic Distributed Reconfigurations in FESTO and EnAS to regulate the whole system performance.

are put in each tin. The following formula is proven to be true by the model checker SESA:

Formula4: $z_0 \models AGAtf8XAGAtc4XPF2$

- **Property4:** If the condition ($nb_{pieces}/nb_{tins+caps} > Threshold$) is satisfied and the EnAS system has to improve the productivity by applying the second mode (i.e. the place $PE1$ is reached), then the FESTO system should also increase the productivity by applying the Medium or High modes (i.e. the state PF1 is reached). The following formula is proven to be true by the model checker SESA:

Formula5: $z_0 \models AGAtf1XAGAtc6XPE1$

7.5 Contribution: Hierarchical Verification of Control Components

Once the Reconfiguration and Coordination Agents are well-checked, we are interested in this section in the detailed model checking of Control Components implementing each complex network of Sys and to be executed after a well-defined reconfiguration scenario. To control the verification complexity, we check step by step each network $Net_{a,b,c,d}$ ($a \in [1, n_{ASM}], b \in [1, n_{CSM_a}], c \in [1, n_{DSM}], d \in [1, n_{DSM_c}]$) by applying a refinement-based strategy. We define at first time a NCES-based abstract model for each $Network_{a,b,c,d}$ which is automatically refined in several steps where NCES-based models of Control Components are automatically generated. The model checker SESA is automatically applied in each step to verify deadlock properties of the new generated components, and is manually applied to verify in addition CTL-based functional properties according to user requirements. The whole control system is feasible if each network $Net_{a,b,c,d}$ of Control Components is correct and safe.

7.5.1 Refinement-based Specification and Verification of a Network of Control Components

Let $Net_{a,b,c,d}$ be a network of Control Components that implement the system according to described conditions in user requirements. We specify at first time this network by a NCES-based abstract model (denoted initially by $M^1_{a,b,c,d}$) in which:

- data are read from sensors to activate actuators of $first(\alpha_{actuators})$,
- all the actuators of $last(\alpha_{actuators})$ are activated.

The Control Components composing this network are automatically modeled in several steps according to the formalism NCES, and are automatically and manually checked in each step by the model checker SESA as follows:

In each *Step i* ($i \geq 1$), Control Components are recursively modeled and checked as follows:

Step i.

- The abstract model $M^i_{a,b,c,d}$ is automatically refined by generating a new set $set^i_{a,b,c,d}$ of NCES-based models of Control Components,
- A new abstract model $M^{i+1}_{a,b,c,d}$ is automatically generated from $M^i_{a,b,c,d}$,
- The model checker SESA is automatically applied to verify deadlock properties of Control Components of $set^i_{a,b,c,d}$ with $M^{i+1}_{a,b,c,d}$. It is manually applied in addition to verify CTL-based functional properties.

  If the new generated Control Components are safe and correct **Then**:

  If $M^{i+1}_{a,b,c,d}$ is not empty **Then** it is automatically refined in *Step i+1*,
  Else the network of Control Components $Net_{a,b,c,d}$ is safe.

  Else The whole system is not feasible because $Net_{a,b,c,d}$ is not safe.

Running example. In FESTO and EnAS, we show in Figure 7.22 the abstract NCES-based model $M^1_{\{2,1,2,1\};\{1,1,0,0\}}$ where the High Production Mode is followed in the first system (i.e. $Reconfiguration_{2,1,2,1}$ is applied and $Net_{2,1,2,1}$ is executed) and the Second Production Policy is followed in the second one (i.e. $Reconfiguration_{1,1,0,0}$ is applied and $Net_{1,1,0,0}$ is executed). This model contains four abstract traces of states. In particular, the first trace $tr1$ is followed by FESTO when the test of the workpiece is failed. $tr1$ is characterized as follows:

** It reads data from the sensors $S1$ and $S2$ to activate thereafter the actuator act_1,

** It allows the activation of the actuator act_4 to evacuate the failed workpiece ($act_4 \in last(\alpha_{actuators})$).

We present in Figure 7.23 the first step of the refinement process to automatically generate NCES-based models of $CC\_Feeder_{2,1,2,1}$ and $CC\_Belt_{1,1,0,0}$ that respectively control the Feeder in FESTO and the Belt in EnAS. The model checker SESA is automatically applied in this step to check the safety of these components (i.e. no deadlock is possible). We present in Figure 7.24 the second refinement step to automatically generate $CC\_Convert_{2,1,2,1}$ and $CC\_Jack1_{1,1,0,0}$ that respectively control the Converter in FESTO and the first Jack station in EnAS. We present in Figure 7.25 the last steps to automatically generate $CC\_Checker_{2,1,2,1}$ and $CC\_Evac_{2,1,2,1}$ that respectively control $Checker$ and $Evacuator2$ in FESTO.

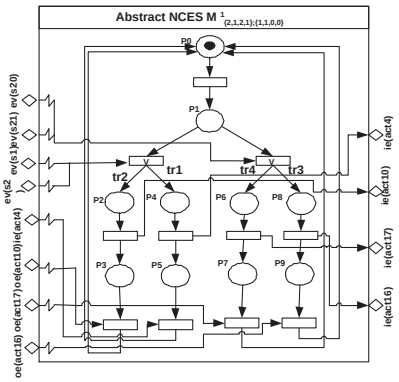

Figure 7.22: The abstract model $M^1_{\{2,1,2,1\};\{1,1,0,0\}}$ of FESTO and EnAS systems

Formalization

We formalize the automatic refinement-based specification and verification of Control Components that implement the whole control system. We denote by $\Sigma^i_{a,b,c,d}$ the set of actuators that should be considered for the refinement of the abstract model $M^i_{a,b,c,d}$ (when the reconfiguration scenario $Reconfiguration_{a,b,c,d}$ is well-applied by the agent) and by $first(\Sigma^i_{a,b,c,d})$ a subset of actuators of $\Sigma^i_{a,b,c,d}$ with no predecessors in $\Sigma^i_{a,b,c,d}$.

$$\forall act \in first(\Sigma^i_{a,b,c,d}), prev(act) \subset \alpha_{actuators} \setminus \Sigma^i_{a,b,c,d}$$

Let $simul\_set^i_{a,b,c,d}$ be a subset of actuators to be activated simultaneously in $first(\Sigma^i_{a,b,c,d})$. We denote in addition by $sensor(simul\_set^i_{a,b,c,d})$ the set of sensors that provide required data before any activation of $simul\_set^i_{a,b,c,d}$, and we denote by $set\_follow(simul\_set^i_{a,b,c,d})$ the following set:

$$set\_follow(simul\_set^i_{a,b,c,d}) =$$
$$\{act\_set \subset \alpha_{actuators} / \exists act \in simul\_set^i_{a,b,c,d}, act\_set \in follow(act)\}$$

The automatic refinement-based specification and verification of Control Components is applied as follows:

Algorithm. Step-By-Step

* For each *Step* i corresponding to the refinement of the NCES-based abstract model $M^i_{a,b,c,d}$:

** For each $simul\_set^i_{a,b,c,d} \subset first(\Sigma^i_{a,b,c,d})$:

*** create Control Component CC,

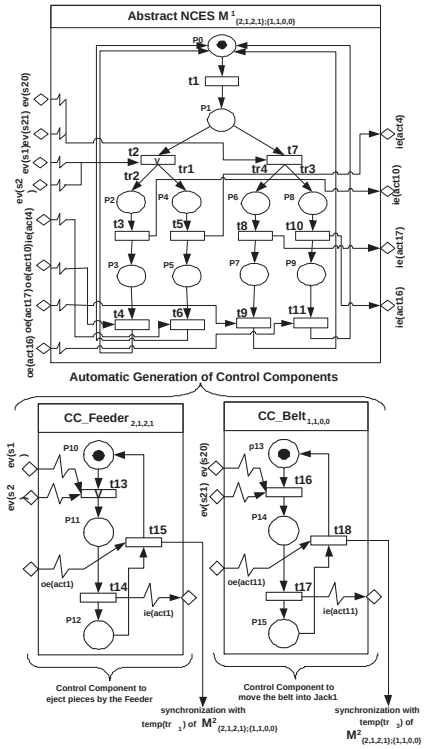

Figure 7.23: The first Step of an automatic refinement process

*** For each $act \in simul\_set^i_{a,b,c,d}$,

**** add $i(CC, sensor(act))$ to CC,

**** add $a(CC, act)$,

**** add $cf(CC, act)$,

*** $I \leftarrow simul\_set^i_{a,b,c,d} \cap last(\alpha_{actuators})$,

*** For each $act' \in I$,

**** $M^i_{a,b,c,d} \leftarrow M^i_{a,b,c,d} \setminus i(M^i_{a,b,c,d}, sensor(act'))$,

**** $M^i_{a,b,c,d} \leftarrow M^i_{a,b,c,d} \setminus a(M^i_{a,b,c,d}, act')$,

**** $M^i_{a,b,c,d} \leftarrow M^i_{a,b,c,d} \setminus cf(M^i_{a,b,c,d}, act')$,

*** $simul\_set^i_{a,b,c,d} \leftarrow simul\_set^i_{a,b,c,d} \setminus I$,

*** If($simul\_set^i_{a,b,c,d}$)

**** For each $act' \in simul\_set^i_{a,b,c,d}$,

***** $M^i_{a,b,c,d} \leftarrow M^i_{a,b,c,d} \setminus i(M^i_{a,b,c,d}, sensor(act'))$,

**** For each $set' \in set\_follow(simul\_set^i_{a,b,c,d})$, For each $act' \in set'$,

***** $M^i_{a,b,c,d} \leftarrow M^i_{a,b,c,d} \cup i(M^i_{a,b,c,d}, sensor(act'))$.

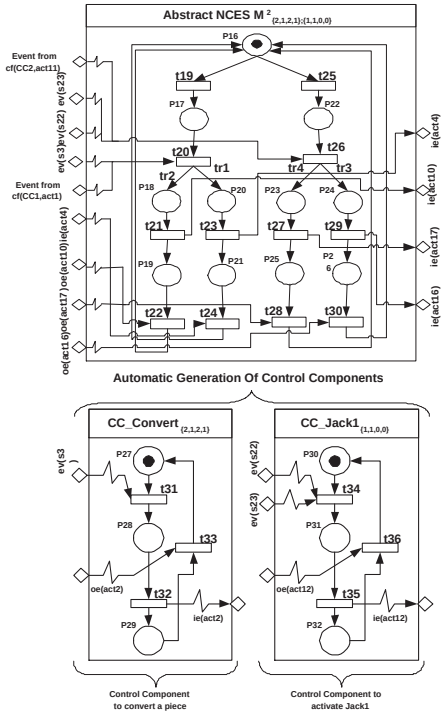

Figure 7.24: The second Step of an automatic refinement process

* If SESA automatically finds a deadlock in the current step, or manually proves that a CTL-based property is not satisfied,

** Then Stop the automatic construction,

** Else

*** $\Sigma^{i+1}_{a,b,c,d} \leftarrow \Sigma^{i}_{a,b,c,d} \setminus first(\Sigma^{i}_{a,b,c,d})$,

*** $M^{i+1}_{a,b,c,d} \leftarrow M^{i}_{a,b,c,d}$,

*** If($\Sigma^{i+1}_{a,b,c,d}$)

**** Then apply *Step* $i+1$ to refine the NCES-based model $M^{i+1}_{a,b,c,d}$,

*** Else display(System is safe).

End Algorithm.

Running Example. In the systems FESTO and EnAS, and according to Figure 7.23, $first(\Sigma^{1}_{\{2,1,2,1\};\{1,1,0,0\}}) = \{act1, act11\}$. We refine therefore the abstract NCES-based model $M^{1}_{\{2,1,2,1\};\{1,1,0,0\}}$ into two Control Components $CC\_Feeder_{2,1,2,1}$ and $CC\_Belt_{1,1,0,0}$ that respectively activate the actuators $act1$ and $act11$. A new abstract model $M^{2}_{\{2,1,2,1\};\{1,1,0,0\}}$ is automatically generated from $M^{1}_{\{2,1,2,1\};\{1,1,0,0\}}$ where $first(\Sigma^{2}_{\{2,1,2,1\};\{1,1,0,0\}}) = \{act2, act12\}$.

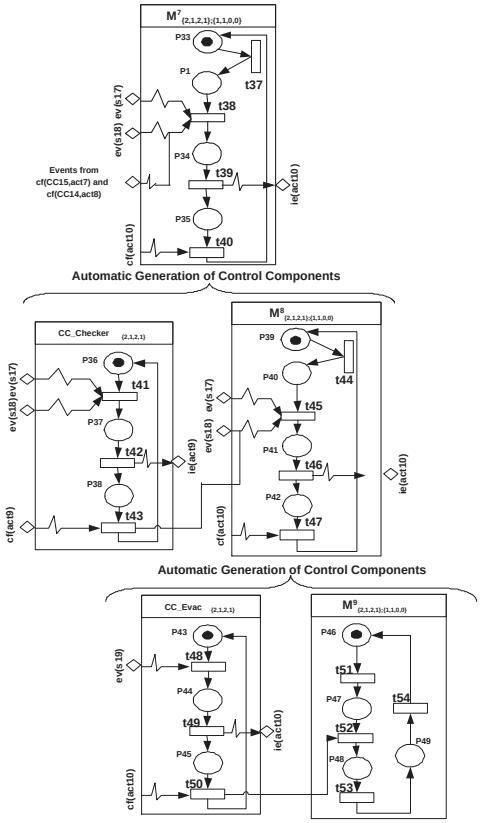

Figure 7.25: The last Steps of an automatic refinement process

Verification of CTL-based Properties with SESA

In addition to automatic verifications of deadlock properties, we manually verify in each refinement step eCTL-based properties of FESTO and EnAS Benchmark Production Systems in order to check if the generated Control Components satisfy user requirements. In particular, the precedence constraint between the actuators act_1 and act_2 (resp. act_{11} and act_{12}) is manually verified in the first refinement step when the NCES-based models of the components $CC\_Feeder_{\{2,1,2,1\}}$ and $CC\_Convert_{\{2,1,2,1\}}$ are generated. We propose the following eCTL formula:

$$z0 \models AGAt20XP10 \text{ (resp. } z0 \models AGAt26XP13)$$

Indeed, whenever a state transition of the reachability graph that fulfills the transition $t20$ (resp. $t26$) of the abstract model $M^2_{\{2,1,2,1\};\{1,1,0,0\}}$ is possible i.e. the Plant Control System reads data from the sensor $S3$ (resp. sensors $S22$ and $S23$), this transition should lead to a successor state in which $P10$ (resp. $P13$) holds true i.e. the actuator act_1 (resp.

act_{11}) is already activated. These properties are proven to be True by the model checker SESA.

7.5.2 Generalization: Refinement-based Specification and Verification of a Reconfigurable System

We apply the proposed refinement-based technique to automatically specify different networks of Control Components that can probably implement the whole embedded system after well-defined reconfiguration scenarios:

****

Algorithm. Specify_Verify
  For each Architecture $ASM_a, i \in [1, n_{ASM}]$
    For each Control policy $CSM_{a,b}, b \in [1, n_{CSM_a}]$
      For each Data configuration $DSM_{c,d}, d \in [1, n_{DSM_c}]$ such that $DSM_c = Data(CSM_{a,b})$
        Step-By-Step($Net_{a,b,c,d}$**)**;
        If a deadlock occurs or a CTL-based property is not satisfied in $Step_i$ $i \geq 1$
          Then Stop Algorithm; Display (system is infeasible);
  Display(System is feasible);
End.

****

We developed and tested in our research laboratory these algorithms by checking the safety and correctness of different networks that can probably implement the Benchmark Production Systems FESTO and EnAS. If each network that corresponds to a particular reconfiguration scenario is correctly specified and verified, Then the correctness of these demonstrators is deduced. The complexity of these algorithms is as follows: let n be an upper bound of n_{ASM}, $\max\{n_{CSM_a}, a \in [1, n_{ASM}]\}$ and $\max\{n_{DSM_c}, c \in [1, n_{DSM}]\}$, and let β be the biggest number of steps to generate in one of the considered scenarios, the complexity is then in $O(n^3.\beta)$. In FESTO and EnAS, 49 reachability graphs are automatically generated by SESA to check the feasibility of Control Components in seven different networks corresponding to seven reconfiguration scenarios. We present in Figure 7.26 the number of states generated in each one of these graphs. By applying the proposed refinement-based approach, the sum of states generated by SESA after the verification of these different networks is 3736 states, whereas the number of states is theoretically about 10^{14} when we apply a classic approach without refinement for the verification of these demonstrators (Figure 7.27). This comparison shows the significance of our contributions in this study.

Running Example. We show in Figure 7.29 a reachability graph which is automatically generated by SESA in a last refinement step when $Reconfiguration_{2,1,2,1}$ (i.e. High

	Steps	1	2	3	4	5	6	7
FESTO: Reconfiguration $_{2,1,2,1}$	States Number	121	35	71	191	94	144	16
EnAS: Reconfiguration $_{1,1,0,0}$								
FESTO: Reconfiguration $_{2,2,2,2}$	States Number	123	36	74	183	91	36	16
EnAS: Reconfiguration $_{1,1,0,0}$								
FESTO: Reconfiguration $_{1,1,1,1}$	States Number	119	37	75	187	33	12	16
EnAS: Reconfiguration $_{2,1,0,0}$								
FESTO: Reconfiguration $_{2,3,2,3}$	States Number	131	38	73	186	31	24	16
EnAS: Reconfiguration $_{2,1,0,0}$								
FESTO: Reconfiguration $_{2,4,2,3}$	States Number	124	34	72	185	92	36	16
EnAS: Reconfiguration $_{1,1,0,0}$								
FESTO: Reconfiguration $_{1,1,1,1}$	States Number	128	35	77	189	31	12	16
EnAS: Reconfiguration $_{1,2,0,0}$								
FESTO: Reconfiguration $_{1,1,1,1}$	States Number	118	37	78	188	29	14	16
EnAS: Reconfiguration $_{1,3,0,0}$								

Figure 7.26: Refinement-based Verification of FESTO and EnAS Benchmark Production Systems

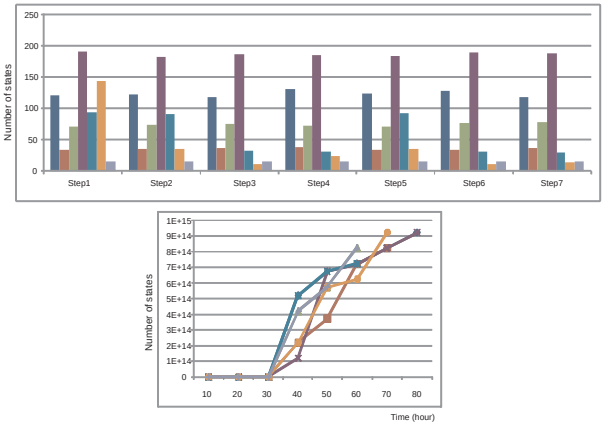

Figure 7.27: (a) Number of states automatically generated by SESA for the refinement-based specification and verification of 7 different networks of Control Components. (b) Number of states generated by SESA for the specification and verification of 7 networks of Control Components without applying any refinement

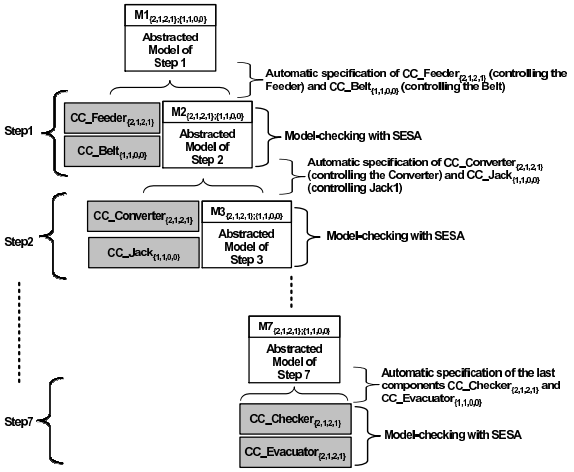

Figure 7.28: Automatic specification of feasible Control Components of FESTO and EnAS systems

Production Policy) is applied in FESTO and $Reconfiguration_{1,1,0,0}$ (i.e. Second Production Mode) is applied in EnAS, to check the Control Components $CC\_Checker_{2,1,2,1}$ and $CC\_Evacuator_{2,1,2,1}$ that control respectively the physical processes $Checker$ and $Evacuator2$ in FESTO. We show in Figure 7.30 another reachability graph which is automatically generated by SESA in *Step 6* when $Reconfiguration_{1,1,1,1}$ (i.e. Light Production Policy) is applied in FESTO and $Reconfiguration_{2,1,0,0}$ (i.e. First Production Mode) is applied in EnAS, to check the Control Component $CC\_Drill1_{2,1,2,1}$ that controls $Drill\_machine1$ in FESTO. We show finally in Figure 7.31 a reachability graph generated in *Step 6*, when $Reconfiguration_{2,2,2,2}$ (i.e. Medium Production Mode) is applied in FESTO and $Reconfiguration_{1,1,0,0}$ (First Production Mode) is applied in EnAS, to check the Control Components $CC\_Drill2_{2,2,2,2}$ and $CC\_G2_{1,1,0,0}$ that control respectively $Drill\_machine2$ in FESTO and the second Gripper station in EnAS.

7.6 Application: Implementation of Multi-Agent Reconfigurable IEC61499 Systems

We define XML-based implementations of Reconfiguration and Coordination agents for distributed reconfigurable embedded control systems following the technology IEC61499. A Control Component is assumed in this section to be a Function Block according to this industrial technology.

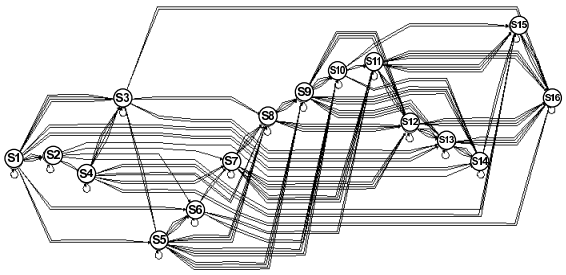

Figure 7.29: A reachability graph generated by SESA in Step7 when $Reconfiguration_{2,1,2,1}$ is applied in FESTO and $Reconfiguration_{1,1,0,0}$ is applied in EnAS

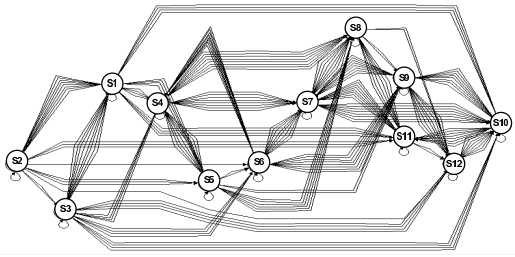

Figure 7.30: A reachability graph generated by SESA in *Step 6* when $Reconfiguration_{1,1,1,1}$ is applied in FESTO and $Reconfiguration_{2,1,0,0}$ is applied in EnAS

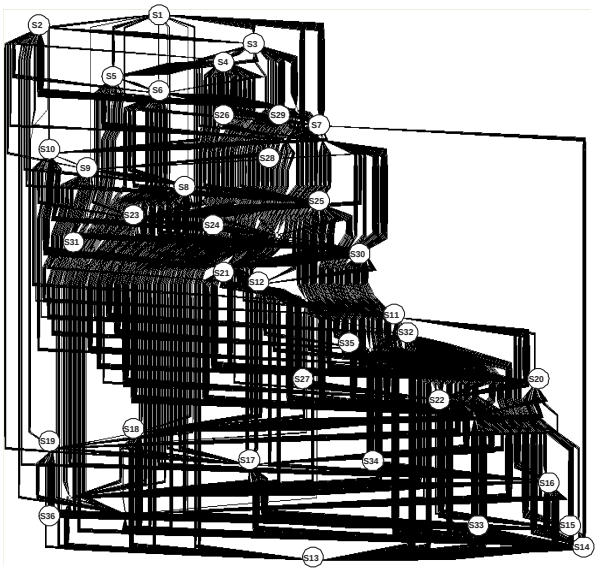

Figure 7.31: A reachability graph generated by SESA in *Step 6* when $Reconfiguration_{2,2,2,2}$ is applied in FESTO and $Reconfiguration_{1,1,0,0}$ is applied in EnAS

7.6.1 Implementation of Reconfiguration Agents

A reconfiguration agent is composed of three modules: Interpreter, Reconfiguration Engine and Converter. The second is the main module to decide automatic reconfigurations at run-time, whereas the rest is used for external interactions (Figure 7.32).

7.6.2 Agent Interpreter

To guarantee a high reactivity of the agent, *Interpreter* ensures by sensors the detection of hardware problems that disturb any normal execution of the system, and ensures also the evaluation of production parameters for any optimization of performance. *Interpreter* ensures in addition all interactions with the Coordination Agent $CA(\xi(Sys))$.

Running Example. *In the Benchmark Production System EnAS, Interpreter should detect by sensors any hardware problem in the plant to apply thereafter automatic reconfigurations that guarantee safe behaviors of the whole system. In addition, it should count the number of pieces nb_pieces and tins (as well as caps) nb_tins + caps per hour. We present in Figure 7.33 an IEC61499-based implementation of Interpreter where the Function Block FB_Inventory counts such numbers and FB_Evaluate evaluates each hour if the First or the Second Production Policy should be followed (evaluation of nb_pieces / (nb_tins + caps* $\times$ *Threshold)).*

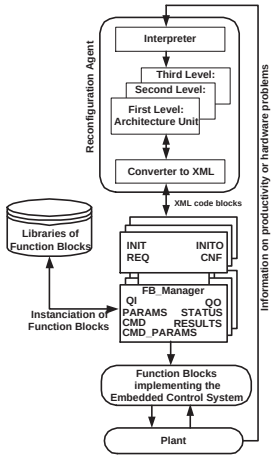

Figure 7.32: Interaction between the Agent and the Embedded Control System.

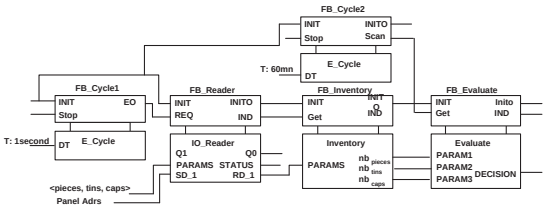

Figure 7.33: Function Blocks-based implementation of the agent interpreter in the EnAS system.

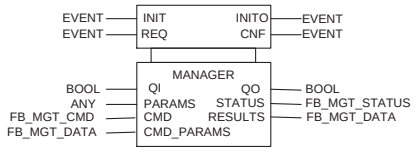

Figure 7.34: Management Function Block".

7.6.3 Reconfiguration Engine

The reconfiguration engine is the agent's mind that receives notifications from *Interpreter* to decide and apply thereafter new automatic reconfigurations of the whole system. It contains the Architecture, Control and Data units (Figure 7.32).

7.6.4 Agent Converter

According to [73], the Standard IEC61499 defines a Standardized Management Function Block *Manager* (Figure 7.32 and Figure 7.34) that offers rich services to Create (load), Initiate, Start, Stop and Delete Function Blocks at run-time (Figure 7.35). This block can be used by specifying different values for the input CMD that allows various service functions which are characterized by the value of the input CMD_PARAMS. The response to each form of service functions is given by the value returned in the output *Status*.

Examples.

- $CMD = CREATE$,
- $CMD\_PARAMS = fb\_instance\_definition$,
- $PARAMS = fb\_instance\_definition$ data,
- $RESULT = fb\_instance\_$ reference.

This operation allows *Manager* to automatically create an instance $fb\_instance\_definition$ in the control system. We note that data types for values (for the inputs $PARAMS$ and $RESULT$) are not defined in IEC61499 and are regarded as implementation specific. However, the standard has defined a data exchange format for porting Function Block definitions based on the XML language which is chosen as a file exchange format for IEC61499 library elements, for example, for exchanging Function Block definitions between engineering support systems and other IEC61499 compliant applications.

To handle interactions between the Reconfiguration Agent and Function Blocks of the system, we use as a technical solution the management Function Block *Manager* that applies desired reconfiguration scenarios. When a scenario is fixed by the engine (i.e. ASM, CSM and DSM state machines) according to environment evolutions and user requirements,

Service Function	Description
Create	Create data type definitions, function block types, instances and connections between function blocks. This will involve downloading definitions from a source, e.g. copying across a network, copying in from a memory smart card.
Initialise	Initialise data type definitions, function block types, instances and connections between function blocks. This concerns setting up function blocks and connections into a runnable state and will include resetting variables to their default initial values.
Start	The Start function triggers the execution of function block networks within a resource. Typically it will start the resource scheduling function and start to run SI function blocks that generate timing events. These in turn trigger chains of events that cause function block execution.
Stop	The Stop service causes all execution to cease by suspending the resource scheduling function.
Delete	The Delete service can be used to delete the definition of any data type, function block or connection.

Figure 7.35: Management Services offered by the Function Block "Manager".

the converter sends this decision in forms of XML code blocks to *Manager* which effectively applies such reconfiguration by exploiting a well-defined library to automatically create, update or delete Function Blocks. The module *Converter* is technically based on a conversion table that defines for each scenario to be fixed by the engine, the XML file that contains the corresponding XML code blocks. In this table, each entry $[< i1, i2 >, < j1, j2 >, < k1, h1, k2, h2 >, < @Manager >, < @XML-File >]$ defines the addresses of *Manager* and the XML file that contains the XML code blocks when the engine reconfigures the system from $Reconfiguration_{i1,j1,k1,h1}$ to the scenario $Reconfiguration_{i2,j2,k2,h2}$ (the ASM state machine evolutes from the state ASM_{i1} to ASM_{i2}, the CSM state machine evolutes from the state $CSM_{i1,j1}$ to $CSM_{i2,j2}$ and the DSM state machine evolutes from $DSM_{k1,h1}$ to $DSM_{k2,h2}$).

Running Example. *In the Benchmark System EnAS, Converter is based on a conversion table containing 18 entries that correspond to different reconfiguration cases to be fixed by the engine. The EnAS agent behaves as follows:*

* ***If*** $nb_{pieces} + nb_{tins+caps} \leq Threshold$ *and no hardware problems occur,*

** ***Then*** *Interpreter signals the Engine to apply the First Production Policy by activating the state* S_{21}*. In this case, Converter sends XML code blocks to the Function Block* Manager *as follows:*

*** Stops the Function Blocks $J2\_FB$ and $G2\_FB$,

*** Deletes from the memory these blocks,

*** Creates the block $G1\_FB$ to implement the First Production Policy,

*** Initializes and Connects this block to the rest of the network of Function Blocks before starting its execution.

** ***Else If*** $nb_{pieces} + nb_{tins+caps} > Threshold$, ***Then*** *Interpreter asks the Engine to apply the Second Production Policy by activating the state* S_{11}*. In this case, Converter sends XML code blocks to the Function Block* Manager *as follows:*

*** Stops the Function Block $G1\_FB$ that controls the Gripper station G_1,

*** Deletes this block from the memory,

*** Creates the blocks $J2\_FB$ and $G2\_FB$ that control respectively J_2 and G_2 to implement the Second Production Policy,

*** Initializes and Connects these blocks to the rest of the network of Function Blocks before starting its execution.

7.6.5 Implementation of The Coordination Agent

We define an implementation for the Coordination Agent $CA(\xi(Sys))$ to be composed of an *Interface* for interactions with Reconfiguration Agents, and an *Engine* which is the mind to decide coherent automatic reconfigurations of the whole distributed embedded control system [60]. The Coordination Matrixes of $\xi(Sys)$ are stored in different files of a database such that each one contains a table in which each line $a \in [1, n]$ defines the following parameters: $< @ - Agent_a >, < i_a >, < j_a >, < k_a >, < h_a >$ that respectively define the address of the agent $Agent_a$ and the reconfiguration scenario $Reconfiguration_{i_a,j_a,k_a,h_a}$ to be automatically applied in the corresponding device. The Engine receives notifications for possible reconfigurations from distributed agents, before decides which scenario corresponding to a well-defined Coordination Matrix should be applied. It interacts thereafter with all concerned devices of the distributed architecture to execute such scenario. We implement the Engine by the following algorithm.

Algorithm. Repeat indefinitely,

- **For any** $Reconfiguration_{i_a,j_a,k_a,h_a}$ received from $Agent_a$ $(a \in [1, n])$
  - $CM \longleftarrow Matrix(Reconfiguration_{i_a,j_a,k_a,h_a})$,
  - **If** $level(CM) = max$, /* CM has the highest priority in $Concur(CM) \cup \{CM\}$*/
    - **Then**, $Tab \longleftarrow file(CM)$, /* *file(CM)* a file presenting all desired reconfigurations by CM*/
    - **For each** $b \in [1, n] \setminus \{a\}$,
      - **Send** XML-bloc $< Tab[b, 1] >, < CA >, < Tab[b, 2] >, < Tab[b, 3] >, < Tab[b, 4] >, < Tab[b, 5] >$ to $Tab[b, 1]$, /*$< Tab[b, 1] >$ defines the address of $Agent_b$*/
    - **If** $\nexists b \in [1, n] \setminus \{a\}$ such that $CA(\xi(Sys))$ receives: $< not - possible >, < Tab[b, 1] >, < Tab[b, 2] >, < Tab[b, 3] >, < Tab[b, 4] >, < Tab[b, 5] >$,
      - **Then, For each** $b \in [1, n]$, **Send** XML-bloc $< apply >, < Tab[b, 1] >, < CA >, < Tab[b, 2] >, < Tab[b, 3] >, < Tab[b, 4] >, < Tab[b, 5] >$ to $Tab[b, 1]$, /* in this case, automatic reconfigurations are applied in distributed devices*/
    - **Else Send** XML-bloc $< reject >, < Tab[a, 1] >, < CA >, < Tab[a, 2] >, < Tab[a, 3] >, < Tab[a, 4] >, < Tab[a, 5] >$ to $Tab[a, 1]$, /*the Coordination Agent rejects the request of $Agent_a$*/

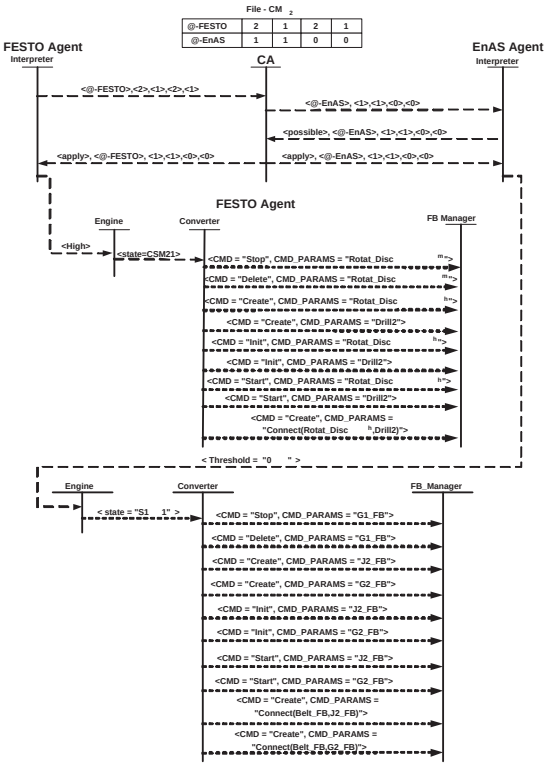

Figure 7.36: Interactions between FESTO and EnAS agents

End.

Running Example. *In the systems FESTO and EnAS, we present in Figure 7.36 the exchanged XML blocks between agents to apply in the first platform the High Production Policy and in the second the Second Production Mode according to the Coordination Matrix* CM_2. *The FESTO agent sends to* $CA(\xi(Sys))$ *a request to improve the productivity by applying* $Reconfiguration_{2,1,2,1}$. *The Coordination Agent sends a request to the EnAS agent that checks if the Second Production Policy is applicable. In this case, an acceptation is sent to* $CA(\xi(Sys))$ *that confirms both automatic reconfigurations in these platforms. In this case, the Function Block* $Rotat\_disc^h$ *is loaded in memory to apply the High Production Policy. The block* $Drill2$ *is loaded in addition to allow drilling of pieces with* $Drill1$. *On the other hand, the blocks* $J2\_FB$ *and* $G2\_FB$ *are loaded in EnAS to apply the Second Production Policy (Figure 7.37).*

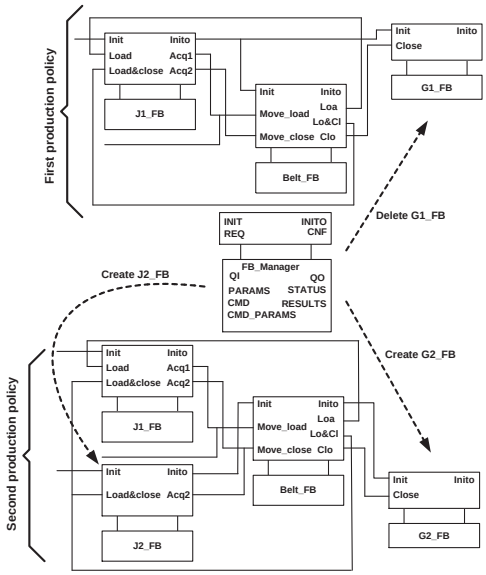

Figure 7.37: The Manager behavior when the EnAS productivity has to be improved

7.7 Conclusion

We define in this chapter a multi-agent architecture for distributed reconfigurable embedded control systems. We assume a Reconfiguration Agent for each device of the execution environment to handle local automatic reconfigurations according to conditions in user requirements, and propose in addition a Coordination Agent for useful coordination between devices when distributed reconfiguration scenarios should be applied. An inter-agents communication protocol is defined to manage this coordination. We propose NCES-based models for both kinds of agents to check the feasibility of the system when reconfigurations should be applied at run-time. Nevertheless, the functional correctness of each possible network of Control Components should be validated for each scenario. We propose therefore a refinement-based approach that generates Control Components step by step and checks their feasibility. The tool *"ProtocolReconf"* is develop to simulate the whole defined multi-agent architecture, and a prototype is implemented to apply the refinement-based approach. If Control Components are assumed to be IEC61499 Function Blocks, then we propose an XML-based implementation of Reconfiguration and Coordination Agents to handle dynamic reconfiguration scenarios of the embedded control system.

7.8 References of the Chapter's Contributions

- M. Khalgui, O. Mosbahi, **Intelligent distributed control systems**, International Journal on Information and Software Technology (accepted for publication), 2010,
- M. Khalgui, **Distributed Reconfigurations of Autonomous IEC61499 Systems**, accepted for publication in ACM Transactions in Embedded Computing Systems, 2011,
- M. Khalgui, O. Mosbahi, **Formal Approach for the Development of Intelligent Industrial Control Components**, International Journal of Modelling, Identification and Control (IJMIC), in print, to be published in 2011,
- M. Khalgui, O. Mosbahi, Z. Li, **Run-Time Reconfigurations of Embedded Controllers**, accepted for publication in ACM Transactions in Embedded Computing Systems, 2011,
- M. Khalgui, H-M Hanisch, **Reconfiguration of Distributed Embedded Control Systems**, IEEE Transactions on Systems, Machine and Cybernetics, Part A, (**Accepted** to be published in 2011),
- M. Khalgui, O. Mosbahi, Zhiwu Li, H-M. Hanisch, **Reconfiguration of Distributed Embedded-Control Systems**, IEEE/ASME Transactions on Mechatronics, 2010,
- M. Khalgui, O. Mosbahi, Z. Li, H-M. Hanisch, **Development of Agent-based Reconfigurable Embedded Control Systems: From Modelling to Implementation**, IEEE Transactions on Computers, (**Accepted** to be published in 2011),
- M. Khalgui, **NCES-based Modelling and CTL-based Verification of Reconfigurable Embedded Systems In Manufacturing Industry**, International Journal of Computers in Industry, accepted, to appear in 2010,
- M. Khalgui, H-M Hanisch, **Automatic NCES-based Specification and SESA-based Verification of Feasible Control Tasks in Benchmark Production Systems**, International Journal of Modelling, Identification and Control, accepted to appear in 2010,

Chapter 8

Feasible Execution Models for Reconfigurable Real-Time Embedded Control Systems

8.1 Introduction

The chapter deals with execution models of reconfigurable real-time embedded control systems to be implemented by different networks of components such that each network should be executed when a corresponding reconfiguration scenario is automatically applied at runtime. We assume that the system's components are located in different containers sharing controls of same physical processes. We mean by execution models the different possible real-time OS tasks that implement the reconfigurable system's behavior in the operational level. We transform at first time each network of components to a system of actions with precedence constraints in order to base our work on previous solid theories on real-time scheduling [26]. Therefore, the event-triggered model of the network is transformed to a time-triggered one. We propose an approach based on the non-preemptive scheduling policy "Earliest Deadline First" (denoted by EDF) to verify temporal bounds of components located in each container. If it is feasible, then we generate a pre-scheduling which is a Direct Acyclic Graph (DAG) to define the corresponding execution sequencing function of the container. If all containers are feasible, then we transform the corresponding pre-schedulings to recurring OS tasks [14]. To meet temporal bounds, we analyze in addition the on-line preemptive schedulability of these tasks because it is often required as denoted in [115] to apply an on-line preemptive policy for scheduling of OS tasks. Thanks to this approach, the network of components corresponding to each reconfiguration scenario is transformed to feasible OS tasks. In addition, the reconfigurable system is considered as sets of OS tasks where each set is loaded in memory when the corresponding reconfiguration scenario is applied (i.e. a particular network of components). We present in Figure 8.1 the different steps to be applied for the feasible assignment of reconfigurable components to OS tasks of

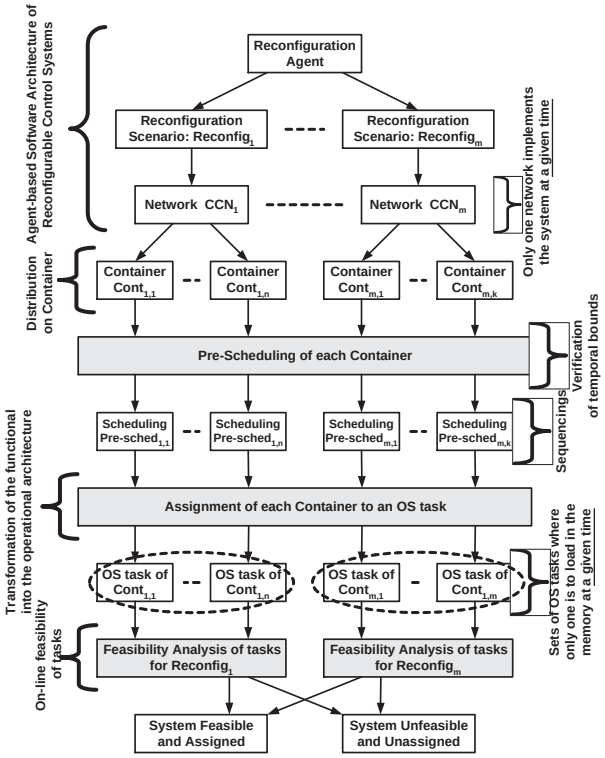

Figure 8.1: Verification and Assignment of Reconfigurable Control Components to sets of Feasible OS Tasks

the execution environment [57].

8.2 State of the Art

Nowadays, many research works on assignments of components have been proposed. In [52, 119], an approach is proposed to assign all the system's components to only one execution thread. This sequential single-threaded approach seems to be inefficient for complex systems as explained in [36, 79]. In [118], another solution is proposed by assigning each component to only one thread. Nevertheless, this approach is limited because it deals with devices containing small numbers of components. In [35], an interesting solution is proposed by assigning subsets of application's components to threads. Each subset is defined as a container handling the execution of the corresponding components. Although this approach seems to be efficient and flexible, the assignment does not take into account temporal constraints to strictly satisfy by the system's components [117].

8.3 Temporal Properties of Reconfigurable Embedded Control Systems

According to user requirements, we define in this section temporal properties to be satisfied by the components of a reconfigurable control system. We introduce the function *cause* to capture end-to-end bounds on $Network_{i,j,k,h}$ that implements the system when the scenario $Reconfiguration_{i,j,k,h}$ is applied. This function specifies a causality between an event input of a component and the corresponding output of another one. We define in addition the set $inputs_{i,j,k,h}$ ($outputs_{i,j,k,h}$, resp) of input (output, resp) events of $Network_{i,j,k,h}$ with no predecessors (successors, resp) in the system (i.e. global input and output events). In this chapter, we assume periodic events of $inputs_{i,j,k,h}$. We classically characterize each event $ie_{i,j,k,h} \in inputs_{i,j,k,h}$ by a release time $r_{i,j,k,h}$, a period $p_{i,j,k,h}$ and a constant deadline $d_{i,j,k,h}$ [112]. We *carefully precise* that the same event can be characterized by different temporal parameters for different reconfiguration scenarios. We denote by $bound\_event(ie_{i,j,k,h}, oe_{i,j,k,h})$ the end-to-end bound to be satisfied between the receive of a global input event $ie_{i,j,k,h}$ ($ie_{i,j,k,h} \in inputs_{i,j,k,h}$) and the sent of a global output event $oe_{i,j,k,h}$ ($oe \in outputs_{i,j,k,h}$).

Running example. *In the system FESTO, we assume in Figure 8.2 that the Function Blocks are distributed on two resources (analogue of containers in the IEC61499 terminology). We assume also the following end-to-end bounds for the different reconfiguration scenarios.*

- **High production.** *The drilling of two pieces has to satisfy:*

$$bound(Send_{2,1,2,1}, Evo_{2,1,2,1}) = 45$$

- **Medium production.** *The drilling of one piece has to satisfy:*

$$bound(Send_{2,2,2,2}, Evo_{2,2,2,2}) = 30$$

- **Light production.** *The drilling of one piece by Drill_machine1 has to satisfy:*

$$bound(Send_{1,1,1,1}, Evo_{1,1,1,1}) = 35$$

8.4 Contribution: Formalization of Reconfigurable Embedded Control Systems

We formalize a reconfigurable control system for verifications of temporal properties. We transform each network of components (corresponding to a reconfiguration scenario) to a

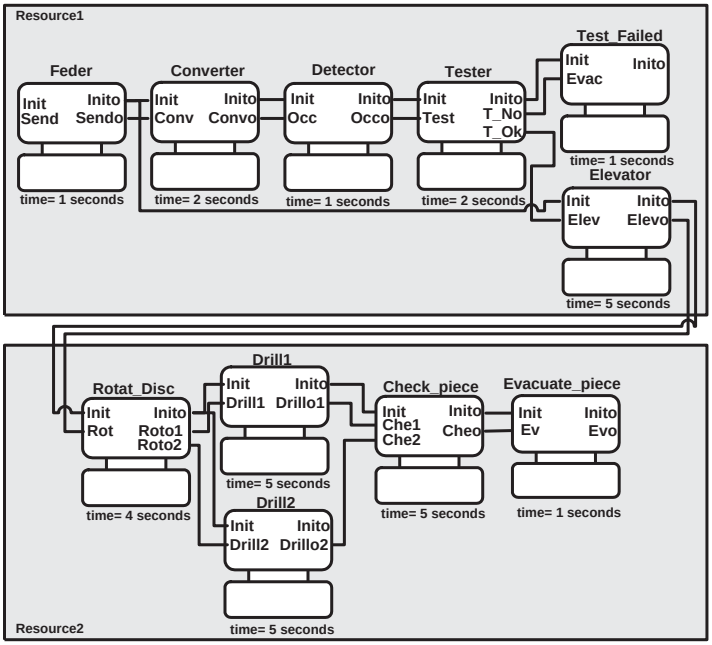

Figure 8.2: Assumed distribution of FESTO's blocks on two resources

system of actions with precedence constraints in order to exploit previous results on real-time scheduling [112, 14]. In a such network, we define an action as the execution of a component when a well-defined input event occurs. Moreover, we define a trace as a sequence of actions under causality relation in the network.

8.4.1 Action of Control Component

We define an action $act_{i,j,k,h}$ as the execution of a component belonging to $Network_{i,j,k,h}$ (execution of one or several algorithms of the component) when a particular input event $ie_{i,j,k,h}$ occurs. We denote in the following by $\sigma_{i,j,k,h}$ (resp, $\sigma^{Cont}_{i,j,k,h}$) the set of the system's actions (resp, the set of actions in a container $Cont$) where we characterize $act_{i,j,k,h} \in \sigma_{i,j,k,h}$ as follows:

- $WCET(act_{i,j,k,h})$ (resp, $BCET(act_{i,j,k,h})$) : the Worst (resp, Best) Case Execution Time of the sequence of algorithms corresponding to $ie_{i,j,k,h}$.
- $pred(act_{i,j,k,h})$: the set of actions to execute in the system before the $act_{i,j,k,h}$ execution. These actions belong to components producing the output events linked to $ie_{i,j,k,h}$.

- $succ(act_{i,j,k,h})$: a set of actions sets. Each actions set corresponds to a possible execution scenario (ie. only one set of actions set is performed at a time). The actions of a set have to be executed once the execution of $act_{i,j,k,h}$ is finished. These actions belong to components activated once the treatment corresponding to $ie_{i,j,k,h}$ finishes.
- $(r_{i,j,k,h}, p_{i,j,k,h}, d_{i,j,k,h})$: the two first parameters characterize the activation of $act_{i,j,k,h}$ [112]. They should be processed while taking into account the execution of $pred(act_{i,j,k,h})$. The deadline $d_{i,j,k,h}$ defines the latest completion date of the execution. To respect temporal bounds, it should be processed while taking into account the deadlines of the $act_{i,j,k,h}$ successors.

Let $first(\sigma_{i,j,k,h})$ (resp, $last(\sigma_{i,j,k,h})$) be a subset of $\sigma_{i,j,k,h}$ where each action is with no predecessors (resp successors) in $\sigma_{i,j,k,h}$. In particular, we denote by $first(\sigma^{Cont}_{i,j,k,h})$ (resp, $last(\sigma^{Cont}_{i,j,k,h})$) a subset of $\sigma^{Cont}_{i,j,k,h}$ such that each action is with no predecessors (resp successors) in $\sigma^{Cont}_{i,j,k,h}$ and we denote in addition by $act^{p,q}_{i,j,k,h}$ the $q-th$ instance of the action $act^{p}_{i,j,k,h}$ belonging to the network of components $Network_{i,j,k,h}$.

8.4.2 Trace of Control Component

By considering precedence constraints between actions in $Network_{i,j,k,h}$, we define a trace $tr_{i,j,k,h}$ of $\sigma^{Cont}_{i,j,k,h}$ the following sequence,

$$tr_{i,j,k,h} = act^{0}_{i,j,k,h}, ..., act^{n-1}_{i,j,k,h} \text{ where,}$$

- $\forall p \in [1, n-1], act^{p-1}_{i,j,k,h} = pred(act^{p}_{i,j,k,h})$
- $act^{0}_{i,j,k,h} \in first(\sigma^{Cont}_{i,j,k,h})$ and $act^{n-1}_{i,j,k,h} \in last(\sigma^{Cont}_{i,j,k,h})$

The execution of the trace $tr_{i,j,k,h}$ has to satisfy an end-to-end response time bound $eertb(tr_{i,j,k,h})$ according to user requirements. In this chapter, we consider non reentry traces [68] : *"the execution of the $z-th$ instance of the trace must not start before the execution end of the $(z-1)-th$ one"*. Therefore, the period of $act^{0}_{i,j,k,h}$ is greater than $eertb(tr_{i,j,k,h})$. To satisfy all the considered bounds in requirements, we propose in [64] a technique to process deadlines for the different actions of traces in $Network_{i,j,k,h}$.

8.5 Contribution: Verification and Assignment of Control Components to OS Tasks

We verify temporal properties of $Network_{i,j,k,h}$ before assign the corresponding components to feasible OS tasks of the execution environment in order to translate the functional to the operational architecture of this network [54]. We assume in the following that $Network_{i,j,k,h}$

is distributed on a set of containers (denoted by *containers*) located in a device. To verify and assign the network, we analyze the schedulability of Control Components in each container $Cont \in containers$ by constructing a reachability graph to verify temporal properties (end-to-end bounds). If it is feasible, then we generate a pre-scheduling that defines the sequencing of components in $Cont$, and if all containers are feasible, then we transform the corresponding pre-schedulings into OS tasks (i.e. one container is assigned into one OS task). To complete the temporal verification of bounds, we check the on-line preemptive feasibility of these tasks by applying a schedulability analysis. As denoted in [115], it is often required to apply an on-line preemptive policy to schedule OS tasks in devices. This methodology of verification and assignment has different advantages: it reduces the number of tasks to schedule by regrouping components of a container in a single task. This advantage is required by several Real Time Operating Systems which restrict such number [115]. Thanks to this regrouping, the complexity of the schedulability analysis of OS tasks is controlled and the context switching is minimized at run-time [65].

8.5.1 Verification and Pre-scheduling of a Container

To verify temporal bounds of $Network_{i,j,k,h}$ that contain components located in a container $Cont$ ($Cont \in containers$), we construct in a well-defined Hyper Period a reachability graph by applying the scheduling policy "Earliest Deadline First" [112]. If it is feasible, then we generate a pre-scheduling as a Direct Acyclic Graph.

- **Hyper Period for a Reconfiguration Scenario**

  Let $lcm_{i,j,k,h}$ be the least common multiple of periods of actions in $first(\sigma^{Cont}_{i,j,k,h})$. Let $act^{max}_{i,j,k,h} = \{r_{max}, p_{max}, d_{max}\}$ and $act^{min}_{i,j,k,h} = \{r_{min}, p_{min}, d_{min}\}$ be two actions of $first(\sigma^{Cont}_{i,j,k,h})$ such that,

  $$\forall act^{a}_{i,j,k,h} \in first(\sigma^{Cont}_{i,j,k,h}),\ r_{min} \leq r_a \leq r_{max}$$

  By exploiting a previous result on the hyper period for asynchronous systems [71], the schedulability analysis should be done in $HP^{Cont}_{i,j,k,h} = [r_{min}; r_{max} + 2.lcm_{i,j,k,h}]$.

  Running example. *In the Benchmark Production System FESTO, the hyper period for Resource*1 *is calculated for all possible reconfiguration scenarios. In particular:*

  - $HP^{Resource1}_{2,1,2,1} = [0, 22]$ *for the High Production Mode,*
  - $HP^{Resource1}_{2,2,2,2} = [0, 60]$ *for the Medium Production Mode,*
  - $HP^{Resource1}_{1,1,1,1} = [0, 100]$ *for the Light Production Mode,*

- **Reachability Graph of a Container for a Reconfiguration Scenario**

  We propose a technique for the generation of the reachability graph $G^{Cont}_{i,j,k,h}$ that presents all possible execution scenarios of components (of $Network_{i,j,k,h}$) located in

the container $Cont$ when $Reconfiguration_{i,j,k,h}$ is applied. Each trajectory of this graph presents a scheduling of system's traces in $Cont$. A state $C_{i,j,k,h}$ of a trajectory contains a selected instance of action to execute among all active ones. We apply the static Earliest Deadline First policy (denoted by EDF) [112] to perform the selection. A state $C_{i,j,k,h}$ of $G^{Cont}_{i,j,k,h}$ is characterized as follows,

$$C_{i,j,k,h} = \{S_{i,j,k,h},\ act^{m,n}_{i,j,k,h},\ t_{i,j,k,h}\} \text{ where,}$$

- $S_{i,j,k,h}$: a set of instances of actions to execute,
- $act^{m,n}_{i,j,k,h}$: the selected instance among all active ones of $S_{i,j,k,h}$ according to the EDF policy,
- $t_{i,j,k,h}$: the start time of the $act^{m,n}_{i,j,k,h}$ execution,

The first state of the graph is denoted by $C^0_{i,j,k,h} = \{S^0_{i,j,k,h},\ act^{min,1}_{i,j,k,h}, t^0_{i,j,k,h}\}$ where $S^0_{i,j,k,h}$ contains instances of actions belonging to $first(\sigma^{Cont}_{i,j,k,h})$. To construct the reachability graph, the following rules are applied recursively for each state $C_{i,j,k,h} = \{S_{i,j,k,h}, act^{m,n}_{i,j,k,h}, t_{i,j,k,h}\}$ without a successor :

Construction Of $G^{Cont}_{i,j,k,h}$

- **Rule 0 : Stop Condition.**
  If $t_{i,j,k,h} > r_{max} + 2.lcm_{i,j,k,h}$, **Then** we reach the limit of $HP^{Cont}_{i,j,k,h}$ and the construction of the current trajectory is stopped,
- **Rule 1 : Verification of Constraints.**
  If there exists an instance $act^{p,q}_{i,j,k,h} \in S_{i,j,k,h}$ missing its deadline, then the network of components is unfeasible in the container $Cont$. Otherwise, an instance $act^{m,n}_{i,j,k,h}$ ($act^{m}_{i,j,k,h}$ belongs to $tr_{i,j,k,h}$ in the container $Cont$) is selected by applying the static EDF policy,
- **Rule 2 : Construction of New States.**
  For each set of $succ(act^{m,n}_{i,j,k,h})$, a new state of $G^{Cont}_{i,j,k,h}$ has to be constructed. If $act^{m}_{i,j,k,h}$ belongs to $last(tr_{i,j,k,h})$, then a new instance of $tr_{i,j,k,h}$ is started,

Running example. *In the system FESTO, we construct the reachability graph for each resource and for each reconfiguration scenario. In Figure 8.3, we present reachability graphs of Resource*2 *for the high and the medium production modes. These graphs are constructed respectively in* $HP^{Resource2}_{2,1,2,1}$ *and* $HP^{Resource2}_{2,2,2,2}$*. In the first mode, we have to execute the two actions Drill*1 *and Drill*2 *to drill two pieces in Drill_machine*1 *and Drill_machine*2*, whereas in the second mode we execute only one of the two actions. Finally, we note that initialization actions are not considered in these graphs.*

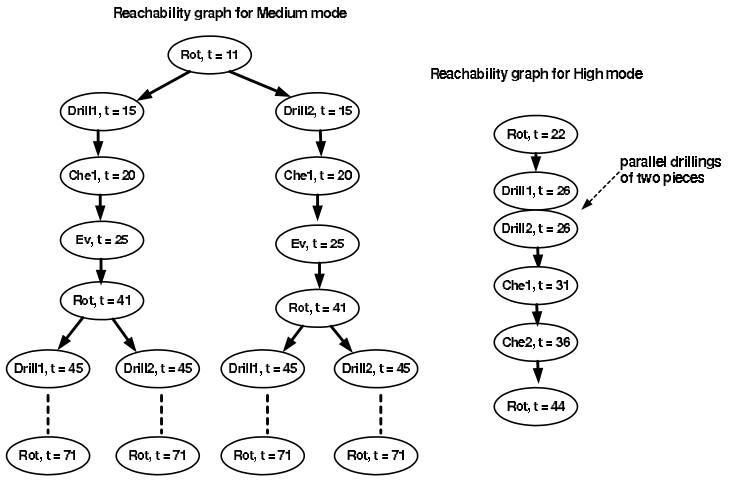

Figure 8.3: Reachability graphs of $Container2$ for the high and medium production modes.

- **Generation Of a pre-scheduling for a Reconfiguration Scenario**

  If the reachability graph is correctly constructed in the hyper period $HP^{Cont}_{i,j,k,h}$ (i.e. all temporal properties are satisfied), then a static scheduling $Stat^{Cont}_{i,j,k,h}$ is generated to be used by the OS when the current reconfiguration scenario $Reconfiguration_{i,j,k,h}$ is applied at run-time. This scheduling is a DAG where each trajectory specifies a possible execution of the system. A state of the graph specifies the execution start time of an action's instance which is selected in the corresponding node of the reachability graph.

  Running example. *In the system FESTO, we generate a pre-scheduling from the reachability graph that we construct for each resource and for each particular reconfiguration scenario. We show in Figure 8.4 pre-scheduling portions of* $Resource1$ *and* $Resource2$ *for the Medium production mode. These pre-schedulings are generated in well-defined Hyper Periods according to the followed production mode.*

- **Assignment of Control Components to OS Tasks**

  We propose a technique to assign the pre-scheduling $Stat^{Cont}_{i,j,k,h}$ defining the sequencing of Control Components in $Cont$ (when the reconfiguration scenario $Reconfiguration_{i,j,k,h}$ is applied) to an OS task. By considering the conditional structure of $Stat^{Cont}_{i,j,k,h}$, we use the recurring task model to apply the assignment [14]. This model was introduced to represent conditional real-time codes.

  Recurring task : *a recurring task* Γ *is characterized by a task graph* $G(\Gamma)$ *and a period* $P(\Gamma)$. *The task graph* $G(\Gamma)$ *is a Direct Acyclic Graph (DAG) with a unique*

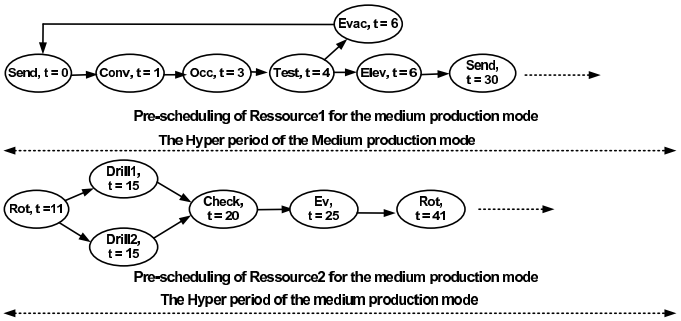

Figure 8.4: A pre-scheduling portion of *container*1 and *container*2 for the Medium production mode.

source vertex (denoted by τ^0*) and a unique sink vertex. Each vertex of this DAG represents a subtask (denoted by* τ*) and each edge represents a possible flow of control. A vertex of* Γ *is characterized by a* WCET *and a deadline* d*. In addition, an edge (*τ*,* τ'*) is characterized by a real number p(*τ*,* τ'*) denoting the minimum amount of time that must elapse after vertex* τ *is triggered (t(*τ*)) and before vertex* τ' *can be triggered (*t*(*τ'*)) [14].*

We encode the graph structure with the set $pred(\tau)$ (resp, $succ(\tau)$) that defines subtasks of Γ such that only one has to be executed before (resp, after) τ. By considering two behavioral modes in the hyper period $HP^{Cont}_{i,j,k,h}$, the pre-scheduling $Stat^{Cont}_{i,j,k,h}$ should be transformed to two recurring tasks $\Gamma^{Cont}_{i,j,k,h}$ and $\Gamma'^{Cont}_{i,j,k,h}$. The task $\Gamma^{Cont}_{i,j,k,h}$ implements the stationary behavior (in $[r_{min}, r_{max}]$) whereas the task $\Gamma'^{Cont}_{i,j,k,h}$ implements the non-stationary one (in $[r_{max}, r_{max} + 2.lcm]$). Note that $\Gamma^{Cont}_{i,j,k,h}$ is periodic with the same period of the stationary mode.

A straightforward transformation consists in associating each subtask to an action's instance. Nevertheless, this transformation produces recurring tasks with several subtasks. This transformation increases the complexity of the schedulability analysis [14]. To control this complexity, we merge as a solution a sequence of instances of actions into a unique subtask. To verify all bounds during the feasibility analysis of these OS tasks, an instance $act^{a,b}_{i,j,k,h} \in Stat^{Cont}_{i,j,k,h}$ such that $act^{a}_{i,j,k,h} \in \text{last}(\sigma^{Cont}_{i,j,k,h})$ must be the last instance of a subtask $\tau_{i,j,k,h}$. According to the EDF policy, the deadline of $\tau_{i,j,k,h}$ is then the deadline of $act^{a,b}_{i,j,k,h}$.

Notations. In the following, $stat\_succ(act^{a,b}_{i,j,k,h})$ (resp $stat\_pred(act^{a,b}_{i,j,k,h})$) denotes the set of instances following (resp preceding) the instance $act^{a,b}_{i,j,k,h}$ in $Stat^{Cont}_{i,j,k,h}$.

A subtask $\tau_{i,j,k,h}$ of $\Gamma^{Cont}_{i,j,k,h}$ has to be implemented as follows,

$$\tau_{i,j,k,h} = act^{0,c}_{i,j,k,h}\ ,\ act^{1,g}_{i,j,k,h}\text{.....},\ act^{d-1,f}_{i,j,k,h} \text{ such that,}$$

- $\forall\ y \in [0, d-2]$, $stat\_succ(act^{y,p}_{i,j,k,h}) = \{act^{y+1,q}_{i,j,k,h}\}$: the sequence implementing $\tau_{i,j,k,h}$ is a sequence of instances in $Stat^{Cont}_{i,j,k,h}$,
- $act^{d-1}_{i,j,k,h}$ is an action without successors in $\sigma^{Cont}_{i,j,k,h}$ or $act^{d-1,f}_{i,j,k,h}$ has more than one successor in $Stat^{Cont}_{i,j,k,h}$,

$$act^{d-1}_{i,j,k,h} \in \text{last}(\sigma^{Cont}_{i,j,k,h}) \text{ or cardinality}(stat\_succ(act^{d-1,f}_{i,j,k,h})) > 1$$

Let $first(\tau_{i,j,k,h})$ (resp $last(\tau_{i,j,k,h})$) be the first (resp last) instance of the subtask $\tau_{i,j,k,h}$. Moreover, let $first\_stat(Stat^{Cont}_{i,j,k,h})$ be the set of instances in $Stat^{Cont}_{i,j,k,h}$ with no predecessors to execute in the stationary mode. We apply the following rules to construct the task $\Gamma^{Cont}_{i,j,k,h}$. The first rule constructs the first subtask $\tau^0_{i,j,k,h}$ of the recurring task $\Gamma^{Cont}_{i,j,k,h}$, whereas the second one is applied recursively to construct the other subtasks,

Rule 0. Construction of the First Subtask $\tau^0_{i,j,k,h}$ in $\Gamma^{Cont}_{i,j,k,h}$.

If $cardinality(first\_stat(Stat^{Cont}_{i,j,k,h})) = 1$, **Then**, $\{\tau^0_{i,j,k,h}\} = first\_stat(Stat^{Cont}_{i,j,k,h})$

Otherwise, a virtual subtask $\tau^0_{i,j,k,h}$ in $\text{G}(\Gamma^{Cont}_{i,j,k,h})$ is constructed as follows:

- $WCET(\tau^0_{i,j,k,h}) = 0$,
- For each state $act^{a,b}_{i,j,k,h} \in first\_stat(Stat^{Cont}_{i,j,k,h})$, a subtask $\tau^z_{i,j,k,h}$ such that $(\tau^0_{i,j,k,h}, \tau^z_{i,j,k,h}) \in \text{G}(\Gamma^{Cont}_{i,j,k,h})$ and $p(\tau^0_{i,j,k,h}, \tau^z_{i,j,k,h}) = 0$ is constructed,

The triggering time of the subtask $\tau^0_{i,j,k,h}$ is equal to the minimum of the execution start times of instances in $first\_stat(Stat^{Cont}_{i,j,k,h})$:

$$\text{t}(\tau^0_{i,j,k,h}) = \min\{\text{t}(act^{a,b}_{i,j,k,h}), act^{a,b}_{i,j,k,h} \in first\_stat(Stat^{Cont}_{i,j,k,h})\}.$$

Rule 1. Construction of Subtasks.

Let $\tau^a_{i,j,k,h}$ be a subtask of $\Gamma^{Cont}_{i,j,k,h}$ such that $\text{last}(\tau^a_{i,j,k,h})$ has a successor in $Stat^{Cont}_{i,j,k,h}$,

$$\exists\ act^{0,q}_{i,j,k,h} \in Stat^{Cont}_{i,j,k,h},\ act^{0,q}_{i,j,k,h} \in stat\_succ(\text{last}(\tau^a_{i,j,k,h}))$$

Let $\tau^b_{i,j,k,h}$ be the successor of $\tau^a_{i,j,k,h}$ in $\Gamma^{Cont}_{i,j,k,h}$ $(\tau^b_{i,j,k,h} \in succ(\tau^a_{i,j,k,h}))$,

$$\tau^b_{i,j,k,h} = act^{0,q}_{i,j,k,h},\ act^{1,w}_{i,j,k,h},\ ....,act^{d-1,p}_{i,j,k,h} \text{ such that,}$$

$$act^{d-1,p}_{i,j,k,h} \text{ in } last(\sigma^{Cont}_{i,j,k,h}) \text{ or } cardinality(stat\_succ(act^{d-1,p}_{i,j,k,h})) > 1$$

By considering the recurring task concept as presented in [14], the following temporal constraints for this new subtask are defined,

– **The ready time** $t(\tau^b_{i,j,k,h})$ is equal to the earliest possible execution time of the instance $act^{0,q}_{i,j,k,h}$. This time should take into account the execution of the $act^{0,q}_{i,j,k,h}$ predecessors in $Stat^{Res}_{i,j,k,h}$. We characterize it as follows,

$$t(\tau^b_{i,j,k,h}) = max\{r(act^{0,q}_{i,j,k,h}); \\ max_{\tau^a_{i,j,k,h}=pred(\tau^b_{i,j,k,h})}\{t(\tau^a_{i,j,k,h}) + \sum_{act^{p,w}_{i,j,k,h}\in\tau^a_{i,j,k,h}} BCET(act^p_{i,j,k,h})\}\}$$

– **The minimum amount of time** $p(\tau^a_{i,j,k,h},\tau^b_{i,j,k,h})$ is classically equal to the difference between the triggering times of $\tau^b_{i,j,k,h}$ and $\tau^a_{i,j,k,h}$: $p_b = t(\tau^b_{i,j,k,h}) - t(\tau^a_{i,j,k,h})$,

– **The deadline** d_b, corresponds to the deadline of the last instance $act^{d-1,p}_{i,j,k,h}$ of $\tau^b_{i,j,k,h}$,

– **The execution requirement** $WCET(\tau^b_{i,j,k,h})$ is the sum of the WCETs of the $\tau^b_{i,j,k,h}$ actions,

Finally, we apply the same method to construct the recurring task Γ'^{Cont} implementing the non-stationary behavior of the container *Cont*.

Running example. *In the system FESTO, we assign the Control Components of each container for each particular production mode to an OS task. Figure 8.5 shows the OS tasks* $\Gamma^{Resource1}_{2,2,2,2}$ *and* $\Gamma^{Resource2}_{2,2,2,2}$ *implementing the system in the stationary behavior when the medium production mode is applied by the agent. We show in Figure 8.6 the OS tasks* $\Gamma^{Resource1}_{2,1,2,1}$ *and* $\Gamma^{Resource2}_{2,1,2,1}$ *implementing the system in the stationary behavior when the high production mode is applied.*

8.5.2 Verification of OS tasks for a Reconfiguration Scenario

At this step, the network of Control Components $Network_{i,j,k,h}$ distributed on containers of the set *Containers* is assigned to *independent* OS tasks of the execution environment (each container is assigned to a recurring task). By considering the transformation technique, the precedence constraints between actions of $\sigma^{Cont}_{i,j,k,h}$ are not lost. Indeed, the temporal characterization of tasks preserves such dependencies. We apply the schedulability condition defined in [14] to check the preemptive on-line feasibility of the tasks implementing $Network_{i,j,k,h}$ when $Reconfiguration_{i,j,k,h}$ is applied. This condition should be applied in a fixed hyper-period *hp* as follows,

$$hp_{i,j,k,h} = [0, \frac{\sum_{\Gamma^{Cont}_{i,j,k,h}\in S} 2*E(\Gamma^{Cont}_{i,j,k,h})}{1-\sum_{\Gamma^{Cont}_{i,j,k,h}\in S} \rho_{ave}(\Gamma^{Cont}_{i,j,k,h})}] \text{ where,}$$

- S : the set of the recurring tasks of $Network_{i,j,k,h}$ to validate,
- $E(\Gamma^{Cont}_{i,j,k,h})$: denotes the maximum possible cumulative execution requirement on any path from the source node to the sink node of the task graph $G(\Gamma^{Cont}_{i,j,k,h})$,

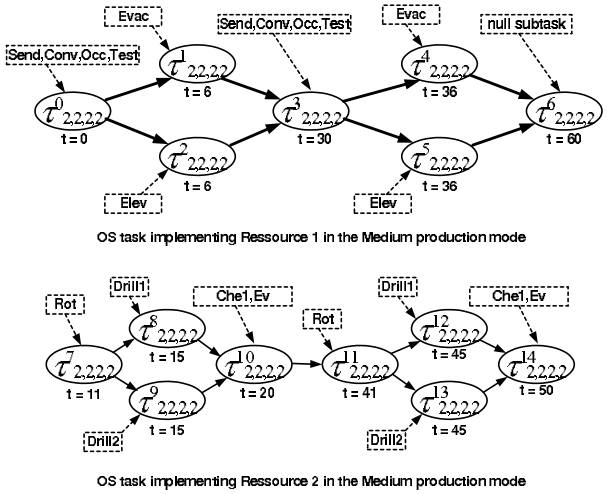

Figure 8.5: The OS tasks implementing the system in the Medium Production Mode.

- $\rho_{ave}(\Gamma^{Cont}_{i,j,k,h})$: denotes the quantity $E(\Gamma^{Cont}_{i,j,k,h})/\mathrm{P}(\Gamma^{Cont}_{i,j,k,h})$,

The schedulability condition indicates that the system is feasible if and only if,

$$\forall t \in hp_{i,j,k,h},\ \sum_{\Gamma^{Cont}_{i,j,k,h} \in S} \Gamma^{Cont}_{i,j,k,h}.dbf(t) \leq t$$

where, $\Gamma^{Cont}_{i,j,k,h}.dbf(t)$ is a function accepting as argument a non negative real number t. This function processes the maximum cumulative execution requirement by jobs of $\Gamma^{Cont}_{i,j,k,h}$ having both ready times and deadlines within any time interval of duration t. Finally, note that [14] proposes an interesting technique to compute this function in the hyper period $hp_{i,j,k,h}$.

8.6 Generalization: Verification and Assignment of Reconfigurable Embedded Control Systems

We generalize in this section the verification of temporal bounds of components encoding the system after different reconfiguration scenarios. We generalize also their assignment to sets of OS tasks. Each set should be load in memory by the agent if the corresponding reconfiguration scenario is applied.

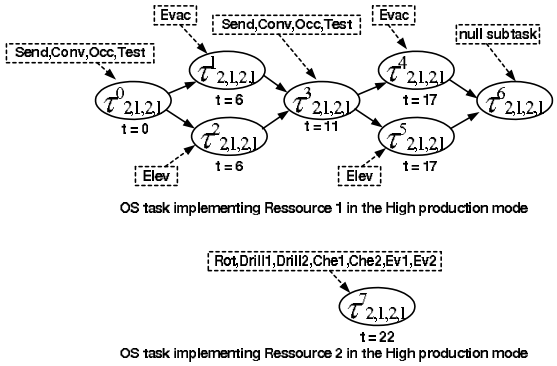

Figure 8.6: The OS tasks implementing the system in the High Production Mode.

8.6.1 Algorithm

To check and assign a reconfigurable system to the execution environment, we should verify end-to-end response time bounds of the network of components $Network_{i,j,k,h}$ corresponding to a particular reconfiguration scenario $Reconfiguration_{i,j,k,h}$ ($i \in [1, n_{ASM}]$, $j \in [1, n_{CSM_i}]$, $k \in [1, n_{DSM}]$ and $h \in [1, n_{DSM_k}]$). This network is distributed on several containers. We have to construct if possible for each one of them a pre-scheduling. If it is feasible, we transform this pre-scheduling to an OS task that we have to check its on-line feasibility. Finally, if all the networks of components corresponding to the different reconfiguration scenarios are feasible and correctly assigned, then the whole reconfigurable system is feasible and correctly assigned.

Algorithm. Verification and Assignment
For each Architecture $ASM_i, i \in [1, n_{ASM}]$,
For each Control policy $CSM_{i,j}, j \in [1, n_{CSM_i}]$,
For each Data configuration $DSM_{k,h}, h \in [1, n_{DSM_k}]$ such that $DSM_k = Data(CSM_{i,j})$,
Construction Of $G^{Res}_{i,j,k,h}$ **for each** $Cont \in containers$,
If a deadline is violated in $Network_{i,j,k,h}$,
Then Display(System is unfeasible),
Else Generation Of a Pre-scheduling for each $Cont \in containers$,
for each $Cont \in containers$ **Assignment of** $Cont$ **to an OS task**,
If all constructed OS tasks meet their deadlines,
Then Display($Network_{i,j,k,h}$ is correctly checked and assigned to the execution environment),
Else System is unfeasible,
Display(Reconfigurable System is correctly checked and assigned),
End Algorithm.

---

we compute the algorithm's complexity by defining n as an upper bound of n_{ASM}, $\max\{n_{CSM_i},\ i \in [1, n_{ASM}]\}$ and $\max\{n_{DSM_k}, k \in [1, n_{DSM}]\}$. The complexity of the algorithm is then in $O(n^3)$. In the research laboratory of automation technology at Martin Luther University, we developed the tool $X - Assign$ that supports this algorithm for the verification and assignment of reconfigurable systems. In addition to the benchmark system FESTO, we applied this tool to the EnAS demonstrator.

8.6.2 Discussion

The approach that we propose to assign networks of components to sets of OS tasks has advantages. As required by several RTOS, it reduces the number of OS tasks implementing the system. In the system FESTO, two tasks (corresponding to $Resource1$ and $Resource2$) are enough to support Distribution, Test and Processing Units. As shown in Figure 8.7, this approach reduces also the context switching between tasks at run-time as follows:

- **Case1. High production mode. if** we consider each component as an OS task (i.e. the assumption proposed in [118]), **then** 22 context switchings are applied in the Hyper period. Thanks to our approach where only two tasks implement the whole system, only 2 context switchings are applied between them.
- **Case2. Medium and Light production modes.** 18 context switchings are applied in the Hyper period if we consider each component as an OS task [118]. In our approach, only 4 context switchings are applied between the two tasks implementing the system.

8.7 conclusion

The chapter proposes a methodology to check and assign Control Components of reconfigurable embedded control systems to feasible OS tasks of the execution environment. These components are assumed to be distributed on several containers and should meet corresponding real-time constraints. The system is implemented by different networks of components such that each one is loaded in memory when the corresponding reconfiguration scenario is applied at run-time. We propose a technical solution that generates a pre-scheduling for each container and for each network of components. This pre-scheduling is assigned thereafter to an OS task that we should check its feasibility. The system is implemented therefore by different sets of OS tasks such that each set should be executed when the corresponding reconfiguration scenario is applied.

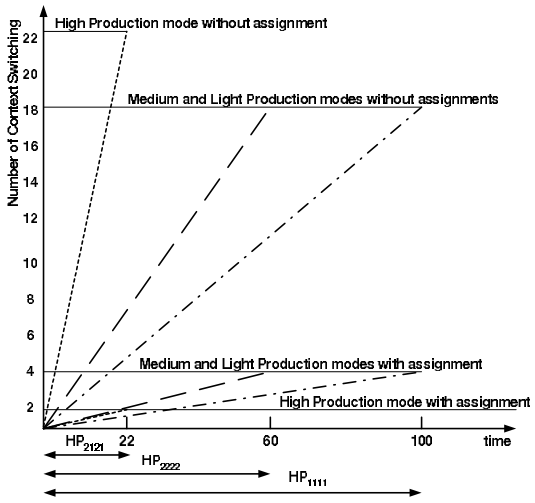

Figure 8.7: Context switchings for the different production modes of the FESTO Manufacturing System.

8.8 References of the Chapter's Contributions

- M. Khalgui, H-M. Hanisch, **Reconfiguration of Industrial Embedded Systems**, Industrial Information Technology Handbook, Editor : Luis Gomes, CRC Press, 2009,
- M. Khalgui, **A Deployment Methodology of Real-time Industrial Control Applications in Distributed Controllers**, International Journal of Computers in Industry. Vol59, N.5, 2008,
- M. Khalgui, H-M Hanisch, **A formal Approach to Check and Assign Reconfigurable Control Applications into Programmable Logic Controllers**, Asian Journal of Control. Vol.11, N.3, 2009,

Chapter 9

Dynamic Low Power Reconfigurations of Real-Time Embedded Systems

9.1 Introduction

Nowadays[1], the minimization of the energy consumption is an important criterion for the development of real-time embedded systems due to limitations in capacities of their batteries, in addition to their tasks which become more and more complex than ever. These systems should provide optimal real-time services with low power consumptions. Several interesting research works have been proposed in recent years for their real-time and low power scheduling [104, 93, 124, 122, 37]. The new generations of such systems are addressing new criteria as flexibility and agility. To reduce their cost, they should be changed and adapted to their environment without disturbances. Several interesting academic and industrial research works have been made in recent years to develop reconfigurable embedded systems [38]. We distinguish in these works two reconfiguration policies: static and dynamic reconfigurations where static reconfigurations are applied off-line to apply changes before the system's cold start [6], whereas dynamic reconfigurations are dynamically applied at run-time. Two cases exist in the latter: manual reconfigurations applied by users [101] and automatic reconfigurations applied by Intelligent Agents [66, 2]. We are interested in this research in dynamic reconfigurations of real-time embedded systems that should meet deadlines defined in user requirements [13]. These systems using CMOS-based processors [102], are implemented by sets of tasks that we assume independent, periodic and synchronous (e.g. they are simultaneously activated at t = 0 time units). Each set is executed when a particular reconfiguration scenario is applied at run-time. According to [76], we characterize each task in this study by a functional priority defining its static priority in the system, a

[1]This research is done in the Research Laboratory of Prof. Zhiwu Li at Xidian University in China for the co-supervision of the PhD student Mr. Xi Wang form February 2010 to january 2013.

period equals to the deadline, and a Worst Case Execution Time (WCET). We define an automatic reconfiguration any operation allowing additions-removals or also updates of tasks at run-time. Therefore the system's implementation is dynamically changed and should meet all considered deadlines of the current combination of tasks. In addition, the energy consumption should not be increased but should be stable or decreased after each possible reconfiguration in order to satisfy the battery capacities. To reach this goal, we define an agent-based architecture where an intelligent software agent is proposed to check each dynamic (manual or automatic) reconfiguration scenario to be applied at run-time, and to help users for feasible and lower power reconfigurations. If some tasks violate corresponding deadlines, or if the power consumption is increased, the agent proposes new solutions for users in order to re-obtain the system's feasibility with low power consumption. The agent proposes first of all to modify periods of tasks in order to decrease the processor speed which is proportional to its power. It suggests as a second solution to modify execution times of tasks in order to decrease the processor utilization. Finally it proposes for users to remove some tasks according to their functional static priorities or also according to their processor utilizations. The minimization of the energy consumption is computed for each solution. The users should decide in this case which solution to apply in order to guarantee feasible low power reconfigurations of the real-time embedded system where all tasks (new and old) meet deadlines with a low power consumption. We developed at Xidian University in China a tool that supports all the services offered by the agent.

In the next section, we analyze previous works on low power and real-time scheduling as well as reconfigurations of embedded architectures, before formalize reconfigurable real-time systems in Section 3 and evaluate the power consumption of such systems in Section 4. We define in Section 5 the agent-based architecture for low power reconfigurations of real-time embedded systems. This architecture is implemented, simulated and analyzed in Section 6. Finally, we conclude and present our future works in Section 7.

9.2 Related Works

We present related works dealing with reconfigurations, real-time and low-power scheduling of embedded systems.

9.2.1 Reconfigurations of Embedded Systems

Nowadays, rich research works have been proposed to develop reconfigurable embedded systems. The authors propose in [6] reusable tasks to implement a broad range of systems where each task is statically reconfigured without any re-programming. This is accomplished by updating the supporting data structure, i.e. a state transition table, whereas the executable code remains unchanged and may be stored in permanent memory. The state transition table consists of multiple-output binary decision diagrams that represent the next-state mappings of various states and the associated control actions. The authors

propose in [101] a complete methodology based on the human intervention to dynamically reconfigure tasks. They present in addition an interesting experimentation showing the dynamic change by users of tasks without disturbing the whole system. The authors in [116] use Real-time-UML as a meta-model between design models of tasks and their implementation models to support dynamic user-based reconfigurations of control systems. The authors propose in [22] an agent-based reconfiguration approach to save the whole system when faults occur at run-time. Finally the authors propose in [2] an ontology-based agent to perform system's reconfigurations that adapt changes in requirements and also in environment. They are interested to study reconfigurations of control systems when hardware faults occur at run-time. We are interested in this study in feasible low power dynamic reconfigurations of real-time systems where additions and removals of real-time tasks are applied at run-time.

9.2.2 Real-Time Scheduling

Real-time scheduling has been extensively studied in the last three decades [13]. Several Feasibility Conditions (FC) for the dimensioning of a real-time system are defined to enable a designer to grant that timeliness constraints associated with an application are always met for all possible configurations. Different classes of scheduling algorithms are followed: (i) Clock-driven: primarily used for hard real-time systems where all properties of all jobs are known at design time. (ii) Weighted round-robin: primarily used for scheduling a real-time traffic in high-speed, (iii) Priority-driven: primarily used for more dynamic real-time systems with a mixture of time-based and event-based activities. Among all priority-driven policies, Earliest Deadline First (EDF) or Least Time to Go is a dynamic scheduling algorithm used in real-time operating systems. It places processes in a priority queue. Whenever a scheduling event occurs (task finishes, new task released, etc.) the queue will be searched for the process closest to its deadline. This process is the next to be scheduled for execution. EDF is an optimal scheduling algorithm on preemptive uniprocessors in the following sense: if a collection of independent periodic jobs characterized by arrival times equal to zero and by deadlines equal to corresponding periods, can be scheduled by a particular algorithm such that all deadlines are satisfied, then EDF is able to schedule this collection of jobs.

We present the following well-known concepts in the theory of real-time scheduling [76]:

- A periodic task T_i $(C_i; T_i; D_i)$ is an infinite collection of jobs that have their request times constrained by a regular inter-arrival time T_i, a Worst Case Execution Time (WCET) C_i and a relative deadline D_i,
- A real-time scheduling problem is said feasible if there is at least one scheduling policy able to meet the deadlines of all the considered tasks,
- A set of tasks is schedulable with a given scheduling policy if and only if no jobs of this set miss their deadlines,

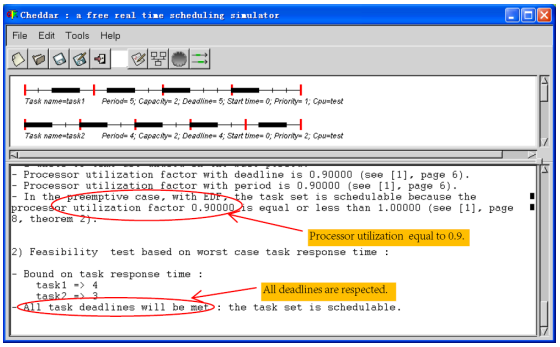

Figure 9.1: An example of periodic tasks

- A task is valid with a given scheduling policy if and only if no jobs of this task miss their deadlines,
- An idle time t of a processor is defined as a time where no tasks released before time t are pending at time t. An interval of successive idle times is classically called an idle period,
- A busy period is defined as a time interval $[a, b)$ such that there is no idle time in $[a, b)$ (the processor is fully busy) and such that both a and b are idle times,
- In the case of independent, periodic and synchronous tasks (e.g. simultaneously activated at $t = 0$), the verification of the system's schedulability is possible to be done in a hyper period $[0, LCM]$ where LCM is the Least Common Multiple [76],
- $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ is the processor utilization factor. In the case of synchronous, independent and periodic tasks such that their deadlines are equal to their periods, $U \leq 1$ is a necessary and sufficient condition for the EDF-based scheduling of real time tasks.

We present an example of periodic tasks simulated by Cheddar [107] in Fig. 9.1, which contains two synchronous periodic tasks. Both of them release at time equals to zero time unit. The first task, $task_1$, with period/deadline equals to 4 time units and C_1 equals to 2 time units. The second one, $task_2$, with period/deadline equals to 5 time units and C_2 equals to 2 time units. In this figure, we can see that the processor utilization of the tasks set is 0.9, and both $task_1$ and $task_2$ respect their deadlines. The LCM is 20 time units.

9.2.3 Low Power Scheduling

Several interesting research works have been proposed for low power and real-time scheduling of real-time embedded systems. Under the well-known Fixed Priority Preemptive Policy

(FPP), Shin and Choi [104] present a simple run-time strategy that reduces the energy consumption and the work in [93] proposes an optimal solution having an exponential algorithmic complexity. Yun and Kim [124] prove that computing the voltage schedule for real-time tasks under FPP is NP-hard and proposes an approximate solution to resolve the problem. If the well-known EDF Policy is applied, Yao *et al.* in [122] propose an off-line algorithm to find a voltage schedule for independent tasks. Over the past several years, many methods and techniques for minimizing power consumption for low power systems have been published, e.g. [16, 85, 89]. Power-reduction techniques can be in general classified into two categories [95]: static and dynamic. Dynamic techniques are generally easy to implement and applied at run-time. Examples of such techniques include those in [104, 78, 17, 45, 92, 96, 110]. We note also that several static power management policies have been investigated, e.g. [122, 46, 51, 94]. Previous investigations on the voltage scheduling problem have focused mainly on real-time jobs running under dynamic-priority scheduling algorithms such as the EDF algorithm [46, 10, 67, 90]. This research is interested in low power reconfigurations of real-time embedded systems where dynamic addition-removal-update of tasks can be applied at run-time.

9.3 Formalization of Reconfigurable Real-Time Systems

Nowadays, dynamic reconfigurations of embedded real-time systems are useful technical solutions to save the whole software/hardware architecture when faults occur at run-time, or also to improve the system's performance under well-defined conditions. A reconfiguration scenario is assumed to be an operation allowing the addition-removal-update of tasks from-to the system. Nevertheless, each scenario should be applied while reducing the energy consumption which is a very important criterion. Indeed, when a scenario is dynamically applied such that new tasks are added to the system, the energy consumption should be stable or decreased. We assume in this research work a real-time embedded system Sys as a set of tasks that should meet real-time constraints defined in user requirements: $Sys = \{T_0, T_1, \cdots, T_n\}$. When a reconfiguration scenario is applied, a subset of tasks can be added/removed to/from the system. Each task T_i of Sys is classically characterized by (i) a function f_i defining its functional priority among all the system's tasks. In this case, the task cannot be removed while the others with lower functional priorities are executed. (ii) the release time R_i defining the execution start time of the task, (iii) the Worst Case Execution Time C_i, (iv) the period T_i, (v) the deadline D_i. We assume in addition that (a) all the system's tasks are periodic and synchronous: all release times are equal to zero time units (see Fig. 9.1), (b) the period of each task is equal to the corresponding deadline. We assume in the following that the system Sys is dynamically reconfigured at run-time such that its new implementation is $Sys = \{T_0, T_1, \cdots, T_n, T_{n+1}, \cdots, T_m\}$. The subset $\{T_{n+1}, \cdots, T_m\}$ is added to the initial implementation $\{T_0, T_1, \cdots, T_n\}$. The processor utilization before and after the reconfiguration scenario is as follows:

$$U_{before} = \sum_{i=1}^{n} C_i/p_i \tag{9.3.1}$$

$$U_{after} = \sum_{i=1}^{m} C_i/p_i \tag{9.3.2}$$

9.4 Power Consumption of Reconfigurable Embedded Real-Time Systems

We want in this section to evaluate the new energy consumption after a reconfiguration scenario. In the literature, the power consumption of a processor following the CMOS technology is determined by two components: static and dynamic power consumptions [102]. The static consumption is the product of the device leakage current and the supply voltage. It is assumed to be negligible in this research. The dynamic consumption is composed of two sections: Transient power consumption and capacitive-load power consumption. It can be expressed as follows:

$$P_D = P_T + P_L \tag{9.4.1}$$

such that:

$$P_T = C_{pd} \times V_{CC}^2 \times f_I \times N_{SW} \tag{9.4.2}$$

$$P_L = \sum (C_{Ln} \times f_{On}) \times V_{CC}^2 \tag{9.4.3}$$

where

- C_{pd} = is the power consumption capacitance (F),
- f_I = is the input frequency (Hz),
- f_{On} = represents the different output frequencies at each output, numbered 1 through n (Hz),
- N_{SW} = is the total number of outputs switching,
- V_{CC} = is the supply voltage (V),
- C_{Ln} = represents the different load capacitances at each output, numbered 1 through n.

We assume in this chapter that the processor speed (S_p) is proportional to the voltage [93] and also to the processor utilization (U):

$$S_p \propto V_{CC} \propto U \tag{9.4.4}$$

where

$$V_{CC} = k_2 U \tag{9.4.5}$$

According to Eq. (9.4.1), the dynamic power consumption (P) of a CMOS circuit is quadratically dependent on the voltage.

$$P = P_D = [(C_{pd} \times f_I \times N_{SW}) + \sum (C_{Ln} \times f_{On})] \times V_{CC}^2 \tag{9.4.6}$$

where,

$$P = k_1 V_{CC}^2 \tag{9.4.7}$$

Such that,

$$k_1 = (C_{pd} \times f_I \times N_{SW}) + \sum (C_{Ln} \times f_{On}) \tag{9.4.8}$$

By considering Eq. (9.4.6);

$$P = kU^2 \tag{9.4.9}$$

where

$$k = k_1 \times k_2^2 \tag{9.4.10}$$

If $U_{after} \leq U_{before}$, then the processor speed after the reconfiguration scenario will be lower than that before, and in this case the power will be stable or minimized. The Eq. (9.4.9) is interesting and will be used to allow low power reconfigurations of embedded real-time systems: if the processor utilization is stable or decreased after any dynamic reconfiguration scenario, then the power will be stable or decreased too.

9.5 Contribution: Agent-based Architecture for Low Power Reconfigurations of Embedded Systems

We define an agent-based architecture for dynamic low power reconfigurations of an embedded real-time system. When automatic or manual reconfigurations are applied at run-time to add, remove or update tasks, the agent should check if the power consumption is increased. In this case, it should propose useful functional (remove tasks) or temporal (change their parameters) solutions for the minimization of energy consumption. We propose three technical solutions to be proposed by the agent: (i) modification of periods (e.g. deadlines) of tasks, (ii) modification of their WCETs, (iii) or removal of some others. In this case,

the users should decide a new low-power configuration of the system according to these solutions.

9.5.1 Modification of periods and deadlines

When a reconfiguration scenario is dynamically applied at run-time to add new tasks, the processor utilization of the system will be certainly increased. If the new utilization U_{after} is larger than 1, then the system is not feasible. The agent proposes as a first technical solution to modify the periods and deadlines of tasks in order to decrease the processor utilization U_{after} to be not only lower than 1 but also than U_{before}. For the reconfigured system, we can get:

$$U_{after} = \sum_{i=1}^{m} C_i/p'_i \tag{9.5.1}$$

We assume that $p'_1 = p'_2 = \cdots = p'_m = p$ in which p'_1, p'_2, ..., and p'_m are the modified periods for each task.

$$U_{after} = \sum_{i=1}^{m} C_i/p \leq U_{before} \tag{9.5.2}$$

$$p \geq \sum_{i=1}^{m} C_i/U_{before} \tag{9.5.3}$$

Because each period should be integer, we can change Eq. (9.5.3) into Eq. (9.5.4):

$$p' = \lceil (\sum_{i=1}^{m} C_i)/U_{before} \rceil \tag{9.5.4}$$

According to Eq. (9.4.9), the new energy consumption is as follows:

$$P_{after} = k(U_{after}^2) \tag{9.5.5}$$

9.5.2 Modification of WCETs

For the low power reconfiguration of embedded systems, the agent proposes as a second technical solution to reduce the WCETs of tasks in order to decrease the processor utilization U_{after}. According to Eq. (9.5.2), we can get:

$$U_{after} = \sum_{i=1}^{m} C'_i/p_i \tag{9.5.6}$$

We suppose that $C'_1 = C'_2 = \cdots = C'_m = C'$ in which, C'_1, C'_2, ..., and C'_m are the new WCETs of tasks.

$$U_{after} = \sum_{i=1}^{m} (C'/p_i) \leq U_{before} \tag{9.5.7}$$

$$C' \leq U_{before} / \sum_{i=1}^{m} (1/p_i) \tag{9.5.8}$$

Because each period should be integer, we can change Eq. (9.5.8) into Eq. (9.5.9):

$$C' = \lfloor U_{before} / \sum_{i=1}^{m} (1/p_i) \rfloor \tag{9.5.9}$$

C' in Eq. (9.5.9) is the modified WCET of each task. The actual utilization of the processor is as follows:

$$U_{after} = C' \times \sum_{i=1}^{m} (1/p_i) \tag{9.5.10}$$

The total energy consumption is shown by Eq. (9.5.5).

9.5.3 Removal of Tasks

The third solution that the agent proposes allows the removal of tasks in order to minimize the energy consumption after any reconfiguration scenario of the embedded system. We propose two policies: the agent suggests to remove tasks according to their functional priorities or according to their processor utilization. This solution is not interesting but useful in many scenarios.

First Policy: Functional Priority Criterion.

By defining for each task T_i a functional priority f_i, the agent suggests to remove tasks with lower functional priorities because their removal can be useful for a low power reconfiguration of the system. Let $List$ be the list of tasks of Sys in ascending order of functional priorities. We can remove the most unimportant tasks to keep the new utilization of the system U_{after} lower than U_{before} as follows:

$$U_{after} = \sum_{f_i=1}^{m} (C_i/p_i) \leq U_{before} \tag{9.5.11}$$

The agent should look for the highest value of U_{after} such that $U_{after} \leq U_{before}$. The total energy consumption is shown by Eq. (9.5.5).

Second Policy: Processor Utilization Criterion.

It is similar to the first policy. The difference is not to order tasks according to their static priorities, but according to their processor utilization. In this case, if the system is not feasible or consumes more energy, then we should remove the tasks with highest utilizations.

9.6 Experimentation

We present in this section an experimentation applying low power reconfigurations of embedded real-time systems. We present first of all the implementation of the agent-based architecture, before we show the simulations and analysis that is made to evaluate the benefits of our contributions.

9.6.1 Implementation of the Reconfiguration Agent

We present the agent's implementation that checks dynamic (automatic or manual) reconfiguration scenarios, in Algorithm 1, and suggests useful solutions for the minimization of the energy consumption. Each solution is generated as an input file from the agent to the well-known model simulator Cheddar [107] to check its feasibility. This implementation is tested in our research laboratory at Xidian University by assuming several cases of systems.

According to Eq. (9.3.1), we can calculate U_{before}. According to Eqs. (9.5.4) and (9.5.9), we can calculate p' in Section 5.1 and C' in Section 5.2, respectively. According to Eqs. (9.5.1) and (9.5.6), we can calculate the processor utilization after the modification of periods and deadlines U^1_{after} and processor utilization after the modification of WCET U^2_{after}, respectively. According to Eq. (9.5.11), we can calculate U^3_{after} and U^4_{after}, respectively. U^3_{after} and U^4_{after} here correspond to the utilization after tasks are removed by the two policies in Section 5.3. According to Eq. (9.4.9), the power consumption is decrease, in EDF, before the system is reconfigured P_{before}, the power consumption decrease after periods and deadlines are modified P^1_{after}, WCET modified P^2_{after}, the tasks removal by function priorities P^3_{after} or the utilization P^4_{after} can be calculated. The decrease of the energy consumption of each technical solution in Algorithm 1 is as follow:

$$P_{before} = 1 - U^2_{before} \tag{9.6.1}$$

$$P^i_{after} = 1 - (U^i_{after})^2 (for : i = 1, 2, 3, 4) \tag{9.6.2}$$

$$P^i_{decrease} = P^i_{after} - P_{before} (for : i = 1, 2, 3, 4) \tag{9.6.3}$$

To the first two technical solutions, the algorithm complexity is $O(n)$, and to the third one, it is $O(n^2)$.

Algorithm 1 low-power reconfigurations

input "system.txt" file;// the system initial configuration
input "add.txt" file;// addition of new tasks
input "priority.txt" file;// functional priorities of tasks
compute (U_{before});
calculate_period p';// solution 1 to compute the new period of tasks
for$(i = 1, i = size(system) + size(add), i + +)$// to compute the new utilization when periods are modified
    $U_i = C_i/p'$;
    $\sum U^1_{after} += U_i$;
endfor;
Evaluate_energy(U_{before},U^1_{after});
calculate_execution_time C';// solution 2 to compute the new WCET of tasks
for $i = 1, i = size(system) + size(add), i + +$// to compute the new utilization when execution times are modified
    $U_i = C'/p_i$;
    $\sum U^2_{after} += U_i$;
endfor;
Evaluate_energy(U_{before},U^2_{after});
sort all the tasks by a descending order based on their static;
Loop1 *remove_tasks_priority*(Sys_{new1});// solution 3 to remove possible tasks (first criterion)
for $i = 1, i = size(Sys_{new1}), i{+}{+}$;// to compute the new utilization when tasks are removed (first criterion)
    $\sum U^3_{after} += U_i$;
endfor;
keep_minimal($U^3_{min\_after}$);
EndLoop1
Evaluate_energy(U_{before},$U^3_{min\_after}$);
sort all the tasks by a ascending order based on their utilization;
Loop2 *remove_tasks_utilization*(Sys_{new2});//solution 3 to remove possible tasks (second criterion)
for $i = 1, i = size(Sys_{new2}), i{+}{+}$;//to compute the new utilization when tasks are removed (second criterion)
    $\sum U^4_{after} += U_i$;
endfor;
keep_minimal($U^4_{min\_after}$);
Evaluate_energy(U_{before},$U^4_{min\_after}$);
EndLoop2
end;

Num	F	R	C	T	D	Num	F	R	C	T	D
1	A1	0	4	200	200	26	F1	0	4	450	450
2	A2	0	5	210	210	27	F2	0	5	460	460
3	A3	0	6	220	220	28	F3	0	6	470	470
4	A4	0	7	230	230	29	F4	0	7	480	480
5	A5	0	8	240	240	30	F5	0	8	490	490
6	B1	0	4	250	250	31	G1	0	5	300	300
7	B2	0	5	260	260	32	G2	0	6	310	310
8	B3	0	6	270	270	33	G3	0	7	320	320
9	B4	0	7	280	280	34	G4	0	8	330	330
10	B5	0	8	290	290	35	G5	0	9	340	340
11	C1	0	4	300	300	36	H1	0	5	350	350
12	C2	0	5	310	310	37	H2	0	6	360	360
13	C3	0	6	320	320	38	H3	0	7	370	370
14	C4	0	7	330	330	39	H4	0	8	380	380
15	C5	0	8	340	340	40	H5	0	9	390	390
16	D1	0	4	350	350	41	I1	0	5	400	400
17	D2	0	5	360	360	42	I2	0	6	410	410
18	D3	0	6	370	370	43	I3	0	7	420	420
19	D4	0	7	380	380	44	I4	0	8	430	430
20	D5	0	8	390	390	45	I5	0	9	440	440
21	E1	0	4	400	400	46	J1	0	5	450	450
22	E2	0	5	410	410	47	J2	0	6	460	460
23	E3	0	6	420	420	48	J3	0	7	470	470
24	E4	0	7	430	430	49	J4	0	8	480	480
25	E5	0	8	440	440	50	J5	0	9	490	490

Figure 9.2: Initial System Tasks

Num	F	R	C	T	D	Num	F	R	C	T	D
1	AA1	0	3	205	205	16	AD1	0	4	245	245
2	AA2	0	4	215	215	17	AD2	0	5	255	255
3	AA3	0	5	225	225	18	AD3	0	6	205	205
4	AA4	0	6	235	235	19	AD4	0	7	215	215
5	AA5	0	7	245	245	20	AD5	0	8	225	225
6	AB1	0	3	215	215	21	AE1	0	4	255	255
7	AB2	0	4	225	225	22	AE2	0	5	205	205
8	AB3	0	5	235	235	23	AE3	0	6	215	215
9	AB4	0	6	245	245	24	AE4	0	7	225	225
10	AB5	0	7	255	255	25	AE5	0	8	235	235
11	AC1	0	3	225	225	26	AF1	0	4	205	205
12	AC2	0	4	235	235	27	AF2	0	5	215	215
13	AC3	0	5	245	245	28	AF3	0	6	225	225
14	AC4	0	6	255	255	29	AF4	0	7	235	235
15	AC5	0	7	205	205	30	AF5	0	8	245	245

Figure 9.3: Added Tasks

9.6.2 Simulations

We present some simulations applying low-power reconfigurations of an embedded real-time system which is initially composed of 50 tasks, and which is dynamically reconfigured at run-time to add 30 new ones. We assume the following temporal characteristics of the system:

- **Initial System's Tasks:** All the following initial tasks of the system are in the file system.txt ((R_i, C_i, T_i, D_i) define the temporal parameters of the task A_i) (Fig. 9.2),
- **Added Tasks:** All the added new tasks are described in the file add.txt (Fig. 9.3),
- **Static priorities:** All the functional priorities of the system's tasks are defined in the file priority.txt (Fig. 9.4).

Num	F	P	Num	F	P	Num	F	P	Num	F	P
1	A1	1	21	E1	41	41	I1	80	61	AC1	40
2	A2	3	22	E2	43	42	I2	78	62	AC2	38
3	A3	5	23	E3	45	43	I3	76	63	AC3	36
4	A4	7	24	E4	47	44	I4	74	64	AC4	34
5	A5	9	25	E5	49	45	I5	72	65	AC5	32
6	B1	11	26	F1	51	46	J1	70	66	AD1	30
7	B2	13	27	F2	53	47	J2	68	67	AD2	28
8	B3	15	28	F3	55	48	J3	66	68	AD3	26
9	B4	17	29	F4	57	49	J4	64	69	AD4	24
10	B5	19	30	F5	59	50	J5	62	70	AD5	22
11	C1	21	31	G1	61	51	AA1	60	71	AE1	20
12	C2	23	32	G2	63	52	AA2	58	72	AE2	18
13	C3	25	33	G3	65	53	AA3	56	73	AE3	16
14	C4	27	34	G4	67	54	AA4	54	74	AE4	14
15	C5	29	35	G5	69	55	AA5	52	75	AE5	12
16	D1	31	36	H1	71	56	AB1	50	76	AF1	10
17	D2	33	37	H2	73	57	AB2	48	77	AF2	8
18	D3	35	38	H3	75	58	AB3	46	78	AF3	6
19	D4	37	39	H4	77	59	AB4	44	79	AF4	4
20	D5	39	40	H5	79	60	AB5	42	80	AF5	2

Figure 9.4: Static Priorities

We performed several simulations to apply the proposed technical solutions for low power reconfigurations of the system. We note that the initial processor utilization $U_{before} = 0.91224$. If we only add the task AA1, the processor utilization $U_{after} = 0.926874$ which is lower than 1. The system is still feasible where all deadlines are satisfied. Nevertheless, the energy consumption is increased ($U_{before} \geq U_{after}$) which is not usually accepted. By applying the first solution where periods and deadlines of tasks are modified to be equal to 355, the processor utilization becomes $U_{after} = 0.909859$. According to Eqs. (9.6.1) and (9.6.2), the power reduction before and after the automatic reconfiguration is as follows:

$$P_{before} = 1 - 0.91224^2 = 16.7819\% \tag{9.6.4}$$

$$P_{after} = 1 - 0.909859^2 = 17.2156\% \tag{9.6.5}$$

We show in Fig. 9.5 a software package that we developed to compute the new periods as well as the new processor utilization. According to Eq. (9.6.3), the decrease of the energy consumption is as follows:

$$P_{after} - P_{before} = 0.433749\% \tag{9.6.6}$$

In a similar way, we apply the second and third solutions to decrease the energy consumption in the system. By modifying the execution times, the energy consumption decreases 2.45583%. It decreases also in the third case when we remove tasks: (i) according to their functional priorities: 3.77711%, (ii) according to their utilization: 3.37665%. We show in Fig. 9.6 the initial utilization U, the utilization including the task AA1 U', the utilization after the reconfiguration scenario U'', the power reduction before the reconfiguration scenario P, the power reduction after such scenario P', and the energy consumption decrease P''. The parameters CP, CW, RP, and RU, correspond to the modification of periods, the

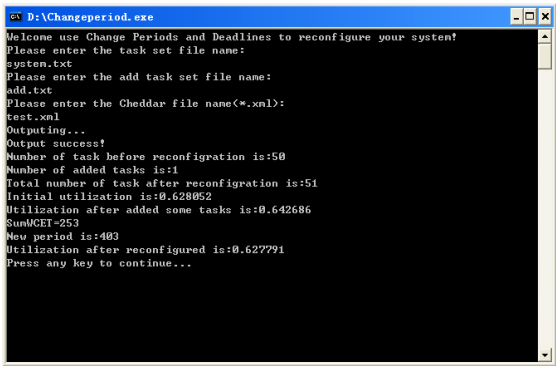

Figure 9.5: Developed Software

Method	Add	C/R	U	U'	U″	P	P'	P″
CP	1	355	0.91224	0.926874	0.909859	16.7819%	17.2156%	0.433749%
CW	1	6	0.91224	0.926874	0.898678	16.7819%	19.2377%	2.45583%
RP	1	2	0.91224	0.926874	0.891297	16.7819%	20.559%	3.77711%
RU	1	1	0.91224	0.926874	0.893541	16.7819%	20.1585%	3.37665%

Figure 9.6: Low power reconfiguration after the addition of $AA1$

modification of WCETs, the removal of tasks according to functional priorities, and the removal of tasks according to processor utilizations, respectively. We show in Fig. 9.7 useful solutions for low power reconfigurations of the system when we add the first 10 tasks, We show in Fig. 9.8 new solutions when we add all the 30 tasks. We show in Appendix the new configuration of the system when we add all the new tasks. We denote in this case by NP: the new period, ENP: the power decrease when we modify the period, NW: the new WCET, ENW: the power decrease in this case, RP: the number of removed tasks according to their static priorities, ERP: the corresponding power decrease, RU: the number of removed tasks according to their processor utilization, and ERU: the corresponding power decrease.

9.6.3 Analysis

We present some analysis that prove the advantages of the different proposed solutions.

9.6.4 Advantages of the First Solution

According to Eqs. (9.5.4) and (9.5.5), the power minimization is proven to be dependent of the WCETs. If all the 30 tasks are added, their WCETs are equal to 165. We made several simulations for $\sum C_i = 3, 4, 5, \cdots, 165$. The result is shown in Fig. 9.9 in which we find a decrease of consumption between 0 and 0.45%, the energy save was piecewise approximate

Method	Add	C/R	U	U'	U″	P	P'	P″
CP	10	406	0.91224	1.12675	0.91133	16.7819%	16.9477%	0.165854%
CW	10	4	0.91224	1.12675	0.754266	16.7819%	43.1083%	26.3264%
RP	10	13	0.91224	1.12675	0.899266	16.7819%	19.132%	2.3501%
RU	10	8	0.91224	1.12675	0.900101	16.7819%	18.9819%	2.19998%

Figure 9.7: Low power reconfiguration after the addition of the first ten tasks

Method	Add	C/R	U	U'	U″	P	P'	P″
CP	30	532	0.91224	1.63352	0.911654	16.7819%	16.8887%	0.106788%
CW	30	3	0.91224	1.63352	0.829756	16.7819%	31.1505%	14.3686%
RP	30	41	0.91224	1.63352	0.911745	16.7819%	16.8721%	0.0902683%
RU	30	26	0.91224	1.63352	0.900925	16.7819%	18.8334%	2.05151%

Figure 9.8: Low power reconfiguration after the addition of the thirty new tasks

linear depend on the $\sum C_i = 3, 4, 5, \cdots, 165$.

Advantages of the Second Solution

According to the Eqs. (9.5.10) and (9.5.5), the power minimization is proven to be dependant of periods. If all the 30 tasks are added, the value of $\sum_{i=1}^{30} 1/p_i$ is equal to 0.131684. We made simulations for subsets of tasks, the range of $\sum 1/p_i$ is between 0 and 0.131684. The result is shown in Fig. 9.10 where the minimization of the energy consumption is between 0 and 35%. This simulation proves the benefits of the second solution than the first. Nevertheless, sometimes, the modification of periods is more simpler than minimization of WCET. We cannot judge that the second solution is more good than the first.

9.6.5 Application and Advantages of the Third Solution

In the third solution where tasks are removed at run-time to minimize the energy consumption after reconfiguration scenarios, we applied several simulations to evaluate the benefits. In Fig. 9.11, the continuous line presents the first policy where the priority function is used as a criterion to remove tasks. In this case, the number of removed tasks is equal or greater than the added one: the tasks remaining in the system are less than 50. The dotted line corresponds to the removal of tasks according to their processor utilizations. The number of removed tasks is smaller than that added. This second policy is more useful than the first. We show in Fig. 9.12 the minimization of the energy consumption when we apply the first (continuous line) and second (dotted line) policy. We note that the energy consumption stills minimal when we apply the second solution where WCETs of tasks are modified.

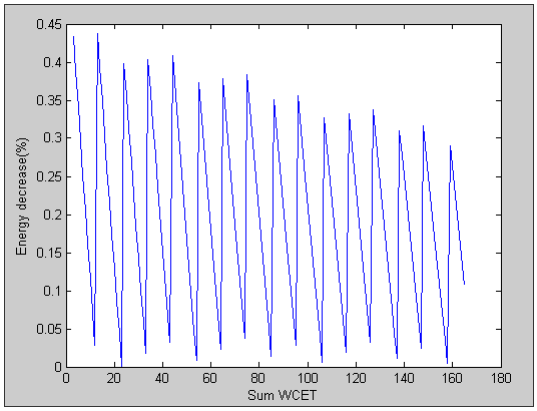

Figure 9.9: Power save by modify periods

9.7 Conclusion

The chapter deals with low power and real-time dynamic reconfigurations of embedded systems to be implemented by sets of tasks that should meet real-time constraints while satisfying limitations in capacities of batteries. A reconfiguration scenario means the addition, removal or update of tasks in order to save the system when faults occur or to improve its performance. The energy consumption can often be increased or real-time constraints can often be violated when tasks are added. To allow a stable energy consumption before and after each reconfiguration scenario, we define an agent-based architecture where an intelligent software agent is proposed to check each dynamic reconfiguration scenario and to suggest for users useful solutions in order to minimize the energy consumption. It proposes to modify periods, reduce execution times of tasks or remove some of them. A tool is developed and tested to support all these services. In our future work, we plan to study low power and real-time reconfigurations of asynchronous tasks that can be loaded in a same processor or can be distributed on different calculators.

9.8 References of the Chapter's Contributions

- X. Wang, M. Khalgui, and Z. Li, **Low Power Manual and Automatic Reconfigurations of Embedded Real-Time Systems**, IEEE Transactions on Automation Science and Engineering (submitted),

Appendix

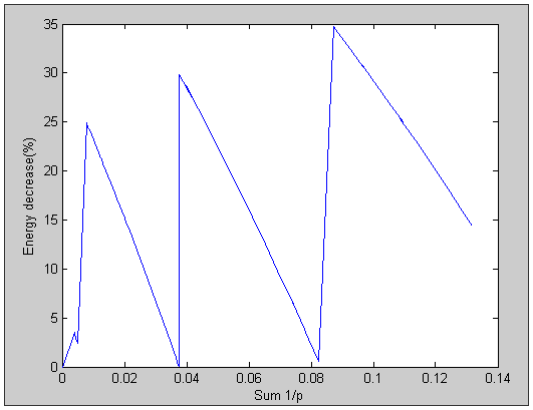

Figure 9.10: Power save by modify WCETs

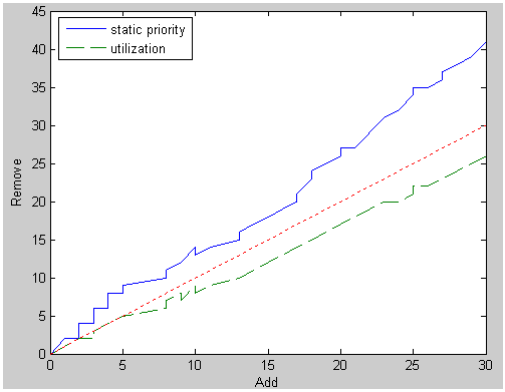

Figure 9.11: Compare the number of removed task between two remove strategies

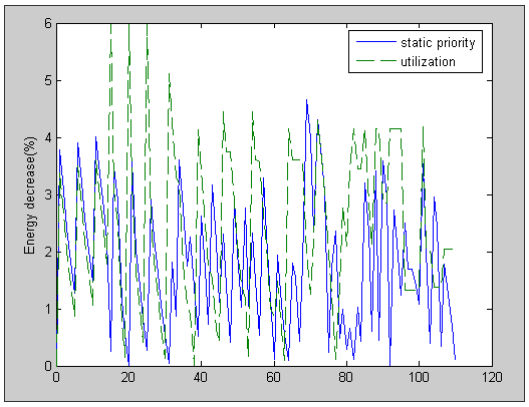

Figure 9.12: Compare power save between two remove strategies

Add	NP	ENP	NW	ENW	RP	ERP	RU	ERU
1	355	0.433749%	6	0.024558	2	3.77711%	1	3.37665%
1	356	0.3877%	6	2.70032%	2	3.06776%	1	2.66551%
1	357	0.341833%	6	2.92276%	2	2.41871%	1	2.01484%
1	358	0.296225%	6	3.126%	2	1.82261%	1	1.41725%
1	359	0.250866%	6	3.3124%	2	1.27323%	1	0.866516%
1	355	0.433749%	6	2.70032%	2	3.8984%	1	3.49825%
1	356	0.3877%	6	2.92276%	2	3.21574%	1	2.81387%
1	357	0.341833%	6	3.126%	2	2.58863%	1	2.18517%
1	358	0.296225%	6	3.3124%	2	2.01054%	1	1.60564%
1	359	0.250866%	6	3.48401%	2	1.47597%	1	1.06975%
1	355	0.433749%	6	2.92276%	2	4.00884%	1	3.60894%
1	356	0.3877%	6	3.126 %	2	3.35102%	1	2.94948%
1	357	0.341833%	6	3.3124 %	2	2.74452%	1	2.34145%
1	358	0.296225%	6	3.48401%	2	2.18354%	1	1.77908%
1	359	0.250866%	6	2.45583%	2	0.260821%	1	5.97047%
1	356	0.3877%	6	3.3124 %	2	3.47515%	1	3.07392%
1	357	0.341833%	6	3.48401%	2	2.88804%	1	2.48533%
1	358	0.296225%	6	2.45583%	2	1.14703%	1	0.739995%
1	359	0.250866%	6	2.70032%	2	0.549872%	1	0.141355%
1	360	0.205775%	6	2.92276%	2	0.00391454%	1	5.97047%
1	356	0.3877%	6	3.48401%	2	3.58946%	1	3.18851%
1	357	0.341833%	6	2.45583%	2	2.02848%	1	1.62364%
1	358	0.296225%	6	2.70032%	2	1.39349%	1	0.987071%
1	359	0.250866%	6	2.92276%	2	0.812806%	1	0.404927%
1	360	0.205775%	6	3.126 %	2	0.279723%	2	5.97047%
1	356	0.3877%	6	2.45583%	2	2.90519%	1	2.50252%
1	357	0.341833%	6	2.70032%	2	2.23279%	1	1.82845%
1	358	0.296225%	6	2.92276%	2	1.61774%	1	1.21186%
1	359	0.250866%	6	3.126 %	2	1.05299%	1	0.645703%
1	360	0.205775%	6	3.3124 %	2	0.532617%	1	0.124057%
2	361	0.160889%	5	24.8906%	2	0.0516027%	2	5.11393%
2	362	0.116286%	5	24.7683%	3	1.82081%	2	4.24864%
2	363	0.0718445%	5	24.6355%	3	0.862135%	2	3.30433%
2	364	0.0276625%	5	24.4906%	4	3.61288%	2	3.18851%
2	366	0.43799%	5	24.3321%	4	2.74584%	2	1.77908%
2	367	0.393276%	5	24.1579%	4	1.75428%	2	1.06975%
3	368	0.348768%	5	21.8572%	4	2.25422%	2	0.89967%
3	369	0.304507%	5	21.7317%	4	1.37328%	2	0.0114068%
3	370	0.260486%	5	21.6062%	4	0.516692%	3	4.13039%
3	371	0.216712%	5	21.4805%	5	2.61749%	3	3.4072%
3	372	0.173177%	5	21.3439%	5	1.69485%	3	2.48909%
3	373	0.12975%	5	21.2072%	5	0.735447%	3	1.75397%
3	374	0.0866383%	5	21.0704%	6	3.16001%	3	1.53435%
3	375	0.0437438%	5	20.9212%	6	2.15849%	3	0.645703%
3	376	0.00101179%	5	20.7718%	6	1.11649%	3	0.404927%
4	378	0.398494%	5	18.7468%	6	2.30667%	4	4.44325%
4	379	0.355344%	5	18.6183%	6	1.39706%	4	3.74559%
4	380	0.31229%	5	18.4786%	6	0.404288%	4	3.74559%
4	381	0.269519%	5	18.3497%	7	2.88304%	4	2.87278%
4	382	0.226997%	5	18.2208%	7	1.97667%	4	2.14387%
4	383	0.184627%	5	18.0807%	7	1.0189%	4	1.40717%
4	384	0.142507%	5	17.7875%	8	2.76524%	4	1.18709%
4	385	0.100604%	5	17.1528%	8	1.04091%	4	0.124057%
5	386	0.0588551%	5	15.1528%	8	2.32615%	5	4.44325%
5	387	0.0172675%	5	15.0207%	8	1.47455%	5	3.60284%

Figure 9.13: New Configuration of the System (1)

Add	NP	ENP	NW	ENW	RP	ERP	RU	ERU
5	389	0.403409%	5	14.8885%	8	0.531186%	5	3.5787%
5	390	0.361387%	5	14.7448%	9	3.2958%	5	2.66158%
5	391	0.31967%	5	14.5882%	9	2.29514%	5	2.11504%
5	392	0.278127%	5	14.3115%	9	1.16535%	5	0.984%
5	393	0.236651%	5	14.1544%	9	0.117004%	5	0.866516%
8	393	0.236651%	5	2.36775%	10	1.93608%	6	1.48556%
8	393	0.236651%	5	1.50543%	10	1.07145%	6	0.618565%
8	393	0.236651%	5	0.989784%	10	0.554457%	6	0.100137%
8	393	0.236651%	5	0.424127%	10	0.100887%	7	4.15374%
8	393	0.236651%	5	0.0635505%	11	1.79859%	7	3.59905%
8	393	0.236651%	5	29.8667%	11	1.55274%	7	3.59905%
9	402	0.326627%	4	27.455%	12	0.431503%	8	3.59905%
10	409	0.0472873%	4	24.8839%	14	1.79043%	9	3.59905%
9	398	0.0327513%	4	28.534%	12	4.65897%	7	1.93387%
9	400	0.408192%	4	28.6681%	12	3.99395%	7	1.25743%
10	406	0.165854%	4	26.3264%	13	2.3501%	8	2.19998%
11	409	0.0472769%	4	23.613%	13	4.30022%	9	4.28659%
11	410	0.00819862%	4	23.7297%	14	3.64363%	9	3.62994%
11	412	0.372759%	4	23.8368%	14	3.03822%	9	3.02449%
13	419	0.099431%	4	18.3526%	15	0.245458%	10	1.97066%
13	421	0.0229552%	4	17.8384%	16	1.7212%	10	1.23317%
13	423	0.378011%	4	18.689%	16	2.35947%	10	0.122166%
14	428	0.186799%	4	15.2771%	17	0.477411%	11	1.407%
15	432	0.0369698%	4	12.559%	18	0.998388%	12	2.80401%
15	434	0.382948%	4	12.559%	18	0.285673%	12	2.09914%
16	438	0.233512%	4	9.78752%	19	0.656958%	13	3.44148%
17	443	0.0503317%	4	7.07455%	20	0.121601%	14	4.14048%
17	444	0.0141144%	4	6.40539%	21	1.0259%	14	3.44148%
17	446	0.350992%	4	6.56439%	22	0.428304%	14	3.44148%
18	451	0.16994%	4	3.79305%	23	3.19333%	15	4.14048%
18	455	0.0276849%	4	3.53416%	24	2.36%	15	2.85424%
18	457	0.356342%	4	2.84965%	24	0.613311%	15	2.09914%
19	460	0.24979%	4	0.592701%	25	3.41682%	16	4.14048%
20	466	0.0406012%	3	34.7246%	26	0.441222%	17	4.14048%
20	467	0.00617653%	3	34.3913%	27	3.58928%	17	2.85424%
20	469	0.326574%	3	34.4864%	27	2.96041%	17	2.85424%
21	473	0.188656%	3	32.7618%	27	0.00847131%	18	4.14048%
22	478	0.0193238%	3	30.6614%	29	2.73165%	19	4.14048%
22	480	0.33225%	3	30.7512%	29	2.15572%	19	4.14048%
23	486	0.130837%	3	28.7104%	31	1.24117%	20	4.14048%
24	489	0.0317946%	3	26.9594%	32	2.49307%	20	1.32605%
24	491	0.33772%	3	26.9594%	32	1.69245%	20	1.32605%
25	499	0.0758991%	3	25.028%	34	1.69504%	21	1.32605%
25	500	0.0437319%	3	25.194%	34	1.41635%	21	1.32605%
25	501	0.0117004%	3	25.3747%	34	1.07588%	21	1.32605%
25	503	0.310379%	3	25.0413%	35	3.5417%	22	4.18913%
26	504	0.277987%	3	22.774%	35	2.01334%	22	2.05151%
27	509	0.118177%	3	20.5848%	36	0.388643%	23	2.05151%
27	512	0.0237703%	3	20.8802%	37	2.95583%	23	1.37556%
27	514	0.316133%	3	20.9624%	37	2.44152%	23	1.37556%
28	519	0.159043%	3	18.757%	38	0.341898%	24	1.37556%
29	523	0.0353605%	3	16.3857%	39	1.78102%	25	2.05151%
29	524	0.00470877%	3	16.4708%	39	1.26297%	25	2.05151%
29	526	0.290449%	3	16.5635%	39	0.696987%	25	2.05151%
30	532	0.106788%	3	14.3686%	41	0.0902683%	26	2.05151%

Figure 9.14: New Configuration of the System (2)

Chapter 10

Simulation of Control Components implementing a Reconfiguration Scenario

10.1 Introduction

We are interested in this chapter in discrete event simulations of reconfigurable embedded systems to verify functional and temporal properties defined in user requirements [75, 113]. In addition to end to end bounds, the blocking problem should be checked for each Control Component to avoid any situation in which the number of input events is higher than the size of the corresponding buffer. Nevertheless, the simulation is known classically as a non exhaustive approach [23, 24] especially for complex systems because it is not possible to simulate all possible execution scenarios and it is not possible to detect all possible faults in these scenarios. We optimize in this work the simulation by applying a technique to be based on fault injections in order to bring the whole system to critical scenarios where functional and temporal properties can be possibly violated. By concentrating the simulation around these scenarios, it will be possible to detect faults and to check major critical behaviors of the system. We are interested in this chapter in the simulation of a particular network of Control Components $Network_{i,j,k,h}$ $(a \in [1, n_{ASM}], b \in [1, n_{CSM_a}], c \in [1, n_{DSM}], d \in [1, n_{DSM_c}])$ to be executed when the reconfiguration scenario $Reconfiguration_{i,j,k,h}$ is automatically applied at run-time. We apply this approach of simulation to an embedded system implementing a footwear factory in Italy where Control Components are designed in different levels of a hierarchy to control the design complexity [56]. The Simulink-Stateflow environment and formal rules are proposed in [12] to implement this software architecture. We formalize a Control Component by introducing two modules: the Control Function Module implementing the component's behavior when a particular input event occurs at run-time, and the Virtual Plant Module supporting interactions of the component with physical processes. Therefore, the application is considered as a set of Control Function Modules with

precedence constraints that should meet temporal bounds. We propose a technique to process a deadline for each control module in order to satisfy the corresponding bound and define thereafter a characterization of critical execution scenarios where a blocking problem can occur or a temporal property can be violated in a Control Component. To consider the system's hierarchy, we define a model of simulators based on the well known "master-slave" architecture to bring the whole system's behavior to critical scenarios. In this model, a slave is defined for each level of the system's hierarchy and the master chooses useful errors to inject in Control Components of each level by the corresponding slave. All the slaves are running in parallel, they do not exchange data and their role is just to inject faults in hardware components of the corresponding levels. We present in the chapter the benefits of this approach by presenting our results that we found in the footwear factory in Italy.

in the next Section, we present a system implementing this factory, and analyze in Section 3 some related works on simulations of embedded systems. We formalize Control Components in Section 4 by introducing two modules defining its behavior. We characterize in Section 5 the critical execution scenarios of a Control Component where a blocking problem is possible to occur or a temporal property can be violated. The architecture of proposed simulators is formalized and implemented in Section 6 before we evaluate in Section 7 its performance and show the results that we found in the footwear factory.

10.2 Industrial Case Study

We are interested in a sub-system of the footwear factory which is implemented according to the IEC61499 technology at the ITIA-CNR Institute in Italy (Figure 10.2).

10.2.1 Presentation

Let us consider 9 belts for the transport of produced pieces (pairs of shoes) from six production stations $Station1$,..., $Station6$ to a storing and packing station $Station7$ (figure 10.1). The bidirectional belt $Belt7$ transports pieces from the discs $disc1$ and $disc2$ to the belt $Belt8$. Each belt (as well as a disc) is characterized by a maximum number of pieces to transport together. According to the production policy in the factory, we assume that the pairs of shoes coming from the different production stations follow **an aperiodic** flow.

By applying the rules proposed in [12], we show in Figure 10.3 the modular implementation of this sub-system as a composition of 9 composite Function Blocks (i.e. composed of Function Blocks) where each block controls the command of a particular belt or a disc. In the following, we will not be interested in the internal behavior of each block. We refer to detailed documentations available at the ITIA-CNR Institute.

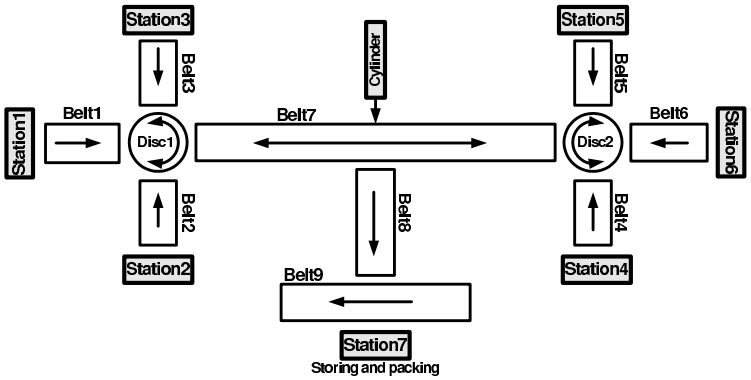

Figure 10.1: A set of belts transporting pieces in the ITIA factory

10.2.2 Problem

In the transportation system, a critical problem that we should avoid is the saturation of belts when the station *Station*7 is storing and packing a set of successive shoes coming from the belt *Belt*8. Moreover, the production of a pair of shoes should satisfy an end to end response time bound described in user requirements between the exit from a *Stationj* ($j \in [1, 6]$) and the packing in the station *Station*7. To guarantee a correct production according to these constraints, we apply in this research work a simulation based on a fault injection technique to check the feasibility of the transportation system in the worst case.

10.2.3 Numerical Characterization

According to documented user requirements available at the ITIA-CNR Institute, the following numerical parameters are considered in this chapter:

- The size of the buffer storing input events in each block FB_Beltj ($j \in [1, 6]$ or $j = 8$) is $m = 1$ and by considering the double transportation direction, the size of the buffer in each block controlling a disc or the belt FB_Belt7 is $m = 2$. Finally, the size of the buffer FB_Belt9 is $m = 14$ which corresponds to the maximum number of shoes waiting the packing that the system can support.

- The worst case execution time of the algorithm in *Belt*9 is 12 time units, the worst time to execute the corresponding Service Interface Function Block is 8 time units and finally the worst time to transport a pair of shoes for FB_Belt9 is 60 time units. For the rest of belts, these times are equal respectively to 5, 5 and 20 time units.

- The production of each pair of shoes from a *Stationj* ($j \in [1, 6]$) to the station *Station*7 should satisfy an end to end bound equal to 220 time units.

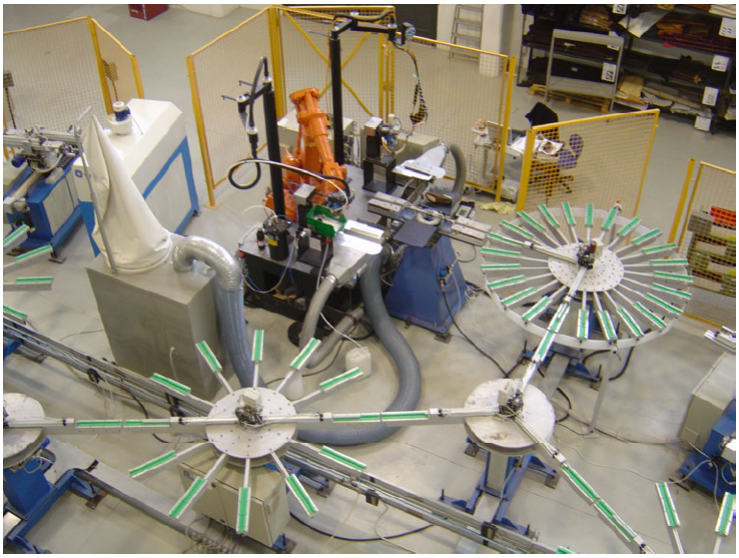

Figure 10.2: The ITIA footwear factory in Vigevano (Italy).

10.3 State of the Art

Several rich research works have been proposed to simulate manufacturing systems [105, 69] since the earliest simulation generator developed by [88]. Among all these contributions, Lee proposes in [70] an automatic simulation modelling methodology based on the part flow specified in a process model and the resource description specified in resource model. In this work, the process model describes the process sequence and the resource alternatives for manufacturing each part. In the same way, a similar methodology is proposed in [11] to generate simulations for KANBAN-controlled manufacturing systems. We note that these two approaches use the generated simulation model for design and analysis of the system. In [109], the authors present an architecture for the generation of Simulation-based Shop Floor Control System (SSFCS) by formally modelling the components (execution model, simulation model and resource model) that comprise the SSFCS. The execution model is a model describing only the execution activities required to control process (i.e. interfacing with physical equipment and performing synchronization among different controllers), the resource model contains a set of required definitions and symbolic descriptions to describe all the individual resources and finally the simulation model comprise all the required information defining the system simulation [108]. Although this architecture is interesting and useful in manufacturing industry, concentrating the system execution around critical scenarios to detect all possible faults and to keep a controlled verification has not been considered. As a solution in our contribution, we base the simulator on a fault-injection technique to bring the system's execution into such scenarios. Nowadays, several fault injection methods and tools have been proposed [18, 48]. These methods are based on simulation-based

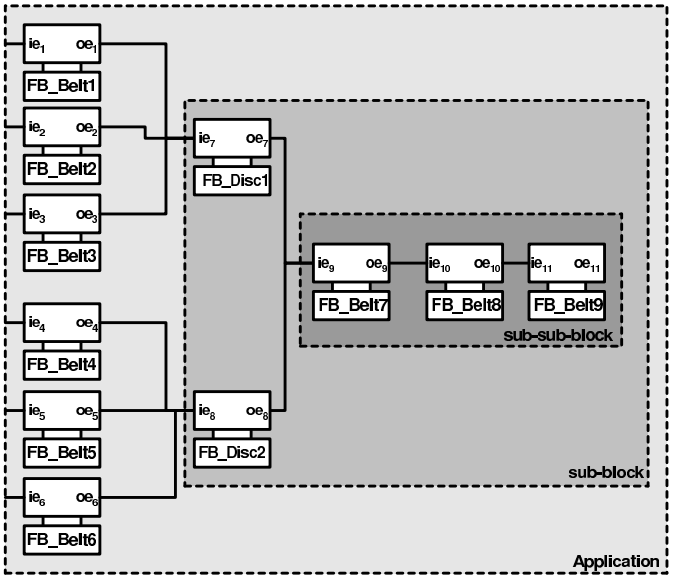

Figure 10.3: Implementation of the footwear system in Vigevano (Italy).

techniques [34], software implemented techniques [25, 53, 77], hardware-based techniques [7] and hybrid techniques where hardware and software approaches are applied together to optimize the performance [123]. None of these techniques seems to be a general solution since they are generally targeted to a particular platform or particular types of applications [19].

10.4 Contribution: Formalization of Control Components

We formalize any system by defining two formal modules encoding the behavior of each Control Component. In order to meet real-time constraints, a technique is proposed thereafter to process deadlines for these components.

10.4.1 Control Modules

We characterize the behavior of a Control Component by defining the following formal modules:

- **Control Function Module** : We define a set of Control Function Modules for a component (denoted thereafter by CFM) such that each one encodes a control function implemented by an algorithm to execute when a corresponding input event occurs in the component.

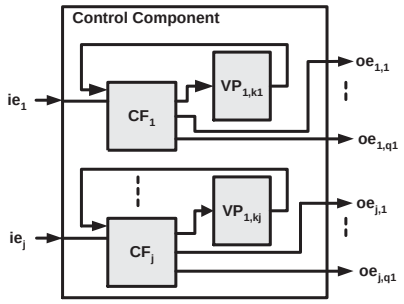

Figure 10.4: Formalization of a Control Component.

Notation. In the following, we denote by $cf(CC)$ the set of Control Functions of a Control Component CC.

- **Virtual Plant Module** : We define for a Control Component a set of virtual plant modules (denoted thereafter by VPM) such that each one is the interface between a unique CFM and a set of physical processes. In the IEC61499 standard, Service Interface Function Blocks are used to implement these modules [74].

  Notation. In the following, we denote by $vp(CC)$ the set of the Virtual Plant Modules of a Control Component CC.

Notation. We denote in the following by *cf_modules* (*vp_modules* ,resp) the set of all the Control Function (Virtual Plant, resp) modules implementing the application.

According to these definitions, the Control Function Modules define the functional architecture of the application, and the Virtual Plant Modules define interactions of Control Modules with physical processes. We abstract in the following these processes in the corresponding Virtual Plant Modules. We show in Figure 10.4 a Control Component characterized by j CFM modules corresponding to different input events such that each one CF_i ($i \in [1, j]$) implements a Control Function controlling physical processes through a VPM module VP_i. Once the control of these processes is correctly done, the module sends output events to following components in the application (Figure 10.5).

Running example. *In the Italian footwear platform located in Vigevano, we present the three levels containing the blocks that control the belts and the two discs. The third level contains CFM and VPM modules controlling belts $Belt1$, ..., $Belt6$, the second level contains modules controlling the two discs and the first level contains modules controlling the rest of belts (figure 10.6).*

10.4.2 Formalization

To simulate a control system, we formalize a Control Function Module $cf \in cf\_modules$ as follows:

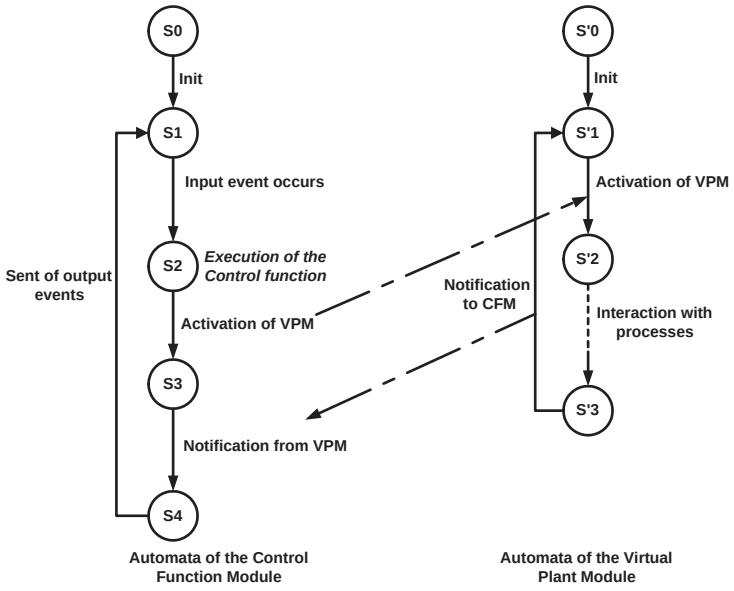

Figure 10.5: Behavior of CFM and VPM modules.

- $pred(cf)$: a set of Control Function Modules preceding cf in the system (i.e. their execution precedes that of cf).
- $succ(cf)$: a set of Control Function Module sets such as only one module set should be executed in the application after the cf execution.
- $d(cf)$: the deadline of the module that we have to calculate while taking into account the successors of cf in order to meet end to end bounds.

Running example. *In the manufacturing platform in Vigevano, we formalize the modules* $CF(FB\_Disc1)$ *and* $CF(FB\_Disc2)$ *as follows:*

$$pred(CF(FB\_Disc1)) = \{CF(FB\_Belt1), CF(FB\_Belt2), CF(FB\_Belt3)\}$$

$$pred(CF(FB\_Disc2)) = \{CF(FB\_Belt4), CF(FB\_Belt5), CF(FB\_Belt6)\}$$

$$succ(CF(FB\_Disc1)) = succ(CF(FB\_Disc2)) = \{\{CF(FB\_Belt7)\}\}$$

Notation. We denote in the following by $first(cf\_modules)$ ($last(cf\_modules)$, resp) the set of Control Function Modules without predecessors (successors, resp) in the system.

Notation. Let CC be a Control Component of a system and let cf and vp be two corresponding modules. We denote by,

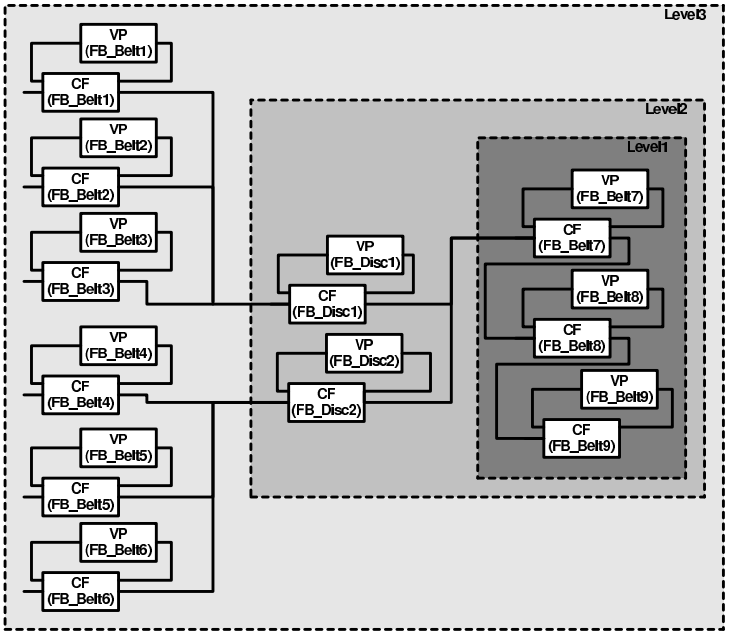

Figure 10.6: The modularity of the system of belts in Vigevano (Italy).

- $\theta(cf)$ the execution time of the algorithm implementing the module cf [64]. We consider this parameter as a *constant*.
- $\theta(vp)$ the duration taken by vp to answer the cf. We characterize this duration by the following parameters,
  - ** $\theta(SIFB)$: the execution time of the software interface modules (middleware) located in the module to interact with physical processes. We consider this parameter as a *constant*,
  - ** $\theta(processes)$: the duration taken by the physical processes to realize the required functionality. We consider this parameter as a *constant*,
  - ** $\theta(errors)$: the duration of external errors that can occur during the treatment of vp. This parameter increases the response time to cf. We consider it as a *variable* equal to zero in the best case.

  By considering this characterization, the parameter $\theta(vp)$ is as follows,

$$\theta(vp) = \theta(sifb) + \theta(processes) + \theta(errors)$$

We show in Figure 10.7 the temporal behavior of a Control Component when a particular input event occurs. In this case, the Control Function Module cf is activated to be executed

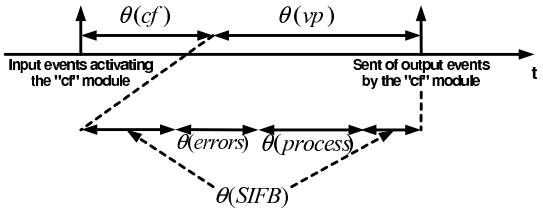

Figure 10.7: Temporal behavior of a Control Component.

according to priority rules in the scheduler [63]. It activates also the corresponding Virtual Plant Module vp based on interface blocks in order to interact with physical processes. As presented in the figure, external errors occur in the hardware components and increase the response time of the vp module. Once the physical process finishes execution, the vp notifies cf which sends corresponding output events to the following cf modules in the control application.

We define *a Sequence* of Control Function Modules a sequence of modules with precedence constraints in the application.

$$Seq : cf_0,\ cf_1,\ ....,\ cf_{n-1} \text{ such as,}$$

- $cf_0 \in first(cf\_modules)$ and $cf_{n-1} \in last(cf\_modules)$,
- $\forall\ i \in [1, n-1]$, $cf_{i-1} \in pred(cf_i)$,

According to this formalization, the system is transformed to a set of Control Function Modules with precedence constraints. We are interested in these modules to optimize the system simulation.

Running example. *In the transportation system of the footwear factory, we distinguish six different sequences defining the behavior of the whole sub-system from the six production stations to the storing and packing station.*

$$\forall j \in [1,3], Seq_j = cf(FB\_Belt_j), cf(FB\_Disc_1), cf(FB\_Belt_7), cf(FB\_Belt_8), cf(FB\_Belt_9)$$

$$\forall j \in [4,6], Seq_j = cf(FB\_Belt_j), cf(FB\_Disc_2), cf(FB\_Belt_7), cf(FB\_Belt_8), cf(FB\_Belt_9)$$

10.4.3 Deadline Processing

According to user requirements, each application sequence Seq of Control Function Modules should meet an end to end bound denoted by $bound(Seq)$. To meet all bounds, we process deadlines for the different modules of sequences and in particular for $Seq : cf_0,\ cf_1,\ ....,\ cf_{n-1}$ as follows:

- The deadline of the last module cf_{n-1} is:

$$d(cf_{n-1}) = bound(Seq)$$

- The deadline of any module $d(cf_j)$, $j \in [0, n-2]$ is processed so that its successors meet also their deadlines as follows, $\forall j \in [0, n-2]$,

$$d(cf_j) = \\ min_{s \in succ(cf_j), cf_k \in s}\{d(cf_k) - \sum_{cf_h \in s}^{d(cf_h) \leq d(cf_k)}(\theta(cf_h) + \theta(sifb_h) + \theta(processes_h))\}$$

Running example. *In the followed platform in Vigevano, the deadline* $d(cf(FB\_Disc1))$ *of* $cf(FB\_Disc1)$ *is processed so that its successor* $cf(FB\_Belt_7)$ *meets also its deadline:*

$$d(cf(FB\_Disc1)) = \\ d(cf(FB\_Belt7)) - \theta(cf(FB\_Belt7)) - \theta(SIFB(FB\_Belt7)) - \theta(FB\_Belt7)$$

10.5 Contribution: Characterization of Critical Scenarios

We define a critical scenario as a bad execution with the high probability to violate functional or temporal constraints. This scenario exists if a particular component *risks* in a first possible case to be blocked when it is waiting a response from a corresponding physical process whereas the buffer cannot receive new input events (i.e. the number of input events is higher than the buffer size) or it risks in a second possible case to violate the corresponding deadline.

10.5.1 Blocking Problem

We define a buffer of size m for a Control Component to store input events waiting their treatment by the corresponding CFM and VPM modules. A VPM module treats one input event at the same time, whereas a CFM module is allowed to treat new input events (according to the FIFO priority) while VPM is occupied by the treatment of a previous event. To avoid in the buffer any blocking problem which risks a non predictable behavior of the component, the treatment of m input events by cf should not be lower than the duration $\theta(vp)$. Otherwise, the component is completely saturated and cannot handle an additional input event. We formalize this problem as follows (figure 10.8),

$$m \ . \ \theta(cf) \geq \theta(vp) \Longrightarrow \text{No deadlock in } fb$$

Note finally that we are not interested in the probability computing of blocking problems but we aim to characterize critical execution scenarios of the system in order to optimize the discrete event simulation.

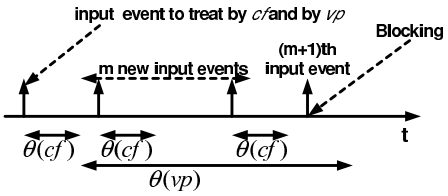

Figure 10.8: Characterization of a blocking problem.

10.5.2 Critical Scenarios

We define a critical scenario as any execution such that we risk to have a blocking problem or to violate temporal constraints. We formalize this scenario as follows,
$\exists l_j \in levels, \exists fb_h \in set_j, \exists cf_k \in cf(fb_h)$ such as,

* **First case:**The treatment of m input events in CC_h is lower than the response time of vp_k.

$$m \ . \ \theta(cf_k) \leq \theta(vp_k)$$

Note that we consider aperiodic input events. Therefore, even if this condition is verified, it does not imply automatically a blocking problem but it increases its probability to occur.

* **Second case:** the deadline risks to be violated because the treatment of cf_k and vp_k finishes in time (t denotes the time to start the execution of cf_k).

$$t + \theta(cf_k) + \theta(vp_k) = d(cf_k)$$

10.6 Contribution: Optimization of the Simulation

We optimize in our research work the simulation of a system by applying a fault injection technique to bring the execution of the corresponding components to critical scenarios in order to control the verification complexity. To take into account the application modularity, we propose an architecture of simulators to simulate the Control Components in the different levels of the corresponding hierarchy.

10.6.1 Architecture of Simulators

We use the *master − slave* architecture to implement the *discrete event simulator* where the master is the main module of the simulation. It is based on heuristics to bring the

whole application to critical scenarios, whereas each slave is a module corresponding to a unique level of the system's hierarchy. It is controlled by the master to apply the simulation strategies in the corresponding level.

Master Simulator

According to our formal characterization, $\theta(errors)$ is the only variable between all temporal parameters, therefore to bring the execution of a particular Control Component $CC \in set_j$ (located in the level l_j) to a critical scenario, the master simulator should act on the corresponding parameters $\theta(errors)$. These errors will be injected by the slave of the level l_j in the hardware components interacting with CC.

Slave Simulator

In the control system, each VPM module uses sensors, actuators and a communication network to control physical processes. To apply any simulation strategy fixed by the master, the slave injects errors in these hardware components such that their duration is the same fixed by the master. Note that all the slave simulators are running in parallel, they do not exchange data and their role is just to inject faults in hardware components of the corresponding levels.

Running example. *In the footwear system located in Vigevano, we show in Figure 10.9 the architecture of simulators composed of one master and three slaves by considering the system hierarchy composed of three levels. These slaves are running in parallel without any data exchange to inject faults in hardware components of the corresponding levels in order to apply the simulation strategies fixed by the master.*

10.6.2 Formalization

We formalize in this section the master and slave simulators to check the feasibility of hierarchical control systems. We define for each Control Function Module $cf \in cf\_modules$ the different values of the corresponding parameter $\theta(errors)$ in order to bring the system execution to critical scenarios. Moreover, we define for the different hardware components the possible values of errors to inject by the corresponding slave.

Formalization of the master simulator

We define at first time the possible values of errors to bring a particular control function to a critical scenario. We generalize thereafter the values of errors for the whole control functions implementing the system.

- **Simulation of a Control Function**

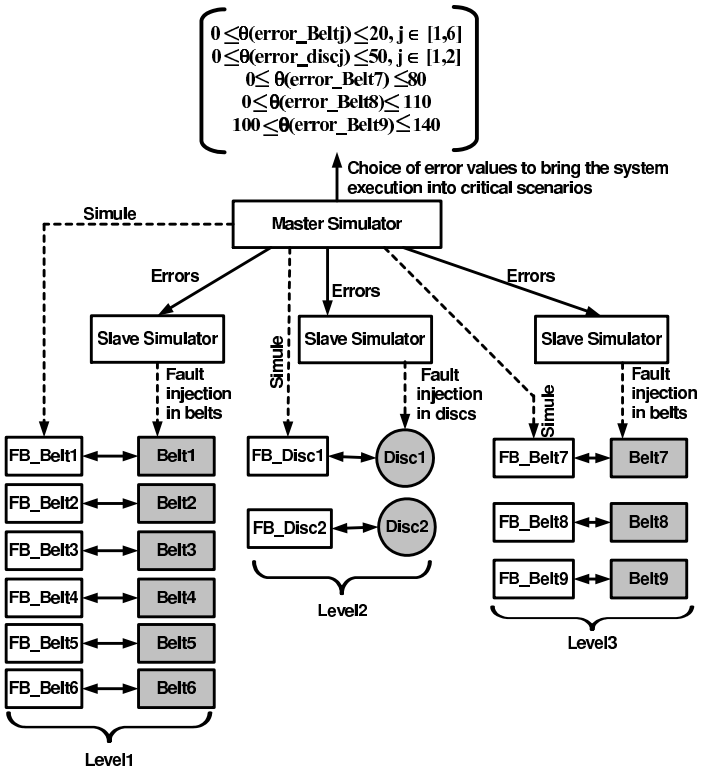

Figure 10.9: Architecture of the simulators implemented in the Vigevano platform (Italy).

During the simulation of a control function cf (with a corresponding virtual plant vp) belonging to a Control Component CC, the master should predict external errors to probably allow the occurrence of $m+1$ new input events waiting cf and vp modules. Therefore, to bring the execution of CC to a critical scenario, the duration of external errors should be bounded as follows,

$$m \ . \ \theta(cf) \text{ - } \theta(process) \text{ - } \theta(sifb)$$

$$\leq \theta(error) \leq$$

$$d(cf) - \theta(cf) - \theta(process) - \theta(sifb)$$

Where the upper bound represents a majoration to meet temporal properties and the lower bound a condition to have $m+1$ input events waiting their treatment. The master should test *all possible values* of $\theta(error)$ in order to create if possible critical execution scenarios of the system.

- **Generalization: Simulation of an Application**

  Let $max\_simulation$ be the duration of the simulation and $period\_error$ a constant. We process for each control function an interval of external errors to bring the corresponding component to critical scenarios.

  $\forall l_j \in Levels, \forall CC_h \in set_j, \forall cf_k \in cf(CC_h)$,

  $$m_h.\theta(cf_h) - \theta(SIFB) - \theta(process) \leq \theta(error) \leq d(cf_h) - \theta(cf_h) - \theta(process) - \theta(sifb)$$

  The master should choose new values of errors for the different control functions at each $\lfloor max\_simulation/period\_error \rfloor$ time unit(s).

Running example. Implementation of the Master Simulator

In the example of the footwear factory in Vigevano, we define the different values of errors to bring each Control Component (i.e. Function Block) to critical scenarios.

- *For each block FB_Beltj ($j \in [1,6]$), $\theta(error\_Beltj)$ should be in [0,20] to create a blocking problem or to violate a temporal property:*

  $$0 \leq \theta(error\_Beltj) \leq 20, (j \in [1,6])$$

- *For each block FB_Discj ($j \in [1,2]$) controlling a rotating disc, $\theta(error\_Discj)$ should be in [0,50] to bring the system to critical scenarios:*

  $$0 \leq \theta(error\_discj) \leq 50, (j \in [1,2])$$

- *For the Function Block FB_Belt7 controlling the belt that transports pieces from the rotating discs, $\theta(error\_Belt7)$ should be in [0,80] to create a blocking problem or to violate a temporal property:*

$$0 \leq \theta(error\_Belt7) \leq 80$$

- *For the block FB_Belt8 activated by FB_Belt7, $\theta(error\_Belt8)$ should be in [0,110] to bring the block to a blocking problem or to violate an end to end bound:*

$$0 \leq \theta(error\_Belt8) \leq 110$$

- *Finally, for the last block FB_Belt9 used to transport final products, $\theta(error\_Belt9)$ should be in [100,140] to possibly obtain a blocking problem or to violate a temporal property. Therefore, only 40 among 140 values of this parameter should be injected by the slave.*

$$100 \leq \theta(error\_Belt9) \leq 140$$

Formalization of a Slave Simulator.

In the simulator architecture, each slave applies the simulation strategies desired by the master. It injects errors in the hardware components used in the corresponding level. For each parameter $\theta(error)$ fixed by the master for a particular Control Function cf of the system, the slave should compose the correct combination of errors in the hardware components.

Notation. Let us denote respectively by $error(s)$, $error(a)$ and $error(net)$ the duration of an error occurring in a sensor s, an actuator a and a network net. In addition, let us denote respectively by $sensor(vp)$ and $actuator(vp)$ the set of sensors and actuators handled by the virtual plant vp.

We formalize the error injection applied by each slave as follows:
$\forall l_j \in Levels, \forall CC_h \in set_j, \forall vp_k \in vp(CC_h)$,

$$\sum_{s \in sensor(vp_k)} error(s) + \sum_{a \in actuator(vp_k)} error(a) + error(net) = \theta(error_k)$$

10.6.3 Implementation

We present in this section the implementation sketch of the master and slave (Table 10.1). The master function **Master_simul()** defines new simulation strategies. It is based on the function **Init()** that processes intervals of errors to be injected by the different slaves. The slave function **Activate_slave()** applies the master strategies by injecting errors in hardware components of the corresponding level.

Running example. Fault Injection Simulation.

Function : Master_simul()
Begin
Init($levels$);
For t from 0 to $max\_simulation$ step by $period\_error$
For each level $l_j \in levels$
For each $CC \in set_j$
For each $cf \in cf(CC)$
$\theta(error) \leftarrow \theta(error) + \lfloor (Imax(cf) - Imin(cf))/\lfloor max\_simulation/period\_error \rfloor \rfloor$
$Activate\_slave(l_j, cf, \theta(error))$;
End.
Function : Init($levels$)
Begin
For each level $l_j \in levels$
For each $CC \in set_j$
For each $cf \in cf(CC)$
$Imax(cf) \leftarrow d(cf) - \theta(cf) - \theta(SIFB) - \theta(process)$;
$Imin(cf) \leftarrow m * \theta(cf) - \theta(SIFB) - \theta(process)$;
$\theta(error) \leftarrow Imin(cf)$;
End.
Function : Activate_slave($l_j, cf, \theta(error)$)
//This code implements the slave simulator of the level l_j.
Begin
$vp \leftarrow VPM(cf)$;
If(network(vp))
/*vp sends messages on the network*/
$error(sensor(vp)) \leftarrow \theta(error)/3$;
$error(actuator(vp)) \leftarrow \theta(error)/3$;
$error(net(vp)) \leftarrow \theta(error)/3$;
$inject(sensor(vp), error(sensor(vp)))$;
$inject(actuator(vp), error(actuator(vp)))$;
$inject(net(vp), error(net(vp)))$;
Else
$error(sensor(vp)) \leftarrow \theta(error)/2$;
$error(actuator(vp)) \leftarrow \theta(error)/2$;
$inject(sensor(vp), error(sensor(vp)))$;
$inject(actuator(vp), error(actuator(vp)))$;
End.

Table 10.1: Implementation of Master and Slave simulators

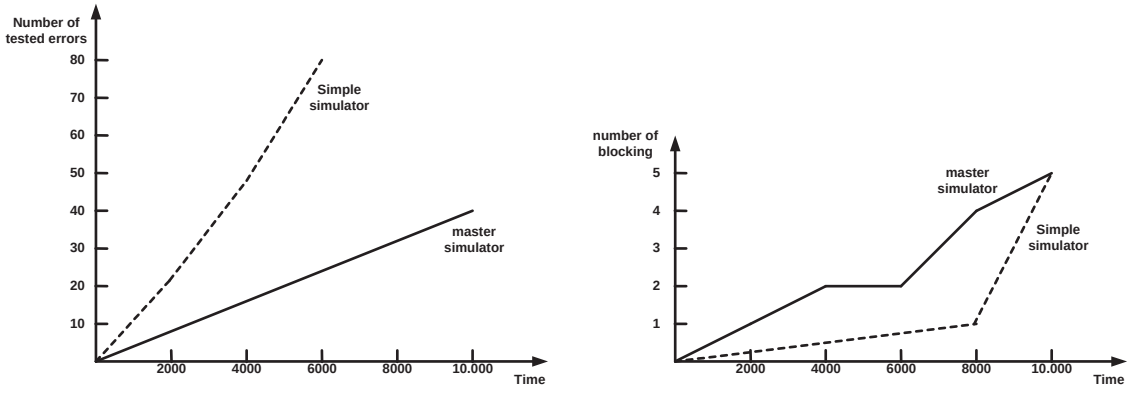

Figure 10.10: Comparison between the proposed simulator and a simple simulator.

In the footwear factory in Vigevano, we implemented all the transportation system in the Simulink Environment (Version 7.1) and applied the simulation in a PC of frequency $1.7GHZ$ *and* $RAM = 512MB$ *(Intel centrino). To analyze the performance of our master simulator based on the injection of errors that we carefully calculate to obtain critical scenarios, we implemented another simulator based on a Random algorithm to inject randomly errors in hardware components. By considering aperiodic input flow of shoes, we applied more than 100 simulation cases. We show in Figure 10.10 the most significant case that proves the advantage of our master simulator. According to this figure, the master should check only 40 values of* $\theta(error\_Belt9)$ *instead of 140 values when we apply a random simulator. Therefore, by verifying the system feasibility around critical scenarios, the master is able to detect as soon as possible any blocking problem (Figure 10.10), whereas the random simulator is not able because it injects randomly errors in hardware components.*

10.7 Evaluation of the Performance

In order to evaluate the performance of our master simulator, we apply the simulation for three cases : periodic, sporadic and aperiodic input events [112]. The best significant case that we found among 100 simulations is presented in Figure 10.11, where the vertical line represents the ideal simulation in which we detect 7 blocking problems at $t = 0$ but this simulation is an unreal and impossible case to reach. Let us denote respectively by *period*, *sporad* and *aperiod* the simulation curves corresponding to the periodic, sporadic and aperiodic input events and let us denote also by *Area* the rectangle area delimited by the ideal bound of simulation and the time $t = 10.000$. The performance of the master simulator for the different cases of periodic, aperiodic and sporadic input events is as follows,

$$performance_{period} = (\int_0^{10000} period(t).dt)/Area$$

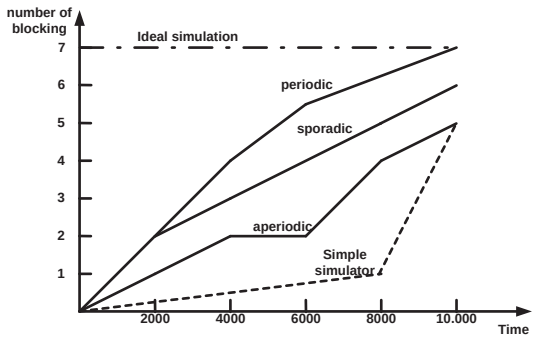

Figure 10.11: Evaluation of performance.

$$performance_{aperiod} = (\int_0^{10000} aperiod(t).dt)/Area$$

$$performance_{sporad} = (\int_0^{10000} sporad(t).dt)/Area$$

We define as follows the general performance $perf$ of our simulator by processing the average of these three performances:

$$perf = ((performance_{period} + performance_{aperiod} + performance_{sporad})/3) * 100$$

Running example: Numerical Results. *According to Figure 10.11, the performance of the master simulator is as follows,*

$$performance_{period} = 0.58;\ performance_{sporad} = 0.48$$

$$performance_{aperiod} = 0.32$$

$$perf = 46\%$$

The performance of our master-slave simulator is 46% *of the ideal simulator whereas the performance of the random simulator is* 16% *(e.g the master is 3 times more performant than the random simulator). This result presents the advantage of simulation that we propose.*

10.8 Conclusion

The chapter deals with optimal verifications of Control Components implementing an embedded control system after a particular reconfiguration scenario. To check functional and temporal properties, we are interested in the system simulation which is known as a non exhaustive method. We apply therefore a fault injection technique to improve its performance.

To achieve a feasible simulation, we define at first time a formalization of a Control Component by introducing two types of modules: the Control Function Module implementing the component behavior when a particular input event occurs and the Virtual Plant Module supporting interactions with physical processes. Therefore, our system is transformed to a system of Control Function Modules with precedence constraints. We propose a technique processing a deadline for each module in order to meet temporal properties. We define critical scenarios for the Control Function Modules to define worst case executions where blocking problems can occur or deadlines can be violated. Therefore, an interesting idea is to apply the simulation around these scenarios by injecting faults in physical processes interacting with Control Components. To consider the system' software design in different levels of a hierarchy, we propose a simulator architecture based on the master-slave model where a slave is affected to each level in order to inject errors in the corresponding hardware components, whereas the master chooses the best values of these errors to generate critical scenarios. We applied this approach to a sub-embedded system of the footwear factory in Vigevano and found interesting results proving the performance of the master that reaches 46% of the ideal simulator's performance.

10.9 References of the Chapter's Contributions

- M. Khalgui, E. Carpanzano, H-M Hanisch, **An Optimized Simulation of Component-based Embedded Systems In Manufacturing Industry**, International Journal of Simulation and Process Modelling. 2009.

Chapter 11

Conclusion

The manuscript deals with reconfigurable embedded control systems following the component-based approach to address new current requirements of our community. We define the general concept of Control Components to be assumed as software units controlling physical processes. A Control Component is composed of an interface for external interactions with the environment, and an implementation that supports its functional tasks. We define a classification of all possible reconfiguration forms that can be automatically applied at run-time: addition-removal of components, modification of their compositions or finally updates of data. We propose in addition a multi-agent architecture for such systems where a Reconfiguration Agent is affected to each device of the execution environment to handle local reconfigurations, and a Coordination Agent is proposed to coordinate between devices in order to guarantee coherent and feasible reconfigurations. We model the whole architecture according to the formalism Net Condition/Event Systems (NCES) and apply the model checker SESA to verify functional and temporal properties described in the temporal logic "Computation Tree Logic" CTL. We verify in addition each network of components that corresponds to a particular reconfiguration scenario of the system. A refinement-based approach is proposed to model and check network's components in step by step. Moreover, we define technical solutions to implement automatic reconfigurations of embedded control systems following the Industrial Technology IEC61499. To construct their operational architectures, we define an approach that generates the different execution models of reconfigurable embedded systems. In this case, different sets of feasible OS tasks are constructed to implement the system for different reconfiguration scenarios. We define in addition an approach proposing useful technical solutions for low power reconfigurations of embedded architectures. We propose in particular to reduce periods and execution times of tasks or also to remove some of them in order to consider limitations in embedded batteries. We define finally an approach allowing optimal simulations of embedded architectures where a solution is proposed to inject faults and to bring the system's behavior to critical executions. We developed prototypes encoding the contributions of this research work which is applied to different case studies, and is published in prestigious international journals.

In our future works, we plan to study the manual reconfigurations of embedded systems by using the UML language for the static redesign of components. This research work is

interesting in many industrial fields. We plan also to assume sporadic and aperiodic tasks for the generation of execution models of embedded systems. In this case, sporadic and aperiodic real-time recurring tasks should be constructed. We plan in addition to assume such tasks for low power reconfigurations of embedded systems. Finally, we plan to develop a complete environment that contains all our research works and that should be applied to real industrial platforms.

Bibliography

[1] B. Al-Hashimi. System-on-chip: next generation electronics. In *Institution of Engineering and Technology, ISBN-10: 0863415520*, 2006.

[2] Yazen Al-Safi and V. Vyatkin. An ontology-based reconfiguration agent for intelligent mechatronic systems. In *Third International Conference on Industrial Applications of Holonic and Multi-Agent Systems*. Springer-Verlag, 2007.

[3] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Sixth ACM Symposium on the Foundations of Software Engineering, pp. 175-188*, 1998.

[4] Rajeev Alur and Thomas A. Henzinger. Logics and models of real-time: A survey. In *Proceedings of Real-time: theory in practice, Volume 600 of Lecture Notes in Computer Science, Springer-Verlg*, 1992.

[5] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Mller, P. Pettersson, C. Weise, and W. Yi. *Uppaal - Now, Next, and Future.* In Proceedings of Modelling and Verification of Parallel Processes (MOVEP'2k), France. LNCS Tutorial 2067, pages 100-125, F. Cassez, C. Jard, B. Rozoy, and M. Ryan (Eds.), 2001.

[6] Ch. Angelov, K. Sierszecki, and N. Marian. Design models for reusable and reconfigurable state machines. In *L.T. Yang and All (Eds): EUC 2005, LNCS 3824, pp:152-163. International Federation for Information Processing.*, 2005.

[7] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. F. Fabre, J. C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation : a methodology and some

applications. In *IEEE transaction on software engineering. Vol 16, N2, pp. 166-182*, 1990.

[8] Artist-Project. *Roadmap : Component-based Design and Integration Platforms.* www.artist-embedded.org, 2003.

[9] E. Asarin, O. Bournez, T. Dang, and O. Maler. Approximate reachability analysis of piecewise-linear dynamical systems. In *Hybrid Systems: Computation and Control, Third International Workshop, LNCS*, 2000.

[10] Melhem R. Mosse D. Alvarez-P. M. Aydin, H. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of Real-Time Systems Symposium*, 2001.

[11] H. Aytug and C. Dogan. A framework and a simulation generator for kanban-controlled manufacturing systems. In *Computers and Industrial Engineering, Vol34, N2, pp. 337-350*, 1998.

[12] Andrea Ballarino and Emanuele Carpanzano. Modular automation systems design using the iec 61499 and the simulink/stateflow toolboxes. In *Proceedings of the 2002 Japan-USA Symposium of Flexible Automation*, 2002.

[13] Goossens J. Baruah, S. Scheduling real-time tasks: Algorithms and complexity. In *In Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Joseph Y-T Leung (ed). Chapman Hall/ CRC Press.3. (2004)*, 2004.

[14] S Baruah. Dynamic and static priority scheduling of recurring real-time tasks. In *Real-time Systems.* vol. 24, n 1, 2003.

[15] Sanjoy Baruah and Joel Goossens. Scheduling real-time tasks: Algorithms and complexity. In *Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Joseph Y-T Leung (ed). Chapman Hall/ CRC Press.3*, 2004.

[16] Bogliolo A. Micheli G. Benini, L. A survey of design techniques for system-level dynamic power management. In *IEEE Trans. VLSI Sys., 8(3). (2000) 299-316*, 2000.

[17] Bogliolo A. Paleologo G. Micheli-G. Benini, L. Policy optimization for dynamic power management. In *IEEE Trans. CAD and Sys., 18(6). (1999) 813-833*, 1999.

[18] A. Benso, P. L. Civera, M. Rebaudengo, M. Sonza Reorda, and A. Ferro. A hybrid fault injection methodology for real-time systems. In *The 28th Annual International Symposium of Fault-Tolerant Computing. Germany. pp. 74-75*, 1998.

[19] A. Benso, M. Rebaudengo, and M. S. Reorda. Fault injection for embedded microprocessor-based systems. In *Journal of Universal Computer Science. Vol.5, N. 10, pp. 693*, 1999.

[20] Mary Berna-koes, Illah Nourbakhsh, and Katia Sycara. Communication efficiency in multiagent systems. In *International Conference on Robotics and Automation, April 26 - May 1, 2004, pp 2129 2134, Vol.3*, 2003.

[21] A. Beugnard, J. M. Jzquel, and N. Plouzeau. *Making components contract aware.* IEEE Computer, 32(7) : 38-45, 1999.

[22] Robert W. Brennan, Martyn Fletcher, and Douglas H. Norrie. A holonic approach to reconfiguring real-time distributed control systems. In *Multi-Agent Systems and Applications: MASA'01.* Springer-Verlag, 2001.

[23] E. Carpanzano and A. Ballarino. A structured approach to the design and simulation-based testing of factory automation systems. In *International Symposium on Industrial Electronics, ISIE2002, Italy.*, 2002.

[24] E. Carpanzano, L. Ferrarini, and C. Maffezzoni. Modular testing of logic control functions with matlab. In *13th European Simulation Symposium and Exhibition.*, 2001.

[25] J. Carreira, H. Madeira, and J. Silva. Xception : Software fault injection and monitoring in processor functional units. In *Conference on dependable computing for critical applications, DCCA-5, USA*, 1995.

[26] Liang Chen, Jianming, and shuqing Wang. Scheduling and control co-design for delay compensation in networked control system. In *Asian Journal of Control, Vol.8, Num.2*, 2007.

[27] A. Chutinan and B. K. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In *Hybrid Systems: Computation and Control, Second International Workshop, LNCS*, 1999.

[28] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *In Proc. Workshop on Logic of Programs, LNCS 131, pages 5271. Springer-Verlag,, YEAR = 1981,.*

[29] E. Clarke, O. Grumberg, and D. Peled. Model checking. In *MIT Press*, 2000.

[30] E. Clarke and R. Kurshan. Computer-aided verification. In *IEEE Spectrum, 33(6)*, 1996.

[31] Daniel D. Corkill, Douglas Holzhauer, and Walter Koziarz. Turn off your radios! environmental monitoring using power-constrained sensor agents. In *In First International Workshop on Agent Technology for Sensor Networks (ATSN-07), Honolulu, Hawaii, pages 3138*, 2007.

[32] I. Crnkovic and M. Larsson. Building reliable component-based software systems. In *Artech House*. UK, 2002.

[33] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool kronos. In *Hybrid Systems III, Verification and Control, LNCS 1066, Springer-Verlag*, 1996.

[34] T.A. Delong, B. W. Johnson, and J. A. Profeta. A fault injection technique for vhdl behavioral-level models. In *IEEE Design and Test of Computers*, 1996.

[35] G. Dukas and K. Thramboulidis. *A Real-Time Linux Execution Environment for Function-Block Based Distributed Control Applications.* 2nd IEEE International Conference on Industrial Informatics. INDIN'04, 2005.

[36] L. Ferrarini and C. Veber. *Implementation approaches for the execution model of IEC 61499 applications.* 2me IEEE International Conference on Industrial Informatics. Germany. INDIN'04, 2004.

[37] Navet N. Gaujal, B. Dynamic voltage scaling under edf revisited, real-time systems. In *Springer Verlag, 37(1), Some results are available as research report INRIA RR-5125. (2007) 77-97*, 2007.

[38] A-L. Gehin and M. Staroswiecki. Reconfiguration analysis using generic component models. In *IEEE Transactions on Systems, Machine and Cybernetics, Vol.38, N.3*, 2008.

[39] G Goessler and J Sifakis. Composition for component-based modeling. In *Proceedings of FMCO'02, the Netherlands, LNCS 2852, pages 443-466.*, 2002.

[40] Mikell P. Groover. Automation, production systems, and computer-integrated manufacturing. In *Prentice Hall, 3 edition, ISBN-10: 0132393212*, 2007.

[41] Sibsankar Haldar and Alex A. Aravind. Operating systems. In *Pearson Education; 1 edition*, 2010.

[42] H-M. Hanisch and A. Luder. Modular modelling of closed-loop systems. In *Colloquium on Petri Net Technologies for Modelling Communication Based Systems, pp. 103-126,Germany,*, 1999.

[43] T. A. Henzinger, P. Ho, and H. Womg-Toi. Hytech: the next generation. In *TACAS95: Tools and Algorithms for the Construction and Analysis of Systems, LNCS*, 1997.

[44] G. Holzmann. The model checker spin. In *IEEE Transactions on Software Engineering, 23(5)*, 1997.

[45] Potkonjak M. Srivastava M B. Hong, I. On-line scheduling of hard real-time tasks on variable voltage processor. In *Proceedings of ICCAD. (1998) 653-656*, 1998.

[46] Qu G. Potkonjak M. Srivastava-M B. Hong, I. Synthesis techniques for low-power hard real-time systems on variable voltage processors. In *Proceedings of RTSS. (1998) 178-187*, 1998.

[47] P-A. Hsiung, Y-R. Chen, and Y-H. Lin. Model checking safety-critical systems using safecharts. In *IEEE Transactions on Computers, vol.56, no.5*, 2007.

[48] M-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. In *IEEE Computer Journal. Vol.30, N.4, pp : 75-82*, 1997.

[49] Kamal Hyder and Bob Perrin. Embedded systems design using the rabbit 3000 microprocessor, interfacing, networking, and application development (embedded technology). In *Publisher: Newnes, ISBN-10: 0750678720*, 2004.

[50] International Standard IEC61499. Industrial process measurements and control systems. In *International Electrotechnical Commission (IEC) Committee Draft*, 2004.

[51] Yasuura H. Ishihara, T. Voltage scheduling problem for dynamically variable voltage processors. In *ISLPED. (1998) 197-202*, 1998.

[52] A. Lobov J-L-M. Lastra, L. Godinho and R. Tuokko. An iec 61499 application generator for scan-based industrial controllers. 3nd IEEE International Conference on Industrial Informatics. INDIN'05, 2005.

[53] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari : a flexible software-based fault and error injection system. In *IEEE transaction on computers. Vol 44, N. 2, pp. 248-260*, 1995.

[54] M. Khalgui. A deployment methodology of real-time industrial control applications in distributed controllers. In *International Journal of Computers in Industry. Vol59, N.5*, 2008.

[55] M. Khalgui. Nces-based modelling and ctl-based verification of reconfigurable embedded systems in manufacturing industry. In *International Journal of Computers in Industry*, 2010.

[56] M. Khalgui, E. Carpanzano, and H-M Hanisch. An optimized simulation of component-based embedded systems in manufacturing industry. In *International Journal of Simulation and Process Modelling*, 2009.

[57] M. Khalgui and H-M. Hanisch. A formal approach to check and assign reconfigurable control applications into programmable logic controllers. In *Asian Journal of Control. Vol.11, N.3*, 2009.

[58] M. Khalgui and H-M. Hanisch. Automatic nces-based specification and sesa-based verification of feasible control tasks in benchmark production systems. In *International Journal of Modelling, Identification and Control*, 2010.

[59] M. Khalgui and H-M. Hanisch. Reconfiguration of distributed embedded control systems. In *IEEE Transactions on Systems, Machine and Cybernetics, Part A, (**Accepted** to be published in 2011)*, 2011.

[60] M. Khalgui and O. Mosbahi. Intelligent distributed control systems. In *Information and Software Technology*, 2011.

[61] M. Khalgui, O. Mosbahi, H-M Hanisch, and Z. Li. Implementation of agent-based reconfigurable embedded control systems. In *IEEE Transactions on Mechatronics, (**Accepted**),*, 2011.

[62] M. Khalgui, O. Mosbahi, Z Li, and H-M. Hanisch. Development of agent-based reconfigurable embedded control systems: From modelling to implementation. In *IEEE Transactions on Computers, (**Accepted** to be published in 2011)*, 2011.

[63] M Khalgui, X Rebeuf, and F Simonot-Lion. A behavior model for iec 61499 function blocks. In *Thrid International Workshop on Modelling of Objects, Components, and Agents (MOCA04)*. Denmark, 2004.

[64] M Khalgui, X Rebeuf, and F Simonot-Lion. Component-based deployment of industrial control systems: an hybrid scheduling approach. In *11th IEEE International Conference on Emerging Technology and Factory automation (ETFA06)*. Czech Republic, 2006.

[65] M. Khalgui and K. Thramboulidis. An iec61499-based development approach for distributed industrial control applications. In *International Journal of Modelling, Identification and Control. Vol5, N.1*, 2008.

[66] Mosbahi O. Zhiwu Li. Hans-Michael Hanisch. Khalgui, M. Reconfigurable multi-agent embedded control systems: From modelling to implementation. In *IEEE Transactions on Computers. (2010)*, 2010.

[67] Kim J. Min S. L. Kim, W. A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. In *Proceedings of Design, Automation and Test in Europe*, 2002.

[68] M H Klein, T Ralya, B Pollack, R Obenza, and M G Harbour. A practioner's handbook for real-time analysis guide to rate monotonic analysis for real-time systems. In *Kluwer Academic Booktitle*, 1993.

[69] O. Labarthe, A. Ferrarini, B. Espinsse, and B. Montreuil. Multi-agent modelling for simulation of customer-centric supply chain. In *International Journal of Simulation and Process Modelling, Vol.2, N. 3/4, pp. 150-163*, 2006.

[70] S. Lee. Automatic generation of simulation model for shop floor control system. In *Master's thesis, Postech*, 1996.

[71] J Leung and J Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. In *Real-time tasks, performance Evaluation.* vol. 2, 1982.

[72] ed Levine, William S. The control handbook. In *New York: CRC Press. ISBN 978-0-849-38570-4*, 1996.

[73] R Lewis. Modelling control systems using iec 61499. In *Institution Of Engineering and Technology.* UK, 2001.

[74] R. Lewis. *Modelling control systems using IEC 61499: Applying Function Blocks to distributed systems.* IEE Control Engineering Series 59. The institution of Electrical Engineers, 2001.

[75] T. Licht, L. Schmidt, C.m. Schlick, L. Dohmen, and H. Luczak. Person-centred simulation of product development processes. In *International Journal of Simulation and Process Modelling, Vol.3, N. 4, pp. 204-218*, 2007.

[76] L. Layland J W. Liu, C. Scheduling algorithms for multiprogramming in a hard real time environment. In *J. ACM, 20. (1973) 46-61*, 1973.

[77] T. Lovric. Processor fault simulation with profi. In *European Simulation Symposium ESS95*, 1995.

[78] Chung EY. Simunic T. Micheli-G De. Benini L. Lu, YH. Quantitative comparison of power management algorithms. In *Design, Automation and Test in Europe Conference and Exhibition. (2000) 20-26*, 2000.

[79] E. Carpanzano M. Colla and A. Brusafferri. *Applying the IEC 61499 Model to the Schoe Manufacturing Sector*. 11th IEEE International Conference on Emerging Technologies and Factory Automation. ETFA'06, 2006.

[80] Lu Ma and J.J.P Tsai. Formal modeling and analysis of a secure mobile-agent system. In *IEEE Transactions on Systems, Machine and Cybernetics, Vol.38, N.1*, 2008.

[81] R. Mailler and V. Lesser. A mediation based protocol for distributed constraint satisfaction. In *The Fourth International Workshop on Distributed Constraint Reasoning, pp. 49-58*, 2003.

[82] R. Mailler and V. Lesser. A cooperative mediation-based protocol for dynamic, distributed resource allocation. In *IEEE Transaction on Systems, Man, and Cybernetics, Part C, Special Issue on Game-theoretic Analysis and Stochastic Simulation of Negotiation Agents, Volume 36, Number 1, IEEE Press, pp. 80-91*, 2006.

[83] R. Mailler, V. Lesser, and Bryan Horling. Cooperative negotiation for soft real-time distributed resource allocation. In *Proceedings of Second International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2003), ACM Press, pp. 576-583*, 2003.

[84] V. Marik, V. Vyatkin, and A. Colombo. Holonic and multi-agent systems in manufacturing. In *Proceedings of HoloMAS'07 conference. Lecture Notes in Computer Science, Vol. 4659, Springer Verlag*, 2007.

[85] G. Micheli and L. Benini. System-level low power optimization: Techniques and tools. In *Trans. Design Auto. of Electr. Sys., 5(2)*, 2000.

[86] I. Mitchell and C. Tomlin. Level set methods for computation in hybrid systems. In *Hybrid Systems: Computation and Control, Third International Workshop, LNCS*, 2000.

[87] Petr Novak, Milan Rollo, Jiri Hodik, and Tomas Vlcek. Communication security in multi-agent systems. In *Multi-Agent Systems and Aplications III*, pages 454–463. Springer-Verlag, 2003.

[88] P.M. Oldfather, A.S. Ginsberg, and H.M. Markowitz. Programming by questionnaire: how to construct a program generator. In *Rand Report, RM-5129-PR*, 1966.

[89] M. Pedram. Power minimization in ic design: principles and applications. In *ACM Transactions on Design Automation of Electronic Systems (TODAES) 1(1). (1996) 56-66*, 1996.

[90] Shin K. G. Pillai, P. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2001.

[91] J. Pitt and A. Mamdani. Communication protocols in multi-agent systems: A development method and reference architecture. In *In F. Dignum and M. Greaves (eds.), Issues in Agent Communication, LNAI1916, pp160-177, Springer Verlag*, 2000.

[92] Wu Q. Pedram M. Qiu, Q. Dynamic power management of complex system using generalized stochastic petrinets. In *DAC. (2000) 352-356*, 2000.

[93] Hu. X. Quan, G. Minimum energy fixed-priority scheduling for variable voltage processors, design, automation and test. In *In: Europe Conference and Exhibition. (2002) 782-787*, 2002.

[94] Hu X S. Quan, G. Energy efficient fixed-priority scheduling for real-time systems on voltage variable processors. In *Design Automation Conference. (2001) 828-833*, 2001.

[95] Pedram M. Rabaey, J. Low power design methodologies. In *Kluwer*, 1996.

[96] Gupta R. Ramanathan, D. System level online power management algorithms. In *DATE. (2000) 606-611*, 2000.

[97] M. Rausch and H-M. Hanisch. Net condition/event systems with multiple condition outputs. In *Symposium on Emerging Technologies and Factory Automation. Vol.1, pp.592-600.*, 1995.

[98] Wolfgang Reisig. Petri nets: an introduction. In *Springer-Verlag New York, Inc, ISBN:0-387-13723-8, 161 pages*, 1985.

[99] S. Roch. Extended computation tree logic. In *Proceedings of the CESP2000 Workshop, number 140in Informatik Berichte, pages225-234, Germany*, 2000.

[100] S. Roch. Extended computation tree logic: Implementation and application. In *Proceedings of the AWPN2000 Workshop, Germany*, 2000.

[101] Martijn N. Rooker, Christoph Sunder, Thomas Strasser, Alois Zoitl, Oliver Hummer, and Gerhard Ebenhofer. Zero downtime reconfiguration of distributed automation systems : The εcedac approach. In *Third International Conference on Industrial Applications of Holonic and Multi-Agent Systems*. Springer-Verlag, 2007.

[102] A. Sarvar. *cmos* power consumption and c_{pd} calculation. In *Texas Instruments*, 1997.

[103] SESA. Signal/net system analyzer. In *http://www.ece.auckland.ac.nz/ vy-atkin/tools/modelchekers.html*, 2008.

[104] Choi K. Shin, Y. Power conscious fixed priority scheduling for hard real-time systems. In *ACM. In: 36_{th} Design Automation Conference. (1999) 134-139*, 1999.

[105] M. Sierhuis, W. J. Clancey, and R. J. J. Van Hoof. Brahms: a multi-agent modelling environment for simulating work processes and practices. In *International Journal of Simulation and Process Modelling, Vol.3, N.3, pp. 134-152*, 2007.

[106] M. Sims, D. Corkill, and V. Lesser. Automated organization design for multi-agent systems. In *Autonomous Agents and Multi-Agent Systems, Volume 16, Number 2, Springer-Netherlands, pp. 151-185*, 2008.

[107] Legrand J. Nana L. Marce-L. Singhoff, F. A holonic approach to reconfiguring realtime distributed control systems. In *Cheddar: A flexible real time scheduling framework, Atlanta, GA, United states, Association for Computing Machinery. (2004) 1-8*, 2004.

[108] Young Jun Son and Richard A. Wysk. Automatic siultion model generation for simulation-based, real-time shop floor control. In *Computers in Industry, Vol.45, pp. 291-308*, 2001.

[109] Young Jun Son, Richard A. Wysk, and Albert T. Jones. Simulation-based shop floor control: formal model, model generation and control interface. In *IIE Transactions, Vol.35, pp.29-48*, 2003.

[110] Chandrakasan A. Brodersen R. Srivastava, M. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. In *IEEE Trans. VLSI Sys., 4. (1996) 42-55*, 1996.

[111] William Stallings. Operating systems: Internals and design principles. In *Prentice Hall; 4 edition*, 2000.

[112] J Stankovic, M Spuri, and K Ramamritham. Deadline scheduling for real-time systems. In *Booktitle: Kluwer Academic Booktitles*, 2005.

[113] G. Subramaniam and A. Gosavi. Simulation-based optimisation for material dispatching in vendor-managed inventory systems. In *International Journal of Simulation and Process Modelling, Vol. 3, N. 4, pp. 238-245*, 2007.

[114] C. Szyperski. *Component Software : Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley. New York, 1998.

[115] H Takada and K Sakamura. mu-itron for small-scale embedded systems. In *IEEE MICRO.* vol. 15, n6, 1995.

[116] K. Thramboulidis, G. Doukas, and A. Frantzis. Towards an implementation model for fb-based reconfigurable distributed control applications,. In *Proceedings of 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp. 193-200*, 2004.

[117] K. Thramboulidis and G. Dukas. *IEC61499 execution model semantics*. International Conference on Industrial Electronics, Technology and Automation. CISSE-IETA'06, 2006.

[118] K. Thramboulidis and A. Zoupas. *Real-Time Java in Control and Automation*. 10th IEEE International Conference on Emerging Technologies and Factory Automation. ETFA'05, 2005.

[119] Function-Block-Run time Toolkit. Rockwell automation. http://www.holobloc.com, 2007.

[120] M. Vardi and P. Wolper. Reasoning about infinite computations. In *Information and Computation, 115(1)*, 1994.

[121] V. Vyatkin. Iec61499 function blocks for embedded and distributed control systems design. In *Book of ISA-o3neida series, Instrumentation, Systems and Automation society*, 2007.

[122] Demers-A.-Shenker S. Yao, F. A scheduling model for reduced cpu energy. In *Proceedings of IEEE annual foundations of computer science. (1995) 374-382*, 1995.

[123] L. T. Young, R. Iyer, and K. K. Goswami. A hybrid monitor assisted fault injection experiment. In *Conference on dependable computing for critical applications, DCCA-3. pp. 163-174*, 1993.

[124] Kim. J. Yun, H. On energy-optimal voltage scheduling for fixed-priority hard real-time systems. In *ACM Transactions on Embedded Computing Systems(TECS) 2(3). (2003) 393-430*, 2003.