Combining Model-Based Engineering in IEC 61499 with Contracts for Cyber-Physical Systems Monitoring



Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Carl von Ossietzky Universität Oldenburg

Dissertation zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

von

M. Sc. Duc Do Tran

geboren am 09. Juli 1985 in Ha Nam, Vietnam

Oldenburg 2024

Gutachter:
Prof. Dr.-Ing. Wolfgang Nebel

Weiterer Gutachter:
Prof. Dr. Alios Zoitl

Tag der Disputation:
01.04.2026

Eigenständigkeitserklärung

Hiermit erkläre ich an Eides statt, dass ich diese Dissertationsschrift selbstständig und ohne fremde unzulässige Hilfe und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe.

Der Inhalt der Dissertation wurde nicht schon überwiegend für eine eigenen Bachelor-, Master-, Diplom- oder ähnliche Prüfungsarbeit verwendet.

Ich erkläre weiter, dass mir die Grundsätze zur Sicherung guter wissenschaftlicher Praxis an der Carl von Ossietzky Universität Oldenburg bekannt sind und dass ich die dort niedergelegten Prinzipien wissenschaftlicher Arbeit und Veröffentlichung befolgt habe.

Ich habe im Zusammenhang mit dem Promotionsvorhaben keine kommerziellen Vermittlungs- oder Beratungsdienste (Promotionsberatung) in Anspruch genommen.

Oldenburg, den *03.06.2025* *Duc Do Tran*

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Prof. Dr.-Ing. Wolfgang Nebel, a distinguished professor at the Department of Information Technology at Carl von Ossietzky University of Oldenburg, and affectionately known as the "*Vater der Informatik in Oldenburg*." It is with great pride and appreciation that I acknowledge him as my *Doktorvater* for this dissertation. I am sincerely grateful for his outstanding mentorship, which has provided me not only with invaluable technical knowledge and experience but also with insightful questions and perspectives that have significantly shaped this work. Beyond his academic expertise, his warm and supportive nature, together with his constant encouragement, has motivated me to strive for the highest quality in this dissertation.

I would also like to extend my heartfelt thanks to my second reviewer, Prof. Dr. Alios Zoitl, a distinguished professor at the LIT Cyber-Physical Systems Lab at Johannes Kepler University Linz, Austria. Prof. Zoitl has made significant contributions to the development of the IEC 61499 standard and the 4diac development framework. His extensive expertise in distributed control systems and Cyber-Physical Systems (CPS) has been highly valuable to my research. I am particularly honored that he agreed to serve as a reviewer of this thesis, and it is a privilege to have him as a member of the evaluation committee.

My sincere gratitude also goes to Dr. rer. nat. Kim Grüttner, who served as my second supervisor. He played a crucial role during the early stages of my work in the R&D Group Hardware/Software Design Methodology at OFFIS. He introduced me to new perspectives on the research topic and consistently provided valuable support, whether through scientific discussions, reports, workshop presentations, or publications. His guidance was essential in keeping me on track throughout the dissertation process.

I am also deeply grateful to Dr. rer. nat. Frank Oppenheimer, who has been an important mentor and companion throughout my work at OFFIS. He provided insightful feedback, constructive criticism, and timely encouragement that kept me

motivated. His friendly demeanor and patience in our discussions, both within and beyond work, have been a constant source of support.

I would also like to express my appreciation to Dr.-Ing. Jörg Walter, my manager and colleague in the Distributed Computing and Communication group at OFFIS. He has been a continuous source of guidance since the beginning of my work there. He supported me in applying IEC 61499 concepts to practical applications and contributed significantly to reviewing my scientific reports and conference presentations.

I would also like to thank my former colleagues from the R&D Group Hardware/Software Design Methodology and the R&D Group Distributed Computing and Communication at OFFIS. I am grateful for your continuous support, constructive feedback, and for being such wonderful colleagues. It has been a pleasure and an honor to work with you.

Last but not least, I would like to express my deepest gratitude to my family, whose love and encouragement have been my strongest support throughout this journey. To my parents, for their unwavering belief in me. To my wife, Nguyen Thi Huong Thuy, for her endless support, patience, and encouragement, and for always standing by my side through both challenges and successes. To my children, Tran Quoc Tuan (Leo) and Tran Leonie Thuy Duong, who have been a constant source of inspiration and motivation in both my professional and personal life.

Thank you all for being with me every step of the way.

Abstract

In the traditional industrial sector, automation systems are typically used to control technological processes, where programmable logic controllers (PLCs) are usually used to run control programs. A PLC conforms to the IEC 61131-3 [1] standard. While PLCs are still used as reliable and powerful automation controllers in more advanced industrial environments, they are not considered suitable to support complex functional architectures of "intelligent systems" because the automation languages of the IEC 61131 standard are obsolete from a software engineering point of view.

Furthermore, the goal of Industry 4.0 is to achieve flexible automation and digital transformation solutions in modern manufacturing through the development of new control systems based on distributed architectures and self-determination capabilities. In this respect, traditional PLC controllers cannot meet the above requirements. [2, 3]. Another drawback of PLCs is that they prevent real interoperability, since their implementation depends on separate "dialects" of the vendors [4].

The term Cyber-Physical Systems (CPS) was originally initiated in the computer sciences domain. It first refers to embedding computing into physical processes. In CPS, embedded computers and networks monitor and control physical processes. There are usually feedback loops where physical processes affect computations and vice versa [5]. Today, the term CPS has become omnipresent, it is understood in different ways in different application domains [6]. We can find this term in various application domains, such as healthcare, energy and utilities, smart cities and communities, manufacturing, aerospace, automotive, chemical processes, civil infrastructure, transportation, entertainment, and consumer appliances [7].

In manufacturing, CPSs are mainly related to distributed real-time control systems and distributed networks. Based on real-time distributed intelligence, they can provide and support capabilities to monitor and control entities in the physical world [6]. They are expected to have self-configuring, self-regulating, self-monitoring and self-optimising capabilities. However, developing for CPSs also presents a number of important challenges that need to be addressed, such as complexity, independent development, heterogeneous component interaction, specification and integration testing.

The need for more advanced modelling frameworks to deal with the complexity and richness of heterogeneous components in a CPS model has been highlighted

in published research. The IEC 61499 standard [8] is an extension of the IEC 61131 standard that provides an open, platform-independent framework for the development of distributed control systems [9]. Other important features of IEC 61499 include full object orientation, event-driven execution behaviour and vendor independence. As a result, this technology is ideally suited for the modeling and simulation of distributed systems, with the goal of replacing PLC in the development of next-generation industrial automation systems.

On the other hand, model-based development for CPSs is often based on the V-model during the design process. A system is associated with system-level requirements. These are then decomposed into subsystems with subsystem-level requirements during the design phase. Due to the complex requirements for the design of CPSs that can involve the fusion of multiple viewpoints/aspects, the elements/components coming from different vendors, developed by different teams, using different design tools and methods, leading to interoperability and integration problems. Contract-Based Design (CBD) is a suitable framework to support the specification of a model-based design process that provides the concepts to overcome the above challenges and problems. CBD provides formal support for decomposing a system into subsystems in a hierarchical structure system [10]. It is also able to address the integration and composition of heterogeneous components in an appropriate way. The assume-guarantee (A/G) contract term initiates in the context of compositional assume-guarantee reasoning for software verification. Within contracts, the complexity of design and verification can be reduced by decomposing activities, under a set of assumptions [11].

Even though IEC 61499-compliant software tools provide model-based testing of applications before and after deployment, a systematic specification of behavioral and timing requirements and its monitoring at run-time is not well supported. Thus, in this thesis, the contracts will be used to formally specify the requirements and monitor the correctness of the behavioral and timing aspects associated with various components in a distributed system. The goal of this work is the combination of IEC 61499 as distributed system component model with a contract-based behavior and timing specification and testing approach to enable correction by construction control systems engineering. Furthermore, this work is an important cornerstone to enable run-time verification in a continuous development cycle for future industrial automation and control systems. Finally, we will demonstrate how our approach can be used to check whether behavioral and timing requirements are met. This can be done during integration testing of an IEC 61499 distributed system model at design-time and, in future work, after being deployed on distributed hardware resources at run-time.

Zusammenfassung

Im traditionellen industriellen Sektor werden Automatisierungssysteme typischerweise zur Steuerung technologischer Prozesse eingesetzt, wobei Programme in der Regel auf speicherprogrammierbaren Steuerungen (SPS) ausgeführt werden. Eine SPS entspricht dem Standard IEC 61131-3 [1]. Obwohl SPSen auch in fortgeschritteneren industriellen Umgebungen weiterhin als zuverlässige und leistungsfähige Automatisierungscontroller eingesetzt werden, gelten sie als ungeeignet zur Unterstützung komplexer funktionaler Architekturen sogenannter „intelligenter Systeme“, da die Automatisierungssprachen des IEC-61131-Standards aus softwaretechnischer Sicht veraltet sind.

Darüber hinaus verfolgt Industrie 4.0 das Ziel, flexible Automatisierung und digitale Transformationslösungen in der modernen Fertigung zu realisieren, insbesondere durch die Entwicklung neuer Steuerungssysteme auf Basis verteilter Architekturen und selbstbestimmter Fähigkeiten. In diesem Zusammenhang können traditionelle SPS-basierte Steuerungen die genannten Anforderungen nicht erfüllen [2, 3]. Ein weiterer Nachteil von SPSen besteht darin, dass sie eine echte Interoperabilität erschweren, da ihre Implementierungen auf herstellerspezifischen „Dialekten“ basieren [4].

Der Begriff der Cyber-Physical Systems (CPS) stammt ursprünglich aus der Informatik und bezeichnet die Einbettung von Rechensystemen in physikalische Prozesse. In CPS überwachen und steuern eingebettete Rechner und Netzwerke physikalische Abläufe. Typischerweise existieren Rückkopplungsschleifen, bei denen physikalische Prozesse die Berechnungen beeinflussen und umgekehrt [5]. Heute ist der Begriff CPS allgegenwärtig und wird in unterschiedlichen Anwendungsdomänen verschieden interpretiert [6]. Beispiele hierfür sind unter anderem das Gesundheitswesen, Energie- und Versorgungsnetze, Smart Cities, Fertigung, Luft- und Raumfahrt, Automobiltechnik, chemische Prozesse, Infrastruktur, Transport, Unterhaltung sowie Konsumgüter [7].

Im Fertigungsbereich stehen CPS vor allem im Zusammenhang mit verteilten Echtzeit-Steuerungssystemen und Netzwerken. Auf Basis verteilter Echtzeitintelligenz ermöglichen sie die Überwachung und Steuerung physischer Prozesse [6]. Von ihnen wird erwartet, dass sie über Selbstkonfigurations-, Selbstregelungs-, Selbstüberwachungs- und Selbstoptimierungsfähigkeiten verfügen. Gleichzeitig

stellt die Entwicklung von CPS eine Reihe bedeutender Herausforderungen dar, darunter steigende Komplexität, unabhängige Entwicklung von Komponenten, Interaktionen zwischen heterogenen Systemen sowie Probleme bei Spezifikation und Integrationstests.

Der Bedarf an fortgeschrittenen Modellierungsansätzen zur Bewältigung der Komplexität und Heterogenität von CPS wird in der Literatur vielfach hervorgehoben. Der Standard IEC 61499 [8] stellt eine Erweiterung von IEC 61131 dar und bietet ein offenes, plattformunabhängiges Rahmenwerk für die Entwicklung verteilter Steuerungssysteme [9]. Zu den wesentlichen Eigenschaften von IEC 61499 zählen vollständige Objektorientierung, ereignisgesteuertes Ausführungsverhalten sowie Herstellerunabhängigkeit. Damit eignet sich dieser Ansatz besonders gut für die Modellierung und Simulation verteilter Systeme und wird als Grundlage für zukünftige industrielle Automatisierungssysteme angesehen.

Die modellbasierte Entwicklung von CPS orientiert sich häufig am V-Modell. Dabei werden zunächst Systemanforderungen definiert und anschließend in Anforderungen auf Subsystemebene verfeinert. Aufgrund der Komplexität von CPS – insbesondere durch die Integration unterschiedlicher Perspektiven, Komponenten verschiedener Hersteller sowie heterogener Entwicklungswerkzeuge – entstehen erhebliche Herausforderungen hinsichtlich Interoperabilität und Integration. Contract-Based Design (CBD) stellt hierfür einen geeigneten Ansatz dar, da es formale Konzepte zur strukturierten Zerlegung von Systemen sowie zur Integration heterogener Komponenten bereitstellt [10]. Insbesondere das Assume-Guarantee-Prinzip ermöglicht eine systematische Reduktion der Komplexität durch die Definition von Annahmen und Garantien [11].

Obwohl IEC-61499-konforme Softwarewerkzeuge modellbasierte Tests von Anwendungen vor und nach der Inbetriebnahme unterstützen, ist eine systematische Spezifikation von Verhaltens- und Zeitanforderungen sowie deren Laufzeitüberwachung bislang nur unzureichend realisiert. In dieser Arbeit werden daher Verträge eingesetzt, um Anforderungen formal zu spezifizieren und die Korrektheit von Verhaltens- und Zeitaspekten in verteilten Systemen zu überwachen. Ziel dieser Arbeit ist die Kombination von IEC 61499 als Komponentenmodell für verteilte Systeme mit einem vertragsbasierten Ansatz zur Spezifikation und Verifikation von Verhalten und Zeit, um ein “Correctness-by-Construction”-Engineering für Steuerungssysteme zu ermöglichen. Darüber hinaus bildet diese Arbeit eine wichtige Grundlage für die Laufzeitverifikation in einem kontinuierlichen Entwicklungsprozess zukünftiger industrieller Automatisierungs- und Steuerungssysteme. Abschließend wird gezeigt, wie der vorgeschlagene Ansatz eingesetzt werden kann, um zu überprüfen, ob Verhaltens- und Zeitanforderungen eingehalten werden – sowohl während der Integrationsphase zur Entwurfszeit als auch perspektivisch im laufenden Betrieb auf verteilten Hardware-Ressourcen.

Contents

List of Figures **xxi**

1 Introduction **1**
1.1 Motivation 1
1.2 Solution Approach for Systems Engineering 4
1.2.1 Design methods 4
1.2.2 Design architecture and framework 6
1.2.3 Requirements and assumptions for design methodology . . . 6
1.3 Research questions 7
1.4 Organization 8

2 Technological background **11**
2.1 IEC 61499 standard 11
2.1.1 IEC 61499 function block 12
2.1.1.1 Basic Function Block 13
2.1.1.2 Service Interface Function Block 14
2.1.1.3 Composite Function Block 15
2.1.2 Application and System model 15
2.1.3 Building and modeling distribution system 17
2.1.4 Development tools and runtime environments for IEC 61499 compliant applications 18
2.1.4.1 FBDK 18
2.1.4.2 4diac 18
2.1.4.3 ISaGRAF Workbench 19
2.1.4.4 NxtStudio 20
2.2 Assume/Guarantee contract 21
2.2.1 Overview of contract concept 21
2.2.2 Properties, operators, and algebra of contract 22
2.2.3 Requirement formalization, contract specification, and support tools 25

3 State of the art **29**
3.1 Model-based design: Methods and tools for CPSs 30
3.2 Modeling and controlling of CPSs using IEC-61499 32
3.3 Contracts for Safety-critical CPS 34
3.4 Summary 36
3.4.1 Limitations of the state of the art 36
3.4.2 Thesis Contribution 38

4 Approach to integration contracts into IEC 61499 **41**
4.1 Methodologies 42
4.2 Building IEC 61499 contract function blocks 48
4.2.1 Time-Sensitive Behavioral Contract (*TSBC*) 48
4.2.2 Timing Contract Function Block(*TCFB*) 51
4.3 The specification language 52
4.3.1 Specification contract for *TSBC* 52
4.3.1.1 Syntax and semantics 53
4.3.1.2 Examples 54
4.3.1.3 BNF 54
4.3.2 Specification contract for *TCFB* 55
4.3.2.1 Syntax and semantics 55
4.3.2.2 Examples 56
4.3.2.3 BNF 60
4.3.3 Specification support tool 61
4.4 Exception handling 63
4.4.1 Motivation and problem definition 64
4.4.2 Syntax for exception handling specification of *TSBC*s 67
4.4.2.1 Exception handling conditions apply within a specified period 67
4.4.2.2 Exception handling conditions apply within the number of a causing event 69
4.4.2.3 Apply exception handling based on the violation rate 70
4.4.3 Syntax for exception handling specification of *TCFB*s 74
4.4.3.1 Syntax 74
4.4.3.2 Examples 75
4.4.4 BNF 76

5 Evaluation **77**
5.1 Evaluation for TSBC 80
5.1.1 Verification for contract specification without exception handling 80
5.1.2 Verification for contract specification with exception handling 83
5.2 Evaluation for *TCFB* 87
5.2.1 Validation of the "*Occurrence*" specification 88
5.2.2 Validation of the "*Age*" specification 90
5.2.3 Validation of the "*Reaction*" specification and exception handling 91
5.3 Combining and extending abilities of *TSBC* and *TCFB* 95
5.3.1 Scenario description 95
5.3.2 Experiment setup and simulation-based validation and evaluation using 4*diac* 97

6 Conclusion and future work **101**
6.1 Summary 101
6.2 Future work 103

A Demonstration of *TSBC* **105**
A.1 System Description 105
A.2 System requirement and contract specification 106
A.3 Implementing and Evaluation 109

B Demonstration of *TCFB* **111**
B.1 System description 111
B.2 Modeling 112
B.2.1 Plant model 112
B.2.2 Controller model 112
B.2.3 Application model 113
B.2.4 System analysis 114
B.2.5 Application of timing contracts 114
B.2.6 Formalization of timing phenomena and effects 115
B.2.7 Contract specification and consistency check 116

Bibliography **119**

List of Figures

1.1 The V Model [18] 5

2.1 Characteristics of an IEC 61499 function block [25] 12
2.2 Interface and ECC of basic function block 13
2.3 Service Interface Function Block 14
2.4 Composite function block 15
2.5 Application and system model of IEC 61499 16
2.6 A hierarchical application using subapplications [25] 17
2.7 Example of utilizing of contract in system design 23

4.1 Description for methodology and research design 42
4.2 Integration examples of the contracts 43
4.3 Description for applying contracts in conception and implementation view 44
4.4 Overview of the proposed integration of contract-based in IEC 61499 FBs 46
4.5 General progress of development and implementation using 4diac and FORTE 47
4.6 Interface of the time-sensitive behavioral contract function block . . 48
4.7 Interface of Timing-Contract function block 51
4.8 Illustration example for the specification of event occurrence 56
4.9 Illustration example for *Reaction* constraint 57
4.10 Illustration example for *Age* constraint 58
4.11 Specification for the example case of *Age* constraint 59
4.12 Pattern-based support of Contract Specification(PassCS) tool . . . 61
4.13 The dialog box for setting contract specifications based on patterns 62
4.14 Assistance window for checking the contract specification 63
4.15 Description of the contract specification areas 65
4.16 Motor current response characteristic 65
4.17 Outline an example of the requirement to pack a container of *6* products in which the total number of products that undergo exception handling is no more than *2*. 71

4.18 An example to permit the exception with 2 violations in 6 times of the last request event 73

5.1 The given values for testing of *TSBC* contract specification 82
5.2 Verification result of the implementation of *TSBC* without exception conditions 82
5.3 Verification result of the implementation of *TBSC* with exception conditions 84
5.4 Verification result of the implementation of *TSBC*. 86
5.5 Verification result of the implementation of TCFB with "*Occurrence*" specification 89
5.6 Verification result of the implementation of *TCFB* with "*Age*" specification. 90
5.7 Application for verification of $TCFB$ in $4diac$ 94
5.8 Distribution of test values for ***Case 2*** and ***Case 3*** 94
5.9 The implementation of *TCFB* with exception handling specification. 96
5.10 Graph of data and test results 98

6.1 Model for runtime monitoring of a centralized control systems using contracts 104
6.2 Model for runtime monitoring of a distributed control systems using contracts 104

A.1 Description of the Pick-and-Place robot system 106
A.2 Block diagram for the PnP controller design 107
A.3 The IEC 61499 system model with contract FBs 110

B.1 Example of the PnP robot in a manufacturing line 111
B.2 The application model for a manufacturing line in IEC 61499 (signals ending at the top part are continued in Fig. B.3) 113
B.3 Application of timing contracts to the ControllerFB subsystem . . . 115

1 Introduction

Contents

1.1 Motivation 1
1.2 Solution Approach for Systems Engineering 4
1.2.1 Design methods 4
1.2.2 Design architecture and framework 6
1.2.3 Requirements and assumptions for design methodology 6
1.3 Research questions 7
1.4 Organization 8

1.1 Motivation

The term industrial control system (ICS) refers to an electronic control system and associated equipment used to control an industrial process [12]. An ICS typically includes supervisory control and data acquisition (SCADA) systems, distributed control systems (DCS), and programmable logic controllers (PLC). They are often found in industrial control applications where SCADA and PLC systems can be scaled down to small systems with few control loops [13]. However, the increasing demand for flexibility and reconfigurability lead to new challenges for increasing complexity of components [14] in industrial control systems (ICS).

Today the functionality of Supervisory control and data acquisition (SCADA) and DCS systems are very similar. They make the boundaries between DCS and SCADA/PLC systems almost erased [12]. Traditional Industrial control systems have evolved in two main directions, either based on the distributed control systems (DCS) or based on PLCs [15]. However, because of the monolithic nature of PLC software, these DCS and PLC-based systems are less flexible than the demands for reconfigurations and adaptations of modern manufacturing systems [15, 16]. On the

one hand, advances in information and communication technology (ICT) and digital electronics are key enablers to improve the combination and coordination between physical and computational elements for increasing the adaptability, autonomy, efficiency, functionality, reliability, safety, and usability [7].

The fourth industrial revolution, also known as Industry 4.0, drives the digital transformation from traditional manufacturing and industrial production processes to intelligent manufacturing solutions. It involves the use and integration of new technologies to improve the flexibility and efficiency of production. One of the concepts often highlighted, and at the heart of Industry 4.0, is the extensive use of cyber-physical systems (CPS). Such systems are expected to bring significant economic and societal benefits and offer great opportunities in several key industries, including industrial automation/manufacturing, chemicals, energy, vehicles/transport, automotive, healthcare, defense and aerospace, and many others.

CPS is an interesting term denoting the networking of physical components (sensors, processors, actuators) using modern information and communication technology (networked computing and networking). The elements in a CPS can influence each other and interact in a feedback loop. They can perform many functions and capabilities. For example, they can exchange and analyze real-time data, solve problems themselves without human intervention, and bridge the physical and digital worlds. Furthermore, they can change and/or contribute to the behavior of the whole system when necessary. As a result, they play a pillar and fundamental role in the context of intelligent manufacturing systems. These systems are expected to provide adaptive control and self-monitoring capabilities to make manufacturing systems more flexible, reliable, and efficient. However, the development of CPS, especially in the production and manufacturing domain, still faces several requirements and challenges that need to be addressed.

In industrial production, CPSs are typically designed and deployed as self-adapting systems in the form of a distributed architecture. In such systems, data from the working environment and processed information can be collected by physical objects or sensing devices and automatically transmitted over networks to computing elements or control devices. Despite the fact that developers today have successfully designed a variety of CPS in many industrial sectors [17], the integration of embedded computing systems and physical elements to develop CPS still faces various challenges as pointed out below.

- The first thing to note is a CPS can be quite complex. It may contain many heterogeneous elements and subsystems that work together to control and monitor the system. For this reason, the development of such systems cannot simply be carried out with the typical approach but has to be carried out as an integration activity. The first research question arising here is *how to ensure the integration of hardware and software in Industry 4.0 components and component networks (IoT devices) and monitor them during operation.* To achieve this, it is necessary to find an adaptive solution and a modern approach, including methodologies and architectures, to overcome the challenges involved.

- The components of a CPS are typically sourced from many different suppliers. They may also be developed by different teams using different frameworks and tools. Building a physical prototype for such systems during the design and development process is both costly and time-consuming. As a result, the model-based development approach has become one of the most effective solutions to the mentioned challenges. However, the next question that needs to be answered is what kind of specialized architecture and design framework is appropriate for modeling the components and applications of a distributed system such as a CPS.

- As the demand for flexibility and autonomous adaptation of such systems increases, so does the number of components integrated into the system. This makes CPS increasingly complex. Furthermore, during development, these systems may consist of many components and reuse sub-functions from previous projects where assumptions are sometimes not fully known, leading to challenges in integrating these components/subsystems into a new system. It is also necessary to ensure compatibility in the connection and interaction between components and to monitor their safe operation. The contract-based design (CBD) approach is appropriate to address these challenges in this situation. However, the integration of the CBD approach into a new development framework (in case the framework does not support it) is a challenge that has to be taken into account as well.

- The operation of CPS relies on distributed computing elements that can communicate and exchange data with real-time physical systems. In this way, CPS can self-adapt to changes in the environment during its operating time. However, problems such as large and multiple measurement ranges, large processing time differences, and real-time response of data exchange can sometimes arise in such systems. In a highly complex distributed system such as the CPS, it is extremely important to ensure the correct behavior and timely arrival of data from the elements and sub-systems. Moreover, the CPS normally operates as a sensor-based autonomous system, where its decision-making capability depends heavily on the fusion of feedback signals from sensors. Let's imagine that if one of these sensors does not work, the communication and data exchange process may be lost due to some reasons, or the signal may arrive at the wrong time (earlier or later than expected), and so on. These problems can lead to the decision-making activities of the system being affected, especially when it comes to the elements operating in the highly sensitive mode, where a minor violation during operation can lead to a dramatic failure of the whole system. In this case, the contract-based design provides solutions to ensure that the system development meets the design specifications and checks the compatibility of the interfaces between the components in the system. Furthermore, throughout the system implementation period, the operational process must monitor and guarantee the correctness of the components in terms of behavior and timing. Let's take the example of a modern automated production line configured as a distributed control

structure. Each workstation is controlled by its own controller and can exchange data and communicate via an industrial communication network to share real-time information. It is assumed that the general requirements for manufacturing a product on this line must be guaranteed, both in terms of the technical specifications of the product and in terms of the time constraints of the manufacturing process. For this purpose, it is necessary to check and monitor both aspects for correct behavior and timely occurrence of expected events/signals for each workstation to ensure that both requirements are met. A behavioral requirement can be expressed by a specification, such as: A value d checked at a component's port must always be within an allowable range ($d \in D$); or the checked value must be within a defined range during a specified time period (if $t \in T$ then $d \in D_1$), outside of which ($t \notin T$) the data value must be in a different range of values (if $t \notin T$ then $d \in D_2$). The time constraint for a component can be defined by specifying that the component's execution time must be reached within a given time interval; or that an expected event must occur at the component's specified port within a given time interval. For example, the arrival of data d at the input of element C must occur every T_0 seconds, or whenever the data d_1 occurs at the input, the data d_2 will arise at the output within the time T thereafter.

1.2 Solution Approach for Systems Engineering

This section presents the rationale for selecting solutions to address the mentioned challenges. There are two parts, including development methodologies and development architecture/framework.

1.2.1 Design methods

In order to cope with the challenges of the development of complex physical systems, Model-Based Design (MBD) offers a good technique to deal with the above-mentioned problems at the design time. The advantages of this method are listed below:

- In the design phase, building a CPS prototype is often time-consuming and expensive. Especially for large CPSs, it's almost impossible. In addition, the evaluation process of such a complex system is always associated with the appearance of unexpected defects and hazards during the test phase. These can sometimes exceed the calculations and predictions of the engineers during the design process. Therefore, a design methodology and tools are needed to support early requirements validation and virtual system integration. A model that enables simulation and analysis is an excellent choice to meet all of the above requirements.

- An application system can be virtually integrated based on its component models and the system architecture specification by using Model-Based Engineering. MBD helps developers create models of physical objects/processes in

industrial automation. This means that the functional behavior can be verified without using the actual plant. MBD enables the verification and validation of components or systems using a simulation-based evaluation method before they enter the real production context.

- Testing based on simulation can also provide a database for analysis and evaluation. This can help the developer imagine possible scenarios for the modeled objects. Furthermore, based on the simulation results, the developer can constrain the properties of the operating environment and specify how components should respond to that environmental conditions. These data are useful for documenting a component's specifications and requirements to another development team. This is particularly helpful for the specification of systems that will only work correctly in a limited subset of the possible behaviors of the environment.

The design process for a CPS is often developed based on the V-shape model (see Figure 1.1). In this model, a product of the development process is divided into two separate phases *design* and *integration*. The design phase involves the decomposition

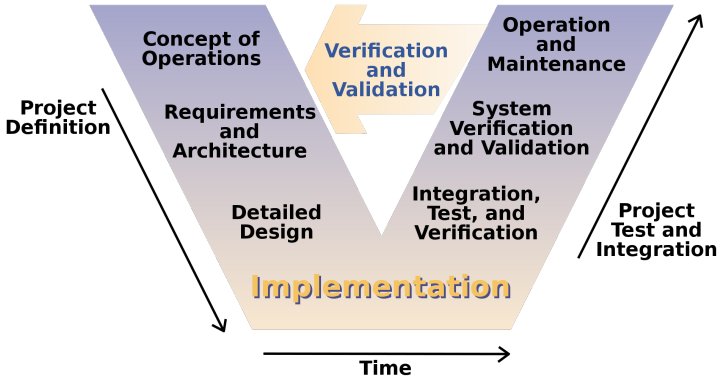


Figure 1.1: The V Model [18]

process, where the system-level (or top-level) requirements are decomposed into different sub-system-level (or lower-level) requirements. Furthermore, requirements for system development may be a fusion of requirements from different design aspects, such as performance, safety, reliability, function, timing, etc. In this regard, each component, subsystem, or process must satisfy a specific set of requirements. On the other hand, the integration phase involves testing activities at each level of integration, starting from the lowest level (component/module level) to the highest level (system level). Testing processes are help to ensure that all the specifications defined in the design phase are met in the integration phase. The integration of components and subsystems requires clear abstractions and interface descriptions to create reliable connections. A suitable framework for dealing with these issues is the contract-based design (CBD) methodology. The first contract concept was conceived by Bertrand Meyer [19] for software specification in object-oriented programming. It focused on the use of pre-conditions (ensuring at program

entry), post-conditions (ensuring at program exit), and invariants (ensuring for all time). Today, the contract-based design methodology relies typically on a pair of assume-guarantee assertions. It is widely recognized as a tailor-made method for addressing security, complexity, and heterogeneity issues for CPSs. Contracts are typically used to specify that component implementations and environments behave correctly. In the context of the contract, the complexity of the design and verification process for a complex component or system can be simplified. This is achieved by decomposing the system-level (upper level) specifications of the requirements/tasks into sub-system-level (lower level) specifications of the requirements/tasks on the condition that they are manageable [11]. Contracts support better requirements refinement and can be applied at any step of the design flow, especially when combined with model-based development [20].

1.2.2 Design architecture and framework

The design of distributed automation control systems requires complex models for the combination and integration of various heterogeneous elements in order to describe the complex behavior of components, subsystems, and the system itself. The IEC 61499 standard was published in 2005 to meet these requirements. It builds on the definitions of IEC 61131-3 and aims to provide a generic reference architecture for model-driven development of distributed automation control systems. This standard has some advantageous features such as object orientation, event-driven execution behavior, and vendor-independent development. Therefore, it is widely accepted as the most suitable architecture to address the structural and behavioral complexity of such distributed systems [14], especially CPSs. The IEC 61499 standard provides and supports modularity through a hierarchical and event-driven functional block model and its deployment on distributed computing nodes. Although IEC 61499-based tools enable model-based testing of applications before and after deployment, they are not well suited for systematically specifying and monitoring design requirements at design time.

As mentioned earlier, one of the appropriate ways to address the challenges of CPS development is system development using the V-model coupled with the CBD method. However, the use and implementation of contract concepts are not available in either the IEC 61499 standard or the current IEC 61499 compliance tools. Therefore, the integration of the contract concepts into an IEC 61499-compliant software design environment in order to support the development of CPS is the main contribution of this thesis.

1.2.3 Requirements and assumptions for design methodology

In CPS, the modeling challenges are related to heterogeneity, concurrency, and sensitivity to timing issues. Since CPSs are heterogeneous distributed real-time systems, the physical processes are a fusion of different events that can happen simultaneously. Therefore, monitoring and controlling CPSs becomes a serious

challenge because their operation has to face the interaction of highly heterogeneous components and a multitude of interactions between them, while still ensuring functional and timely correctness. The research work presented in this thesis, proposes to combine three technologies: Model-Based Engineering, Contract-Based Design, and IEC 61499 for the modeling, control, and monitoring of safety-critical cyber-physical systems. The requirements and assumptions for the application of the design methodology for safety-critical CPS systems used in this thesis are listed and briefly described in the following points.

- Model-based design engineering for CPS development includes models of components, applications, systems, and devices. It is applied using IEC 61499-compliant development software.
- There are several options for IEC 61499-compliant development software. However, *4diac* [21] version 1.12.0 will be chosen as the software development tool in this thesis. The reasons for this choice are: 1) the IEC 61499 standard is expected to become a key standard in the industrial control solution, where the PLC standard has dominated so far, and 2) *4diac* is a software supported by a large user and developer community. It is also used in the author's research project to develop and implement the IEC 61499 compliant systems.
- To maintain and monitor the correctness of the timing behavior and timing constraints for safety-critical CPSs, the contract-based design methodology and concepts should be integrated into the IEC 61499-compliant runtime environment.
- A system is modeled in a hierarchical structure, where the subsystems and the overall system can be described by composite function blocks (CFBs) or subapplications in *4diac*. The contracts are defined and monitored at each level of the system.
- There is also a need for a suitable syntax for requirements specification. It can be used to specify the behavior of the components in terms of functionality and timing. This allows components to be monitored while running in an IEC 61499 compliant runtime environment, especially *4diac*.
- Introducing contracts into *4diac* should not only address specific types of requirements, but also provide the ability to extend the requirements specification to a broader, more complex, and more flexible range of applications.
- The implementation and simulation-based evaluation process are performed in *4diac* in its runtime environment called *FORTE* (version 1.12.0).

1.3 Research questions

In order to bring these conceptual designs together as a solution for the issues outlined above, the following research questions need to be asked:

1. As mentioned in the previous sections, the motivation of this thesis is to find a solution approach for model-based engineering with contracts for safety-critical CPS using IEC 61499. So the first question is: ***(Q1)*** *Do any existing IEC 61499-compliant development tools/frameworks offer features for specifying the functionality and timing constraint requirements?.*

2. Once the answer to the motivation question is found, a follow-up question should be raised regarding the methodology. In particular, critical requirements for the development of safety-critical CPS include ensuring that data arrives on time and that the value of that data is correct (e.g., the data must be within a bounded range). Therefore, the following questions are: ***(Q2)*** *How to ensure the correctness of behavior and the timely occurrence of the data of a component in the IEC 61499-compliant application? How can such requirements be semantically expressed and integrated into that 61499-compliant development tool/framework, especially in* 4diac*?*

3. For a distributed system, we need not only to ensure the performance of every single component (which is modeled by a type of IEC 61499 FB) but also the performance of the subsystems and the entire system (which is modeled by an IEC 61499 FB network), that must also be taken into account. The next question that needs to be asked relates to the feasibility of using contracts across different levels of a hierarchical structure. ***(Q3)*** *How can the functional and timing behavior of a system composed as a hierarchy of components, which were specified according to* ***Q2****, be verified?*

4. Once an answer to the above question is found, the implementation process may encounter some problems and new requirements. Therefore, the subsequent question that need to raise here is: ***(Q4)*** *What are the limits of the expressiveness of the proposals on* ***Q1*** *-* ***Q3****? How well does the approach scale for complex designs?*

5. Methods for applying and evaluating the concepts and solutions proposed in this thesis should be outlined. Therefore, the last question is: ***(Q5)*** *Are the specification syntaxes for the aforementioned requirements sufficiently expressive? Can they cover complex use cases? How to monitor the correctness and safety of those components during implementation?*

In the following chapters of this thesis the answers to these questions will be presented and discussed.

1.4 Organization

This thesis is structured as follows:

Chapter 2 provides the necessary background concepts and theories related to IEC 61499 and contract-based design, especially assume-guarantee contracts. A brief overview of different contract specification languages and supporting tools is also provided.

Chapter 3 presents other relevant work which covers various topics included in this thesis, for example, model-based design methodology for CPS; modeling and controlling CPS using IEC 61499; using contracts for safety-critical CPS; and applying exception handling to contracts. A summary and analysis of the needs and contributions of this work, as well as the answers to the questions *Q1* can also be found there.

Chapter 4 provides the core methodology and the solution approaches for this work. In particular, the methodology for combining model-based design and contract-based design using IEC 61499 can be found here. In addition, the details for the development of contract FBs (*TSBC* and *TCFB*) in *4diac* are represented. The syntax and semantics for the specification of the contracts and the exception mechanism for *TSBC* and *TCFB* are also explained in fact. You can also find the content of the answers to the questions *Q2* and *Q3* in this chapter.

Chapter 5 evaluates the proposed solutions in this thesis. The IEC 61499-compliant development tool (*4diac*) and its real-time runtime environment (*FORTE*) are used for the evaluation processes. In particular, the research in this thesis was validated by some use case examples, which are established to introduce different scenarios in practice to test the responsiveness and performance of the developed contract FBs. The verification and evaluation are performed using a simulation-based validation methodology in which implementations are executed based on the use of a set of assumptions for these activities. The chapter also provides the answers to the question *Q4* and the question *Q5*.

Chapter 6 concludes the thesis and provides some suggestions for future research.

2
Technological background

Contents

2.1 IEC 61499 standard 11
2.1.1 IEC 61499 function block 12
2.1.2 Application and System model 15
2.1.3 Building and modeling distribution system 17
2.1.4 Development tools and runtime environments for IEC 61499 compliant applications 18
2.2 Assume/Guarantee contract 21
2.2.1 Overview of contract concept 21
2.2.2 Properties, operators, and algebra of contract 22
2.2.3 Requirement formalization, contract specification, and support tools 25

2.1 IEC 61499 standard

In the traditional industrial sector, manufacturing automation systems are typically controlled through programmable logic controllers (PLCs). These PLCs are compliant with the IEC6131-3 standard. The most common characteristics of these control systems are monolithic and centralized. In the last few decades, the growing need for more flexibility and higher performance has led to a trend toward distributed control approaches from centralized systems.

The IEC 61499 standard was published in 2005 with the primary objective of providing a common architecture for the development of event-driven distributed control applications. It aims to improve and extend the existing established practices based on the IEC1131-3 standard [22]. The IEC 61499 extends the IEC 61131-1 by replacing the cyclic execution model of PLCs with an event-driven

execution model and by improving the encapsulation of software components. These enhancements are intended to meet the requirements of modern technology, such as modularity, reusability, reconfigurability, portability, interoperability, flexibility, and distribution [23].

2.1.1 IEC 61499 function block

The IEC 61499 standard provides a graphical and textual description language for defining the structure and behavior of a distributed application. The central modeling element of this architecture is function block (FB) [24]. A general IEC 61499 FB is defined by an external interface and an encapsulated functionality inside the block. Furthermore, it can contain some internal variables.

The interface generally includes a set of event inputs, event outputs, data inputs, and data outputs, as illustrated in Fig. 2.1. A data input needs not be associated with an input event if it is only used to define a constant value. Otherwise, a data input must have at least one input event associated with it. Similarly, for output data, any data output from the interface must be associated with at least one output event. On the other hand, an input/output event does not need to be linked to input/output data. Note that a function block may be missing input or output data, or both, but it should not be missing input and output events.

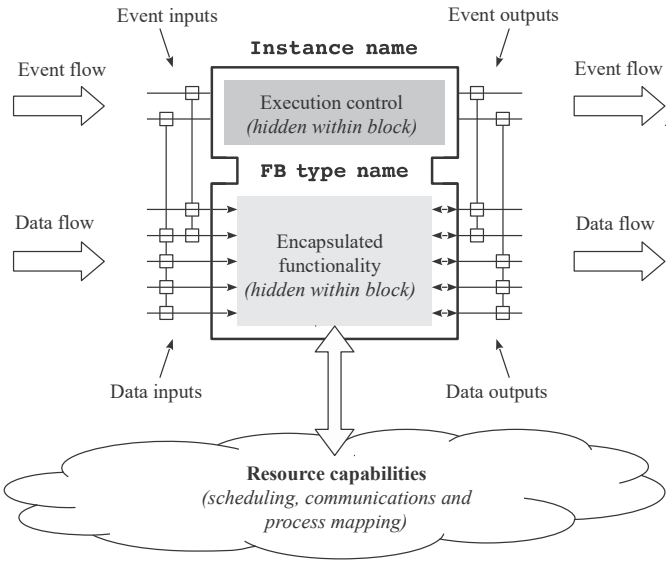


Figure 2.1: Characteristics of an IEC 61499 function block [25]

The encapsulated functionality within a FB may include some algorithms. An algorithm can be written by various programming languages such as Java [26, 27], C [28], C++ [29, 30]. It can also reuse legacy algorithms written in the PLC languages (Structured Text (ST) [31, 24]). In addition, the algorithm can use internal variables, which are protected, and therefore can not be directly accessible from outside the FB. The algorithms are used to define the behavior and properties of a FB.

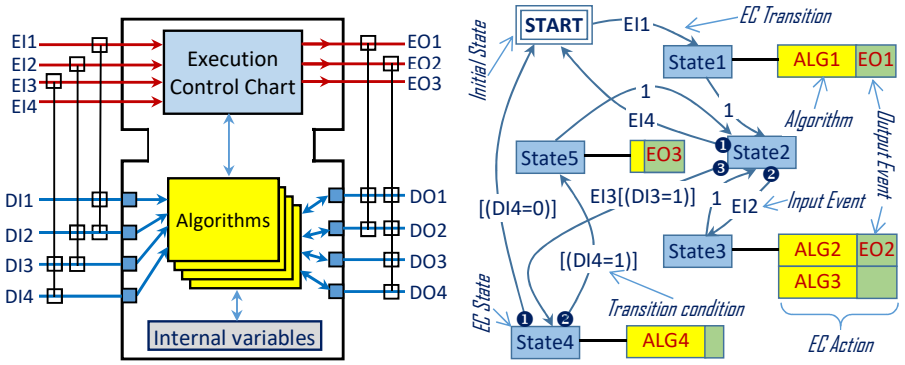
The execution of the FB is described as follows: When an awaited input event is triggered, all of the input variables that are associated with that event will be

updated. At the same time, this event activates condition checking to allow the program to call and execute one or more algorithms. The checking of the condition could be based on the occurrence of this event itself or the matching of the internal variables of the logical expression. Upon completion of those algorithms, the values of the involved variables (internal and output data) will be updated. If an algorithm is declared with the emission of an output event once its execution has been done, then the FB will emit that event at the output side. The output values associated with this emitted output event are updated and written to the output data ports of the function block. Execution can then proceed either to the next scheduled algorithm or to the next state to wait for another input event.

Apart from that, the behavior of the FB can also be defined by encapsulating a network of different types of FBs. The IEC 61499 FBs are classified into three different kinds include Basic Function Blocks (BFB), Composite Function Blocks (CFB), and Service Interface Function Blocks (SIFB). These FB types are executed by the event-driven model as mentioned above. In addition, they have the same external interface including event input/outputs and data input/outputs as illustrated in Fig. 2.1. The difference among these FB types involves how the encapsulated functionality is specified to describe and generate the behavior inside the block. These differences will be pointed out as follow:

2.1.1.1 Basic Function Block

The behavior of this FB type is defined by a graphical state transition diagram so-called execution control chart (ECC) to specify the state machine for control event execution(see Fig. 2.2(b)). A state in an ECC can have an action, or it



(a) Interface and characteristics (b) Execution control chart

Figure 2.2: Interface and ECC of basic function block

possibly has no action (*State2* in the figure). An action can contain an (or several) algorithms, or it is only used to trigger an (or several) output events. On the completion of an algorithm or an action, an output event can be fired if this event is associated with that algorithm/action. A transition from one state to another will only occur if the condition for the EC transition is met. A transition condition can be an awaited input event or a logical expression or a combination of both.

A state can contain many transition conditions. If in a state that has multiple possibilities to transition to another (for example *State2* and *State4* in fig. 2.2(b)), then the EC transitions will be sorted by their priority. This will be useful when a state fulfills various transition conditions at the same time.

2.1.1.2 Service Interface Function Block

This type of FB provides services for the exchange of data values and events between the FB and external resources over a network. These SIFBs can be of the following types [25, 32]:

- *Publish/Subscribe* FBs are used to send and receive event and data connections across the boundaries of the resources. A *Publish* FB can transmit one or more data values to one or more external resources within other controllers over a communication channel. On the other side, a *Subscribe* FB is configured to receive the values sent by the *Publish* from another resource. The communication is established through an identification address that is specified on both the *Publish* and the *Subscribe*. In addition, the SIFBs are triggered from the hardware over the mapped I/O pins as well as from an incoming event. This means that the event of a *Responder* FB can be triggered each time a package is received [21].
- *Client/Server* FBs are used to set up an interlocked exchange of data with external resources. In particular, a *Client* sends requests to the *Server* and receives corresponding answers from the *Server*. The *Server* receives the request(s) sent by the *Client*. The *Server* sends back the corresponding response.

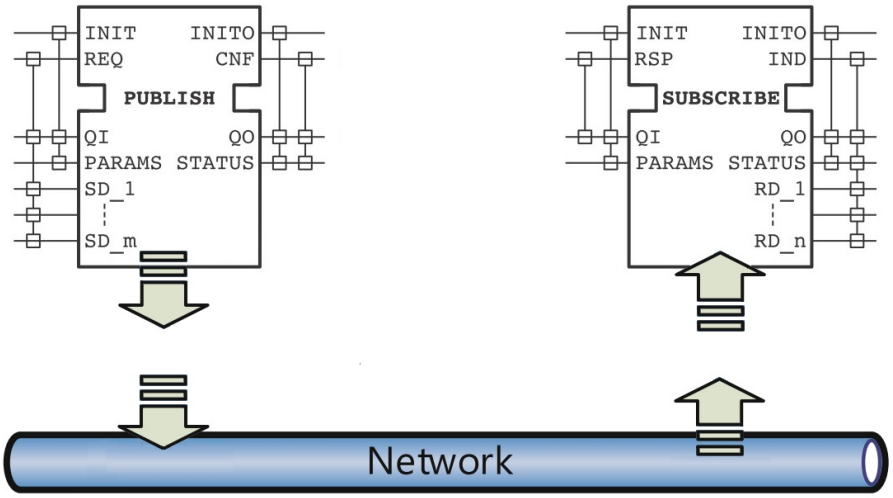


Figure 2.3: Service Interface Function Block

Unlike the other FB types, the behavior of a SIFB cannot directly specify using the IEC 61499 modeling tools. Instead, it is defined by using a native programming language, e.g., C or C++.

2.1.1.3 Composite Function Block

This type of FB provides the ability to encapsulate a network of internal FBs within another external FB. The behavior of a CFB relies on the behavior of its encapsulated FB network. A CFB can be encapsulated inside one (or some) BFB(s), one (or some) lower-level CFB(s), and one (or some) SIFB(s), as shown in Fig. 2.4. Therefore,

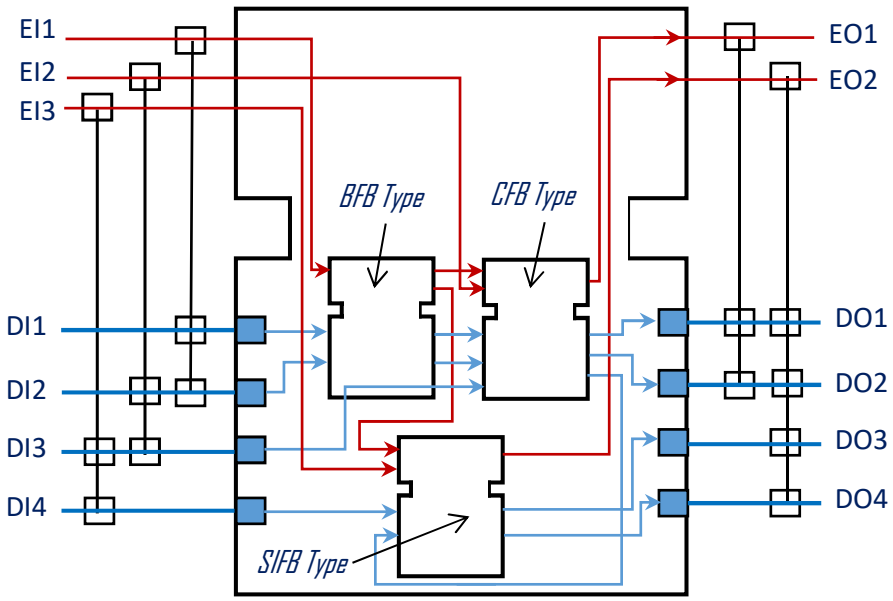


Figure 2.4: Composite function block

CFBs are typically used to define the behavior of complex objects/components. They provide a useful way to build hierarchical application modeling with IEC 61499. In addition, they also help to simplify modeling simulations and improve the reusability of components in complex systems.

2.1.2 Application and System model

An IEC 61499 application is created by linking events and data variables of different FBs in a network to solve a particular automation control task. The example of an IEC 61499 application is shown in the left-upper part of Fig. 2.5. Here, the application model called *Application 1* is represented by a FB network of *FB1 - FB4*. In this example, the event flows are indicated by the red arrows, and the data flows are indicated by the blue arrows.

In a network of FBs, the events of one FB can be linked to the events of another FB in both the fan-in and fan-out modes with no limitations. This means that it is possible to connect several event outputs to one event input (fan-in) and one event output to several event inputs (fan-out). There are two points to note when connecting data between FBs: First, data can only connect if they have the same data type or they are compatible (*ANY* type); Second, data has only a fan-out mode, it has not fan-in mode. It means that a data output of one FB

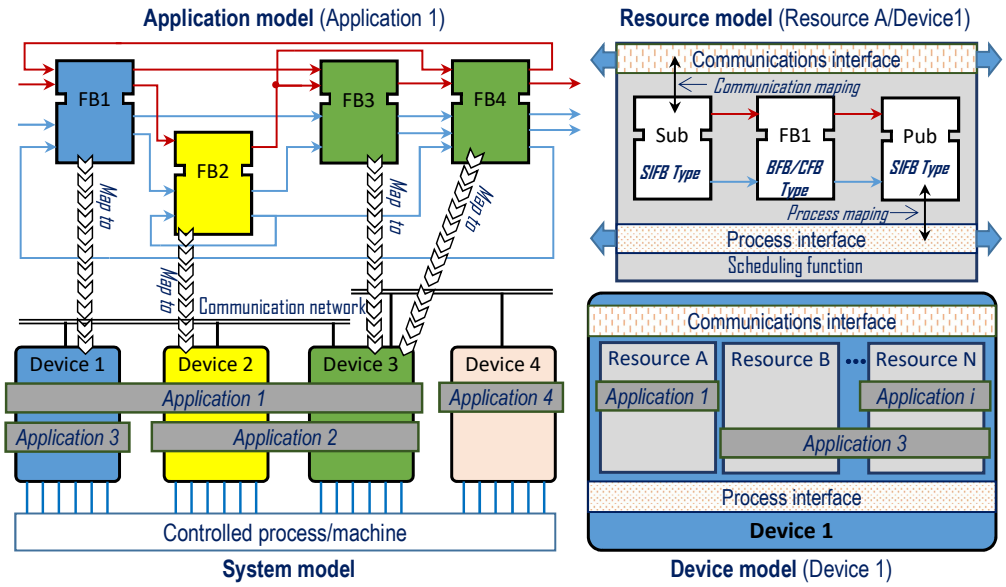


Figure 2.5: Application and system model of IEC 61499

can be linked to several data inputs of one (or more) FB(s), while a data input can only receive one data from a specific FB.

The separate modules or functionality parts of a distributed application (e.g. *Application 1*, *Application 2*) can be allocated to run on various device (e.g. *Device 1*, *Device 2* and *Device 3* of *Application 1* as illustrated at the links corner-lower part of Fig. 2.5).

The IEC 61499 standard provides a device model for exchanging data with other devices via inter-device communication links/communication networks using the communication interface. It also allows data exchange between its resources and the input and output (I/O) interface ports of the real physical devices (e.g. sensors, actuators, etc.) via the so-called process interface. A device that can be used to run a complete application (*Application 3* runs on *Device1*, *Application 4* runs on *Device4*) or just a few FBs (fragments) of a distributed control system (a fragment of *Application 1* runs on *Device1*, a fragment of *Application 1* and a fragment of *Application 2* operated by *Device2* and *Device3*).

Note that a device can consist of more than one resource. A resource is a location that contains the IEC 61499 application FBs assigned to the device. It provides support for the execution of these FBs. As we can see from the rights-lower part of Fig. 2.5), the FBs of an application can be mapped to not only a single resource but also multiple resources in a device. Inside a resource model, there is a FB network and interface layers. For interaction with the communication systems, the SIFBs can be used to establish the connection between the FBs and the resource interfaces. A scheduling function is used to make sure that the encapsulated functionality in the FBs that is executed in the correct order. A resource provides support for independent execution of applications; it enables applications to be loaded, configured, started, and stopped without affecting other resources on the same device or across the network. [25].

The IEC 61499 standard also provides a system model for the characterization of an industrial automation system. In a system model, a set of devices is interconnected by multiple inter-device communication links or communication networks. By mapping applications to resources, it enables several applications to collaborate and work together in a distributed system.

IEC 61499 also provides a system model to characterize an industrial automation system, whereby a set of devices are interconnected via several inter-device communication links or communication networks to help several applications inter-operate and collaborate in a distributed system.

2.1.3 Building and modeling distribution system

An IEC 61499 distribution model is created by linking the application and system models. The following steps summarize this process: In the first step, a distributed application must be developed. This application consists of a network of FBs to define some desired behaviors and properties of an automation system. In the next step, a system model must be created. This model connects the available control devices and various networks to provide a communication network between these devices. Finally, the IEC 61499 mapping function supports the assignment of application parts to different device resources. In particular, depending on the specified objectives, one or more of the FBs of this distributed application model will be selected and configured to run on different resources.

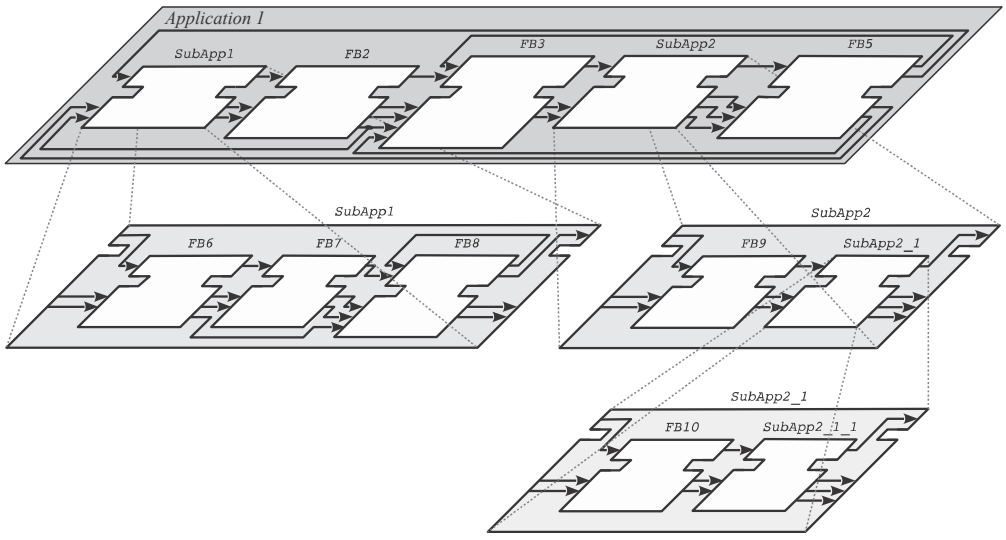


Figure 2.6: A hierarchical application using subapplications [25]

A control solution can be built in a hierarchical structure with subapplications, as shown in Figure 2.6. Similar to a CFB, a subapplication can be viewed as a lower level of the application. It is composed of a network of different types of FBs and possible sub-applications at a lower level as well. The difference between a CFB and a subapplication is that the subapplication is distributable [15]. It means that the FBs in the sub-application can be mapped and run on different resources,

while the FBs in the CFB can only map and run on a specific resource.

Further explanation of the distinction between subapplication and CFB, as well as more details on the various concepts and elements of IEC 61499 can be found in [25].

2.1.4 Development tools and runtime environments for IEC 61499 compliant applications

This section is a summary of some significant IEC 61499 software tools that have been introduced in recent years. Each of these development tools usually comes with an appropriate runtime environment (RTE) for the implementation of IEC 61499-compliant models. Among the commercially supported tools are *ISaGRAF* [31] and *NxtStudio* [33]. On the other hand, *FBDK* [26] and *4diac* [21] are open-source tools that are well-known so far. These software tools provide features for modeling, testing, and deploying IEC 61499-compliant FBs and applications in industrial distributed control systems.

2.1.4.1 FBDK

Function Block Development Kit (*FBDK*), introduced by Holobloc Inc, is the first programming tool to follow the IEC 61499 standard. It is a Java-based software developed for applications in the development and modeling of the 61499-compliant FBs. It plays a significant role in developing, teaching, and evaluating the IEC 61499 standard [34]. Like most of the other IEC 61499 compliant frameworks, it provides two elements including a Java-based graphical interface for the editor environment (*FBEditor*); and a Java-based runtime execution environment (*FBRT*) for the implementation and deployment.

- FBDK is widely used in academic communities and research trial implementations [35]. In *FBDK*, the FBs are built into Java classes. The typical implementation of an FB requires an external event to be active at the event input port (usually named *REQ*). Once execution is complete, the FB emits an event out at the event output port (usually named *CFN*)[9].
- In addition, the *FBDK* provides a Java-based runtime platform called the Function Block Runtime Environment (*FBRT*) for the implementation and execution of IEC 61499-compliant systems. Until today, the non-embedded version of the *FBRT* is still available as part of the *FBDK* family software package. It continues to be useful not only for testing, teaching, and research but also for developing IEC 61499 [24].

2.1.4.2 4diac

The Framework for distributed industrial automation and control (*4diac*) is an open-source project provided by *Eclipse* that offers a solution for the distributed control domain. It consists of the *4diac* Integrated development environment (*4diac-IDE*) and *4diac* Runtime Environment (*FORTE*). The general goals of

the *4diac* are as follows: it aims to provide an infrastructure for developing and executing distributed industrial process measurement and control systems compliant with the IEC 61499 standard. It gives support for designing the modern control systems in Industry 4.0 scenario.

- *4diac-IDE* is an Eclipse-based engineering tool for developing IEC 61499-compliant FB and industrial application/distributed systems. This tool is written in Java, widely used in research trial implementations. It provides an extensible engineering environment for developing all types of the IEC 61499 FB as mentioned in section 2.1.1 and for modeling IEC 61499 compliant applications.

- In addition to real-time execution support for IEC 61499 applications, *4diac* provides an IEC 61499 compliant runtime platform called the *FORTE* or *4diac* runtime environment (*4diac-RTE*). *FORTE* is a C++ based runtime environment for the implementation of IEC 61499 applications. It runs on a real-time operating system. So far, it is supported on many operating systems, including *eCos*, *NET+OS 7*, *Posix*: *Cygwin*, *Linux* (*i386*, *xScale*, *PPC*, *ARM*), *rcX*, *VxWorks*, *PikeOS*, and *Windows*. It can also run on several different devices and boards (i.e., *BeagleBone Black*, *Digi Connect ME*, *emBRICK*, *KIPR's CBC v2*, *Lego Mindstorms EV3*, *NXH 51-ETM*, and *Raspberry PI*), and PLCs include *Bachmannelectronic M1 PLC* , *CONMELEON C1*, *Raspberry-SPS*, *WAGO PFC200*, *MicroControl uMIC.200*, and *icom MR-router series*. Furthermore, *4diac FORTE* offers a flexible communication infrastructure by using different communication protocols consists of Eclipse *SCADA SFP*, *Ethernet (TCP/UDP)*, *MQTT*, *OPC UA*, *OPC DA*, *RS232*, etc. [21].

- The number of supported operating systems and communication protocols is growing.

2.1.4.3 ISaGRAF Workbench

ISaGRAF is based on Microsoft Visual Studio Shell, one of the first commercial tools to provide all the functionality needed to develop distributed control applications. It combines two standards, IEC 61131-3 and IEC 61499, allowing users to choose the most appropriate way to develop applications. As a result, developers can use PLC languages while taking advantage of the features of IEC 61499.

- In *ISaGRAF*, functions and FBs code is written in C language and IEC 61131-3 standard languages. Besides, the FB execution model is based on the cyclic program execution of the IEC 61131-3 standard instead of the event-driven program execution like most other IEC 61499 compliant software tools.

- *ISaGRAF* Runtime is an IEC 611313 runtime platform running on virtual machines providing additional support for IEC 61499. It is configurable by

ISaGRAF Workbench with interoperability between devices utilizing the same runtime platform, which is based on a manufacturer-supplied conformance profile.

2.1.4.4 NxtStudio

The *nxtStudio* is another commercial IEC 61499-compliant development tool. It is developed by the Austrian company *nxtControl* and based on *.NET 3.5 Framework*. *nxtStudio* is widely considered the most efficient tool for control logic and distributed system development.

- *nxtStudio* has developed a special hybrid solution by combining the advantages of IEC 61499 and IEC 61131 for the control design framework. It allows hardware-independent engineering and can establish a connection between theoretical research and practical applications. Furthermore, it encapsulates multi aspects in software objects for representing real devices and functions in the engineering tool by using the concept called *Composite Automation Type*(*CAT*). A *CAT* is represented by a FB that combines functions of machines with their simulation and visualization. It supports the control application development and HMI/SCADA visualization, allowing testing, simulation, and communication activities without the real installation. Therefore, it makes the engineering of plants and equipment much simpler, quicker, and consistently of higher quality [33].
- *nxtControl* has also introduced a hybrid runtime system called *nxtIECRT*. It helps the controller execute both IEC 61499 and IEC 61131 programs. *nxtIECRT* has been developed based on the open source *4diac FORTE* with enriched functions, features, and additional services enabling the change of control applications during the system [36]. As a result, this tool can be used not only for professional applications in the industry but also for purely academic research and practical applications that comply with IEC 61499 standard[35].

Summary: Along with the aforementioned tools, many academic and research-oriented development tools have been used by researchers to model, test, and deploy IEC61499-compliant applications, such as *CORFU* [37], *BlokIDE* [38], *FBench*, etc. They introduced and summarized in [39, 9, 40, 35]. In [34, 24, 36], you can find other studies that summarize and compare in detail several development tools and their corresponding IEC 61499-compliant runtime environment.

In this work, the 4diac platform (consisting of *4diac-IDE* and *FORTE*) is used for modeling, testing, and implementation activities. It was chosen for the following reasons: It is an open-source environment. It is used in numerous research projects by many academics and researchers. The user community is very large, so there is great community support. Its functions and features are really useful and are regularly updated, extended, and maintained. In addition, it is fully compliant with the models and guidelines of the IEC 61499 standard. The *4diac-IDE* provides a graphical interface based on the widely used Eclipse.

2.2 Assume/Guarantee contract

This section presents a summary of the key concepts and characteristics that are related to the notion of a contract-based design. These concepts are introduced and clarified around the notion of a component. A component represents an element of a design and is characterized by A/G contracts [10]. This means that the component must have the expected properties/behaviors if it is in the desired environment. A component interface contains a set of input and output ports that enable interaction between that component and its environment to exchange data and events with others. Using contracts in cyber-physical system development can be found everywhere and at all stages of system design [17].

2.2.1 Overview of contract concept

In the context of system design, components are the result of the decomposition process normally of a subsystem or, in some specific cases, of a system. It owns a terminated input/output variable set to interact and cooperate with its environment. Besides, it can be connected to other components to give a behavior in an abstract layer.

The CPS design is often developed based on the V-shape model, where the product development process comprises two separate phases *design* and *integration*. The design phase concerns the decomposition of the design requirements and tasks of the system level (or higher level) into different subsystem-level (or lower level), down to components. Additionally, system development can be a conjunction of design requirements from numerous aspects, such as performance, safety, reliability, function, timing, etc. Therefore, each component, subsystem, or process shall also meet all requirements for these aspects.

On the other hand, the integration phase includes testing activities at each level of integration, starting from the lowest level (component/module level) to the highest level (system level). The testing target is to ensure that the integration of components/objects meets all defined design specifications. The integration of components and subsystems requires clear abstractions and interface descriptions to create reliable connections. Therefore, applying the contracts is an appropriate methodology for compositional specification to address these problems. In the context of contract-based design, components at different levels of abstraction are characterized by contracts, in which the components must ensure their properties/behaviors when these components meet their environment [10].

The design-by-contract approach was first introduced by Bertrand Meyer [19] for software specification via pre- and post-conditions of functions in object-oriented programming. The assume-guarantee (A/G) contract term initiates in the context of compositional assume-guarantee reasoning for software engineering and object-oriented programming. Within contracts, the complexity of design and verification can be reduced by decomposing activities, under a set of assumptions [11]. However, the assumption-guarantee (A/G) contract proposed by Benveniste et al. [41] is mostly used in the context of verification and integration for complex system design activities, contracts are requirements specifications for the correctness of integration

between components at the same level (horizontal contracts). They can also use to specify the consistency between the higher and lower level of abstraction (vertical contracts), down to basic components in a hierarchical structure system [17].

An assume-guarantee (A/G) contract is expressed by assertions as a pair of assumptions and guarantees that are denoted by $\mathscr{C}(A, G)$ of $\langle Assumptions, Guarantees \rangle$. An assumption describes the expected properties of the component's environment. A guarantee describes the component properties/behaviors that must be satisfied according to the satisfaction of those assumptions.

2.2.2 Properties, operators, and algebra of contract

Contract-based design can understood as using a set of methodological guidelines in system design. It exploits an algebra of contracts characterized by operators such as refinement, conjunction, and composition [41].

In the following, we consider a component M is associated with a specification contract $\mathscr{C} = (A, G)$ and component implementation $\mathcal{I}$. A component E will be called a legal environment of contract $\mathscr{C}$, if it can also be associated with the contract $\mathscr{C}$ as an environment $\mathcal{E}$, and $\mathcal{E}$ and $\mathscr{C}$ have the same variables, and $\mathcal{E} \subseteq A$.

Definition 1: Contract satisfaction The satisfaction (denoted by $\vDash$) involves linking implementations to contracts. Once an implementation $\mathcal{I}$ satisfies a contract $\mathscr{C}$, written by $\mathcal{I} \vDash \mathscr{C}$, its behavior matches the guarantees under the assumptions, that can formally give by $\mathcal{I} \cap A \subseteq G$

Definition 2: Compatibility A contract $\mathscr{C}$ is called compatible if and only if it exists at least one environment $\mathcal{E}$. For A/G contracts, this relates to $A \neq \emptyset$. On another side, a contract without any environment is called incompatible.

Definition 3: Consistency A contract $\mathscr{C}$ is called consistent if and only if it exists at least one implementation $\mathcal{I}$. For A/G contracts, this relates to $G \neq \emptyset$. On the other hand, a contract without any implementation is called inconsistent.

There are two basic principles followed by contract-based design techniques: abstraction/refinement and composition/decomposition. Abstraction and refinement are processes relating to the design flow between different layers of abstraction. Composition and decomposition operate at the same level of abstraction [41]. Fig. 2.7 illustrates an example using contracts to support the development process. There are three main contract operators, namely conjunction ($\wedge$), composition ($\otimes$), and refinement ($\preceq$). The meanings and properties of these operators are described below:

Contract conjunction As mentioned in section 2.2.1, developing a complex system may involve different viewpoints/aspects, such as function, timing, safety, etc. Thus, it is necessary to fuse these viewpoints mathematically to obtain the complete system description. System requirements can be structured and formalized using contracts. Contract composition provides formal support to reasoning about

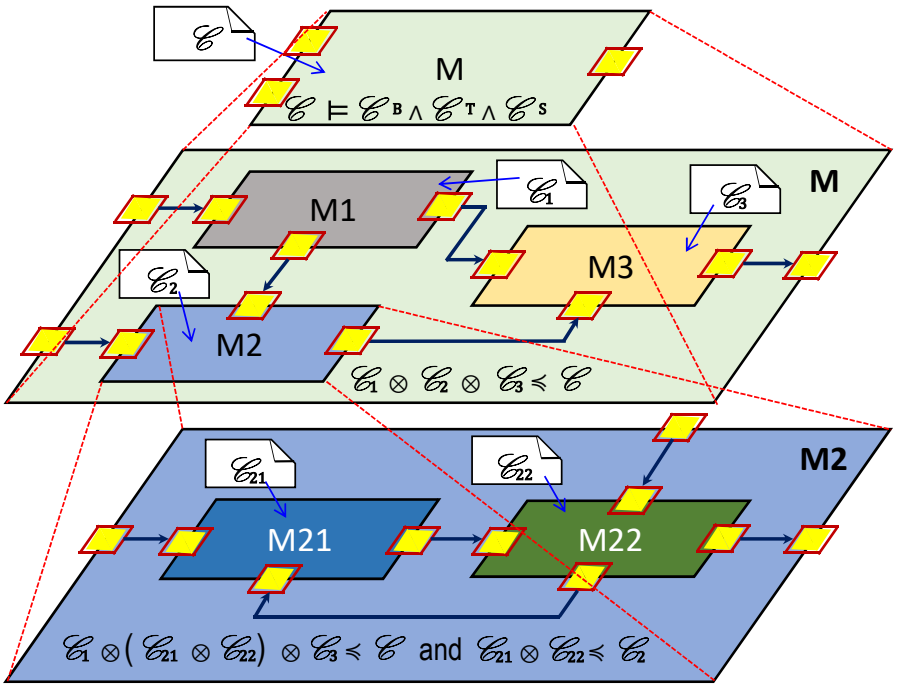


Figure 2.7: Example of utilizing of contract in system design

the conjunction of different design viewpoints reflecting that the system must satisfy all of these viewpoints [41].

Fig. 2.7 shows an example of a conjunction of the contracts relating three aspects, including the behavioral contract ($\mathscr{C}^B$), the temporal contract ($\mathscr{C}^T$), and the security contract ($\mathscr{C}^S$), where: The behavioral contract ($\mathscr{C}^B$) describes the specification of the functionality; The temporal contract ($\mathscr{C}^T$) specifies timing constraints associated with different activities; The security contract ($\mathscr{C}^S$) refers to the specification of the propagation, impact, and handling of errors. Satisfying the above requirements/aspects can be formalized as: $\mathscr{C} \vDash \mathscr{C}^B \wedge \mathscr{C}^T \wedge \mathscr{C}^S$.

The logic conjunction of contracts is formalized as a boolean algebra as follows: Let $\mathscr{C}_1 = (A_1, G_1)$ and $\mathscr{C}_2 = (A_2, G_2)$ be contracts, the conjunction of these contracts is a contract $\mathscr{C}(A, G) = \mathscr{C}_1 \wedge \mathscr{C}_2$, defined by:

$$\begin{aligned} A &= A_1 \cup A_2 \\ G &= G_1 \cap G_2 \end{aligned}$$

Contract composition In the CBD paradigm, the design of a complex system is based on the architectural decomposition of the system into smaller subsystems. In this context, system requirements, represented by system-level contracts, are decomposed into corresponding requirements of subsystems, referred to as subsystem-level contracts.

Contract composition characterizes the construction of a composite contract from the contracts of individual subsystems. For instance, the contract composition of a higher-level system S_1 is defined based on the contracts of its subsystems S_{11}, S_{12}, and S_{13}, denoted by $\mathcal{C}11$, $\mathcal{C}12$, and $\mathcal{C}_{13}$, respectively.

Accordingly, the composite contract of the system can be expressed as:
$\mathcal{C}_{11} \otimes \mathcal{C}_{12} \otimes \mathcal{C}_{13}$.

The combination of contracts associated with different components can be formalized through the parallel composition concept. Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be contracts (possibly over different sets of ports). The composite contract $\mathcal{C}_1 \otimes \mathcal{C}_2$ is a contract that can be specified by assumption and guaranteed as follows [11, 17]:

$$\begin{aligned} A &= (A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)} \\ G &= G_1 \cap G_2 \end{aligned}$$

Contract composition is associative and commutative property which means the commutative property gives a result in equivalent contracts for the composition of $\mathcal{C}_1 \otimes \mathcal{C}_2$ and $\mathcal{C}_2 \otimes \mathcal{C}_1$. Similarly, the associative property gives a result in equivalent contracts for the composition of $(\mathcal{C}_1 \otimes \mathcal{C}_2) \otimes \mathcal{C}_3$ and $\mathcal{C}_1 \otimes (\mathcal{C}_2 \otimes \mathcal{C}_3)$.

Contract refinement The refinement is a kind of pre-order on the contracts. Its purpose is to be a guarantee of substitutability. The refinement relationship specifies compliance concerns, both system behavior and valid usage contexts[41]. It also offers the ability to replace a component with a better one [42]. Let's consider an example in the context of independent development. The design of system S_1 (characterized by $\mathcal{C}_1$) will be separately specified and independently performed for sub-contracts $\mathcal{C}_{11}$, $\mathcal{C}_{12}$, and $\mathcal{C}_{13}$. The requirement for designing is that any development compliant with $\mathcal{C}_{11} \otimes \mathcal{C}_{12} \otimes \mathcal{C}_{13}$ should also comply with $\mathcal{C}_1$. Furthermore, any context is valid for $\mathcal{C}_1$ that should also be valid for $\mathcal{C}_{11} \otimes \mathcal{C}_{12} \otimes \mathcal{C}_{13}$. The merged system should behave as specified by $\mathcal{C}_1$ under such context. This is expressed by the refinement relation as follows: $\mathcal{C}_{11} \otimes \mathcal{C}_{12} \otimes \mathcal{C}_{13} \preceq \mathcal{C}_1$[41].

Contract refinement must guarantee the following argument to support independent implementability. Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be contracts. $\mathcal{C}_2$ refines contract $\mathcal{C}_1$, which is written as $\mathcal{C}_2 \preceq \mathcal{C}_1$ if any implementation of $\mathcal{C}_2$ should implement $\mathcal{C}_1$ and should be able to operate in any environment for $\mathcal{C}_1$ [41]. If $\mathcal{C}_1$ and $\mathcal{C}_2$ are two saturated contracts[1], then $\mathcal{C}_2$ refines $\mathcal{C}_1$ ($\mathcal{C}_2 \preceq \mathcal{C}_1$), if and only if:

$$\begin{aligned} A_2 &\supseteq A_1 \\ G_2 &\subseteq G_1 \end{aligned}$$

Summary: Contract-based design paradigm plays a key role in reducing complexity for the specification, design, and verification of complex systems. Developing a complex system like CPS often concerns independent development activities of different teams using different tools and skills. It relates to the decomposition of design requirements in a top-down process and the fusion of viewpoints/aspects during the integration process. Accordingly, the use of contracts in the development of CPS is related to assigning responsibilities and separating requirements into

[1] $\mathcal{C} = (A, G)$ is saturated if $A = A^{\downarrow}, G = (G \cup \neg A)^{\downarrow}$, where $A^{\downarrow}$ and $(G \cup \neg A)^{\downarrow}$ are nonempty set[41]

manageability ones in a simpler practice. Furthermore, contracts also specify the prerequisites for the correctness of component integration.

Three fundamental contract operators support the design methodologies: conjunction, composition, and refinement. In a hierarchical structure system, horizontal and vertical contracts can be used to support reasoning about complex interactions of components. The horizontal contract relates to integrating components into the composite one at the same level of abstraction. The vertical contract represents a contract refinement relationship involving a different level of abstraction.

2.2.3 Requirement formalization, contract specification, and support tools

Specification requirement in the context of system development plays an important role in all phase of the development of a product. It is used to pass and describe the necessary information and requirements to developers. The specification can also cover the development requirements in different aspects/views. It describes correctly not only the desired but also the prohibited behavior of the system components [43, 44], and gives possibilities for verification and validation. However, the use of requirements in informal or natural language may lead to ambiguities, incompleteness, or inconsistencies, because it has no constraints in syntax and semantics [20] as well as the lack of formality [45]. Hence, it is not suitable for applying formal analysis and verification processes, in particular, in the development of complex systems.

CPS design often involves capturing and controlling requirements for independent developments and integration activities. This requires a formalization of the design and integration specifications to achieve a holistic understanding to support the collaboration across different teams and the verification process during the development period. A reasonable specification formalization can make the contracts more accurate, reduce ambiguity, be simpler to verify[45], and keeps the design going in the right scope.

Using a contract-based design combined with a model-based design provides better support for compositional reasoning and refining requirements in CPS development. However, there is a need for a suitable specification language for capturing and analyzing requirements to construct well-established contract-based reasoning.

Applying natural language for requirement specification has some advantages, such as it is very intuitive and "friendly" especially for non-expert users. However, it can make the specification hard or even impossible to translate into a formal specification contract without being unambiguous and unclear. A requirement specification exploiting natural language depends on the styles of usage for vocabulary and grammar of the creators/writers, and sometimes, it requires a depiction in more detail than expressed with a formula. Therefore, natural language represents an inconvenient language for characterizing the contract specification in the context of complex system development.

In system-level design, formal specification languages are rigorous techniques

for the specification to characterize system requirements and properties within precise, unambiguously defined syntax and semantics. With a well-defined syntax and semantics, it can offer the ability for automatic verification and validation. Hence it is usually noticed as better support for requirement specifications in the contract development framework. Temporal logic (TL) and its variants, which include Linear Temporal Logic (LTL)[46], Metric Temporal Logic (MTL) [47], and Signal Temporal Logic (STL) [48] are reviewed as good support tools for formalizing system requirements and contract specifications. However, in contrast to nature language, a formal language for expressing system specifications is hard to write and understand [49]. Therefore it is a challenge to use, especially for non-experts.

Semi-formal specification combines natural and formal languages and can bridge the gap between them. Semi-formal specification often represents a pattern-based (template-based) or text block-based language. Furthermore, it features well-defined semantics and can easily translate into a verifiable formal language. Using pattern language coming with a support tool gives a better solution for specifying contracts. Here, the contract specifications can have rigorous semantics like formal methods. Additionally, based on a set of template text, the requirements can be expressed similarly using natural language. Therefore, the contract specification can be easily characterized even by non-experts.

There are several frameworks and support tools have been introduced to apply contract-based design techniques for the design of complex systems.

HRELTL [50] (Hybrid Linear-time Temporal Logic with Regular Expressions) is a temporal logic for hybrid systems. It is a combination of temporal operators and regular expressions for characterizing both discrete and metric real-time constraints. It can therefore support contracts specified in a temporal logic for hybrid systems [51]. On the other hand, the contract-based specification language named OTHELLO ("**O**bject **T**emporal with **H**ybrid **E**xpressions **L**inear-time **LO**gic") was introduced as a specification language for hybrid properties. This language combines the HRELTL language with the object-oriented temporal logic [52]. It can use graphical representations and natural language expressions to support the formal requirements applying for hybrid domains [53].

The OCRA (**O**thello **C**ontracts **R**efinement **A**nalysis) [54] tool is an Othello-based tool support for developing a contract-based framework for compositional verification and reuse of components. It builds on top of the NuSMV [55] model checker framework to support the specification of components enriched with Othello contracts[2]. Furthermore, the tool can be used for automated checking and analyzing the correctness of the contract refinement and implementation. In the OCRA tool, contracts are characterized by a TLT or HRELTL [50].

CHASE (**C**ontract-based **H**eterogeneous **A**nalysis and **S**ystem **E**xploration)[56] is considered the first contract-based framework to cover the end-to-end solution of the design of complex systems. It can perform a comprehensive validation of different approaches to requirements validation. The CHASE framework provides an end-to-end practice for applying contract-based design using a combination of front-end and back-end modules. The front-end supports a pattern-based method

[2]contracts are specified in discrete or hybrid linear-time temporal logics

for expressing requirements. The libraries in CHASE (written in C++) provide the necessary support tools for methodologies and design flows, for example, structuring, managing, and analyzing specifications. A set of back-ends allows translating those specifications into suitable formal languages for assessing requirement correctness, completeness, consistency, and refinement checking [57]. The tool supports the automatic translation of requirements expressed in natural language to formal methods. Therefore, contract reasoning, validation, and verification can easily perform automatically. CHASE supports requirement analysis for various richer logic languages[57], for instance, HRELTL [52], STL (Signal Temporal Logic) [58], and StSTL (Stochastic Signal Temporal Logic) [59].

A design framework for Multi-Layer Time Coherency (MULTIC) [60, 61] has been developed for support timing specification purposes. This tool provides the syntax and formal semantics for applying contract-based techniques of timing aspects in Advanced Driver Assistance Systems and Automated Driving (ADAS/AD). By using pattern-based requirement specification language(RSL) [45], the specification of timing properties following the contract-based can be formulated through different sets of patterns. In MULTIC, the timing specifications are inherited from AUTOSAR frameworks and defined extensions for reasoning about timing within complex scenarios. MULTIC provides four different design paradigms including *Compositional Semantic Framework*, *Timing Specifications*, *Models of Computation*, and *Converter Channels*.

- *Compositional Semantic Framework* provides support for system design in a hierarchical structure. It gives formally well-defined concepts allowing to reason about consistency between the individual models in that structure.
- *Timing Specifications* provides essential support to simplify expressing timing specifications for applying the contract-based method in the system design within complex scenarios.
- *Models of Computation* provides the formal basis for modeling the behavior of components based on different notions of time. It also supports the integration of component implementations into the semantic framework.
- *Converter Channels* provides support to the interaction components with different MoCs. For example, a component implemented in C++ and a component implemented using Matlab/Simulink can interact with each other through *Converter Channels*.

In MULTIC, the syntax and semantics of contracts are used to characterize timing phenomena and effects by the so-called MULTIC Timing Specification Language (MTSL). This tool can automaticaly translate specifications into executable generators and monitors. The simulation-based validation and verification and the contract-based virtual integration testing (VIT) are the highlights of this tool.

3
State of the art

Contents

3.1 Model-based design: Methods and tools for CPSs . . . 30
3.2 Modeling and controlling of CPSs using IEC-61499 . . 32
3.3 Contracts for Safety-critical CPS 34
3.4 Summary 36
3.4.1 Limitations of the state of the art 36
3.4.2 Thesis Contribution 38

Most researchers consider cyber-physical systems (CPS) a critical enabler for Industry 4.0. The term CPS originates from embedded systems engineering. Therefore, in the beginning, the definition of CPS focused only on the aspect as a product of the synthesized integration of heterogeneous components, where components cooperate and exchange data via technical communication networks. In recent years, the CPS term describes different systems in many application domains. In the manufacturing domain, CPSs are automated distributed systems [62, 63]. It targets support controlling, auto-monitoring, and data sharing in real-time for physical processes. Depending on the standpoints on the definition of this system influence the establishment of solution approaches, the building of design methods/techniques, and the selection/development of support tools.

As mentioned in chapter 1.1, this thesis presents an approach combining three technologies: Model-based Engineering, Contract-based Design, and IEC 61499. It aims to give solutions to address challenges for safety-critical CPS development focused on the industrial application domain. This chapter provides an overview and study of some related works that draw strongly on the exploration of aspects and concepts of *Model-based design: Methods and tools for CPSs*, *Modeling and controlling of CPSs using IEC-61499*, *Contracts for Safety-critical CPS*.

3.1 Model-based design: Methods and tools for CPSs

CPSs are complex systems that emerge from the integration of dynamic physical systems/environments and cyber (computing and networking) systems. The research task in CPS development is to discover solutions/methodologies to combine knowledge and engineering principles across the computer and engineering disciplines to produce new CPS and supporting technologies [64]. Due to the mentioned complexities, the development of CPSs usually cannot be performed using a specific framework. It should be instead a design process based on the composition of elements of independent development activities in a hierarchical structure. It is also known as component-based development that permits modular design for complex systems.

Designing a CPS with heterogeneous components/subsystems distributed across networks is a challenge that needs to be overcome. Model-based design (MBD) has been the most widely exploited as a powerful paradigm for the compositional reasoning of components in complex system design [65, 66, 67, 68, 69, 70, 71, 72]. It provides an advantageous solution for component-based design in modeling, testing, and verifying processes [64] during the development cycle. A model of CPS characterizes the incorporation of multidiscipline models such as hardware, software, networks, and physical processes models. In this regard, using multi-paradigm modeling is highlighted as an appropriate preference to deal with the challenge of the heterogeneity of models in the CPS design. However, the selection of modeling frameworks/tools for models in CPS is different and highly depends on the perspective of developing methodologies and the application domains from which they start. Some literature reviews provide surveyed languages and tools for involving multi-paradigm modeling (MPM) approaches for CPS[73, 74, 75].

[73] delivers a catalog of the most general languages and tools concerning the formalism aspect of CPS development using MPM such as Petri Nets, Hybrid Automata, Timed Automata, and Temporal Logic, etc. The paper in [75] exhibits the result of a survey on tools and modeling languages for building CPS using MPM approaches. It shows that the tools included Unified Modeling Language (UML)/Systems Modeling Language(SysML), Modeling and Analysis of Real-Time and Embedded (MARTE), Architecture Analysis & Design Language (AADL), MATLAB/Simulink, Modelica, UPPAAL, 20-sim [76], which are usually selected to characterize the behavior of models.

On the other hand, the hybrid model approach for CPS modeling was also introduced by many researchers as a comprehensive solution [77, 78, 79, 80]. The hybrid model approach involves the development of a mixed model in which it covers multiple levels of abstraction and different paradigms. In other words, a model of CPS will be a mixture of continuous- and discrete-time models. In this regard, the model of physical components can be analog elements, physical dynamic systems, and the environment. Their behavior is specifed by continuous time models through differential equations. These component models are mostly built using Simulink, Labview, or Modelica [78]. The model of cyber components involves the

discrete-time/event models and the logic-driven elements. It is used frequently for characterizing the management, control, observation, and data interaction operation, that is often built based on discrete-event system (DES) models. In DES, automata (timed and hybrid automata) finite-state machines or Petri nets are typical discrete-event formalisms that are used to describe the discrete and logical behavior of these components.

Hybrid Automaton (HA) [81] is a mathematical model for the precise description of systems. HA combines the continuous, dynamical evolution of states described by differential equations at each location with transition states. There are two types of state transitions: continuous flow transitions, which occur over time, and discrete jump transitions, which occur immediately. *Timed automata*(TA) [82] is a critical subclass of HA, extended with real-valued clock variables, usually used to model real-time systems. The features of HA can be found in the various tools such as *Modelica*, *Ptolemy II* [83], *SpaceEx* [84]. The features of TA can be used with the *UPPAAL* [85] tool.

The work introduced in [77] uses modeling and simulation environments in *Ptolemy II* to address the challenges in CPS models such as hybrid system modeling and simulation, concurrent and heterogeneous models of computation. In [86], the authors introduced the use of a model-based design methodology with *SystemC* [87] based simulation framework to deal with different models of CPS. Unlike *Ptolemy II*, which is an environment for simulation of diverse *Models of Computation*(MoC), *SystemC* provides only discrete-event simulation. Thus, to model and simulate continuous-time physical processes, the *Transaction Level Modeling* (TLM) and *SystemC-AMS* extensions were added to handle abstract communication and continuous behavior issues. The authors in [88] proposed to use *Modelica* for modeling and simulation of CPSs. By integrating the timing simulations based on the *TrueTime* simulator into a support library called *Modelica_EmbeddedSystems*, this work presented and demonstrated the capability to model and simulate CPS realistically with timing effects.

Some studies have presented co-simulation methods [89, 90] using the *Functional Mock-up Interface*(FMI) standard for model exchange and simulation by different tools. The models may then cooperate with the simulators for network and communication. The *INTO-CPS* [91, 92] is a project aimed to create an integrated toolchain for comprehensive Model-Based Design support the multidisciplinary, collaborative modeling (co-modeling) and simulation (co-simulation) of CPSs [70]. By using the *INTO-CPS* technology through co-simulation based on *Functional Mock-up Units*(FMUs), the authors in [72] demonstrated the flexible methodology of modeling CPSs. The works proposed by [93] introduced using the standardized model exchange and co-simulation interface FMI to connect two modeling and verification tools for CPSs. The behaviors of component models and sub-models are described via *SpaceEx* by using HA. The emphUPPAAL uses the TA model to describe behaviors of a timed system.

The authors in [94] suggested a co-simulation framework called *Cyber-Physical Co-simulator* (CPS-Sim) to integrate the physical system simulator using *MATLAB/Simulink* [95] and the network simulators using *QualNet* [96] and *OMNeT++*

[97]. In [98, 99] represented a co-simulation framework using High-Level Architecture (HLA) and Functional Mock-up Interface (FMI) standards. The Embedded Control System (ECS) model is specified in the *Parallel Object-Oriented Specification Language* (POOSL) [100]. POOSL is a discrete-time modeling language. Therefore, it is a suitable tool for the modeling of software architectures. The physical system model can be characterized in OpenModelica [98] or 20-sim [99]. The discrete-time model of control software can be developed in either 20-sim [99] or POOSL [98].

On the other side, [101, 102] argues that co-simulation techniques have to fight against various problems such as free-running, jitter, slow sampling rates, and other synchronization. For this reason, the author [101, 102] presented a new methodology for plant design and simulation. In this regard, the plant can be directly developed in IEC 61499 through an extension called Hybrid Function Blocks (HFBs). The HFBs are based on Hybrid Input-Output Automata (HIAO) and are defined with formal synchronous execution semantics. As a result, they provide the possibility to describe the continuous dynamics of the physical plant.

Many authors emphasize that MBD is a powerful design to deal with the complexity of CPSs. The advantages and benefits of the features of model-based techniques and tools have been widely adopted. However, CPS development is a complex multidisciplinary process. The design of such a system requires often the service of multiple modeling paradigms and methods. It also involves the independent design activities of different teams using diverse tools and languages. Therefore, it is necessary to have a suitable development framework where the multi-domain components can be modeled, executed, and monitored in a unified framework. If so, it would be a great solution to overcome the limitations of co-simulation and to ensure the correctness of the timed functionality and real-time operation of each component and the entire system.

3.2 Modeling and controlling of CPSs using IEC-61499

From the early stages of the standard's development, the IEC 61499 has been defined and designed as a generic reference architecture for building new flexible low- and high-level control concepts to develop distributed control software [32, 3, 9], especially Cyber-Physical Systems (CPS). The survey in [40] examined several standards and reference architectures, such as *OpenRTM*, IEC 61804, and IEC 61499. The authors also provide an overview and assessment of various modeling frameworks, such as *MATLAB/Simulink*[95], *SCADE*, and *OpenRTM*. As a result of the comparison among all assessed reference architectures and modeling frameworks, the author underlines that IEC 61499 provides the best properties for model-driven development of control software for distributed automation.

The central modeling element of IEC 61499 is the function block (FB), a composable representation of behavior activated through event ports, with optional data ports. Its execution rules rely on the event-driven paradigm. Event-driven execution is the fundamental mechanism for transparently modeling distributed

systems [9]. In addition, using IEC 61499-based FBs enables modular construction of control design. It can model miscellaneous physical components, services, and systems. Therefore, it makes reconfiguring, reusing, and integrating components in complex automation systems easier. Many research works have been trying to introduce how IEC 61499 can use as service-oriented architecture to model and improve a flexible and reconfigurable control applications in industrial automation context. [9] presented an overview of research activities and results using IEC61499 on industrial automation context, support tools, and aspects of IEC61499 execution on devices, applications, and distributed automation systems. The developing, simulation, and execution of distributed automation and reconfigurable control systems implemented on the IEC 61499 standard were introduced and demonstrated [32, 103, 16, 3, 35, 104]. Many researchers have agreed that using the component-based design method, where the components are developed and implemented in the IEC 61499-conformant framework, is an excellent strategy. [105] proposed an approach for applying component-based design to reusable intelligent software components of batch processes based on ISA S88 and implemented in IEC 61499 Standard. [16] introduced a multi-agent control approach for a baggage handling system, where the reusable intelligent components are designed and implemented based on the component-based architecture and the IEC 61499 standard. The use of the IEC 61499 standard in combination with component-based design is accepted as a solution to provide scalable, reconfigurable, and fault-tolerant distributed control. The feasibility of IEC 61499-based implementation and control of CPS using a low-cost device such as Raspberry Pi was demonstrated in [106, 107].

IEC 61499 applications are usually developed and implemented as hierarchical solutions. [108, 109] presented a successful transfer of hierarchical automation solutions, which were developed and implemented in MONACO (**MO**deling **N**otation for **A**utomation **CO**ntrol) to better structure component-oriented hierarchical components in IEC 61499. These works [108, 109] used the 4*diac* tool to model and build the IEC 61499-compliant control application.

The works presented in this section are primarily concerned with the development methods and modeling and simulation techniques using IEC 61499. Moreover, many works have also described the guidelines for designing, implementing, and controlling a distributed control system based on the component-based design method implemented in an IEC 61499 framework. However, the specifications describing the desired system behavior are still missing and not provided in IEC 61499. There are diverse research works [37, 110, 111, 112, 113, 114] which present a hybrid approach to handle the requirements specifications of the behavior of IEC 61499 control applications. For example, the application of a combined methodology using IEC 61499-based models with a set of diagram types in the Unified Modeling Language (UML) [115] or Systems Modeling Language (SysML) [116], which supports specification, analysis, verification, and validation. With this approach, the behavior specification of the component models will be first described using different types of diagrams, such as class diagrams, activity diagrams, sequence diagrams, use case diagrams, state machine diagrams, Block Definition Diagrams, and requirement diagrams to express different aspects/views. Those models (UML/SysML) will

be then transformed to IEC 61499 FB models through a specific software toolset. Although the ideas of combining UML and SysML with IEC 614499 provide excellent support for distributed automation systems development, those approaches were centered mainly on defining the specification of components at an abstract level before the actual design and implementation.

In CPS development, model-based design can enrich interface specification and composition capabilities. Many works focused on reflecting behavioral properties in interfaces, especially in general-purpose computing. It is still necessary to reflect causality properties for expressing abstract temporal behavior, real-time resource usage, and timing constraints [117]. Even the IEC 61499 standard provides a solution to describe the behavior of FB through a service sequences feature. That is still not enough to describe the behavior in more complex cases and detailed requirements, especially in CPS design. Several other studies have focused on addressing issues related to the execution time of IEC 61499 applications. For example, the work [118] introduced the comparisons of different implementation strategies for IEC 61499 FBs and the execution overhead in a Java-based environment. The issue of guaranteeing hard real-time constraints of IEC 61499 applications was comprehensively handled by [119]. [120] presented an approach of formal description for runtime behavior and real-time constraints into the verification process of IEC 61499-based automation systems. [121] argues that IEC 61499 and other component-based architectures offer outstanding support for dynamic reconfiguration and adaptability. However, it is crucial that the real-time behavior is deterministic and well-defined. To provide accurate and rigorous deterministic real-time execution of safety-critical real-time systems, enhanced functional and temporal correctness are required. Therefore, combining the contract-based method and the component-based design using IEC 61499 offers a suitable technique for distributed automation system developments, especially CPS. This may provide a way for specifying and monitoring the functional and timing behavior of components during the development and integration of such systems.

3.3 Contracts for Safety-critical CPS

Assume-Guarantee contract for CPS has been widely explored. The proposed methodology in [17] describes a way to combine contracts with platform-based design (PBD) to address challenges of CPSs design such as the complexity and heterogeneity of multiple engineering domains. In particular, contracts are applied to assure the correctness of element integration at the same level of abstraction and for consistent verification across lower and higher levels of abstraction in the context of system-level design. With a similar approach, the works in [11, 122] presented the methodology to formalize different levels of abstraction and viewpoints involving horizontal and vertical contracts in the design process by combining the PBD and contracts. Establishing consistency, compatibility, and different refinement relationships between these contracts allows the design of systems to be implemented in a hierarchical and compositional way.

For reasoning and expressing the contracts about different requirement aspects

(e.g., timing constraints, functional behavior) that need to use a suitable specification language. The approach in [123] introduced the framework of hybrid automata that is considered as a generalization of timed automata, used as a generalized formal model and specification language for hybrid systems and automatic verification purposes. Follow this approach, the authors in [124, 125] used a formal verification tool called ARIADNE [126] for hybrid systems. In particular, the use of the common meta-model called *Heterogeneous Rich Component*(HRC), where functional and non-functional aspects of the system are combined, enables the allocation of responsibility among different developers during the design process and the support of verification models in the integration process was proposed. The authors in [127] presented several different types of contracts, referred to as "design contracts" to address the timing aspects of interaction and communication behavior involving control and computation/communication problems across two design groups. In particular, the sampling-to-actuation (StA) delay problem in control theory and response time has been covered using the time tolerance (TOL) contract in various experiments that have been conducted with the use of Simulink and Ptolemy. However, formalisms and languages that can help to concretely capture design contracts, are still lacking in this work [127].

There are a lot of researchers who have suggested different languages and techniques for expressing contract specifications and requirements. For the use of timing contracts, the pattern-based requirement specification language(RSL) [45] is used to formulate requirements as assumption and guarantee contracts form. The work [60, 61] identified suitable design paradigms for the handling of time and timing effects in ADAS/AD (Advanced Driver Assistance Systems/Automated Driving) by applying contracts from a set of timing specification patterns to specify timing properties. These contracts are modeled in the SysML model formalism for the timing specifications of the system and utilized to ensure traceability as well as compositional analysis throughout the development process. By separating assumptions and guarantees from the timing behavior of the interacting components, contracts are capable of tracking timing properties within subsystems by assuming a certain guarantee. The approach in [49] implemented the use of CBD methods for the design of safety-critical CPS, in particular, the contract concept is classified into two types of component and interface contracts to obtain better support for system design correctness validation. A compositional approach in [128] presented an effort to bridge the gap across CBD and dynamically changing systems to control the traffic networks, for instance, timing requirements for traffic lights using the concept of dynamic contract, which is formalized by the temporal logic specification. The use of formal languages based on temporal logic [46] has been proposed by many researchers with various approaches. For instance, the formal property specification languages based on linear temporal logic (LTL) for contract-based design by extending the LTL with Regular Expressions to formal expression of requirements in hybrid domains have been introduced in [51, 50, 129], the signal temporal logic (STL) is used to handle dense-time real signals and hybrid dynamical models[130]. The specification method called metric temporal logic (MTL), an extension of the temporal logic to formalize timing requirements and properties

of real-time systems was proposed in [47].

The foregoing investigations have demonstrated the benefits and impacts of adopting a contract-based design framework in many contexts of complex and safety-critical system engineering. Likewise, different types of contracts have been created and implemented on different design platforms. However, the combining of the model-based design framework with contracts under the use of the IEC 61499 standard is not addressed or discussed here at all. We can recognize that, on one side, the IEC 61499 standard is an extension of the 61131-3 standard, which is widely used in industrial automation solutions, especially distributed control systems in the manufacturing and production sectors. On another side, contracts exhibit usefulness in handling the development and management issues of distributed and multi-domain systems. This combination is essential and is expected to be of high benefit in the process of developing and controlling complex systems, such as CPS. Another aspect to note is that when the use of contracts in IEC 61499-compatible tools becomes a reality, the contract specification language should be constructed to allow it to be used in IEC 61499 function blocks properly.

3.4 Summary

3.4.1 Limitations of the state of the art

So far, the most important considerations concerning the investigation of the design of complex distributed control systems such as CPS can be summarized as follows:

Model-based design methodologies are widely recognized and adopted in the design of large-scale complex systems like CPS. The construction of the system based on a hierarchical model and the use of contracts to formalize the specification in the design phase, and to integrate components into the overall system in the test phase, is a useful solution chosen by many researchers to cope with the distributed, autonomous and complex nature of CPS development. On another side, the introduction and applying of the IEC 61499 standard have also demonstrated advantageous achievements and practical feasibility in the modeling and development of distributed systems. On the other hand, the introduction and application of the IEC 61499 standard as a supporting framework to perform the model-based design of CPS has also demonstrated advantageous results and practical feasibility in the modeling and implementation process. With the same opinion as the authors in [103], the plant simulation and the distributed control system can be satisfactorily carried out with a discrete model of computation. Therefore, all IEC 61499-compliant models in the mentioned work will be directly accomplished by using the IEC 61499-compliant development tool. However, there is still effort to integrate contracts into functional blocks as well as applications that comply with IEC 61499.

An approach using IEC 61499 for building hierarchical control systems in a master-slave arrangement has been introduced by [131]. In particular, the interaction between components on different hierarchy levels has been verified by behavioral event contracts. In this sense, assumptions must cover the order of received input events. For output guarantees, the order of emitted events has to be ensured, if

the received input event sequence fulfills the assumption. However, the contract specification in this work is limited to event sequences, acceptable for the function blocks, and does not deal with the transfer of data through data ports. This leads to a verification algorithm considering only event sequences as defined in ECCs without any check for condition values of data variables. The work presented in [132] targets static verification at the component, algorithm, and *ECC* levels of IEC 61499 function blocks, where the IEC 61449 components are manually translated into *WhyML* language and enriched with the necessary contracts under pre- and post-conditions form. Verifications are conducted in the *Why3* platform, which takes into account the correctness of IEC 61499-compliant components at different levels such as BFB algorithm (*ECC* level), component (functional block level), and application (functional block network). However, [132] showed unsolved issues including automatic contract generation from function blocks and the lack of defined semantics for reasoning on causal and timing properties on IEC 61499 models.

From the above-mentioned points, we discovered two answers to the question *Q1* and the question *Q3*. In particular, the answer to the question *Q1* is that the existing IEC 61499 compliant development software tools do not yet support the requirement specification in various aspects, especially timing and behavior. The answer to question *Q3* found here is: for specification languages, the ability to test the functionality of the IEC 61499 function block network is limited because they are either not yet fully explored or still have a huge limitation, e.g., it only addresses some issues at the function block level, but the test and observation of an application or function block network in regards to timing constraints or functional behavior are not yet fully complete.

To realize the implementation of contracts in IEC 61499 compliant applications and run-time systems, the proposed approach in [133] was presented as a pilot experiment of this thesis, in which the different types of contracts were integrated into *FORTE* (*4diac* run-time environment)[1] under different contract function blocks that had been created using the IEC 61499 compliant tool, *4diac*. In addition, the feasibility with respect to achieving this purpose has also been demonstrated in that work through an evaluation example. However, several limitations are still present, e.g. syntax and semantics for the specification language of the proposed contracts are still missing. This mentioned lack was filled and extended by later publications in [134, 135]. Especially, in the context of using contract-based design combined with an IEC 61499 compliant tool, the inheritance of existing formal specification languages is not possible in the current version of the software due to the limitation of the use of Unicode special characters and symbols. Therefore, along with the creation of contract FBs including time-sensitive behavior contract (*TSBC*)[134] and timing contract function block (*TCFB*), a pattern-based language to support contract-based reasoning to monitor functionality and timing behavior of IEC 61499 FBs were also presented. More details on these contract function blocks (*TSBC* and *TCFB*), as well as syntax and semantics related to pattern-based specification language are given in Chapter 4.

[1] *4diac-IDE* and *FORTE* version 1.9.0 were used for this work.

3.4.2 Thesis Contribution

Even though each of the aforementioned design concepts is perfectly well-known, there are also some research efforts focusing on the combination of them to cope with a complex issue for instance the combination of model-based design and IEC 61499 architecture for modeling and simulation of CPS [101], or the approaches of combining model-based design with a contract-based design methodology for safety-critical CPS development, or the approaches of applying contracts to address the verification issue such as the correctness of IEC 61449 models or the ordering of events received and sent by a component in hierarchical IEC 61499 solutions. However, there are no efforts to integrate all of them for ensuring the correctness of operation and monitoring behaviors in functionality and timing aspects for safety-critical CPSs. This thesis will present the following contents:

- Model-based development of CPS using IEC 61499: A model of a CPS includes physical process models combined with models of the software, computing platforms, and communication networks. Therefore, it needs a rich integrated modeling framework for specification, architectural modeling, and monitoring the interactions between these elements or components. In this thesis, the plant modeling and controller design of such systems will be accomplished directly and based on the discrete computational model (*MoC*) using the IEC 61499-based design framework *4diac*.

- Integration of Requirement Specifications for IEC 61499 Function Blocks: In this thesis, requirement specifications and assertions for IEC 61499 function blocks are formalized using a proposed syntax to ensure and monitor the correctness and safety of IEC 61499 components during implementation. The contract specification syntax can be defined either directly within contract function blocks or through a supporting tool called *PassCS* (Pattern-based Language Support for Contract Specification). This tool provides semantic support for contract specification and translates high-level specifications into executable syntax, enabling the automatic integration of IEC 61499 contract function blocks developed in *4diac*. Integration of contract-based monitors: Contracts will be integrated into the IEC 61499-based modeling tool and simulation environment, which aims for implementing contracts on the IEC 61499 runtime environment. Especially, this work introduces the concepts of contract function blocks *TSBC* and *TCFB* to build IEC 61499-compliant applications with contracts, where the *TSBC*s will be applied for a functionality specification and assuring the timed functional behaviors of a component, while the *TCFB*s are used to specify timing constraints for assuring and monitoring the component operation in timing aspects.

- This work also introduces an exception handling mechanism for contract function blocks. The purpose of this mechanism is to allow values resulting from a contract violation to be re-evaluated under alternative conditions (i.e., exception conditions). This approach reduces rigidity in the application of contracts during system operation. For instance, if a contract violation

occurs at a specific time but does not affect the stability of the component itself or the overall system, it may be accepted through the defined exception handling mechanism. Furthermore, this work presents extensibility options for specifying more complex requirements.

- Through establishing assumptions about the test environment to be used for evaluation purposes, the methodology used to address issues of ensuring and monitoring the correctness of behavior and time constraints will be introduced and evaluated through simulation-based validation procedures.

Note, that these contributions are established and evaluated based on the assumptions from the operating environment of the IEC 61 499-compliant development software tool *4diac*. The details of these assumptions will be given more fully in chapter 5. Moreover, it is theoretically possible to obtain similar results when deploying an application using these approaches/contributions within other working environments/platforms, but the verification of this argument is beyond the scope of this thesis.

4

Approach to integration contracts into IEC 61499

Contents

4.1 Methodologies 42
4.2 Building IEC 61499 contract function blocks 48
4.2.1 Time-Sensitive Behavioral Contract (*TSBC*) 48
4.2.2 Timing Contract Function Block(*TCFB*) 51
4.3 The specification language 52
4.3.1 Specification contract for *TSBC* 52
4.3.2 Specification contract for *TCFB* 55
4.3.3 Specification support tool 61
4.4 Exception handling 63
4.4.1 Motivation and problem definition 64
4.4.2 Syntax for exception handling specification of *TSBC*s . 67
4.4.3 Syntax for exception handling specification of *TCFB*s . 74
4.4.4 BNF 76

This chapter will start with an overview of the methodologies and used techniques in this thesis. Additionally, detailed descriptions of the construction of contract functional blocks, such as time-sensitive behavioral contracts and timing contracts, will be introduced in the following sections. Besides, the syntaxes and semantics of the corresponding contract specification language, which are used in the IEC 61499-compliant tool and the contract specification support tool are also explicitly outlined.

4.1 Methodologies

The research method is briefly described in Fig. 4.1. The main focus of this thesis is the combining model-based engineering with contracts for CPS design and monitoring using the IEC 61499 framework. At the beginning of development, the

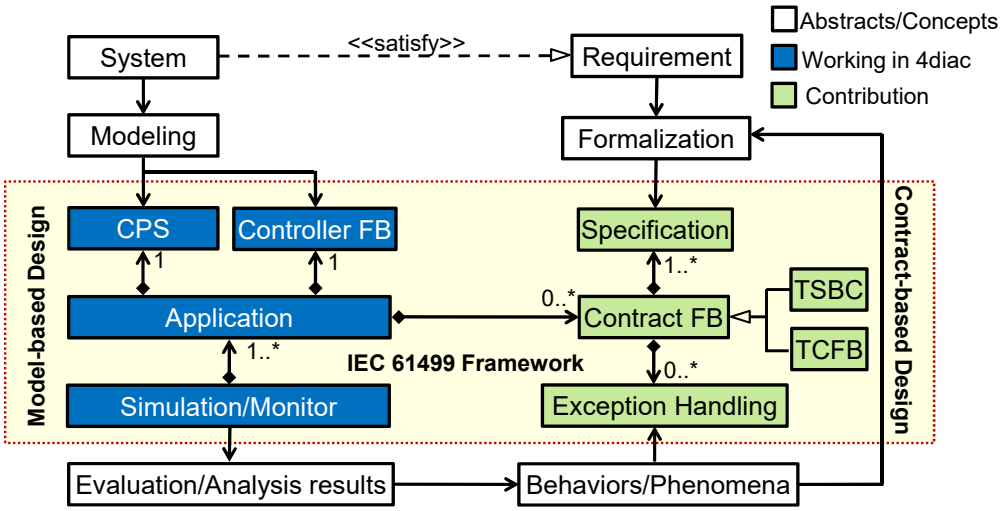


Figure 4.1: Description for methodology and research design

requirements define the desired behavior of components (plant, controller) and the expected properties of the environment of those components. The requirements can be a fusion of various requirements concerning different viewpoints such as functional, timing, safety, etc. All these requirements will be expressed under terms of A/G contracts where the assumptions and guarantees are formalized and specified through the new proposed syntaxes. These contracts can be integrated into the IEC 61499-compliant application via the newly developed contract FBs.

In the context of MBD using IEC 61499 to model the system, the component models are characterized by FBs. The *Application* is a network of FBs built in a hierarchical structure. The model of the component, subsystem, or system will be concatenated to contract FBs and then encapsulated in *CFB*s. In this work, the development will be carried out based on the IEC 61499-compliant tool *4diac*. One of the highlights of this thesis is the building of contract FBs for enclosing the requirements/agreements into the IEC 61499-compliant applications in the *4diac* tool. There are two types of contracts, which have been developed including *Time-Sensitive Behavior Contract*(*TSBC*) and *Timing Contract Function Block*(*TCFB*). *TSBC* is used for monitoring and ensuring the correctness of the timed functionality of a component. *TCFB* is used for monitoring and ensuring the timing constraints for the execution of an object (component, subsystem, etc.) or a process. Thanks to these contract FBs, the requirements specifications for environment properties, entity behaviors (components, subsystems, systems), and timing constraints of processes can be integrated into the *4diac* runtime environment called *FORTE*. As a result, simulation of the system deployment can be executed under supervision of those integrated contracts in *FORTE* to collect the response behaviors/phenomena of

components, subsystems, and the system. The simulation results will be analyzed to help the formation of contract specifications and conditions for exception handling appropriately and safely. That makes the use of contracts and their exception conditions for components and processes in the systems possible that can smoothly execute under the properties/scenarios of the implementation environment in the real-time context. Note that the simulation is not only conducted during the system integration phase but also performed for every component model in the design phase. It helps to obtain the specifications of that component early and actually. Thus, the actual requirements can be updated and shared promptly with other independent development teams helping to early identify risk, rather than waiting for the system integration phase.

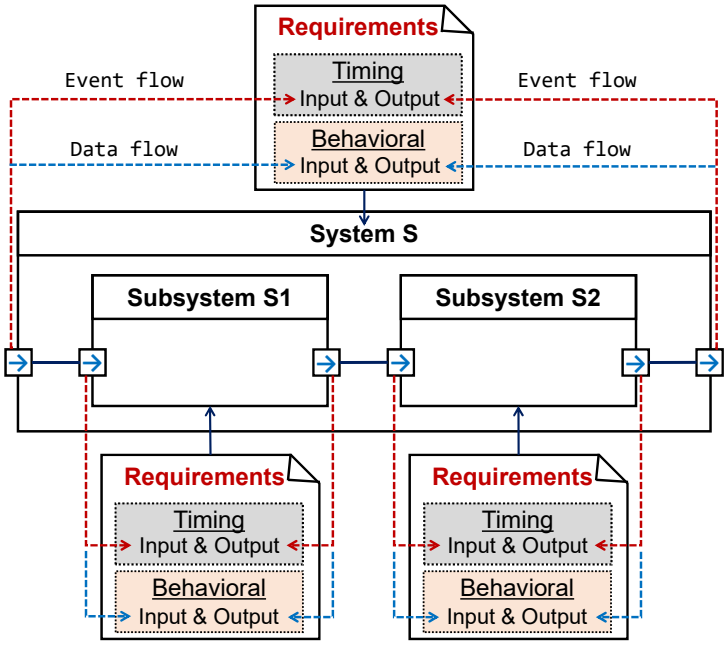


Figure 4.2: Integration examples of the contracts

One of the advantages of the IEC 61499 standard is that the interface FBs are divided into two separate parts, event and data ports. In addition, a data port must be associated with at least one event port to enable data exchange. For example, if a data port expects to read/write values from/to other data ports, it must wait for the triggering of the event with which it is associated. That is the premise for building the ideas using contracts for monitoring the timed behaviors and timing constraints for components and processes in the systems.

Fig. 4.2 illustrates how two types of contracts would be applied to components and the overall system. Behavior requirements refer to observing the correctness of component behavior. In this case, the assumption checks the satisfaction of input data, and the guarantee verifies the fulfillment of output data compared to the expected range values. The timing requirements refer to checking that the input and output events have occurred during the desired time interval. The contracts apply to every component, subsystem, and overall system in a hierarchical

structure. Thus, checking for compatibility, consistency, and refinement of those contracts is also required.

To explain this idea more clearly, let's take an example of requirements for a component that are expressed under aspects of functionality and timing as follows:

- **For behavioral requirement**: An input value of the component must not exceed a limited range of values. If this requirement meets, its output value must be in the given range of values.
- **For timing requirement**: The input event, which triggers the activation of input data, must occur every $T[ms]$. Whenever that input event occurs, the output event, which associates with the output data, must occur within $[t_1, t_2]$ after that.

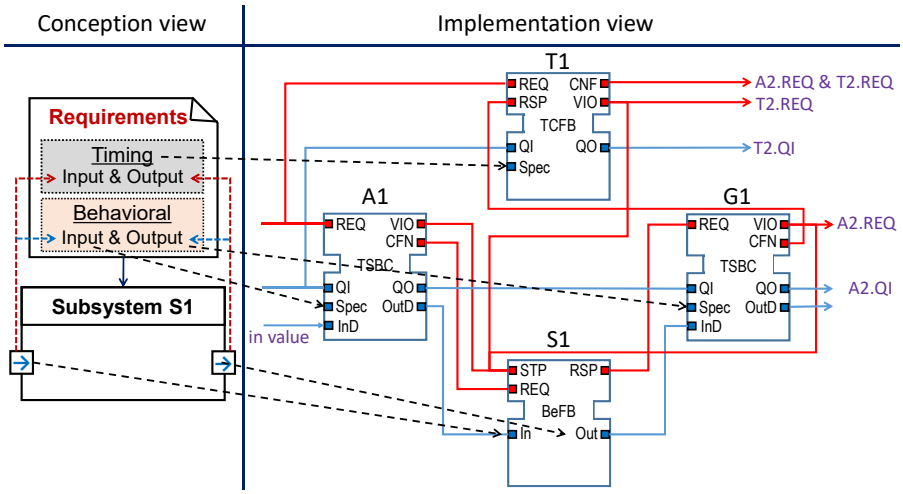


Figure 4.3: Description for applying contracts in conception and implementation view

Fig. 4.3 depicts the solution by employing contracts in the conceptual and implementation view in the framework of IEC 61499. The primary features of the contract FBs *TSBC* and *TCFB* will be employed to ensure the above requirements in the following manner:

- For ensuring behavioral requirement: A *TSBC* is used to express the assumption (*A1*) and another *TSBC* is used to express the guarantee *G1*. The syntaxes of behavioral requirements for input (Assumption) and output (Guarantee) are formally specified at the port *A1.Spec* and *G1.Spec* respectively. The *A1* checks contract satisfaction for input data (*val1*) of behavior FB (*S1*) at port *A1.InD*. [1]. In this regard, there are two scenarios that can occur.
  - In the first case, if the value *val1* satisfies the assumption (in other words, *val1* meets the check condition expression $val1 \in [a_1, a_2]$) $\rightarrow$

[1]For an example of *Assumption*, the requirement for the component's input at *S1.In* is that the input value *val1* must never exceed a given range of values $[a_1, a_2]$

The event *A1.CNF* associated with *A1.QO = TRUE* and *A1.OutD = A1.InD* is triggered to activate the *S1.REQ* event for normal execution of behavior FB (*S1*). As a result, the satisfied data (*val1*) will be allowed to read at port *S1.In*. Simultaneously, the data input *A1.QO*, which is linked to port *G1.QI* of *G1*, write the logic value *TRUE* for enabling the execution of the guarantee FB (*G1*). Upon completion of execution of the behavior of FB, the event *S1.CFN* is activated to request the execution of *G1* at *G1.REQ*. The *G1* will run to check the correctness of the component's behavior through the value of the output data at *S1.Out* [2]. When the output value does not meet the guarantee, *G1* will trigger the event *G1.VIO* associated with *G1.QO = FALSE* to stop the execution of *S1* at *S1.STP*. Parallel with that, this event (*G1.VIO*) also triggers the event *A2.REQ* associated with *G1.QO = FALSE* $\rightarrow$ *A2.QI = FALSE* to avoid the execution of the following behavior FBs. In case the output of *S1* fulfills the guarantee, *G1* emits the output event (*G1.CNF*) associated with *G1.QO = TRUE*. Therefore, the downstream assumption FB (*A2*) will be requested for the execution through a trigger of event *A2.REQ* associated with *A2.QI = TRUE* enabling the check of the next component.

- In the remaining case, when the assumption *G1* does not meet the check condition (contract violation occurs at *A1*, because of $val1 \notin [a_1, a_2]$), then the event *A1.VIO* will be triggered. This event *A1.VIO* is associated with *A1.QO = FALSE*. It does not trigger the event *S1.REQ* of the behavioral FB (*S1*) to start the operation of *S1*. Instead, it triggers the event *S1.STP*, for example, to switch *S1* to deactivation mode for safety reasons. At the same time, the event *A1.VIO* will trigger the event *G1.REQ* associated with *G1.QI = FALSE* to block the execution of the guarantee FB (*G1*). The *G1* then emits the *G1.VIO* event associated with *G1.QO = FALSE*, to notify the violation to the following contract FBs.

- For ensuring timing constraint requirement: A *TCFB* is used to check the timely occurrence of input and output events of an object. In this example, it uses to ensure the timing constraints for the execution of a behavioral FB (*S1*) comprised of the *TSBC*s (*A1*, *G1*). The timing constraint syntax for input and output is formally defined at the port *T1.Spec*. When the event *A1.REQ* occurs corresponding to the occurrence of the input data at *A1*. *T1* is also triggered concurrently with the input event *T1.REQ*. It will first check if the input data arises at the desired time interval. If the event *T1.REQ* arrives outside the defined time interval, then the operation of the *S1* and the following components will still be stopped even though the input value (*val1*) satisfies the requirement of *A1*. Otherwise, it transits to the waiting state for the appearance of the event *T1.RSP* that corresponds to the happening

[2] For an example of *Guarantee*, the requirement for the component's output is that the output value at *S1.Out* is not allowed to exceed a range of values $[g_1, g_2]$

of the output data *G1.CFN* at the *G1*. *T1* next checks the duration time between the occurrence of input events *T1.REQ* and *T1.RSP*. If they meet the timing relation requirement, *T1* emits the output event *T1.CFN* associated with *T1.QO* = *TRUE* for enabling the execution of sequence components. Otherwise, the operation of *S1* and the next components will be blocked by triggering event *T1.VIO* associated with *T1.QO* = *FALSE* of *T1*.

- These contract FBs are controlled through the logic value of the data ports *QI* and *QO*. In particular, if the *REQ* event associated with *QI* = *TRUE* happens, the FB is activated and ready to operate in the successive procedures. Otherwise, the FB is not allowed for checking activities and will immediately fire the *VIO* event associated with *QO* = *FALSE* for intercepting the operation of the following components. In this way, these subsequent behavioral components can switch to a safe working mode or turn off.

Fig. 4.4 exhibits how contracts would be applied in a development framework using the IEC 61499 standard. It can also provide the answer to the question of why using contracts is essential for developing and monitoring safety-critical systems.

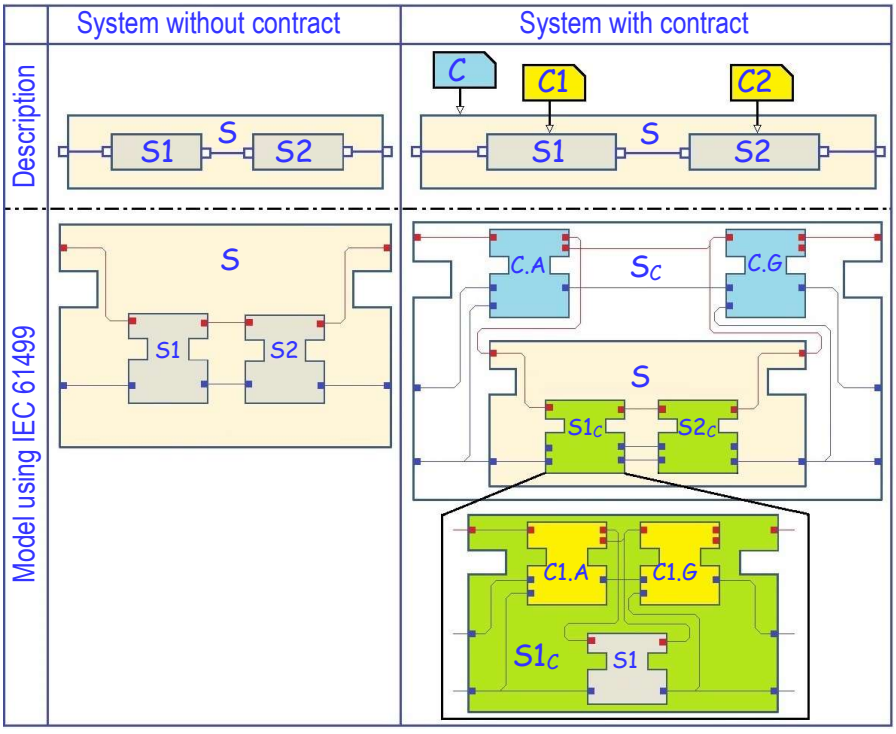


Figure 4.4: Overview of the proposed integration of contract-based in IEC 61499 FBs

Let's consider an example of a system S that consists of two subsystems of $S1$ and $S2$. If this system does not use contracts, it cannot check the planned behavior of S, $S1$, and $S2$ because there is no reference to a consistent and formalized specification. On the opposite is the same system using contracts. In this regard, the requirement for the desired properties/behaviors of the system S is formally specified through the top-level contract C. Furthermore, the lower-level

contracts $C1$ and $C2$ are used to characterize the requirements for the behaviors of subsystems/components ($S1$ and $S2$).

In the context of the modeling system control using the IEC 61499, the models of components and the overall system are described using various FB types such as *BFB*, *CFB*, and *SIFB*. Each FB refers to the component model (imply behavior FB), as illustrated in the lower-left corner of Fig. 4.4. Within the use of contracts, a behavior FB (for example, the "original" $S1$) will be encapsulated by contract FBs (for example, a $TSBC \rightarrow C1.A$ and another $TSBC \rightarrow C1.G$) and encapsulated in a *CFB* type characterizing a component with contracts (denoted $S1_C$). Furthermore, the FBs representing the component behaviors, can be interconnected in a network, and then be encapsulated in a *CFB* to characterize a subsystem. Contract FBs are then added to this *CFB* for modeling the use of contracts at the subsystem level. This principle can be repeated for integrating the contract FBs into other behavior FBs, higher-level systems, and the top level of the system. As a result, contracts can be employed in the horizontal and vertical integrations, and from component to system level of a hierarchical system, as described in the lower right part of Fig. 4.4.

Fig. 4.5 illustrates how to create an FB representing the model of components and contracts in the *4diac* framework and how to integrate and implement those FBs in *FORTE* (the *4diac* runtime environment).

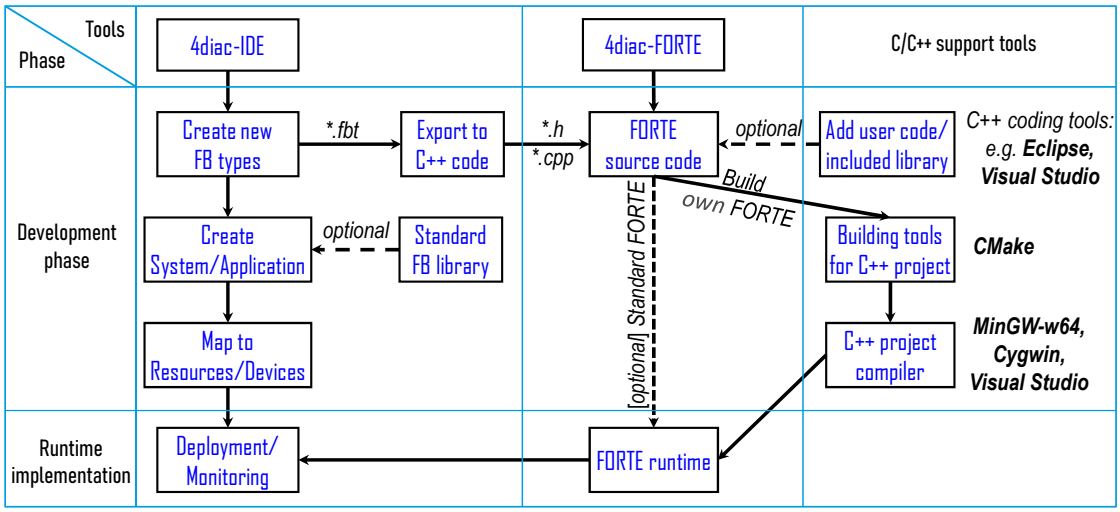


Figure 4.5: General progress of development and implementation using 4diac and FORTE

In *4diac*, a standard library consists of the existing built-in FBs. These FBs model several commonly used events, functions, and properties, such as delay (E_DELAY), cycle (E_CYCLE), enable (E_PERMIT), data conversion ($F_STRING_TO_INT$), etc. In addition, a standard *FORTE* is an executable file also provided to implement these existing FBs. However, the development of an application usually requires other FBs beyond the standard library. For this reason, creating new FBs to describe different applications is a mandatory task in modeling systems with *4diac*.

There are two methods to create a new FB. The first method concerns picking

up the existing FBs in the standard library, linking them together in a network to model the planned behavior, and then encapsulating them in a *CFB* type. However, this method is not always possible because of the limitation of FBs in the standard library. The remaining technique involves developing a new FB of type *BFB*. In this regard, the desired behavior of the component is generated using the graphical state transition called the execution control chart (*ECC*). In particular, the algorithms of *BFB* in *4diac* are composed in either ***C++*** language or Structured Text (***ST***). The newly created FBs will then translate to the ***C++*** project of the FORTE source code thanks to the export function of *4diac-IDE*. This ***C++*** project is compiled to produce an executable file of *FORTE* for implementing the IEC 61499-compliant application in *4diac*. Note that any newly created FB requires an export and then re-compile efforts to reach a new expanded *FORTE* file. If not, this new FB cannot be identified during the implementation of the application in the runtime environment. Explaining the development, integration, and compiler process using *4diac* and all the necessary tools is shown in Fig. 4.5.

4.2 Building IEC 61499 contract function blocks

As mentioned in the previous sections, one of the objectives of this thesis is to integrate the contracts into the IEC 61499 compliant framework. For this purpose, two kinds of contract FBs are developed and introduced to specify the requirements regarding the timed behavior and timing constraint aspects.

4.2.1 Time-Sensitive Behavioral Contract (*TSBC*)

The purpose of building this kind of contract FB is to specify and integrate timed behavioral restrictions for components in an IEC 61499-compliant application. Fig.

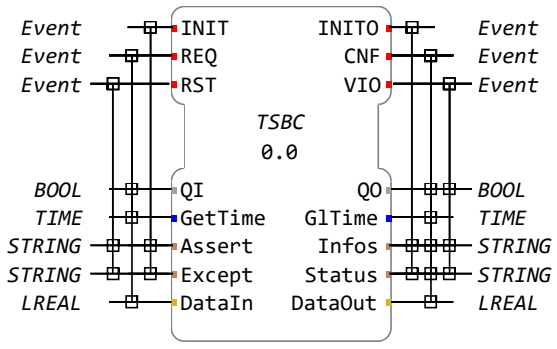


Figure 4.6: Interface of the time-sensitive behavioral contract function block

4.6 indicates the interface as well as the details of the data types of the input/output data ports of this function block type, which can be formally represented by the tuple as follows:

$$IFB_{TSBC} = (\mathbf{EI}, \mathbf{EO}, \mathbf{DI}, \mathbf{DO}, \mathbf{AI}, \mathbf{AO})$$

where:

- $\mathbf{EI} = \{INIT, REQ, RST\}$: set of input event variables.
- $\mathbf{EO} = \{INITO, CNF, VIO\}$: set of out event variables.
- $\mathbf{DI} = \{QI, GetTime, Assert, Except, DataIn\}$: set of input data variables.
- $\mathbf{DO} = \{QO, GlTime, Infos, Status, DataOut\}$: set of out data variables.
- $\mathbf{AI} = \{INIT \mapsto \mathbf{WI}_1, REQ \mapsto \mathbf{WI}_2, RST \mapsto \mathbf{WI}_3\}$: set of associations between input events and input data. Where:
  - $\mathbf{WI}_1 \equiv \mathbf{WI}_3 = \{Assert,\ Except\}$
  - $\mathbf{WI}_2 = \{QI, GetTime, DataIn\}$
- $\mathbf{AO} = \{INITO \mapsto \mathbf{WO}_1, CFN \mapsto \mathbf{WO}_2, VIO \mapsto \mathbf{WO}_3\}$: set of associations between output events and output data. Where:
  - $\mathbf{WO}_1 = \{Infos, Status\}$
  - $\mathbf{WO}_2 \equiv \mathbf{DO} = \{QO, GlTime, Infos, Status, DataOut\}$
  - $\mathbf{WO}_3 = \{QO, Infos, Status\}$.

The functions and tasks of these events and data ports in the *TSBC* are described in detail below: The initialization event *INIT* triggers to activate getting the initial values of internal variables, the contract specification (defined at port *Assert*) and the exception handling condition (if exists at port *Except*). The contract specification and exception (if any) are then decoded and meaningfully checked through the algorithm enclosed in the initialization state. If the parsing process is successful without errors (the specifications are reasonably well defined), then the event *INITO* will fire to indicate the ready state of *TSBC*. Otherwise, the *INITO* event will not fire, which means the specifications have errors. There are many reasons for the error, such as the set syntax being incorrect, the set values being illogical (e.g. $min > max$), etc.

Note that after the initialization process is successful (the output event *INITO* emitted), if a trigger on the *INIT* event arises anywhere during the implementation time striving to change the specifications (for example, the contract, the exception condition, etc.), that trigger will be ineffective. For this reason, the event *RST* was presented to serve the resetting purpose of the *TSBC*. Once the *RST* is triggered, it drives the *TSBC*'s execution back to the initialization state. At this state (initialization), the *TBSC* can get and set the new specifications. Besides, it resets the initial values of the counter, timer, internal variables, etc. In particular, when the timed contracts are specified to observe the switch behavior of components, the *RST* event gives powerful support. Let's take an example of requirement specification for the timed behavior of a motor used to control a robot joint. The motor state changes between start and stop movements during the robot's performance. Therefore, the expression of the contract is as follows: Whenever a motor startup, the motor

current must be less than I_0(A) for $t < t_0$ (*). Beyond that period ($t \geq t_0$), the motor current must be in the range of $[I_{min},\ I_{max}]$(A) (**). Imagine that when the time values are not reset, then *TSBC* will check condition (*) only once, and then it checks requirement (**) during the remaining operation because of $t > t_0$. It is necessary to trigger the *RST* event simultaneously with the motor starting to reset all the timer values to solve this problem. Another benefit of using the *RST* event is that it allows *TSBC* to change contract specifications for different time intervals. For example, the requirements for a behavior FB (*S1*) are expressed via a list of contracts (*C11*, *C12* ...), and they will be applied at various time points during its implementation. This idea can be a reality by connecting each feeding contract event to the *RST* event of *TSBC*. In this way, we can apply the specification contracts to a wider scope of applications and more complex use cases compared to using standard syntax.

The data input port *GetTime* is used to set the start time value of *TSBC*. In particular, if the *GetTime* is set to the value of 0 (default value), then the start time value will be determined by its local time (For example, *StartTime* = *NOW_MONOTONIC*();). If the *GetTime* gets a non-zero time value (for example, *getTime*) from another FB, that value will be set as the start time value of the TSBC (*StartTime* = *getTime*;). That means that the start timestamp of this *TSBC* depends on the timestamp of another FB. A data output port named *GlTime* is presented to support the dispatch of this TSBC start timestamp value to other *TSBC*s through a setting of *GlTime* = *StartTime*;. The ***C++*** code snippet shown below gives an example for the mentioned.

```
...
if(GetTime == 0){
    //Get current time
    StartTime = std::chrono::system_clock::now();
    //Equivalent to NOW_MONOTONIC() in ST
}
else{
    //Get StartTime value at the GetTime port
    StartTime = GetTime;
}
//Pass StartTime to GlTime port
GlTime = StartTime;
...
```

Let's consider the following example to understand the purpose and meaning of the *GetTime* and *GlTime* ports of *TSBC*. Assume that *TSBC1* and *TSBC2* are connected, where the $TSBC1.GlTime$ links to $TSBC2.GetTime$. Besides, $StartTime1$ and $StartTime2$ are the starting time variables of *TSBC1* and *TSBC2* respectively. Let's set $TSBC1.GetTime$ as the value of 0, then $StartTime1$ will get the value of *NOW_MONOTONIC*() in ***ST*** code. That leads to $TSBC1.GlTime = StartTime1$. As a result, $TSBC2.GlTime = TSBC1.GlTime = StartTime1$. That implies $StartTime2 = StartTime1$ and $TSBC1.GlTime = StartTime1$. It

is a way of passing global time in a network of *TSBC*s.

Furthermore, the characteristics and functionalities of the event ports *REQ*, *CNF*, and *VIO*, and the data ports *QI* and *QO* are similar to what had been described in the section 4.1. In addition, the data ports *Infos* and *Status* are used to indicate the information of the execution status, such as checking status (satisfaction, violation), the execution time, input and output data value, etc. The syntax of the contract specification (represented at the *Assert*) and the exception condition (can be set at the *Except*) will be more details introduced in the following sections.

4.2.2 Timing Contract Function Block(*TCFB*)

The Timing Contract Function Block (*TCFB*) is another type of contract FB developed and introduced in this thesis. It supports the definition of timing and causality specifications for a component's operation or process. Using *TCFB*, the specification of timing constraint will be integrated into an IEC 61499-compliant application.

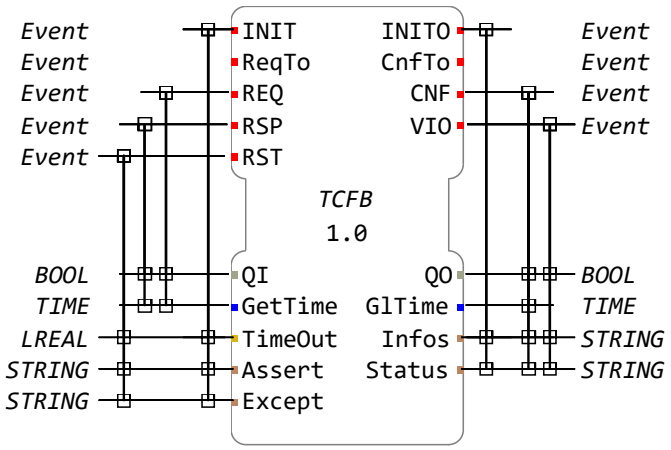


Figure 4.7: Interface of Timing-Contract function block

Fig. 4.7 depicts the interface and the details of the data types of the input/output data ports of this function block type, which can be formally represented by the tuple as follows:

$$IFB_{TCFB} = (\mathbf{EI}, \mathbf{EO}, \mathbf{DI}, \mathbf{DO}, \mathbf{AI}, \mathbf{AO})$$

where:

- $\mathbf{EI} = \{INIT, ReqTo, REQ, RSP, RST\}$: set of input event variables.
- $\mathbf{EO} = \{INITO, CnfTo, CNF, VIO\}$: set of out event variables.
- $\mathbf{DI} = \{QI, GetTime, SetTimeOut, Assert, Except\}$: set of input data variables.
- $\mathbf{DO} = \{QO, GlTime, Infos, Status\}$: set of out data variables.

- $\mathbf{AI} = \{INIT \mapsto \mathbf{WI}_1, REQ \mapsto \mathbf{WI}_2, RSP \mapsto \mathbf{WI}_3, RST \mapsto \mathbf{WI}_4\}$: set of associations between input events and input data. Where:
  - $\mathbf{WI}_1 \equiv \mathbf{WI}_4 = \{SetTimeOut, Assert, Except\}$
  - $\mathbf{WI}_2 \equiv \mathbf{WI}_3 = \{QI, GetTime\}$
- $\mathbf{AO} = \{INITO \mapsto \mathbf{WO}_1, CFN \mapsto \mathbf{WO}_2, VIO \mapsto \mathbf{WO}_3\}$: set of associations between output events and output data. Where:
  - $\mathbf{WO}_1 = \{Infos, Status\}$
  - $\mathbf{WO}_2 \equiv \mathbf{DO} = \{QO, GlTime, Infos, Status\}$
  - $\mathbf{WO}_3 = \{QO, Infos, Status\}$.

Some event and data ports in *TCFB* have the same features and functions as the event and data ports, which have the same name in *TSBC*. They include the events of *INIT*, *INITO* and *RST*, and the data of *QI*, *QO*, *Assert*, *Except*, *GetTime*, *GlTime*, *Status* and *Infos*. The addition of the data port *TimeOut* is the first difference with *TSBC*. This port is associated with the *INIT* event to get the setting value for the *TCFB* timeout. The timeout value defines a specified period during which the TCFB waits for an expected event (*REQ* or *RSP*). When the *REQ* or *RSP* event has not arrived within that given time frame, TCFB will emit a *VIO* event to notify the violation of the timing constraints of that event. For serving this timeout counter operation, *TCFB* is equipped with two event ports *ReqTo* and *CfnTo* to generate a checking cycle for the timeout counter. These two event ports are normally connected to a delay FB (E_DELAY) to check the timeout of TCFB. As mentioned, the purpose of the *TCFB* is to specify the timing constraints for the occurrence of events/signals so that data input/output ports do not need to be present in this FB. Instead, the *RSP* event port is added to support monitoring the timed causality chains (it implies a causal relation between events *REQ* and *RSP*) of a component or process in the IEC 61499-compliant application.

The roles and functions of the event ports such as *REQ*, *RSP*, *CNF*, *VIO* and the data ports *QI* and *QO* operate similarly to what has been explained in the section 4.1. The details of the syntax and semantics of these specifications will be fully presented in the next section.

4.3 The specification language

4.3.1 Specification contract for *TSBC*

This section provides the concept of expressing the timed behavioral reasoning for components in an IEC 61499-compliant application using contracts. The timed behavioral contract specification refers to requirement expressions, in which the requirement expression can change depending on time during the implementation process. The following example illustrates and explains the concepts for accomplishing the above objective.

The behaviors of a component require that the input and output data must ensure the following characterizing: For the operating time before t_0 $(t < t_0)$*, the values of the input (denoted by* x*) must be in the interval of* $[vi_1, vi_2]$*. Afterward, for* $(t \geq t_0)$*, the value of that input must be within the interval of* $[vi_3, vi_4]$*. When that input meets those requirements, then the output of the component (denoted by* y*) should ensure the following properties: The output values must be in the range of* $[vo_1, vo_2]$ *for* $(t < t_1)$*, and the component's output values must be within the interval of* $[vo_3, vo_4]$ *if* $t \geq t_1$. In the context of using an assume/guarantee contract, the specification for this behavior can be expressed under the formation of the mathematical expressions as follows:

- **Assumption:** $\begin{cases} x(t) \in [vi_1, vi_2] \; if \; t < t_0 \\ x(t) \in [vi_3, vi_4] \; if \; t \geq t_0 \end{cases}$
- **Guarantee:** $\begin{cases} y(t) \in [vo_1, vo_2] \; if \; t < t_1 \\ y(t) \in [vo_3, vo_4] \; if \; t \geq t_1 \end{cases}$

To specify these requirements in IEC 61499-compliant applications, it is necessary to have a suitable syntax for integrating this specification into *TSBC*. The following section will provide more details on the syntax and semantics of the specification for the timed behavior contract.

4.3.1.1 Syntax and semantics

The syntax of the *Specification* of *TSBC* is characterized through the following expression:

$t\,\langle CTyp\rangle\,\langle V||I\rangle\,[\langle u\rangle]?d\,\langle CTyp\rangle\,\langle V||I\rangle\,:\,d\,\langle CTyp\rangle\,\langle V||I\rangle$

Where:

- t is the keyword for the time variable; d is the data variable;
- $\langle CTyp\rangle$ represents the contract type or comparison type that includes a set of characters "=", "≠", ">", ">=", "<", "<=", "| =", "|#". Due to the limitation of using Unicode characters in *4diac*[3], therefore some Unicode characters (∈, ∉, ≥, ≤) can not be used directly. Instead, some alternative formats are applied to represent the Unicode characters. For example, the characters "| =" stands for [∈], ">=" stands for [≥], "<=" stands for [≤], and "|#" stands for [∉];
- $\langle V||I\rangle$ references the text area that defines the comparison values. Depending on the comparison type or mathematical operator, $\langle CTyp\rangle$ can be specified by either a value of V or an interval of values I. If $\langle CTyp\rangle$ is $\neq$ or $=$, then this place will be V, which stands for a specific value (va). Otherwise, it will exist two values ($\{sn, gn\}$ or $\{-\infty, va\}$ or $\{va, +\infty\}$), where sn represents a value of smaller number and gn refers to a value of greater number.

[3]At least until *4diac-IDE* version 1.19.0

- For the expression of the time, the unit of time ($\langle u \langle$) has to be displayed next to the finite constrained values. The following units of time are available for the specifications "ns", "us", "ms", and "s" standing for "*nanosecond*", "*microsecond*", "*millisecond*", and "*second*" respectively.

Note that, in the suggested contract syntax of *TSBC*, the mathematical expression for comparing the data for two different values, for example, $sn \leq d \leq gn$, must be described using the syntax $d| = [sn, gn]$. On the other hand, the mathematical expression for comparing data for a finite value can be described either via the straightforward form or under the interval format. For example, the requirement for the data d is described through the mathematical term as $d \leq val$. This can be specified by the contract syntax as either $d <= val$ or $d| = (-inf, val]$ [4].

For expressing data that does not depend on a time variable, one of the following two syntaxes $t \in \mathbb{R}$ or $t \in (-\infty, +\infty)$ can be used to specify in the timing condition expression. They refer to the time variable is always fulfilled (*for all times* - $\forall t$). If the data d is expressed by $d \in \mathbb{R}$, it implies that data d is any real value, and the result of this comparison expression is always satisfied. The following examples will describe the semantics and syntax of the proposed contract specification for *TSBC*.

4.3.1.2 Examples

The semantic of the contract specification is characterized as "*d must always be greater than or equal to* 0.5 *and less than* 2.5". That is represented under the mathematical expression as $0.5 \leq d < 2.5; \forall t$. The syntax for the specification in *TSBC* is written by:

$t| \ = \ R \ ? \ d| \ = \ [0.5, 2.5]$

The semantic of the contract specification is: "*If the operation time t is less than or equal to* 4 *seconds, then data d must be always less than* 2.5. *Otherwise data d is presented by any real value*". That is characterized under the mathematical expression as $d < 2.5; \forall t \leq 4$ and $d \in \mathbb{R}; \forall t > 4$. The syntax for the specification in *TSBC* will be written by:

$t \ <= \ 4s \ ? \ d \ < \ 2.5 \ \ : \ \ d| \ = \ R$

The semantic of the contract specification is: "*If the operation time t is between* 1.5 *to* 3.5 *milliseconds, then data d must always be greater than or equal to* 5 *and less than* 25, *otherwise data d must be smaller than* 55". It is presented by the mathematical expression $5 \leq d < 25; 1.5ms < t \leq 3.5ms$ and $d < 55; (t \leq 1.5) \cup (t > 3.5)$. The syntax for the specification in *TSBC* will be written by:

$t|\#\,(1.5, 3.5]\,ms \ ? \ d| \ = \ (-\infty, 55) \ \ : \ \ d| \ = \ [5, 25)$

4.3.1.3 BNF

The following grammar summarizes the specification patterns for time-sensitive behavioral contracts.

$\langle BehaviorSpec \rangle$ ::= $\langle TimCond \rangle \langle LogicalComparison \rangle \langle IfTrue \rangle \mid \langle TimCond \rangle \langle LogicalComparison \rangle \langle IfTrue \rangle \langle Otherwise \rangle \langle IfFalse \rangle$

[4] it stands for $d \in (-\infty, val]$

⟨*LogicalComparison*⟩ ::= '?'

⟨*TimCond*⟩ ::= 't|=R'|⟨*Condition*⟩⟨*TimeUnit*⟩

⟨*Condition*⟩ ::= ⟨*Variable*⟩ ⟨*Operator*⟩ ⟨*ValueField*⟩

⟨*Variable*⟩ ::= 't' | 'd'

⟨*Operator*⟩ ::= '=' | '<' | '<=' | '>' | '>=' | '#' | '|=' | '|#'

⟨*ValueField*⟩ ::= ⟨*Value*⟩ | ⟨*Interval*⟩

⟨*Value*⟩ ::= ⟨*Number*⟩ | ⟨*Number*⟩ '.' ⟨*Number*⟩ | ⟨*Infinity*⟩

⟨*Number*⟩ ::= 0 ... 9

⟨*Infinity*⟩ ::= '-inf' | '+inf' | '..'

⟨*Interval*⟩ ::= ⟨*Boundary*⟩ ⟨*Value*⟩ ',' ⟨*Value*⟩ ⟨*Boundary*⟩ | 'R'

⟨*Boundary*⟩ ::= '['|']'|'(' |')'

⟨*TimeUnit*⟩ ::= 'ns' | 'us' | 'ms' | 's'

⟨*IfTrue*⟩ ::= ⟨*DataCondition*⟩

⟨*DataCondition*⟩ ::= 'd|=R'| ⟨*Condition*⟩

⟨*Otherwise*⟩ ::= ':'

⟨*IfFalse*⟩ ::= ⟨*DataCondition*⟩

4.3.2 Specification contract for *TCFB*

In this work, the specification of the timing contract is inspired by the contract specification language in the MULTIC [61]. The syntaxes here have some modifications compared to the MULTIC to match the context of applying to the IEC 61499 FBs. Specifically, the keywords for a causal relation of timing specifications, such as "*Reaction*" and "*Age*" will be borrowed from the MULTIC. Further, another keyword called "*Occurrence*" was added to specify the requirement for arriving event of a component.

4.3.2.1 Syntax and semantics

The syntax of the *Specification* of *TCFB* is characterized through the following expression:

$$\langle CausalRelation \rangle :: \lfloor q\,[(\langle tc \rangle)] - > \rceil\, e\,(\langle tc \rangle)$$

- ⟨*CausalRelation*⟩:*Occurrence* or *Reaction* or *Age*.

- The text field of $\lfloor q\,[(\langle tc\rangle)] - >\rceil$ describes the assumption text field. This text field is only required when defining the timing contract specification of the causal relation *Reaction* or *Age*. For expressing the *Occurrence* causal relations, the contract specification will be understood that its assumption will always be *TRUE*. Therefore, the display of the assumption text field is not necessary.
- $q\,(\langle tc\rangle)$ and $e\,(\langle tc\rangle)$ are used to define timing constraint requirements for the causal relation between the request event (q: represents the event of the assumption) and response event (e: represents the event of the guarantee). Within that, the execution time of a component or process can be bound within a specified timing interval.
- The text field of timing constraint $\langle tc\rangle$ is characterized through the keyword i and the bounded values $MinT$ and $MaxT$. In particular, it is formally defined as $i| = [MinT, MaxT]\,\langle tu\langle$. Similar to the contract specification of *TCFB*, the characters "$| =$ " will be used to replace the symbol $[\in]$, and the time units are also defined similarly in the previous section.
- [...]: indicates an optional field definition.

4.3.2.2 Examples

Occurrence constraint The following example illustrates the use of the keyword "*Occurrence*" for timing restriction of the appearance of a single event on a determined port.

The semantic of the event occurrence (e) *"**e** occurs every [1ms,3ms]"* [5]. It is specified through the following syntax:

$$Occurrence :: e(i| = [1, 3]ms)$$

Fig. 4.8 shows an example for this use case.

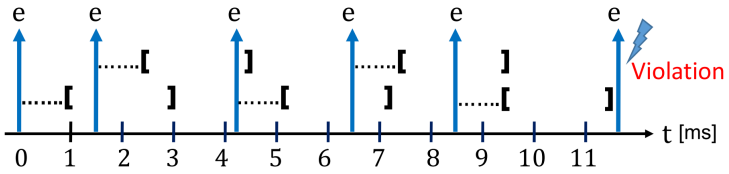


Figure 4.8: Illustration example for the specification of event occurrence

Reaction constraint The reaction constraint would be used with the keyword "*Reaction*" to specify a timing response restriction concerning the occurrence of two different events (request q and response e). In particular, once a causing event q occurs, the following event e(or resulting event) must occur in a specified interval afterward. The semantic of the causal relation of the following event e which is

[5]Please note that there is no display of assumption. So e must occur regardless of its environment.

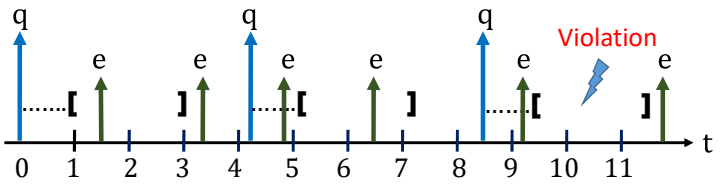


Figure 4.9: Illustration example for *Reaction* constraint

affected by a cause event q as: "*every time* ***q*** *occurs (**A**),* ***e*** *must occur within the interval [1s,3s] (**G**)*". It is described by the syntax as follows:

$$Reaction :: q- > e(i| = [1,3]s)$$

The semantic for a reaction relation is as follows: "q *occurs every [2s,3s] (**A**); when whenever* q *occurs,* e *must be occurred within the interval [1ms,3ms] (**G**)*". It can be represented by the syntax as follows:

$$Reaction :: q(i| = [2,3]s)- > e(i| = [1,3]ms)$$

Fig. 4.9 depicts the examples of how the *Reaction* relation works.

Note: For the definition of a standard for satisfaction of the *Reaction* relation, there are the following possibilities:

- [**Option 1**]: A causing event q always generates exactly one following event e. This following event must be within the time interval that is defined in the specification.
- [**Option 2**]: A causing event q may generate more than one following event e. However, it is a contract satisfaction if only one of these following events occurs within the given time interval (defined in the specification). In this case, any following events that occur before or after this following event will not result in a violation of the contract.
- [**Option 3**]: A causing event q may generate more than one following event e, but there must be at least one following event within the time interval defined by the specification. This means that other following events (associated with the causing event q) occurring at any time will not affect this satisfaction if there is one following event that satisfies the contract.
- [**Option 4**]: A causing event q can generate many following events e, all of which must be within the specified time interval defined by the specification.

We can imagine that a physical component typically causes only one response event to a triggered input event. Nevertheless, we cannot rule out the possibility that it may cause several response events which could be influenced by a disturbance. Among these aforementioned options, **Option 1** and **Option 4** would seem too rigorous, whereas **Option 3** seems too flexible and may create a loophole for violations. Therefore, **Option 2** is selected to specify the standard satisfaction

for the timing contract according to the *Reaction* constraint in this work. This option can be combined with exception specifications to provide more flexibility in considering the satisfaction of events. The details of the exception handling will be introduced in section 4.4.

Age constraint : The specification for this causal relation begins with the keyword "*Age*". It expresses the causal relationship in a reverse-directional manner. In other words, it is necessary to check that when the following event (e) occurs, whether there had already occurred any causing event (q) in the given time interval before or not.

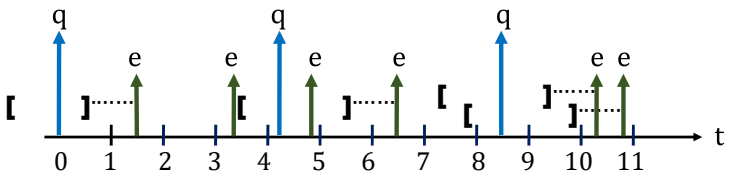


Figure 4.10: Illustration example for *Age* constraint

For example, the semantic of the timing contract specification as: "*every time* ***e*** *occurs (**A**),* q *has occurred within the interval [1s, 3s] (**G**)*". It will be characterized by the syntax as follows:

$$Age :: q- > e(i| = [1,3]s)$$

Fig. 4.10 illustrates example for the working principle of the *Age* constraint. The standard of satisfaction of the *Age* relation relation depends on the selected standard of satisfaction of the *Reaction* relation. In the example shown in Fig. 4.10, the last following event will cause satisfaction if **Option 3** or **Option 4** is applied. This event will generate a violation if **Option 1** or **Option 2** is involved (two following events meet the time constraint associated with one causing event).

The use of *Reaction* and *Age* can give the same result in most cases. However, there are also some application cases where the timing contract specification with *Age* cannot replaced by a *Reaction* expressing. The following example will illustrate this issue. A robot arm moves and operates at three different positions named A, B, and C. The arrival/departure of the robot at these points can be recognized by on/off signals of sensors placed at points A, B, and C. At the beginning of a working cycle, the robot arrives and stops at A, picks up a workpiece, then moves to B detected by the sensor signal S_B. At point B, it waits for the processing of the workpiece within the time interval $[t_1, t_2]$. The finishing of this process will be notified by the sensor signal S_T; once S_T occurs, the robot picks up that finished workpiece and moves it to C. It puts the workpiece at point C and then returns to A (via B and without a stop at B) to perform a new working cycle. Fig. 4.11 represents the clarification for choosing contract specification for this example case. We can see in the upper part of Fig. 4.11 that the signal S_B will be characterized under two different labels $S_B^{'}$ and $S_B^{''}$. The $S_B^{'}$ label implies the signal S_B the robot receives when it moves in the direction from A to C and will stop at point B. The

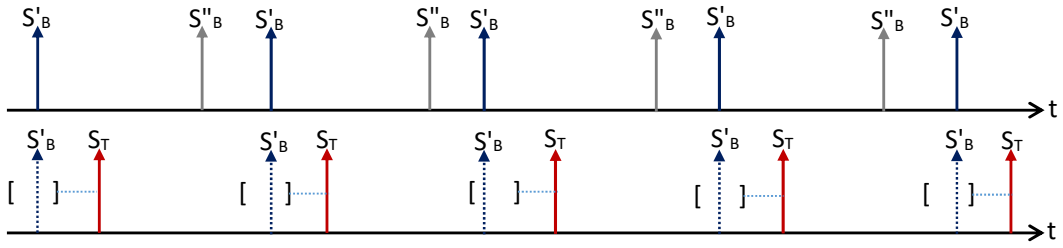


Figure 4.11: Specification for the example case of *Age* constraint

S_B'' label refers to the signal S_B the robot receives when it moves in the direction from C back to A. In this moving case, it passes through B but does not stop at B.

Let's assume that we define the timing contract with the *Reaction* specification for waiting for the finish of the workpiece processing as follows: *Whenever* S_B *signal occurs,* S_T *signal occurs within the time interval* $[t_1, t_2]$. With this specification, we can imagine that the contract will only be satisfied when the robot moves from point A to point B. For the robot movement from point C back to point A, the robot does not need to stop at point B (even though the sensor signal at point B still raises). As a result, the contract will notify a violation since the signal S_B'' occurs, the S_T signal does not occur within the time interval $[t_1, t_2]$ afterward.

On the other hand, if we express the timing constraint specification in this case using the *Age* relation as *Whenever* S_T *occurs,* S_B *has occurred within the time interval* $[t_1, t_2]$ *before*. By using this specification, the mentioned problem can be solved correctly.

Notes : Applying $TCFB$ for timing constraint monitoring is based on the three relations/constraint concepts of *Occurrence*, *Reaction*, and *Age*. The following notes should be considered when using these constraint concepts:

1. Theoretically, it is possible to have contracts that constrain an event to occur at a specific time (e.g., *every 1s*). However, due to the performance of the employed hardware devices, occasionally this point may never be reached. For example, it may happen that $T_{k-1} = 0.999s$ -> $T_k = 1.0001s$. Therefore, the time constraints here recommend the use of an interval definition. However, if needed, we can also use a definition of $t| = [T, T]$ to represent the definition of $t = T$.

2. As we can see, applying a time contract can be done by a single $TCFB$, where the expression of *Assumption* (A) and *Guarantee* (G) is composed in a string of *Specification* written to the *Assert* port. It is different from applying time-sensitive contracts, where observing *Assumption* (A) and *Guarantee* (G) must be done by two different $TSCB$s. In fact, for applying a *Reaction* relation, we can use two different $TCFB$s to express the timing contract. In that case, we must use the *Occurrence* (instead of *Reaction*) to express the contract specification. For this intent, a $TCFB$ is used to express the requirement for the occurrence of the causing event (*Assumption*), and another $TCFB$ is used to define the occurrence of the following event after the causing

event has occurred and satisfied its constraint time (*Guarantee*). However, this alternative option is not favored to apply because of the following reasons:

- Using two $TCFB$s increases the number of interconnecting between components in a system. Besides, it will increase time overhead for its upper-level component/subsystem. Therefore, it is a weakness to the purpose of strictly controlling the timing aspect of the occurrence of events.
- If we have to use a $TCFB$ to express the assumption: *"Whenever the causing event (q) occurs..."*. That means that we must use a $TCFB$ without checking any time constraints. In this case, using $TCFB$ really does not make much practical sense.
- Determining the occurrence time of the following event e and the time out for the $TCFB$ as *Guarantee* must be based on the timestamp at the $TCFB$ as *Assumption*, which adds complexity to the handling process.
- In this thesis, exception conditions for $TCFB$ are often considered based on the number of satisfied following events corresponding to a causing event. Meanwhile, the *Occurrence* specification only monitors the occurrence within a specified time constraint of a single event (there is only an event connected to the REQ port and no event connected to the RSP port). Therefore, the exception conditions for $TCFB$ are just built and apply for the causal relation of *Reaction* and *Age* and do not apply to the *Occurrence* constraint. As a result, when we use this alternative option, we cannot apply the exception condition that we can easily apply when we use the *Reaction* specification.

3. The *Age* constraint can be considered a specification for retrieving the causal relation. However, if we use suitable connections and configurations, we can fully apply the *Reaction* constraint to monitor causal relationships without using any other specification (e.g., *Age* relation).

4.3.2.3 BNF

The following grammar summarizes the specification patterns for the *TCFB*.

⟨*TimeSpec*⟩ ::= ⟨*CausalRelation*⟩ '`::`' ⟨*TimContract*⟩

⟨*CausalRelation*⟩ ::= '`Occurrence`' |'`Reaction`'|'`Age`'

⟨*TimContract*⟩ ::= ⟨*Guarantee*⟩ | ⟨*Assumption*⟩ '`->`' ⟨*Guarantee*⟩

⟨*Guarantee*⟩ ::= '`e`' '`(`' ⟨*TimeConstraint*⟩ '`)`'

⟨*Assumption*⟩ ::= '`q`' | '`q`' '`(`' ⟨*TimeConstraint*⟩ '`)`'

⟨*TimeConstraint*⟩ ::= '`i|=`' '`[`' ⟨*Value*⟩'`,`' ⟨*Value*⟩ '`]`' ⟨*TimeUnit*⟩

⟨*Value*⟩ ::= ⟨*Number*⟩|⟨*Number*⟩ '`.`' ⟨*Number*⟩

⟨*Number*⟩ ::= 0 .. 9

⟨*TimeUnit*⟩ ::= 'ns' | 'us' | 'ms' | 's'

4.3.3 Specification support tool

As mentioned in section 2.2.3, semi-formal specification language can be represented by a pattern-based (template-based) or text-block-based. It represents one of the reasonable options to address the inherent weaknesses and disadvantages of natural and formal language. For this reason, it is selected to express the contract specifications in this thesis.

In this work, a graphic user interface (GUI) named PassCS (**Pa**ttern-ba**s**ed **s**upport of **C**ontract **S**pecification) has been developed to simplify for characterization and integration of contract specifications for IEC 61499 FBs. This tool is written in Java and aims to support the automatic translation of the contract expression from template-based specifications to the comparable syntax for contract FBs (*TSBC* and *TCFB*). Fig. 4.12 depicts a screenshot of the interface of this tool.

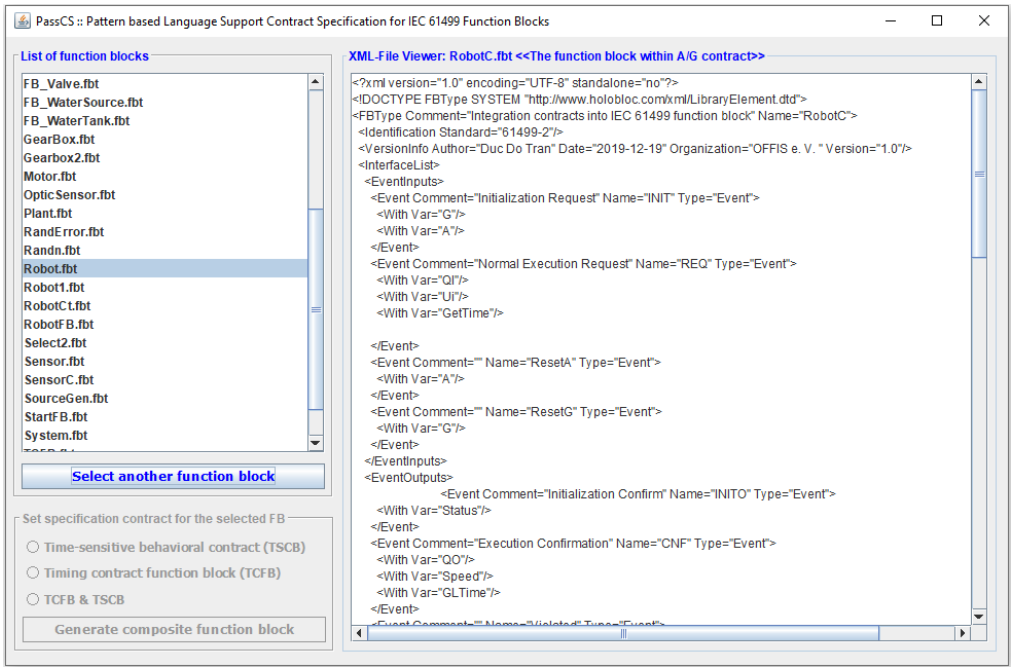


Figure 4.12: Pattern-based support of Contract Specification(PassCS) tool

The workflow and main functions of this tool will be presented below. The first thing to do is select a behavior FB in the FB list, which we want to express the requirement specification. When we complete this step, a view window will display details about that selected FB. The attribute information of this behavior FB represents under an XML format of type FB (**.fbt*).

After choosing the needed function block, the next step is to choose one of the three possible options, to add the specification of contract blocks to the selected function block in the first step.

In the next step, we need to choose one of three options to describe how to specify the contract. The three options are: *TSBC*, *TCFB* and *TSBC&TCFB*. The aim of this action is to integrate the contract FBs of *TSBC* or(and) *TCFB* along the appropriate syntaxes of contract specifications into the selected behavior FB. After that, a text-block-based window will appear to allow characterizing the contract specifications as shown in Fig. 4.13.

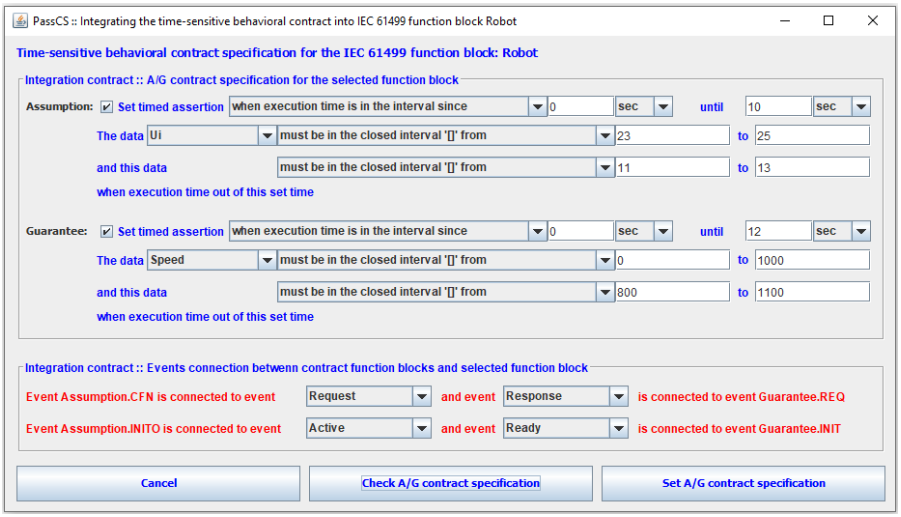


Figure 4.13: The dialog box for setting contract specifications based on patterns

From this window, we can select one input variable and one output variable of the behavior FB from the input and output data lists to set the assertion for *Assumption* and *Guarantee*. The contract specifications are defined by dropping down and selecting from the pattern text lists and inserting the appropriate values to characterize the comparable requirements. Furthermore, we can perform connecting among input/output events of the *Assumption*, the behavior FB, and the *Guarantee*. For example, the event *INITO* of the *Assumption* links to the event *INIT* (or *Active*) of the behavior FB, the *CNF* event of the *Assumption* links to the event *REQ* of the behavior FB. On the outputs side, the event *INITO* (or *Ready*) of the behavior FB can connect to the event *INIT* of *Guarantee*, and the event *CNF* of the behavior FB will link to the event *REQ* of the *Guarantee*. However, the event connections are just optional actions because we can perform or edit them after the generation of the CBF[6] is complete. The remaining connections (event ports and data ports) inside the FB network between the $TSBC$s (A and G) and the selected behavioral FB will be completed automatically by this software.

Once defining contract specifications is done, we can check these specified assertions again by clicking the "*Check A/G contract specification*" button. After clicking the button, another window will appear and exhibit the set contract specification in 3 different expressions to support the check. The expressions shown include natural language, mathematical expression, and specification syntax as illustrated in Fig. 4.14.

[6]It is a network of the behavior FB, *Assumption*, and *Guarantee*

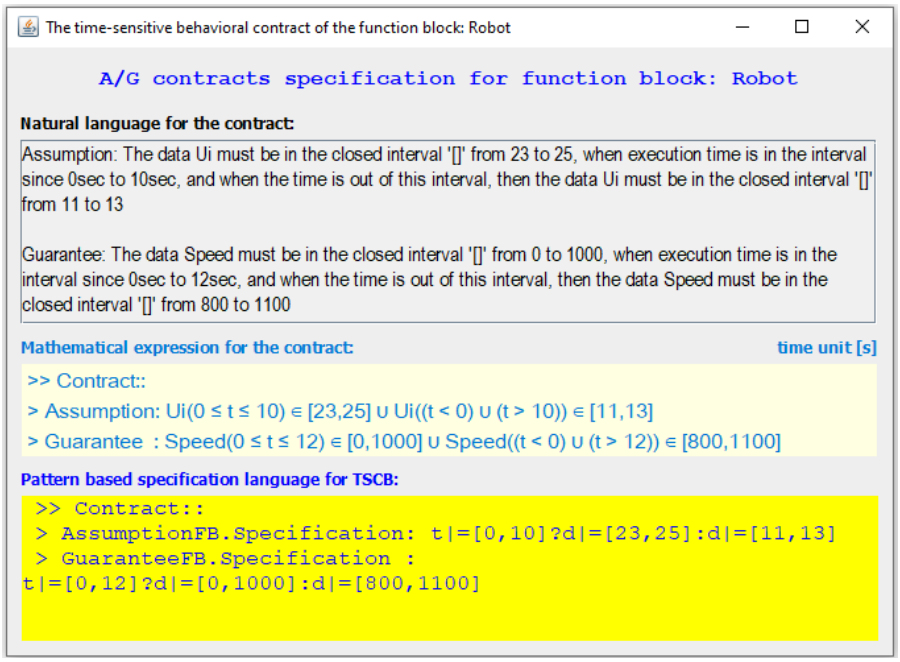


Figure 4.14: Assistance window for checking the contract specification

After everything has been checked carefully and the settings are correct, we need to click the "*Set A/G contract specification*" button to complete establishing the contract. The "*Generate composite function block*" button will change from a deactivated state to an activated state after the button "*Set A/G contract specification*" is clicked.

A CFB will be created in *4diac* after we click the "*Generate...*" button. This CFB presents the encapsulation of the behavior "*original*" FB interconnected with *Assumption* and *Guarantee* FBs. We can see the syntax of assertion in the *Initial Value* of *Properties* field[7] of that CFB in *4diac*.

Note that although the CFBs in *4diac* can be generated automatically by the tool, to use them, we have to export these FBs to C++ code for compiling a new *FORTE* executable. Without this, these FBs will not be recognized during the implementation of their application.

4.4 Exception handling

Applying for contracts to develop and monitor systems regarding safety criteria is essential and can have many advantages. However, the use of contracts can face many obstacles and unexpected problems/situations when they are applied in practical scenarios. In some cases, we want to apply the contracts flexibly, such as we can ignore a certain number of minor violations instead of catching all errors in the specification contract. Sometimes, we also do not want to apply for a contract for a fixed period. For example, let's imagine that a contract is applied to monitor

[7]The syntax of *Assumption* and *Guarantee* can be found in the text area of *Initial Value* of *Properties* field of data port named *A* and data port named *G* of the CFB respectively.

a data port of a component where a step function characterizes its property. That implies its data will be mainly given in two interval values, one before and one after the jump (denoted t_0). Pay attention to the object value at the time of the jump t_0. It is difficult (even impossible) to determine in which interval of values (before or after the jump) the object should be there. In addition, the precise determination of the time of the jump t_0 is also inconceivable. Furthermore, the suggested syntax supports the contract specification with the *if...else* statement. As a result, if we do not use an additional solution, we must use more than one contract FB to express the mentioned specification completely.

4.4.1 Motivation and problem definition

The purpose of the contract implementation is typically to control and monitor compliance regarding the requirement specifications compared to the actual operational behavior of a component. The contract gives an acceptance and the component implementation continues working when the component behavior satisfies the contract (normal behavior). It will prevent or break the component implementation if the component behavior violates the specification contract (abnormal behavior). However, an abnormal behavior does not always cause damage or negatively affects the component operation. Occasionally, it happens due to a careful restriction of the contract specifications. Therefore, we need a mechanism to review abnormal behaviors for using and implementing the contracts more flexibly and accurately. The following arguments will justify the need for an exception handling:

- Firstly, the use of exception handling can extend the scope of the implementation contract. In particular, some requirements cannot be fully specified and reflected by a fixed syntax of the contract specification. Instead, they need to add some extra descriptions for applying in the operational environment of a contract.

- A question that can be raised here is, *if we want to apply the contract in a broader scope of values, why we don't define it directly in the contract specification instead of using exception handling?*. Let's consider an example of a serial production line where a workpiece can be delivered slightly later (or earlier) than the expressed requirement of the contract. We can imagine that with a certain number of minor violations of timing constraints or the frequency appearance of violations being too small, the violations probably do not cause a critical disruption to the operation of the following components. Therefore, these violations should be ignored instead of prevented. When we define the contract in a broader range of values (All workpieces will have a wider span for timing constraint, and a workpiece can arrive earlier or later than its previous constraint range) and if subsequent components do not undergo comparable changes, the performance of the system can be affected. For example, if one workpiece arrives late and the following one arrives early, the robot may not have enough time to pick up and transport these workpieces. Therefore, it could lead to a jam on the conveyor.

Fig. 4.15 illustrates the implementation results of a contract without exception handling. We can easily recognize that the results have only two states: satisfaction and violation.

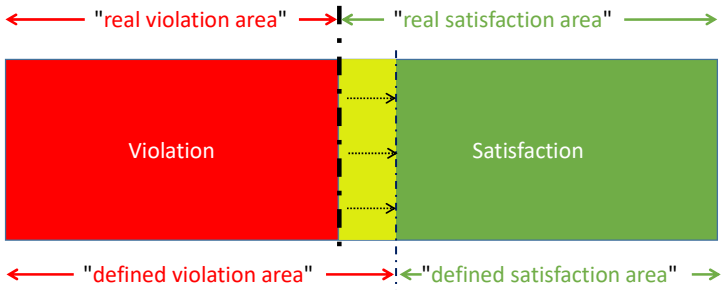


Figure 4.15: Description of the contract specification areas

The borderline between satisfaction and violation areas is identified depending on assigning limitation values from the contract specification. To accurately determine the "real" borderline (bold line) between satisfaction and violation areas is not easy because the component implementation results can be affected by unknown factors. Therefore, the "estimated" borderline (thin line) is often selected towards the satisfaction area to keep the safety. As a result, there are some violations of contract values (where the violated values are near this "estimated" borderline), but they do not cause any injury or threat to the component operation.

For *TSBC* contract types, its syntax for contract expressing is specified by the structure as follows: if <*component operating time is inside of period* $t \in T$>, then <*component data d must be within the interval of values* I_1>. Otherwise (<*component operating time is outside of period* $t \notin T$>) <*component data d must be within the interval of values* I_2>. We can notice that data verification activities depend on the adaptation expression relying on time value t. However, determining the exact timing span T is not comfortable to accomplish. The following example of a requirement specifying the motor's current intends to clarify these issues.

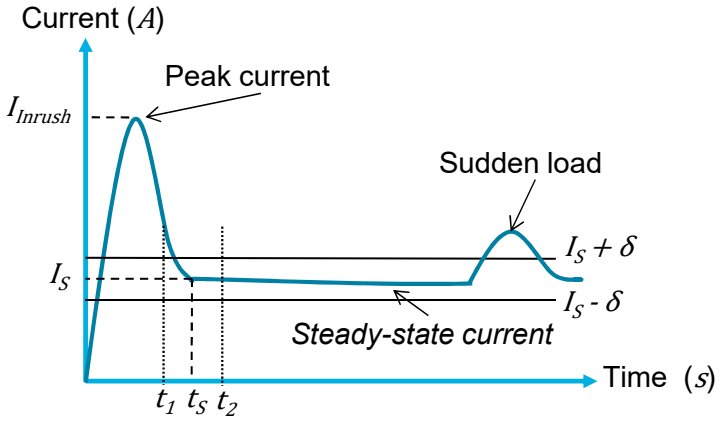


Figure 4.16: Motor current response characteristic

The requirements for the safe operation of a motor can be expressed in terms of its current response as follows: "*The motor must reach steady-state current*

value within $I_S - \delta$ and $I_S + \delta$ after t_S [s] (e.g. $t \geq t_S$), during the start-up time ($t < t_S$) the current must not exceed the value of I_{Inrush}". These requirements will be specified in the form of a mathematical expression as follows:

$$\begin{cases} i(t) \leq I_{Inrush} & \text{if } t < t_S \ (1) \\ i(t) \in [I_S - \delta, I_S + \delta] & \text{if } t \geq t_S \ (2) \end{cases}$$

The behavior requirement specification can be expressed through the following syntax for *TSBC*:

$$t < t_S ? d \leq I_{Inrush} : d \in [I_S - \delta, I_S + \delta] \quad (3)$$

We observe that determining the exact value of t_S is challenging due to discrepancies between the calculated model and the actual system behavior. Suppose that t_S is selected as the switching point of the requirement expressions based solely on calculated results. In practice, however, the actual switching point may occur either before (at t_1) or after (at t_2) the predefined value t_S, as illustrated in Fig. 4.16.

If the actual switching point occurs earlier than the predefined value ($t_1 < t_S$), any contract violations arising within the interval (t_1, t_S) will not be detected, since the contract still evaluates the requirement expression $i(t) \leq I_{\text{Inrush}}$.

Conversely, if the actual switching point occurs later than the predefined value ($t_2 > t_S$), a contract violation may be falsely detected for $t > t_S$, because the contract has already switched to evaluating the requirement $i(t) \in [I_S - \delta, , I_S + \delta]$, while the system is still in the startup phase.

The suggested solution to this problem is as follows: For the *TSBC* contract specification (its syntax is described by the label (3)), we apply an additional requirement expressing as an exception handling condition. This exception condition will only be considered when any violation contract occurs within the timing interval of (t_1, t_2). Some ideas for handling contract violations during this interval are as follows: we can specify a new bounded range of values, in which, if the violated value satisfies that new restriction, the violation is ignored. Otherwise, it will be prevented. Another possibility is to count the number of contract violations corresponding to the contract FB's *REQ* event trigger. That number of violations can be compared to a certain allowable number (For example, *the number of contract violations within (t_1, t_2) must be less than or equal to 3*). We can also set the boundary frequency of violations by request events. For example, *in 100 occurrences of REQ events, there should be no more than 10 violations.*

For *TCFB* contract types, the causal relation characterizes the association between a triggering event (q) and a subsequent event (e). In general, a single triggering event q may lead to multiple subsequent events e. However, in this thesis, contract satisfaction for a standard timing contract specification is defined more strictly as follows: for each occurrence of a triggering event q, there must exist exactly one corresponding subsequent event e that satisfies the specified timing constraint associated with q.

If fewer or more than one subsequent event satisfies the defined timing constraint for a given triggering event, this is considered a contract violation.

In cases where a triggering event is intended to produce multiple subsequent events within a bounded time interval, such behavior must be explicitly modeled using the exception handling mechanism.

Building an exception-handling mechanism applying both contract FB types (*TSBC* and *TCFB*) can be considered as building the contract annexes or subcontracts. These subcontracts will be activated only when a violation arises during the implementation of the main contract.

The proposed syntax for expressing the exception handling contains two different condition parts. The first part (denoted by C_1^{EH}) specifies the condition under which the exception handling can apply. The second part (denoted C_2^{EH}) represents the extended range of conditions where the violation value must be met to be ignored by the exception mechanism. The rule for ignoring a contract violation in the exception mechanism is that the condition in C_1^{EH} must first be satisfied before being considered in the condition expression C_2^{EH}. Therefore, the syntax for the exception handling mechanism be expressed in a format as $(C_1^{EH} -> C_2^{EH})$. A contract violation will be ignored if the value of that violation fulfills both conditions in C_1^{EH} and C_2^{EH}. Otherwise, it will be classified as violation. Sections 4.4.2 and 4.4.3 will present more details about the syntax and semantics of the exception handling condition specification and the use cases.

4.4.2 Syntax for exception handling specification of *TSBC*s

For *TSBC*'s exception handling syntax, its "*First Condition*" C_1^{EH} is used to set the applied condition. The applied condition can specified based on either the expression of the time variable or the formula condition of the number of occurrences of the *REQ* event of the FB. The "*Second Condition*" C_2^{EH} is used to check the value of the contract violation after the "*First Condition*" has been met. The expression in C_2^{EH} represents a comparison of the violation value within a new range, which expands the "normal" range of the contract.

4.4.2.1 Exception handling conditions apply within a specified period

The ideas for building the syntax and semantics of the exception handling specification will be presented and clarified with the following use case example.

A conveyor belt transfers workpieces between two workstations, where a workpiece is placed at one position and picked up at another. When this conveyor serves in a "normal" operation, its motor current is stable within an interval value (e.g., $i \in [I_{min}, I_{max}]$). This property is monitored and guaranteed by a corresponding contract. During the conveyor working procedure, when a workpiece is loaded or unloaded from this conveyor, this can cause the motor's current value to change. If the change occurs within the loading or unloading time interval (suppose $t \in [t_1, t_2]ms$) and the difference of the value is not too large compared to the set value in the contract specification, we can ignore these contract violations. Therefore, we can establish the exception handling condition in several cases as follows:

Case 1 : In the first case, we can set up an exception specification for ignoring all values that violate the contract, where those values are within the allowable range (suppose $[I_{min} * (1 - x\%), I_{max} * (1 + x\%)]$) within a specified period ($t \in [t_1, t_2]ms$). This case can be expressed by the following syntax:

$$t| = [t_1, t_2]ms- > d| = [I_{min} * (1 - x\%), I_{max} * (1 + x\%)]$$

Case 2 : This case is almost similar to ***Case 1***. However, we only want to accept a certain number of violations (for example, N times) that satisfy the exception specification in ***Case 1*** (instead of all violations). That means that if the number of violations exceeds N times or the violation value is out of range, the contract violation will be prevented rather than ignored. We can use the following syntax to specify this exception condition:

$$t| = [t_1, t_2]ms- > d| = [I_{min} * (1 - x\%), I_{max} * (1 + x\%)] \times N$$

Other cases : In the above two cases, we only consider exception conditions for contract violations within an allowed range of values and only within a certain period. Similarly to the contract specification syntax, we can use the syntax $d| = R$ (stands for $d \in \mathbb{R}$) for exception condition specification to express that the violation value is any (that implies that with any value of the violation contract). It allows us to define exception conditions for the following cases:

- The syntax of the exception condition $t| = [t_1, t_2]ms- > d| = R$ represents that any violation with any value in the interval $[t_1, t_2]ms$ will be ignored.
- The syntax of the exception condition $t| = [t_1, t_2]ms- > d| = R \times N$ implies that only N times of the contract violations with any value in the timing interval of $[t_1, t_2]ms$ will be ignored. If there are more than N times of the contract violations or the contract violation is outside the above timing period, that violation will be considered as real violation.

Summary: The syntax for exception handling conditions applies within a specified period for *TSBC*s that can be summarized as follows:

$$t\,\langle CTyp\rangle\,\langle V||I\rangle\,[\langle u\rangle] - > d\,\langle CTyp\rangle\,\langle V||I\rangle\,[\times N]$$

Where:

- $t\,\langle CTyp\rangle\,\langle V||I\rangle\,[\langle u\rangle]$ and $d\,\langle CTyp\rangle\,\langle V||I\rangle$ are exactly interpreted as in section 4.3.1.
- $[\langle u\rangle]$ indicates the time unit of the set time values.
- $[\times N]$ is an optional text field, a natural number $N > 0$ used to define the allowed number of times for ignoring violations.

4.4.2.2 Exception handling conditions apply within the number of a causing event

As a reminder, the operation of an IEC 61499 compliant component is caused by an input request event (it is typically named *REQ*). That is, the output value of this component is affected by the input value triggered by the corresponding *REQ* event. In other words, it can be expressed under a discrete function as $y_i = f(x_i)$. Where $i = 0...N$ represents the order of the request event, x_i represents the input value corresponding to the occurrence of the request event at times of i, and y_i is the output value corresponding to the input value x_i. Therefore, the specification for exception conditions can also be based on the number and positions of that causing event (*REQ*). It means that the setting of "*First Condition*" C_1^{EH} can be based on an identified ordering number of the occurrence of the request event *REQ*. For example, the exception condition will only be considered if the occurrence of the violating contract was caused by triggering the *1st REQ* event. In addition, the expression of C_1^{EH} can also be determined based on the number of occurrences of a causing event. For example, if the occurrence of the violation contracts corresponds to the interval between the *50th* and the *80th* triggering of the causing event *REQ*, those violations could be rechecked in the "*Second Condition*" C_2^{EH}.

Regarding the expression of the "*Second Condition*" C_2^{EH}, we can specify in the same way as the examples described in section 4.4.2. In particular, the violating contracts can be considered and ignored within a limited or unlimited number of violations. We can also restrict that the violating values are arbitrary or that they must be within an acceptable range of values, for example, $[I_{min} * (1 - x\%), I_{max} * (1 + x\%)]$. The following use case examples give a more detailed explanation for these ideas.

Case 1 : The specification of the exception condition for the case: If any violation is induced by the request event (suppose REQ_k), where the order of that request event k is in the range of $[a, b]$ (that means $k \in [a, b]$, where $a, b \in \mathbb{N}$ and $a < b$). At the same time, if the value of this violation contract is in the range value of $[I_{min} * (1 - x\%), I_{max} * (1 + x\%)]$, then this violation will be ignored. The syntax for this case will be then described as follows:

$$e| = [a, b] -> d| = [I_{min} * (1 - x\%), I_{max} * (1 + x\%)].$$

Case 2 : This case represents the combination of the ***Case 1*** in this section and the ***Case 2*** in section 4.4.2. That means that the specification exception condition just aims to ignore a specified number of contract violations (for example, only N times, instead of all violations) that satisfy the exception specification in ***Case 1***. We can use the following syntax to specify this exception condition:

$$e| = [a, b] -> d| = [I_{min} * (1 - x\%), I_{max} * (1 + x\%)] \times N$$

Other cases : Similar to section 4.4.2, we can use the expressing part $d| = R$ to represent that the violating value is arbitrary. Therefore, by replacing the

condition expressions in C_1^{EH} in section 4.4.2, we can express other exception specifications as follows:

- The syntax of the exception condition $e| = [a, b] - > d| = R$ expresses that any violation caused by the request event REQ_k, where its order k is in the range $[a, b]$ ($k \in [a, b]$), will be ignored within any value of that violation.
- The syntax of the exception condition $e| = [a, b] - > d| = R \times N$ implies that only N times of the contract violation caused by the request event REQ_k, where its order k is in the range $[a, b]$, will be ignored within any value of that violation. If there are more than N times of the contract violations or the contract violation is outside the range of [a, b], that violation will be prevented.

Summary : The syntax for exception handling conditions applies within the number of causing events for *TSBC*s that can be summarized as follows:

$$e\,\langle CTyp\rangle\,\langle V||I\rangle - > d\,\langle CTyp\rangle\,\langle V||I\rangle\,[\times N]$$

Where:

- $e\,\langle CTyp\rangle\,\langle V||I\rangle$ represents the setting of conditional expressions (C_1^{EH}) based on the ordering number of the causing event.
- $d\,\langle CTyp\rangle\,\langle V||I\rangle$ are exactly interpreted as in section 4.3.1.
- $[\times N]$ is an optional text field, a natural number $N > 0$ used to define the allowed number of times for ignoring violations.

4.4.2.3 Apply exception handling based on the violation rate

In the cases introduced above, the exception conditions in the "*First Condition*" C_1^{EH} apply just for a fixed span or the number of causing events. That means that there are no more exception conditions after that definition interval. It is sometimes not enough to cover some application cases. Illustrative use case examples presented below will clarify the motivations and provide detailed solutions for enriching the requirements specifications in the contract applications.

Case 1 - Violation rate based on the execution cycle : Let's consider a use case example as follows: A conveyor delivers products within a desired mass to the palletizing operation. It is necessary to ensure that the weight of a product is within the range value of $[m_1, m_2]$. Let's assume that there are *100 products* on each pallet. A pallet can comprise up to *5 products* with a weight greater than m_2 and less than or equal to m_3. Our task now is to specify these requirements for the *TSBC*. First, we can define the requirement specification for the weight of a product using the contract expressed through the following syntax:

$$t| = R?d| = [m_1, m_2]$$

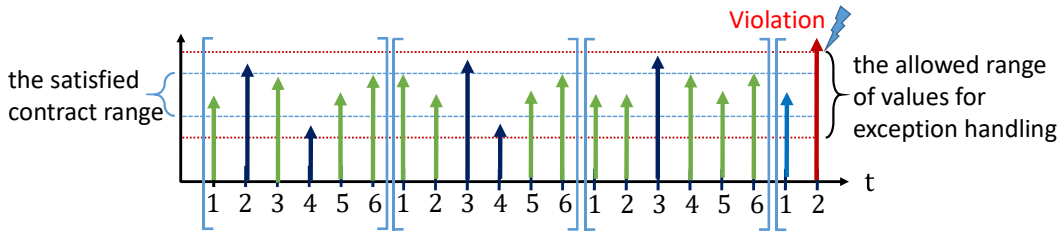
Next, we can establish requirements for palletizing operation by adding a specification as the exception condition for the monitoring contract defined on the *TSBC*. We can imagine that a product needs to check its weight every time the *REQ* event is triggered. So, if the *REQ* event of the *TSBC* occurs *100* times, this corresponds to *100* products being verified. The following syntax is used to specify this abnormality condition:

$$ec = 100 -> d| = (m_2, m_3] \times 5.$$

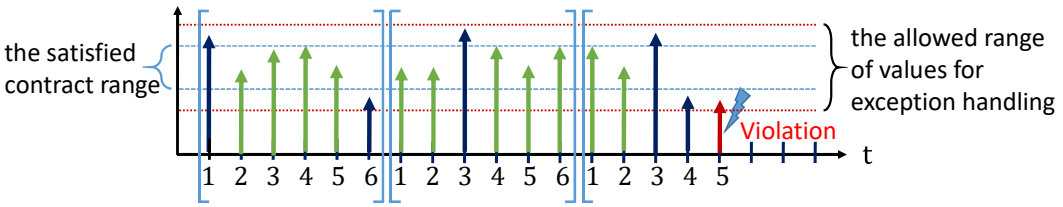
It is important to remember that after *100* product checks since start-up, if the contract and its exception conditions meet the requirements, then the next occurrence of the *REQ* event will be counted as *1*. Besides, the number of errors will also be reset to *0*.

Fig. 4.17 provides more detailed descriptions of applying the mentioned type of exception condition specification. In Fig. 4.17(a) and Fig. 4.17(a), data whose values fall within "*the satisfied contract range*" (e.g. $[s_1, s_2]$) are the contract satisfactions. Besides, "*the allowed range of values for exception handling*" area (e.g. $[p_1, p_2]$) defines the new permitted range for contract violations. The exception condition specification is expressed as follows: For the cycle of *6* request events, if there are some contract violations occur ($d \notin [s_1, s_2]$), but the violation value is still within the range new permitted range ($d \in [p_1, p_2]$). At the same time, if the number of those violations is not greater than *2*, then those violations are ignored. This specification is expressed by the following syntax:

$$ec = 6 -> d| = [p_1, p_2] \times 2.$$



(a) Violation due to exceeding the acceptable range



(b) Violation due to exceeding the frequency of allowable exceptions range

Figure 4.17: Outline an example of the requirement to pack a container of *6* products in which the total number of products that undergo exception handling is no more than *2*.

In Fig. 4.17(a), we see that contract violation occurs because the contract violation value exceeds the allowable value range. In 4.17(b), the last violation is

interfered with because the total number of violations (the sum of the previous violations and this violation) exceeds the allowed number.

These aforementioned cases would be abstracted in the generic form by the following syntax:

$$ec = Y- > d\,\langle CTyp\rangle\,\langle V||I\rangle \times X$$

Where:

- Y:is a natural number that indicates the set value of the number of request events in a cycle.
- X: is a natural number used to determine the number of ignored violations thanks to the exception condition.

Case 2 - Violation rate based on a time cycle : This case is almost similar to the presented **Case 1** above. There is only one difference in replacing the condition in the C_1^{EH} expression. In particular, instead of the number of request events (denoted by the "***ec***" keyword), the exception condition in the C_1^{EH} is determined by a time cycle term (represented by the "***tc***" keyword). This specification type is valuable when there is difficulty in determining the number of request events due to the high frequency of triggering request events. Note that when we use the expression of time variables, we also need to add the time unit for the time value.

For example, we can specify the exception condition that if the number of contract violations does not exceed X times during a cycle time $T[s]$, then these violations are ignored. The contract will intervene immediately if the violation value exceeds the permitted range or the total number of violations exceeds the allowed number of X. This case can be described by the following specification syntax:

$$tc = T\,[\langle u\rangle] - > d\,\langle CTyp\rangle\,\langle V||I\rangle \times X$$

Where T is a natural number, it refers to the set time value.

When a cycle has finished, if there are violations and those violations meet the exception condition specified, then the number of violations will reset to *0*.

Case 3 - Violation rate over last causing events : In this case, the exception condition expression is built and checked based on the number of violations over the last causing events. Unlike the described cases above (***Case 1*** and ***Case 2***), the violation count in this case will not reset to *0* after a specific number of the causing events. Instead, it is continuously updated over the last number of triggering events. The following example will further explain the idea applied to this case. Imagine this: A conveyor belt transports products between two workstations in a production process. Products are loaded at the upstream workstation (position 1) and unloaded at the downstream workstation (position 2). Let us assume that the process is synchronous. It implies that when a product is fed at the workstation 1, another one will be taken away at the workstation 2. Therefore, there will always be a specific amount of product (assume Y products) on the conveyor during the

operation of the process. It is further supposed that a product has a standard size within $[s1, s2]$. If a product has a size that is out of the range of $[s1, s2]$ but still within the interval values of $(s2, s3]$, it can be then considered for ignoring under the exception condition. Furthermore, another requirement is to ensure the distance between the products and the load capacity on the conveyor. To achieve this purpose, it must be ensured that no more than X products with dimensions between $(s2, s3]$ are on the conveyor during operation.

We can use *TSBC* to specify and monitor the above requirements as follows: First, checking the size of a product will be accomplished through contract specification syntax as:

$$t| = R?d| = [s_1, s_2]$$

Here, whenever *TSBC*'s *REQ* event is triggered, that corresponds to a product requested for the check. Therefore, the exception condition can be expressed as an assertion that it is not allowed that more than X violations exist over the last Y times of the request event (*TSBC.REQ*). This exception condition will be characterized by the syntax as follows:

$$le = Y- > d| = (s_2, s_3] \times X$$

Here, using the keyword "***le***" means that the number of contract violations is counted based on the last of the request events of the *REQ*.

Fig.4.18 shows the component's result for implementing the exception condition specification that there must not be more than 2 violations over the last 6 request events.

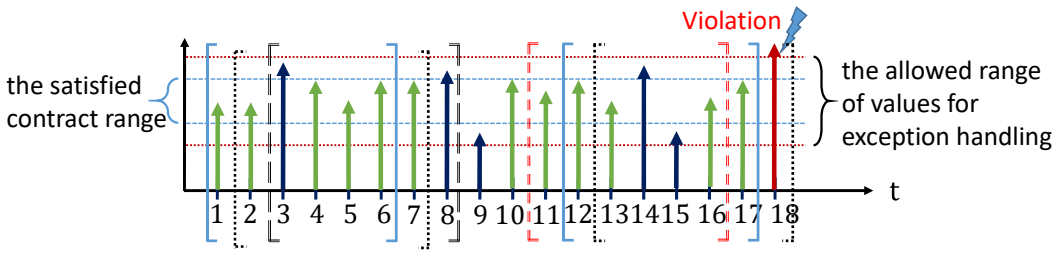


Figure 4.18: An example to permit the exception with 2 violations in 6 times of the last request event

The exception condition of the mentioned example cases is represented through the general syntax as follows:

$$le = Y- > d\,\langle CTyp\rangle\,\langle V||I\rangle \times X$$

Where:

- Y: is a natural number that refers to the set value of the number of last request events.
- X: is a natural number used to determine the number of violations that will be ignored over the last of causing events (the last of Y).

Case 4 - Violation rate based on a time frame : A difference compared to the introduced case above is the replacing the keyword "***le***" with "***lt***" in the C_1^{EH} expression. The purpose of using this specification type is that it can deal with the difficulty in determining the number of request events due to the high frequency of triggering request events. The functional and practical principles are almost similar to the introduced ***Case 3*** above.

For example, the semantics for exception conditions are: No more than X violations are allowed to exist in the last of Y [*time units*] (e.g., *seconds*) during the whole component's operation. This exception condition will be presented by the specification syntax as follows:

$$lt = Y\,[\langle u \rangle] - > d\,\langle CTyp \rangle\,\langle V||I \rangle \times X$$

4.4.3 Syntax for exception handling specification of *TCFB*s

As mentioned in Section 4.3.2, a standard timing constraint satisfaction in this thesis is only one causing event (denoted by q) associated with one satisfying following event (denoted by e). It means that it will be a timing contract violation if we can find some following events (they satisfy the timing constraints) associated with a causing event. However, in some cases, we need to allow for one cause event that can generate more than one following event that meets the timing constraint condition. You cannot express this kind of requirement in the actual syntax of the timing contract. Instead, you can only note it as an exception condition of the timing contract specification. This section will introduce the syntax and some applying cases of exception conditions to timing contracts. These contents will be explained and clarified through some specific use case examples.

4.4.3.1 Syntax

The exception condition specification for *TCFB* consists of two expression parts similar to the explanation of *TSBC* that is introduced in the section 4.4.2. The definition and use of the expression in the "*First Condition*" C_1^{EH} is the same as in the exception condition specification for *TSBC*. Here, the difference with the *TSBC* exception is only in the expression of the "*Second Condition*" C_2^{EH}. In particular, we are not curious about the contract violation values. Instead, we are only interested in the number of following events that satisfy the timing restriction corresponding to the triggering event. Therefore, the syntax for specifying exception handling of *TCFB*s can be summarized as follows:

$$\langle KoC \rangle\,\langle CTyp \rangle\,\langle V||I \rangle - > NoS\,\langle CTyp \rangle\,\langle V||I \rangle\,[\times N]$$

Where:

- $\langle KoC \rangle$ consists of $\{e,\ t,\ ec,\ tc,\ le,\ lt\}$ that represents the keyword to set the "First Condition" C_1^{EH}.
- NoS is a keyword that refers to the number of satisfied events corresponding to a request event.

- The settings for value text field $\langle CTyp \rangle \langle V||I \rangle$ and the optional field $[\times N]$ are exactly interpreted as in previous sections.

4.4.3.2 Examples

Some examples to explain the semantics of syntax in some use cases will be introduced in this section. Note that the exception handling for *TCFB* only applies to the causal relationships of *Reaction* and *Age* and does not apply to the *Occurrence* constraint. This is because the *Occurrence* specification only monitors the occurrence of a single event. Nevertheless, $TCFB$ exception conditions are considered based on the number of satisfied following events corresponding to a causing event.

Example 1:The requirement specification for the exception condition for a component is as follows: *During the entire execution of the component, each causing event will generate exactly two following events that satisfy the contract instead of one.* This requirement specification for exception condition is represented by the syntax as follows:

$$t| = R- > NoS = 2$$

Example 2: The requirement specification for the exception condition for a component is as follows: *There can be 2 to 3 of the satisfied following events for a causing event* REQ_k *if the order of this request event k is in the range [100, 500]. If the order of this request event* REQ_k *is outside that range, then there should be only one following event that satisfies the contract.* This exception condition specification is represented by the syntax as follows:

$$e| = [100, 500]- > NoS| = [2, 3]$$

Example 3: The semantic of the specification for an exception condition is characterized as follows: *For any time* t *less than* 5s*, there will be* 2 *following events satisfying the time constraint associated with a causing event, out of which there should be only* 1 *satisfying event for a causing event.* This exception condition specification is represented by the syntax as follows:

$$t < 5s- > NoS = 2$$

Example 4: The semantic of the specification for an exception condition is: *No more than* 5 *times is allowed to exist in the last of 10 causing events, in which one causing event generates from 2 to 3 of the satisfied following events.* The syntax for this exception condition specification will be written by:

$$le = 10- > NoS| = [2, 3] \times 5$$

Note that the above examples represent only a few specific cases. Exception condition specifications can be set for various actual use case requirements, as long as the parameter changes conform to the given formats of the condition expressions (C_1^{EH} and C_2^{EH}) in the syntax.

4.4.4 BNF

The following grammar summarizes the specification patterns for establishing the exception handling condition syntax for both types of contract function blocks *TSBC* and *TCFB*.

⟨*EHCondSpec*⟩ ::= ⟨*AppliedCondition*⟩ -> ⟨*AppliedValues*⟩ [⟨*PermitTimes*⟩]

⟨*AppliedCondition*⟩ ::= ⟨*ConditionVar*⟩ ⟨*Operation*⟩ ⟨*ValueField*⟩ [⟨*TimeUnit*⟩]

⟨*ConditionVar*⟩ ::= '`t`' | '`e`' | '`ec`' | '`tc`' | '`le`' | '`tl`'

⟨*Operation*⟩ ::= '`=`' | '`<`' | '`<=`' | '`>`' | '`>=`' | '`#`' | '`|=`' | '`|#`'

⟨*ValueField*⟩ ::= ⟨*Value*⟩ | ⟨*Interval*⟩

⟨*Value*⟩ ::= ⟨*Number*⟩ | ⟨*Number*⟩ '`.`' ⟨*Number*⟩ | ⟨*Infinity*⟩

⟨*Number*⟩ ::= 0 ... 9

⟨*Infinity*⟩ ::= '`R`' | '`inf`' | '`..`'

⟨*Interval*⟩ ::= ⟨*Boundary Value*⟩ '`,`' ⟨*Value Boundary*⟩

⟨*Boundary*⟩ ::= '`[`'|'`]`'|'`(`' |'`)`'

⟨*TimeUnit*⟩ ::= '`ns`' | '`us`' | '`ms`' | '`s`'

⟨*Value*⟩ ::= ⟨*Number*⟩|⟨*Number*⟩ '`.`' ⟨*Number*⟩

⟨*Number*⟩ ::= 0 .. 9

⟨*TimeUnit*⟩ ::= '`ns`' | '`us`' | '`ms`' | '`s`'

⟨*AppliedValues*⟩ ::= ⟨*AppDataVar*⟩ ⟨*Operation*⟩ ⟨*ValueField*⟩

⟨*AppDataVar*⟩ ::= '`d`' | '`NoS`'

⟨*PermitTimes*⟩ ::= × ⟨*Number*⟩

5 Evaluation

Contents

5.1 Evaluation for TSBC 80
5.1.1 Verification for contract specification without exception handling 80
5.1.2 Verification for contract specification with exception handling 83
5.2 Evaluation for *TCFB* 87
5.2.1 Validation of the "*Occurrence*" specification 88
5.2.2 Validation of the "*Age*" specification 90
5.2.3 Validation of the "*Reaction*" specification and exception handling 91
5.3 Combining and extending abilities of *TSBC* and *TCFB* 95
5.3.1 Scenario description 95
5.3.2 Experiment setup and simulation-based validation and evaluation using 4*diac* 97

The contents of this section will focus on the evaluation of the correct working of the IEC 61499 contract FBs (*TSBC* and *TCFB*), which have been created and introduced in this thesis. Verification aims to answer the following questions: *Can requirements of contracts and exception conditions accurately be expressed using the proposed specification syntax and semantics*? *Do such specified contract FBs correctly respond to their associated contract and exception specifications*?

The assessment progress is accomplished based on the simulation-based evaluation method. It is conducted through the following steps:

First, we manually generate a set of test values for each contract specification. The purpose of this step is that we can manually determine from this set which value will result in contract satisfaction and which value will cause contract violation.

For example, we want to generate *5000* requests to simulate the implementation

of a component, meaning that there needs to be *5000* values corresponding to the number of request executions of that component. Suppose that we need to monitor that the input data of that component must be within the range values of $[10, 20]$ during the entire implementation.

Next, simulation-based evaluation will be conducted with a simple application and implemented in *4diac*. In the first experiment, we generate all values of the test set that satisfy this contract. If the simulation results show any violations, it means that the contract FB did not work correctly. Otherwise, we can conduct the subsequent experiment where there will be some violating values in the test set (Assume that the value at position *2345* is *23*, at position *4001* is *8*, and at position *4532* is *25*). Within these values, we can assume that when the exception condition is not applied, this simulation results show the satisfaction contract for the first *2344* request events and the violation contract for the 2345^{th} request event. It will result in the normal execution state being disabled. The goal is to ensure security if a contract violation is detected.

In the next step, we change the violating value at the 2345^{th} position to a satisfactory value (e.g., *13*). We can predict that the contract violation will occur at the 4001^{st} position instead of at the 2345^{th} position. When the simulation is restarted, if the results are as expected, we can repeat the value change at position *4001* and check the response of the $TSBC$ to the contract violation at the next position (*4532*).

The evaluation process can repeat many times with different numbers of violations and at diverse positions of the violation values. In addition, the verification and evaluation will also be in conjunction with some exception condition specifications in different scenarios. The manual evaluation is performed and analyzed before implementing the simulation-based evaluation process. If the manual evaluation results match the simulation results, we can conclude that they are correct according to the simulation-based evaluation.

The evaluation will be carried out under the following assumptions/conditions:

- **Hardware platform**: Tab. 5.1 presents details of the hardware platform specifications used for the evaluation processes.

Table 5.1: The technical specifications of hardware platform

Processor	Intel Core i5 CPU M540 @ 2.53GHz ×4
Memory	3.6GB
OS Name	Ubuntu 20.04.4LTS
OS Type	64-bit

- **Software platform**: The IEC 61499-compliant development tool *4diac-IDE* version 1.12.0 will be used to develop the contract FBs (*TSBC* and *TCFB*) and the IEC 61499 FBs of different components. These FBs will then be used to construct and model the IEC 61499-compliant control applications with

contracts. The simulation of those applications will operate on the IEC 61499 compliant runtime environment *FORTE* version 1.12.0.

- **Others assumptions**: The time overhead and the precision of the FBs, which concern the timing functionality such as *E_CYCLE* or *E_DELAY* in *4diac*, strongly depend on the hardware platform configuration and the operating system on which it is running. For instance, an *E_CYCLE* will not be able to work stably if its set value is less than *20 ms* in the Windows operating environment, but it can still work well and stably with a set value of *1 ms* or even less without any problems on Linux operating environment.

  Therefore, some assumptions that are necessary for the evaluations are listed as follows:

  - To have a well-defined specification of timing constraints, we should also check and determine the precision of runtime response. For example, if we set the time value to $x[ms]$ at DT port of the FB *E_CYCLE* or *E_DELAY*, the value of jitter may be $y[ms]$, that means the response time could be within $x \pm y[ms]$. Based on the experiment results, the sampling time value is selected at *50 ms* to get a steady process of evaluation activities. The jitter value is designated at *10 ms* during implementation on (*FORTE*). The experimental results in section 5.2.1 will prove this choice.
  - The number of contract FBs (*TSBC* and *TCFB*) that can be used in an application is unlimited if this application does not have any constraints for its operation time. If an application has a timing constraint, the number of contract FBs should be calculated based on that time constraint. Since each contract FB will consume a certain amount of time, the number of contract FBs used will obviously increase the execution time of this application.
  - The time overhead of a component FBs, as well as *TSBC* and *TCFB*, executed and running on the selected hardware platform, is assumed for all experiments of evaluation activities in the worst case at *10 ms*. The goal is to ensure the suitable choice of sampling time used for the evaluation implementation of the FBs *TSBC* and *TCFB*. Let's imagine that we set a sample time at *10 ms* for an application as a serial network of 10 different FBs. It leads to a trigger of the next cycle occurring while the application has not finished its execution of the trigger of the previous cycle. It can cause incorrect results in the assessment activities.

The evaluation for correct operation of a single contract FB concerning its specified syntaxes of contract and the exception condition specification will be performed by following steps:

- A contract specification syntax (defined at the data port *Assert*) and maybe the exception condition specification syntax (defined at the data port *Except*) are used to express the semantics of a defined requirement of an example case.

- Based on the requirement description, we create a set of test values where the satisfaction and violation values can be given manually. For monitoring, the violation values can be marked in a table.
- The next step is to compare the manual evaluation values with the results of the simulation implementation. If there is any difference between the manual evaluation results and the simulation results, it means that the FB is not working correctly.

Notes: These are some notes for the process of evaluating the results as follows:

- The simulation-based evaluations were performed with a reasonably large amount of test values and over a long period (they have been performed by implementing more than 500,000 values and over more than 6 hours of continuous execution time). However, the number of the test values and the execution time have been reduced to make it easier for the reader to follow the graphs and manual analysis.
- The evaluations for the implementation and performance of contracts (*TSBC*s and *TCFB*) in the IEC 61499-compliant applications were published in some scientific papers [133, 134, 135]. Therefore, these evaluations will not be included in this chapter. Instead, they are presented as additional information in the appendices. Note that the interface of contract FBs in the old versions, including events and port names, may differ from the current versions, but the main features and application methods will remain the same.

5.1 Evaluation for TSBC

This section will describe the verification and evaluation of the syntaxes of specifications and the performance of the *TSBC* in two use cases involving contracts with and without exception handling specifications.

5.1.1 Verification for contract specification without exception handling

For analyzing and assessing the correctness of the implementation of the *TSBC* in this case, the examinations are carried out in a general procedure as follows:

- **Step 1**: First, a list of values is created corresponding to each occurrence of the request event(*REQ*) of *TSBC* with a specified cycle time (for example, *50ms*).
- **Step 2**: Subsequently, these values will be analyzed and checked manually based on the contract specification. From those analytical results, a prediction of the position of the request event (*REQ*) driving the violation contracts is provided and marked.

- **Step 3**: An application in *4diac* is built and implemented for testing and verification in the IEC 61499-compliant real-time runtime environment. For driving this experiment, we create a new FB named *DataReader* and use it to get and feed the values from the mentioned test list to the *TSBC* data input port (*DataIn*). Another function of this FB is to convert the provided values to the compatible data type (*LREAL*) of the data input port (*DataIn*) of the *TSBC*.

- **Step 4**: Observe and compare the results of simulation implementation. If the simulation results match the predicted results, then the contract specification and the *TSBC* work as expected. As a result, the operation of the *TSBC* will stop at the position where the first violating value exists.

- **Step 5**: Edit the first violated value so that this edit value satisfies the contract. Repeat the experiment from **Step 3**. These procedures will be repeated until the last violation value, which is in the original list of **Step 1**.

Assessing the correctness of *TSBC* can be explained using the following example.

Example 1 : *The requirement for the operation of a component is expressed as follows: During the time* $t < 10$ *seconds since the startup of the operation, the input value of that component must be in the interval of* $[23, 25]$. *After that period, its value must be in the range of* $[29, 31]$. *If any input value does not meet the specified requirements above (violation of the contract), the component's operation must be stopped immediately.* The following syntax will be used to characterize the specification requirements for the desired behavior of that component:

$$t < 10s \ ? \ d| = [23, 25] \ : \ d| = [29, 31]$$

A set of 1000 values is generated to check the correct operation of the *TSBC* based on the above requirements. These values are saved as a file in **.txt* format with the name "*vtsbc.txt*" and fed with a sampling time at a rate of *50 ms*. Fig. 5.1 shows the test values used for the simulation-based validation process in the next step. In this test list, there is a total of 11 values that do not meet the contract specification. The positions of the request event associated with the violation value include *690(28.49)*, *716(31.46)*, *730(28.40)*, *754(28.76)*, *830(31.89)*, *848(31.35)*, *902(31.45)*, *913(31.51)*, *920(28.87)*, *929(31.38)*, and *967(32.69)*.

Here are some remarks regarding Fig. 5.1:

- Firstly, the labels *Min1*, *Max1*, *Min2* and *Max2* stand for the threshold values 23, 25, 29 and 31 respectively. The values are in the ranges [*Min1*, *Max1*] and [*Min2*, *Max2*] that will cause the satisfaction results associated with the set time of the contract.

- Secondly, the first request event(*REQ #1*) corresponds to the starting time value $t = 0s$.

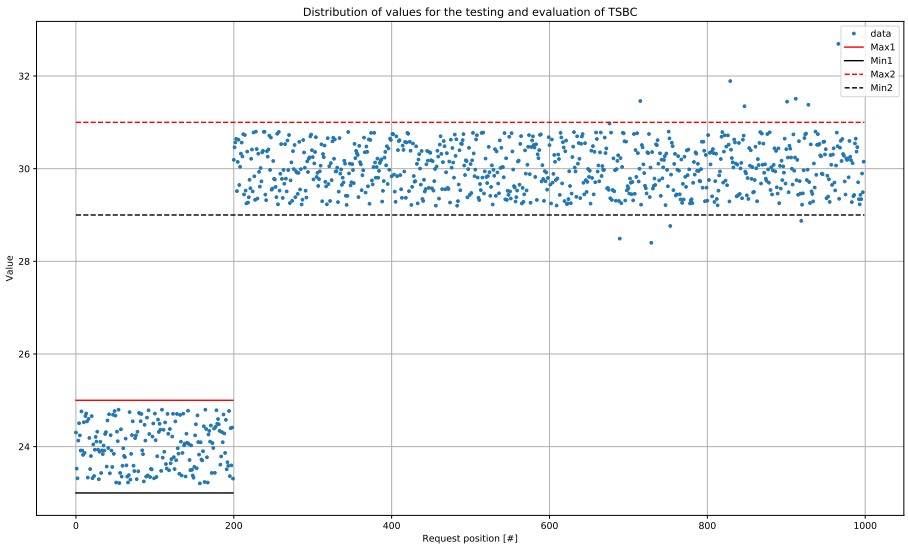


Figure 5.1: The given values for testing of *TSBC* contract specification

- Finally, the *E_CYCLE* FB triggers the simulation operation with a sampling time of *50 ms*. It means that *201* first request events are equivalent to the execution time ($t < 10s$), and from the 202^{sd} request event that is equivalent to the execution time ($t \geq 10s$).

An IEC 61499 compliant application in *4diac* is built and implemented to serve verification processes based on simulation in the *FORTE*. This application includes a *E_CYCLE* FB, a *DataReader* FB, and a *TSBC* that is interconnected in a network as shown in Fig. 5.2. The timed behavior is simulated by connecting the *E_CYCLE* and the *DataReader*, where each test value in the file *vtsbc. txt* is sequentially fed to *TSBC* by *E_CYCLE* every 50 ms.

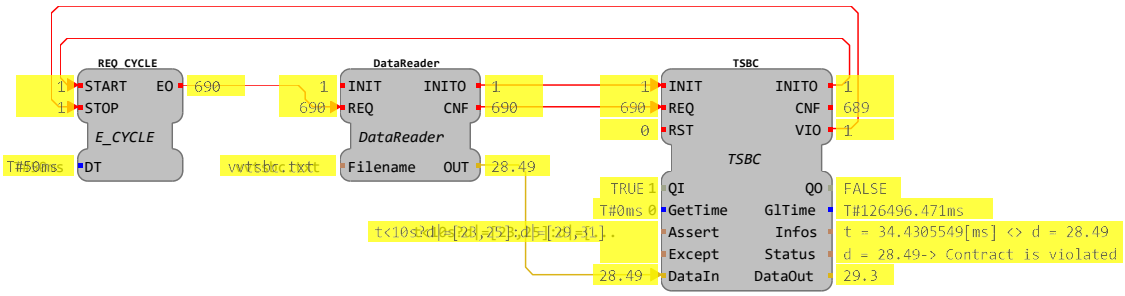


Figure 5.2: Verification result of the implementation of *TSBC* without exception conditions

The expected result for implementing this experiment in *4diac* is that the tested values from *1* to *689* will give a satisfying result. It corresponds to the trigger of the event *TSBC.REQ* from the 1^{st} to the 689^{th}. If *TSCB* reacts correctly, then

at the 690^{th} trigger of *REQ* event, it will raise a violation event *VIO* because the value *28,49* is not within the allowed range [29,31].

We can easily see from the simulation results shown in Fig. 5.2 that the violation occurred at the 690^{th} trigger of *REQ* event associated with the value of *28,49*. It perfectly matches the expected results in the manual analysis step.

Next, we need to correct the violation value to a satisfaction value (e.g., *29.49* instead of *28.49*) and then perform the simulation again. In this experiment, the simulation results show a violation occurring at the 716^{th} trigger of *REQ* event associated with the value of *31.46*. This experiment will continue to repeat until the last violation value in the test list is detected and replaced. During these executions, the *TSCB* has always reacted correctly to the violation values existing in the mentioned test list. This proves that the *TSCB* and its specification syntax for the requirement worked correctly.

5.1.2 Verification for contract specification with exception handling

This section presents a simulation-based evaluation method for the applications with *TSCB* using a specification of the behavior requirements combined with various exception conditions. In this case, the evaluation sequence is most similar to the sequence in the section 5.1.1. It is also performed based on the list of test values stored in the file *vtsbc.txt* and repeated the experiments according to the described steps in the section 5.1.1. The following examples represent how to assess the correctness of *TSCB* execution under the requirement specification in conjunction with different exception condition specifications.

Example 2 : We consider again the component's behavior conforms to the requirement, as specified in ***Example 1***. But in this example, the behavior requirement will add the exception condition for the violations as follows: For the last of *60* consecutive request events, there is not allowed to exist more than *3* minor violations. Here, a violation will be a minor violation if the value of that violation is within the range of $[28, 32]$. The operation of this component will immediately stop at any time if a violation value is outside the range $[28, 32]$ (not a minor violation) or the number of minor violations is greater than *3*. This exception specification can be defined by the syntax as follows:

$$le = 60 - > \ d| = [28, 32] \times 3$$

The manual analysis is performed on the basis of the list of violation values in combination with the specification of the exception condition. We expect that the simulation operation for this experiment will not stop before the 929^{th} trigger of the *REQ* because all violations before this time satisfy the exception condition. At the 929^{th} trigger of the *REQ* event, the *TBSC* must raise the *VIO* event because there are a total of *4* minor violations that occurred in the last of *60* requests (from 870^{th} to 929^{th}). They include the violations at the position (value) of *902(31.45)*, *913(31.51)*, *920(28.87)*, and *929(31.38)*. The results of this activity are summarized

in Tab. 5.2 below. Note that on Tab. 5.2, the statuses marked *Ignored* are the values that cause a violation, but they meet the exception condition.

Table 5.2: Analysis result for the violation values with exception condition

#	Req. position	Value	Status
1	690	28.48	Ignored
2	716	31.46	Ignored
3	730	28.40	Ignored
4	754	28.76	Ignored
5	830	31.89	Ignored
6	848	31.35	Ignored
7	902	31.45	Ignored
8	913	31.51	Ignored
9	920	28.87	Ignored
10	929	31.38	Violated
11	967	32.69	Stop

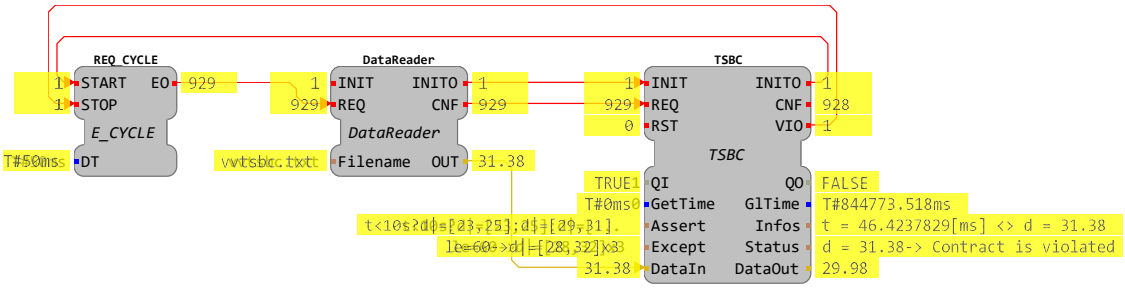


Figure 5.3: Verification result of the implementation of *TBSC* with exception conditions

The simulation-based method is conducted on the same application model used in ***Example 1***. The difference is that the *TSBC* will be added a specification of exception condition at data port *Excepts*. Here, the simulation still operated in a "normal" for *928* times of first requests even though *9* violation values already existed in the test list before the occurrence of the 929^{th} trigger of the *REQ* event. The simulation operation of the application only stops at the 929^{th} request (see Fig. 5.3) due to a violation, and that violation value does not meet the exception condition.

When we edit the violate values in the range from *690* to *848* to satisfaction values, the simulation results of the application remain unchanged. If we change any violation value in the range of the *REQ* event position from *870* to *929*, the simulation will stop at the 967^{th} trigger of the *REQ* event. This violation occurs because the violation value goes beyond the range of exceptional conditions (not a minor violation) rather than exceeding the allowable amount in the last *60* requests.

The results from these various experiments demonstrate that *TBSC* operates within the given specification for the requirement and its exception condition.

Example 3 : Let's continue exploring the example of the requirement for a component's behavior as introduced in *Example 1* and *Example 2*. Here, we conducted some experiments to check the response of *TSBC* within the given specification for the requirement combined with the exception condition. Principally, this example is mainly similar to *Example 2*. The difference, in this case, is the exception condition specification defined based on a specific period of the execution time instead of the number of request events as in *Example 2*. In particular, the specification for the exception condition is restricted in this example case as follows: The exception condition for the violations will be considered only in the interval of the component execution time of $[30, 50]s$ since start-up. Concurrently, violation values must meet the requirement in the following use cases:

- **Case 1**: *If a violation has a value within the range of* $[28, 32]$ *(minor violation), and it occurs within the time interval of* $[30, 50]s$*, then it will be ignored. If a violation occurs outside the given timespan, or if it is not a minor violation, the application must immediately stop its execution.* The specification for this exception condition is expressed by the syntax as follows:

$$t| = [30, 50]s- > \ d| = [28, 32]$$

- **Case 2**: *In this example case, the exception requirement is almost similar to the above case. The difference is that it is not allowed to exist more than 5 minor violations within the restricted timespan.* The specification of the exception condition for this case would be expressed by the syntax as follows:

$$t| = [30, 50]s- > \ d| = [28, 32] \times 5$$

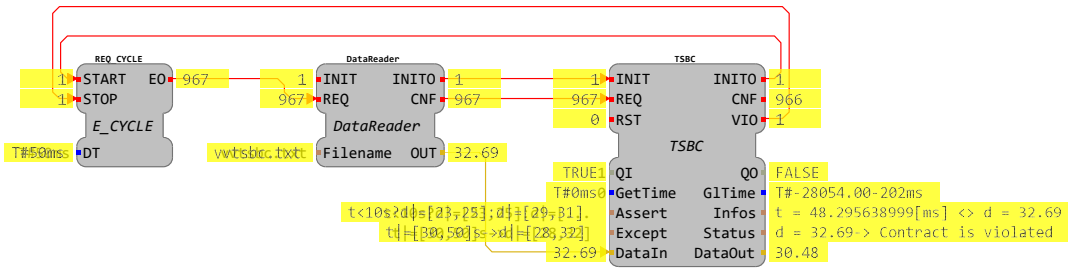
Note that for this specification, the "*First Condition*" of the exception condition (C_1^{EH}) is defined based on the execution time. However, for the manual analysis purpose, we can also estimate the result based on the set value for the E_CYCLE. In particular, within a cycle time of *50ms*, the above time interval would theoretically relate to the trigger order of the request events *REQ* from *601* to *1001*. Based on the set value of E_CYCLE, the calculation results for the estimated time for the violation values in the test list (stored in the file *vtsbc.txt*) are shown in Tab. 5.3.

For *Case 1*, the manual analysis shows the expected results that violations before the 967^{th} trigger of the request event are ignored because all of them are minor violations. At the 967^{th} trigger of the request event associated with the value of 32.69, it is not a minor violation. Therefore, the *TSBC* will raise a *VIO* event and stop the operation of the application, although this violation is still within the considered time span for exception handling. For *Case 2*, we can foresee that the *TSBC* must emit a violation event (*VIO*) when the trigger of the request event (*REQ*) is the 848^{th} occurrence. The violation at this position is the 6^{th} occurrence of minor violations within the specified time interval for the exception condition.

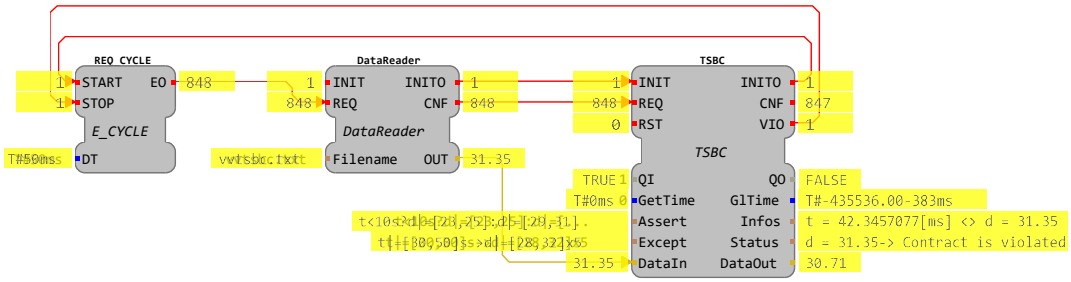
Fig. 5.4(a) and Fig. 5.4(b) illustrate the simulation results of the application implementation in *4diac* for *Case 1* and *Case 2*, respectively. The results demonstrate that they match the expected results and the prediction of the manual analysis step.

Table 5.3: Manual analysis results of the exception condition in ***Example 3***

Req. position	Time(s)	Value	Case 1	Case 2
690	34.45	28.48	Ignored	Ignored
716	35.75	31.46	Ignored	Ignored
730	36.45	28.40	Ignored	Ignored
754	37.65	28.76	Ignored	Ignored
830	41.45	31.89	Ignored	Ignored
848	42.35	31.35	Ignored	Violated
902	45.05	31.45	Ignored	Stop
913	45.6	31.51	Ignored	Stop
920	45.95	28.87	Ignored	Stop
929	46.4	31.38	Ignored	Stop
967	48.3	32.69	Violated	Stop



(a) Violation due to exceeding the acceptable range of value



(b) Violation due to exceeding the allowable exception amount

Figure 5.4: Verification result of the implementation of *TSBC*.

Similar to the evaluation strategy in the other use cases presented above, for each editing of a violation value, we restart the simulation again to verify the correctness of the syntaxes and the response of the *TSBC*. In all experiments afterward, the simulation results have no difference from the expected results. This confirms that the proposed syntaxes of the requirement specifications and the functionality of the *TSBC* work correctly.

5.2 Evaluation for *TCFB*

This section will describe the verification and evaluation of the syntaxes of specifications and the performance of the TCFB. Similar to the evaluations for *TSBC*, the correctness of the specification syntaxes and the service performance of the *TCFB* will be evaluated through various scenarios. The requirements of the *TCFB* concern different timing requirements and causal relation specifications such as *Occurrence*, *Reaction*, and *Age*. The *TCFB* contract can also add the exception conditions to expand the requirement specifications in some cases. The general sequence used for the assessment activities of the *TCFB* is listed as follows:

- **Step 1**: The first step of the procedure is to generate test data. It is a set of values for feeding the set time value to *4diac*'s timing functions. In this test list, we can define at which position a test value will cause a violation contract in the testing execution phase.

- **Step 2**: The manual analysis and estimation of results will be conducted based on the test values generated in **Step 1** combined with the requirement specification. In particular, when a value in this test list is set for a timing function in *4diac* (such as *Delay* or *Cycle*), then we can give an estimated result caused by that setting value. Note that the configurations and the operating system of the execution hardware platform on which the experiments for the evaluation are running will affect the timing of the confirmation event of the timing FBs. As mentioned in the earlier part of this chapter, the response timing can occur *10ms* earlier or later than the set value for the timing FBs (*Delay* or *Cycle*). For example, when a *Delay* function receives a set value of *500* along with the setting time unit is *ms*, and whenever this delay FB is triggered at its input event, there will be a confirmation event emitted from this FB in the interval of *[490, 510]ms* afterward.

- **Step 3**: Two different applications are created to serve the evaluation of the *TCFB*'s operation and its timing requirements specifications. The first application concerns the constant time values setting for the timing functions in *4diac* (E_CYCLE and E_DELAY). The expected result of the experiment is that if we give constant time values for these FBs, then the results of implementing this application will not change as well. The second application will link to the test values created in **Step 1**. The purpose is that we want to intentionally write some values to make the timing FB generate the events that occur outside the constraint time. The *TCFB* must then emit a *VIO* event

corresponding to that set value. To generate the signals for the simulations, besides using the standard timing FBs such as E_CYCLE and E_DELAY (obtainable in the standard library of the *4diac*), we also need to create some new FBs because the mentioned standard FBs can not read the value in a test data in $*.txt$ file.

- **Step 4**: Monitor and compare the simulation results of the application implementations. The contract specification and the *TCFB* work as expected if the simulation results match the predicted results.
- **Step 5**: Edit the violated value so that this edit value causes the event to occur within the timing constraint (satisfy the timing contract). Repeat the experiment with the second application from **Step 3** to check the reaction of the *TCFB* with the next violated value. These procedures are repeated until the last violation value in the original list of **Step 1**.

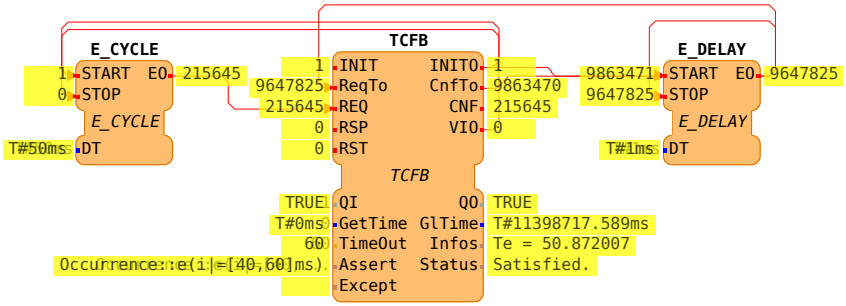
5.2.1 Validation of the "*Occurrence*" specification

The main goal of the experiment in this section is to evaluate the correct operation of *TCFB* by applying the "*Occurrence*" syntax. Simultaneously, this experiment attempts to find the answer to the specialized question concerning the simulation-based evaluation for the *TCFB*. Note that a general feature for all experiments for evaluating the specification syntaxes of the *TCFB* and its operation is that the simulation processes are periodically triggered. The applications of *TCFB* will be used to serve the evaluation activities during implementation on the selected hardware platform. Therefore, the question is, *what is the minimum value to set for E_CYCLE FB, at which it can maintain stability and reliability for the applications of TCFB?* Let's imagine that if we select a request cycle that is faster than the time overhead of the *TCFB*s, then these contract FBs do not have enough time to react according to those request events.

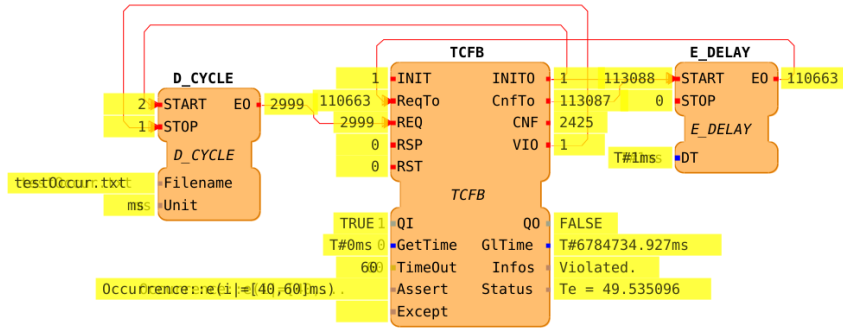
A simple application is created in *4diac* as shown in Fig. 5.5(a) to evaluate the implementation of *TCFB* that uses the "*Occurrence*" specification. This evaluation will be performed under the following assumptions: 1) The worst-case time overhead of the *TCFB* is *10 ms*. 2) The estimated jitter value of implementing the *E_CYCLE* on the selected execution hardware platform is *10 ms*. Therefore, if the set time value for the *E_CYCLE* is *50 ms*, then we expect the output event *EO* of the *E_CYCLE* to occur within the interval $[40, 60]ms$. As a result, we can specify the time restriction for the occurrence of the output event *EO* of the *E_CYCLE* by the following contract: *The EO signal of the E_CYCLE occurs within the interval* $[40, 60]ms$. This contract specification can be defined by the syntax as follows:

$$Occurrence :: e(i| = [40, 60]ms)$$

In this experiment case, the *E_CYCLE* plays the role of monitoring the actual occurrence time of the EO output event compared with the nominal occurrence time (defined at port DT) of the *E_CYCLE*. We can also check whether the *E_CYCLE* jitter value during implementation on the execution hardware platform



(a) The implementation of *TCFB* without violation



(b) The implementation of *TCFB* with violation

Figure 5.5: Verification result of the implementation of TCFB with "*Occurrence*" specification

exceeds the assumed value of *10ms* or not. One thing to note for the application of *TCFB* in this experiment and all applications of *TCFB* in other experiments is the link of the *TCFB* with an *E_DELAY* set to a value of *1ms*. It aims to create a 1ms scan cycle for the *Timeout* function of the *TCFB* (The timeout value is set to the value of *60*, and the time unit is seconds for all experiments in this section).

Fig. 5.5(a) shows the implementation result of the application in *4diac*. We can notice that the *E_CYCLE* has generated the event *EO 215645* times (up to the time of the picture capture, equivalent to *3 hours* of execution time). The *TCFB*) shows that there have been no contract violations up to this execution time.

This result demonstrates that with a trigger cycle of *50ms*, an application with one *TCFB* can still operate stably in the implementation environment of the selected hardware platform. Additionally, the jitter value of the *E_CYCLE* function also does not exceed *10ms*. Consequently, it matches the mentioned assumption.

In the next step, we will examine how the *TCFB* reacts when the expected event occurs later than the constrained time. For this purpose, another application is created by using a newly developed timing FB named *D_CYCLE* to replace the *E_CYCLE* in the application shown in Fig. 5.5(a). The purpose of using *D_CYCLE* is to be able to read the values from the test data in the *.txt* file for generating the trigger cycle, which the standard *E_CYCLE* cannot do.

For this experiment, a set of test values is created to cause the trigger cycles for the *D_CYCLE*, including the first *2999* values of *50* and at the *3000*th value

being *70*, and saved in the *testOccur.txt* file. The expected result for the application implementing these test values in *4diac* is that there will be contract fulfillment for the first *2999* triggers. For the 3000^{th} trigger, there must be a contract violation. The execution results shown in Fig. 5.5(b) demonstrate that the *TCFB* responds correctly to the manual analysis outlined above. Besides the presented experiment, some other experiments were performed. These were conducted by editing the values that cause contract violations at different positions in the test file and restarting the simulation procedure afterward. The results of these experiments indicate that the *TCFB* reactions always match the manual analysis results.

5.2.2 Validation of the "*Age*" specification

To evaluate the performance and the correctness reaction of *TCFB* for the syntax of the specification for the "*Age*" relation, the "*Age*" example for a robot arm in the 4.3.2.2 section will be reused here. However, the constraint time interval is adjusted to get more effects (output events) from the simulation in a shorter period. In particular, the following expression characterizes the time constraint for the material handling process: *Whenever* S_T *occurs,* S_B *has occurred within the time interval* $[400, 600]$ *before*. The timing restriction specification for this case would be expressed by the syntax as follows:

$$Age :: q- > e(i| = [400, 600]ms)$$

The evaluation process for this case will be done by running experiments on the application created in *4diac* as shown in Fig. 5.6.

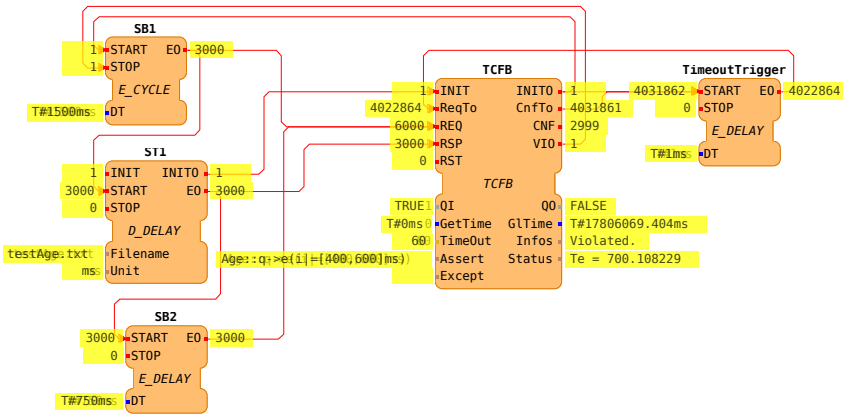


Figure 5.6: Verification result of the implementation of *TCFB* with "*Age*" specification.

In this example case, we only focus on the time constraint of the signals when the robot arm works at the position *B*. As explained in the example in the 4.3.2.2 section, the *ST* signal represents the finish signal of the workpiece processing. The time constraint for the workpiece processing is defined by the time between the *ST* and *SB* events (the robot arm reaches the *B* position). The sensor signal at position *B* is emitted twice in an operating cycle. Once when the robot moves from position *A* to position *B* (represented by S'_B) and once when the robot returns from position *C* to *A* and passes through *B* (represented by S''_B).

For the *4diac* application, a standard FB *E_CYCLE* is used to model the occurrence of the S'_B signal (labeled *SB1* in the *4diac* application). Here, the signal *SB1* does not need to operate under a time constraint. However, it is assumed that this signal (*SB1*) will be within a cycle of *1.5s*, and the output event of this *E_CYCLE* is linked to the *REQ* event of the *TCFB*. To simulate the timing effect of the *ST* signal, a newly developed FB named *D_DELAY* is developed in *4diac* to generate delayed output events. The delay time of *D_DELAY* is caused by sequentially reading the values in the list of values stored in the file **.txt*. According to the order of event occurrence in a cycle, the *ST* signal arises within a specified time interval after the S'_B signal appears. Therefore, the *E_CYCLE.EO* (imply the S'_B signal) will be connected to the *D_DELAY.START* to start the causing of the *ST* signal. The output event *D_DELAY.EO* (imply the *ST* signal) is linked to the input event *RSP* of the *TCFB*, while the *E_CYCLE.EO* connects with the input event *REQ* of the *TCFB*. As mentioned, the position sensor at *B* will trigger again when the robot arm returns from position *C*. Assume that this signal (denoted by S''_B) will occur within *750ms* after the *ST* signal occurs. A standard *E_DELAY* with a set value of *750ms* is used to model the timing effect of this signal. The *D_DELAY.EO* will trigger the start of this *E_DELAY*. Its output event *EO* is linked to the input event *TCFB.REQ*.

A set of *3000* values is created and used to check the correct reaction of the *TCFB* for the mentioned timing constraint specification. These values are saved in the file named *testAge.txt*, where the first *2999* values are *500*, and the 3000^{th} value is *700*. This means that for each trigger of *E_CYCLE.EO* (implies that the *ST* signal occurs), the *D_DELAY* will output *EO* events (represents that the *ST* signal occurs) approximately *500ms* later for the first *2999* triggers. If everything works ideally, then at the 3000^{th} trigger, the *EO* events will occur about *700ms* later. In other words, if *TCFB* is working accurately, there should be no violations on the first *2999* request triggers, and a violation must occur on the 3000^{th} request trigger. The results of the application implementation in *4diac*, illustrated in Fig. 5.6, demonstrate that the intervention of *TCFB* perfectly matches the expected behaviors that we have analyzed.

The evaluation process for this case is also carried out with some experiments, using the cause values for violations at different positions in the test file. This process is conducted in a similar way to the experiments described earlier. In all of these experiments, the results of the simulation of the application are the same as those expected from the manual analysis step. This proves that the presented syntaxes of the timing specifications for the "*Age*" constraints and the functionality of the *TCFB* work correctly.

5.2.3 Validation of the "*Reaction*" specification and exception handling

The experiments in this section are used to evaluate the *TCFB* performance and the correctness of the syntax for specifying "*Reaction*" relation constraints. For this purpose, let's consider an example of a cause-and-effect relationship for the model

of a component behavior. Assume that the request event (REQ) of the component model occurs within [900,1100]ms. We are now going to consider some of the following example cases for the reaction of the response event of this behavior FB.

- **Case 1**: *Whenever a request event (**REQ**) occurs, a response event (**RSP**) will occur within* $[400, 600]ms$. It is a standard "*Reaction*" specification. It means that for a request event, there is only one response event that satisfies the contract. It will be a contract violation if there are more or less satisfied response events. To define the contract for use cases where the specification exceeds this standard, we can use the exception specification. That will be shown in the following example cases.

- **Case 2**: *Whenever a request event (**REQ**) occurs, a response event (**RSP**) occur within* $[400, 600]ms$. *When a request event causes two satisfying events, it will not be a violation of the contract if two conditions are met simultaneously. 1) These events occur within* $[400, 900]s$ *since the startup of the execution, and 2) the mentioned thing does not occur more than 5 times during that period (*$[400, 900]s$*).*

- **Case 3**: *The normal behavior of the timing response of the behavior FB is that whenever a request event (**REQ**) occurs, a response event (**RSP**) occurs within* $[400, 600]ms$. *However, if a request event causes two satisfying events, it can be ignored if it satisfies the extra condition that there are no more than* 5 *such violations in the last of* 100 *request events.*

The time contract specification in the three examples above is the same. It is expressed by the following syntax:

$$Reaction :: q\,(i| = [900, 1100]\,ms) - > e\,(i| = [400, 600]\,ms)$$

For the exception specification, the ***Case 1*** has no exception, for the ***Case 2***, the exception handling mechanism will be added to $TFCB$ by using the following syntax:

$$t| = [400, 900]s - > NoS = 2 \times 5$$

and the exception condition specification syntax for ***Case 3*** is given as follows:

$$le = 100 - > \;\; NoS = 2 \times 5$$

The evaluation process is always the same as presented in the previous sections. Furthermore, the *4diac* application used for the experiments in this section is most similar to that used for the *Age* constraint case. In particular, compared to the application in the *Age* experiments, the trigger order of E_DELAY and D_DELAY is swapped.

An E_CYCLE is used to model the trigger signal for the execution of this application. This FB generates a trigger cycle of *1s* during the application implementation phase. Its output event EO is linked to the REQ event of the $TCFB$ and the $START$ event of the E_DELAY to trigger these events simultaneously.

The E_DELAY is named $RspSig1$ to model the timing effect of the following signal. It will fire the output event EO after approximately *500ms* whenever its $START$ event is triggered. The event EO of this E_DELAY is connected to the RSP event of the $TCFB$. As a result, this E_DELAY FB will always cause a response event that satisfies the timing contract. The $START$ event is activated by the EO event of this E_DELAY. The values stored in the **.txt* file will be read sequentially to generate the delay time for this FB. We notice that the first response signal ($RspSig1$) appears about *500ms* after the signal $ReqSig$ occurs. If the value of the $RspSig2$ signal arises within the period $t < 100ms$, it may generate two response signals that satisfy the constraint of the timing contract.

To check and evaluate the correct operation of the $TCFB$ with the syntax of the *Reaction* constraint specification, a set of *1000* values is generated and stored in the file named *testReact1.txt*. Fig. 5.7 shows the distribution of test values used for the simulation-based validation process in the next step. In this list, most of these values are in the interval $[140, 160]$ to cause output events that are outside the time constraint defined in the contract. There are also a total of *11* values in the interval $[15, 75]$ to generate other output events that satisfy the timing contract (occurring within $[400, 600]ms$ after the occurrence of the REQ event). These values and their position in the test list can be found in Tab. 5.4. This table also shows the manual analysis results for the three cases mentioned above.

For **Case 1**, the violation event (VIO) of the $TCFB$ will fire and block the execution of the application at the 577^{th} occurrence of the REQ event. This is due to the fact at this position, there are two events that occur within $[400, 600]ms$ after the occurrence of the REQ event.

For **Case 2**, the application is terminated by the $TCFB$ at the 881^{st} REQ. The reason is that there are *6* times in which we can find *2* satisfied response events related to a request event. It exceeds the permission defined in the exception condition. Therefore, it is aborted by the $TCFB$.

In **Case 3**, the application is only stopped by the $TCFB$ at the 979^{th} REQ instead of at 881^{st} REQ like in **Case 2**. This is because the count and update of the number of violations will only be within the last *100* request events. In this example case, there are a total of *6* violations from the 880^{th} REQ to the 979^{th} REQ (in the last *100* requests). As a result, the $TCFB$ fires the VIO event to block the execution of the application.

Fig. 5.9 shows the simulation results of the application implementations in the first experiments concerning the different specifications mentioned above. We can see that the ***TCFB*** produces the correct reaction event associated with the contract and exception handling specifications to which it is assigned.

To measure the reliability, the next trials will be executed by sequentially substituting a violation value with a satisfaction value, and then the simulation will run again for each case. In particular, for ***Case 1***, as long as the $TCFB$ receives the first causing violation value during the execution time of the application, it will immediately fire the VIO event. For ***Case 2***, the $TCFB$ fires the VIO event whenever it receives the first *5* causing violation values while the application runs. For example, if we change the value at the 577^{th} position from *50* to *160*, then

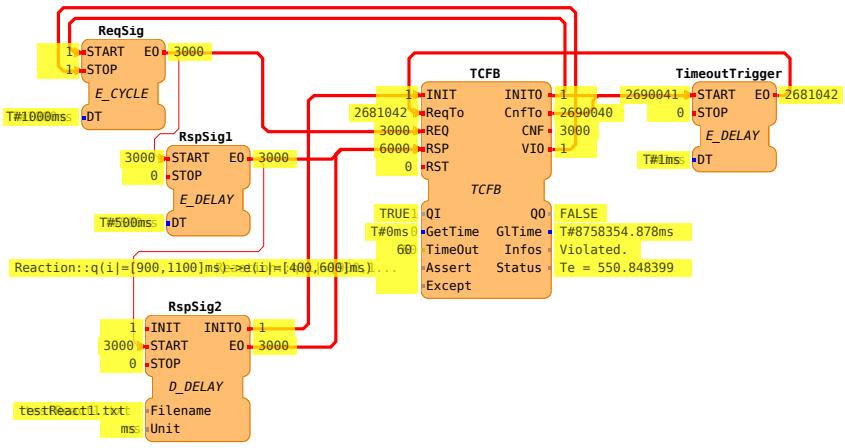


Figure 5.7: Application for verification of $TCFB$ in 4*diac*

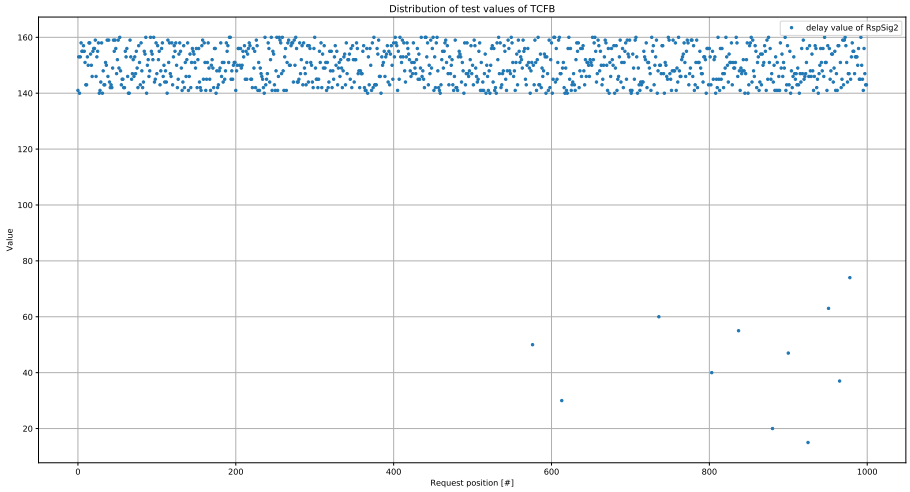


Figure 5.8: Distribution of test values for ***Case 2*** and ***Case 3***

$TCFB$ will emit the VIO event on the 901^{st} cycle. This process repeats until the last causing violation value in the list is reached. In **Case 3**, when we modify a cause violation value in the first *5* positions in Tab. 5.4 (e.g., from *577* to *838*), the simulation output will not change compared to the first trial. In other words, the $TCFB$ will always emit the VIO event on the cycle of *979*. On the other hand, if we change any value in the last *6* positions in Tab. 5.4 (e.g. from *881* to *979*), the VIO event will not fire during the execution time of the application. This is because the number of violations is always less than or equal to *5* in the last *100* events. In all subsequent experiments, the simulation results show that $TCFB$ always gives a correct answer, which is consistent with the results of the manual analysis. This proves that the proposed syntax of the requirement specifications and the functionality of $TCFB$ work correctly.

Table 5.4: The manual analysis for the application used different specifications

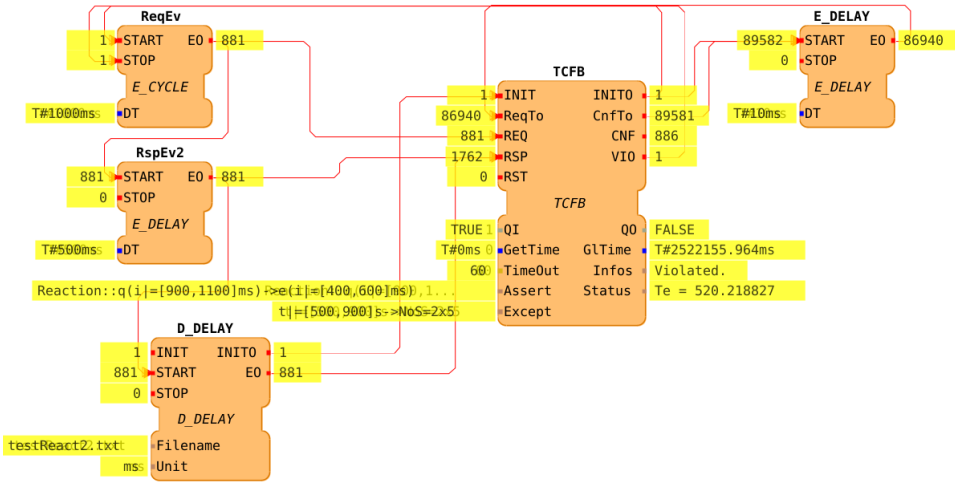
#	Req. position	Value	Case 1	Case 2	Case 3
1	577	50	Violated	Ignored	Ignored
2	614	30	Stop	Ignored	Ignored
3	737	60	Stop	Ignored	Ignored
4	804	40	Stop	Ignored	Ignored
5	838	55	Stop	Ignored	Ignored
6	881	20	Stop	Violated	Ignored
7	901	47	Stop	Stop	Ignored
8	926	15	Stop	Stop	Ignored
9	952	63	Stop	Stop	Ignored
10	967	37	Stop	Stop	Ignored
11	979	74	Stop	Stop	Violated

5.3 Combining and extending abilities of *TSBC* and *TCFB*

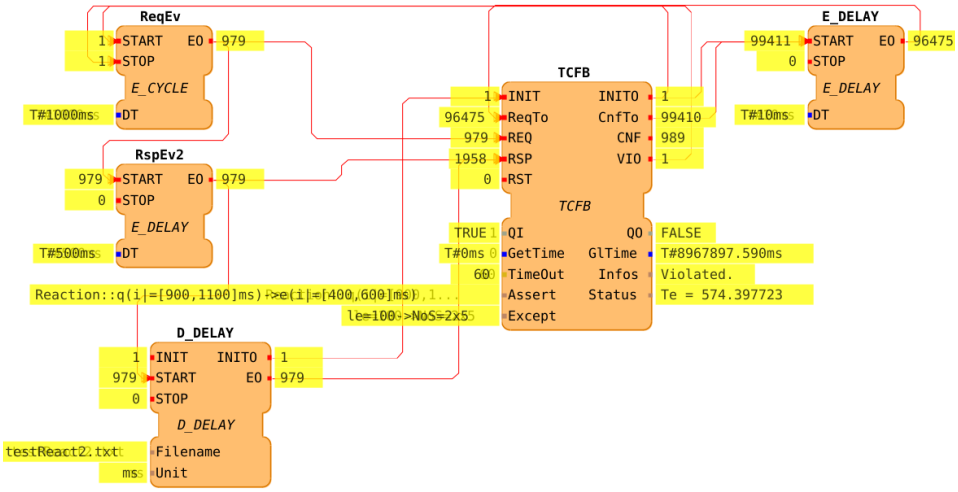
The above sections have used different standard specification syntax and the possible inclusion of additional exception conditions to examine and assess the correctness of the function and the reaction of *TSBC* and *TCFB*. In practice, however, the requirements for components in complex systems are sometimes more complex than the presented specification syntax. Therefore, it raises the question of whether we can handle a complicated requirement using the existing contract specification syntaxes. The purpose of this section is to present some concepts and solutions concerning the mentioned issues above.

5.3.1 Scenario description

Let's consider an example of speed control for an autonomous vehicle. We want to ensure that the automatic driving function must meet the requirements for safety and energy-saving driving. Assume that this vehicle will reach the optimal state in terms of safety and energy saving when its speed (denoted by s) is in the range $[105, 135]$ (unit: km/h). Therefore, the controller needs no adjustments to maintain speed in this range. This means that the control value (denoted by c) is set to 0. If the vehicle speed is less than 105 km/h, the controller must increase the engine speed by setting the control value to 1. If the vehicle speed is greater than 135 km/h, the controller has to make a change to decrease the engine speed by setting the control value to -1. These requirement specifications for this case would be expressed by the expression as follows:



(a) The implementation result of the ***Case 2***



(b) The implementation result of the ***Case 3***

Figure 5.9: The implementation of *TCFB* with exception handling specification.

$$\begin{cases} s < 105 \rightarrow c = 1 \\ s \in [105, 135] \rightarrow c = 0 \\ s > 135 \rightarrow c = -1 \end{cases}$$

We can see that the quality of the self-driving function and ensuring the safety requirements of an autonomous vehicle depend on the correctness of monitoring and controlling the speed. If the controller makes a wrong decision or the speed monitoring is not good enough, it can cause dramatic problems and unsafe situations for the vehicle. Therefore, the requirement is that the controller must provide the control value with respect to the measured speed.

5.3.2 Experiment setup and simulation-based validation and evaluation using *4diac*

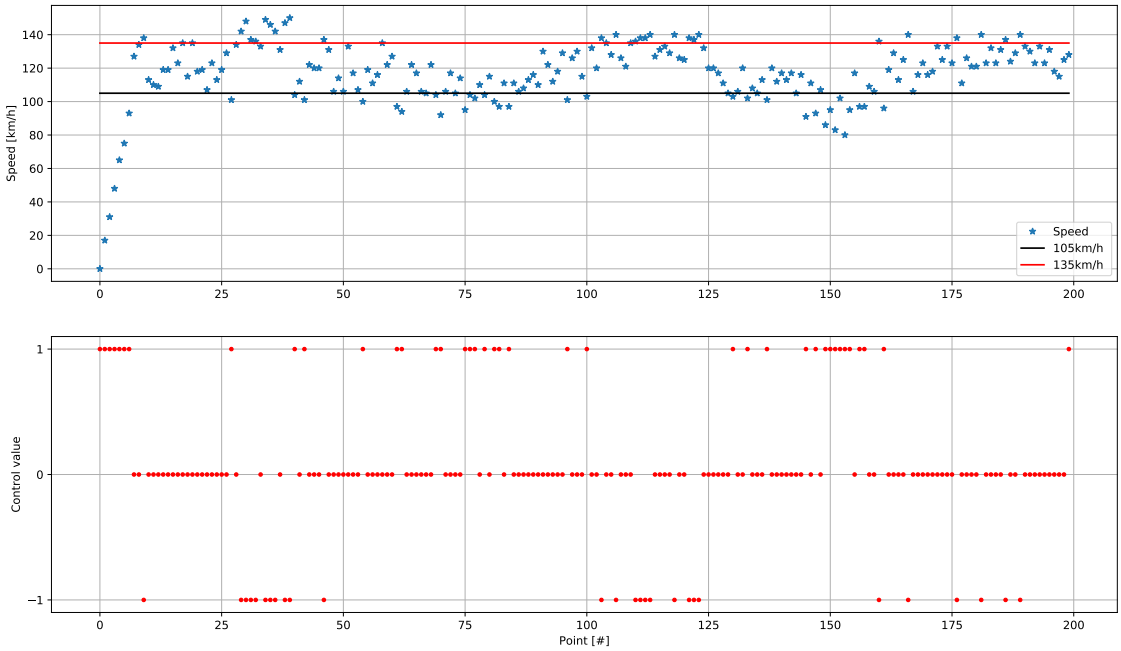
In this case, the presented requirement specification is completely outside the standard specification for $TSBC$s introduced in the previous sections. In particular, the standard $TSBC$ specification is expressed under the $if...else$ statement, while in this example case, the specification is presented under the $if...elseif...else$ statement. This section presents how to combine the existing contract FBs and the standard syntaxes to specify and address the non-standard requirements for the $TSBC$.

To produce the simulation-based evaluation for this example case, let's assume that we have a set of the measured speed values of the vehicle, as shown in the upper part of Fig. 5.10(a). The control values will then be manually generated to model the controller behavior with respect to the measured speed values as shown in the lower part of Fig. 5.10(a). These values are stored in the files named *Speed.txt* and *CtrlSig.txt* respectively.

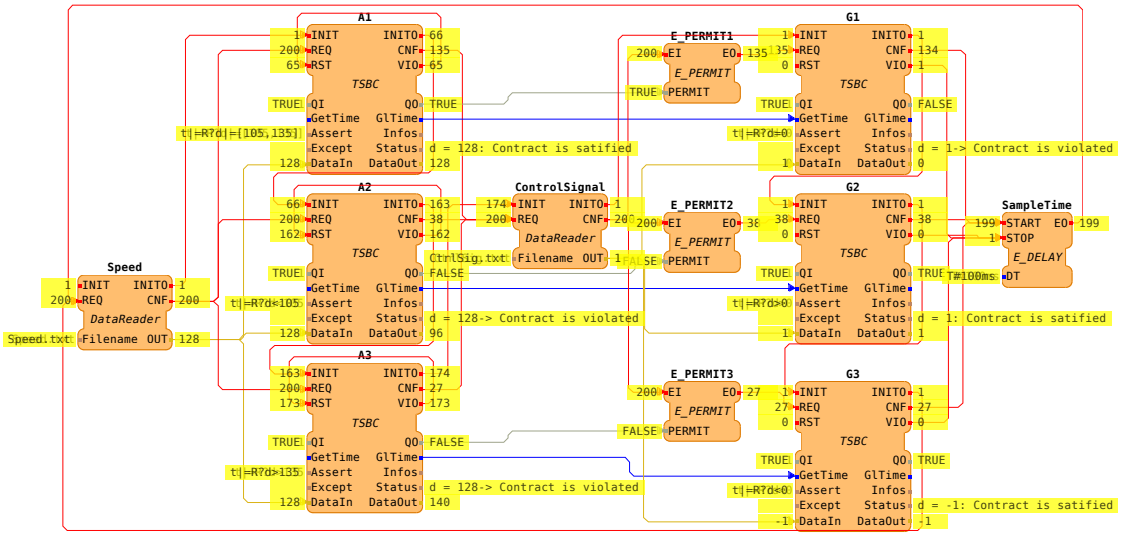
As shown in Fig. 5.10(b), a *4diac* application is built and used to model the vehicle behavior. The speed and control values will be monitored by the contract FBs (*TSBCs*). In this application, a *DataReader* (labeled *Speed*) is used to read the speed values of the vehicle from the file *Speed.txt*. A read speed value is checked simultaneously by 3 *TSBC*s (labeled *A1*, *A2* and *A3*) with respect to 3 different speed ranges. Note that the speed test ranges defined at $A1$, $A2$, and $A3$ are independent, which means that a speed value can only satisfy the condition at one FB (at $A1$, $A2$, or $A3$). For example, $A1$ is specified by the syntax of $t| = R?d| = [105, 135]$, $A2$ by $t| = R?d < 105$ and $A3$ by $t| = R?d > 135$. As a result, in normal operation of this application, there will always be one $TSBC$ that fires the *satisfying* event (*CNF*) and the other two remaining $TSBC$s will fire the event *violation* (*VIO*). The FBs that trigger the VIO event are reset immediately by linking the VIO event to the RST event in the same $TSBC$. The reset aims to allow these FBs to continue operating in the next cycle. In this application, the *Guarantee* FBs ($G1$, $G2$, and $G3$) can only act if their corresponding *Assumption* FB ($A1$, $A2$, and $A3$) satisfies the specification. The E_PERMIT[1] FBs are used to support this objective. Here, $G1$ is specified by the syntax of $t| = R?d| = 0$, $G2$ by $t| = R?d > 0$, and $G3$ by $t| = R?d < 0$.

For example, the speed value is 128. This value meets the set condition on $A1$ and does not satisfy the set conditions on $A2$ and $A3$. Therefore, $A1$ will emit the CFN event associated with $QO = TRUE$, while $A2$ and $A3$ fire the VIO event associated with $QO = FALSE$. $A1.CNF$ event will then trigger the *ControlSignal* FB to request reading a control value from file *CtrlSig.txt*. Since $A1.QO$ linked to $E_PERMIT.PERMIT$ with a $TRUE$ value, the E_PERMIT linked $A1$ will fire the EO event to request the serving of the $G1$. For $G2$ and $G3$, there is no request to work because their E_PERMITs are triggered with a $PERMIT$ value of $FALSE$. Therefore, the EO event will not fire from these E_PERMIT FBs

[1]By using an available function block in the *4diac* standard library called *E_PERMIT*. Its functional principle is that the *EO* event is only fired whenever an *EI* event is triggered and associated with the *PERMIT = TRUE*



(a) The speed values and the controlled values



(b) The implementation result in 4diac

Figure 5.10: Graph of data and test results

($E_PERMIT2$ and $E_PERMIT3$). In this example, involving speed with value 128, the application checks for $A1- > G1$ agreement. It means that $G1$ expects the controller to deliver a value of 0. However, the controller delivers a control value of 1, which does not satisfy the contract, and therefore, it causes $G1$ to fire the VIO event.

For the experiment, we created 200 speed values along with 200 control values (see Fig. 5.10(a)) to evaluate the performance of the control and monitoring activities. In 200 control values, 199 control values meet the requirement specification, and the last control value (200^{th} value) is incorrect (e.g., set to 1 instead of 0) to check the *TSBC*'s response to this error. That means, for the first 199 requests, the operation of this application will be in a normal state. At the 200^{th} request, a violation occurs. The simulation results show that the $TSBC$s in the application implementation comply with the expected results. There have also been other experiments to assess the response of the $TSBC$s. For example, if we change the control value from 1 to 0, the simulation results will show that the application will not have any contract violations. However, when we change the control value from 1 to -1, the violation occurs due to its non-compliance with the contract. In addition, the experiment was performed by keeping the control value of 1 but changing the speed value from 128 to 99. For this set speed value, $A2$ will trigger the controller operation, and the control value is checked by $G2$. Within this pair of values, the application works without violation. If we change the speed value from 128 to 150 and keep the control value at 1, then in that case, $A3$ will trigger the $ControlSignal$ FB and the control value will be checked by $G3$. The operation of the application is blocked at this point because the control value does not satisfy the condition at $G3$. In all of the above situations, $TSBC$s always reacted as expected. We can recognize that, by a suitable combination of FBs, we can apply these contract FBs in complex and diverse requirements specifications in applications.

In addition, the evaluation of the performance of $TSBC$ and $TCFB$ applied in hierarchical systems was presented in publications [134] and [135]. It will be presented again in Appendix A and B) of this thesis. It is important to note that these evaluations were conducted with earlier versions of $TSBC$ and $TCFB$, in which the event ports and data ports of these FBs can be named differently from the current version. However, the operating principles and basic features of those FBs are unchanged.

6 Conclusion and future work

Contents

6.1 Summary 101
6.2 Future work 103

6.1 Summary

This thesis introduced an effort to investigate an application-oriented solution for the virtual integration of IoT (Internet of Things) components, supporting the digitalization of processes in the context of Industry 4.0. The primary need is to propose a method for monitoring and ensuring accuracy regarding both aspects of the event's occurrence time and the desired behavior of the components through concrete use cases and reference implementations. The objects in this thesis intend to model and develop the distributed control system and its components in an IEC 61499 standard-compliant framework. Furthermore, the contract-based concept was used and developed in the IEC 61499 compliant framework to test the interaction of I4.0 components through virtual integration.

In this thesis, model-based engineering combined with contract-based design methodology is used to develop the distributed control system. In particular, an IEC 61499-compliant development framework, 4*diac*, and its runtime environment, *FORTE*, has been selected to support the development activities. The main research objectives and contributions of this thesis are listed below:

1. System development conforms to the V-model, where the system, components, and functionalities are developed by using the IEC 61499-compliant framework, specifically the software development tool *4diac*.

2. Through the developed contract FBs, the contract-based design approach is integrated into the IEC 61499-compliant applications. These contract FBs allow requirements to be specified and monitored during the development and implementation phases. The functionalities of these contract FBs are described in detail below:

    - The Time-Sensitive Behavioral Contract (*TSBC*) FB was developed to ensure and monitor the timed behavior correctness of components and systems under the form of the assume/guarantee contracts. The *TSBC*s are used as assuming and guaranteeing FBs that encapsulate the behavioral FB and act as pre- and post-conditions for executing the behavior. A behavioral FB can represent a component, a subsystem, or a system. When developing a safety-critical CPS, it is essential to make assumptions about components and ensure that a behavioral FB operates correctly under those assumptions. This ensures that components, subsystems, and the system are always constrained and verified during the design and integration phases.
    - The Timing Contract Function Block (*TCFB*) was developed and applied to express the timing requirements in early design stages to check system requirements. It is also used to monitor the requirements in the timing aspect during the integration phases for subsystems and components and during the system operation. For such contracts, a formal specification is attached to the FBs to capture the expected timing behavior and timing constraints, such as the interaction between components/(sub)systems of industrial control systems. Therefore, they can be used to ensure the safety of components and systems.

3. Some syntaxes are proposed and used in this work to express the requirement specifications for these contract FBs ($TSBC$ and $TCFB$). There are two possibilities to define the contract specification for those contract FBs. It can be written directly as a string at the port called *Assert* of the contract FBs in *4diac* or defined by using a developed support tool called *PassCS*.

4. Syntaxes for expressing exception handling have also been proposed and developed. When using the exception conditions, the contract FBs can ignore a certain number of contract violations during the operation process as long as those violations still satisfy the defined exception condition. This gives us the ability to define contracts in a more advanced and flexible way.

5. Contract implementation as IEC 61499 FBs allows seamless integration into the standard IEC 61499 deployment process and enables online monitoring of IEC 61499 applications. A simulation-based evaluation method was used to verify and evaluate the performance of the contract FBs along with their syntaxes. The simulation process was run on 4*diac* and its runtime environment $FORTE$. Some simple use case examples have been presented to illustrate and explain how requirements and exception conditions can be included in the IEC 61499-compliant applications through the contract FBs and their syntaxes. The

evaluation is performed on separate single-contract FBs. It is also executed and evaluated in the context of collaboration in hierarchical applications (shown in Appendix sections A and B). Additionally, the ability to use these contract FBs to specify complex requirements is demonstrated and evaluated.

It is important to note that the applications of these contract FBs are not limited to the given use cases presented and demonstrated in this thesis. These contract FBs can be used in a much broader range of domains. They can also be applied to more complex system development scenarios in the context of flexible and complex specifications. Furthermore, the feasibility and utility of these contract FBs will continue to be investigated with other demonstrators on other platforms. The idea of implementing *TSBC*s as monitors to specify allowed system states within determined time periods in the FPGA has also been explored in [136].

6.2 Future work

Apart from the presented usefulness and feasibility of using the Contract FBs for CPS development, some improvements need to be conducted and achieved to make them more convenient to use in a practical context. Some topics and issues that need to be addressed in future work are:

- Applying contracts to a FB in IEC 61499 still requires manual work. Although a support tool for contract specification (*PassCS*) has been developed to allow non-experts to easily and automatically integrate behavioral requirements for a FB in 4diac, the tool still has some limitations. One limitation of this tool is the ability to specify requirements. It can only specify one input and one output variable of a behavioral FB, and it currently lacks options to support the specification of exception conditions. Therefore, it needs better solution support for contract specification and the capability for generating and integrating contract monitors into IEC 61499 FBs and applications.

- It is not always easy to properly integrate the contract FBs into an IEC 61499 FB network. Therefore, a problem that needs to be solved is how to automatically test for compatibility and consistency ("*Virtual Integration Testing*" VIT) when constructing connections between components (represented by IEC 61499 FBs) in the network (represented by IEC 61499 compliant applications). Furthermore, support for automatic communication refinement of FBs and code generation for distributed monitors must be investigated.

- When these contract FBs are applied to applications, they take some time to execute during the running of the application. When an application runs on a single controller device (see figure 6.1), the time overhead of the contract FBs must be known to determine timing constraints for subsequent component events/data occurrence and component/system executions. When these contract FBs are applied to applications, they take some time to execute during the running of the application. When an application runs on a single

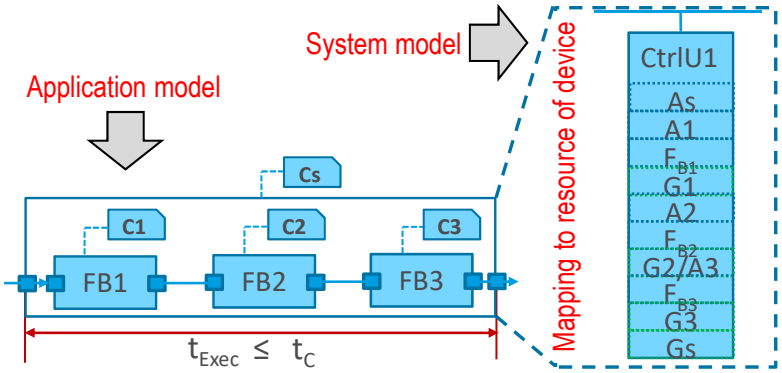


Figure 6.1: Model for runtime monitoring of a centralized control systems using contracts

controller device (see figure 6.1), the time overhead of the contract FBs must be known to determine timing constraints for subsequent component events/data occurrence and component/system executions. On the other hand, if the applications within these contract FBs are implemented on a distributed control system (see figure 6.2), this can lead to many problems that need to be considered. If an application is running on a single control device (see Fig. 6.1), the time overhead of the contracts must be known to determine time constraints for the occurrence of events/data of subsequent components and time constraints for the executions of components/systems.

- On the other hand, if the applications within these contract FBs are implemented on a distributed control system (see Fig. 6.2), this can lead to many problems that need to be considered. The reasons are time constraints are affected not only by communication time but also by contract time overhead, which can differ between mapping sources on different hardware platforms. Therefore, an assessment of the execution time overhead of the *TSBC* and

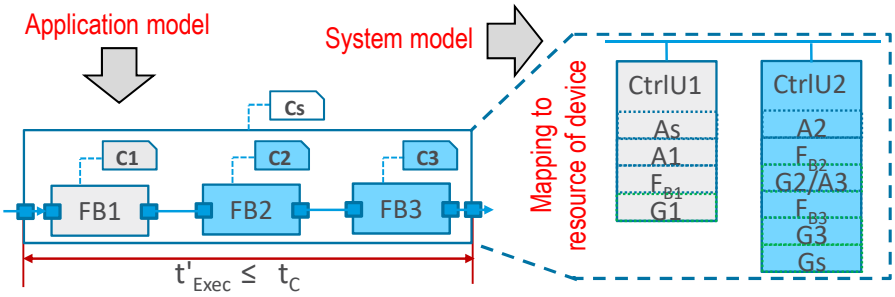


Figure 6.2: Model for runtime monitoring of a distributed control systems using contracts

TCFB for specification and application contract-based online monitoring of the distributed automation system in IEC 61499 is necessary to address.

A Demonstration of *TSBC*

A.1 System Description

This section presents a representative pick-and-place robot control system to demonstrate the application of *Timed State-Based Contracts* (TSBCs) in IEC 61499-compliant systems. The system consists of a robotic manipulator that interacts with a delivery conveyor at the pick-up location and a transport conveyor at the unloading location. The two positions are denoted as A and B, corresponding to angular positions of 0° and 180° within the robot workspace.

The robot is driven by a DC motor equipped with a gearbox to enable rotational motion, and a pneumatic cylinder is used to actuate the gripper. The gripping and releasing operations are detected by sensors *Sg* and *Sd*. The operational sequence of the system is illustrated in Fig. A.1.

The clockwise rotation from A to B is denoted as *Fw*, while the counter-clockwise rotation from B to A is denoted as *Bw*. The durations of the gripping and releasing operations (T_g and T_d) are assumed to be completed within 100 ms. Additionally, it is assumed that a workpiece is always available at the pick-up location prior to the robot's arrival.

The robot transports a workpiece to position B with a torque of 0.2 Nm and returns to position A without load. The supply voltage U_i is nominally 24 V with a tolerance of ±2%, i.e., $23.52 \leq U_i \leq 24.48$. The robot operates within a semicircular workspace bounded between A and B.

The maximum rotational speed ω_M (for PWM = 1) is bounded within [0.6335, 0.6595] rad/s with load and [0.6357, 0.6616] rad/s without load. The rotational speed ω and angular position ϕ are measured using a 16-bit rotary encoder. The executed angle is computed as the absolute difference between the current and initial positions. The execution time t_e is defined as the duration from the start to the completion of a motion.

Due to inertia, the robot may stop within the extended interval $[-2°, 182°]$.

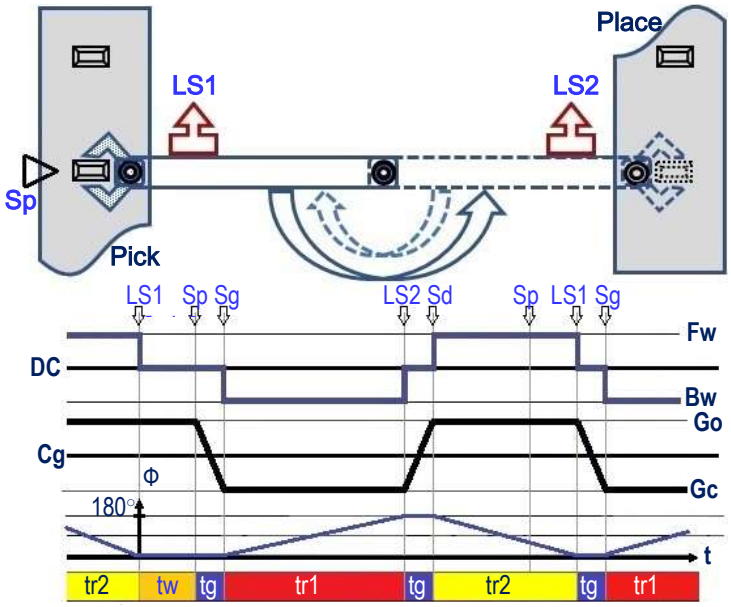


Figure A.1: Description of the Pick-and-Place robot system

Upon completing the unloading process, the robot immediately returns to the pick-up location.

A.2 System requirement and contract specification

Since the sensors (*LS1* and *LS2*) may potentially fail, the robot must ensure that its executed rotation angle ϕ remains within the workspace bounded by the interval $[-2°, 182°]$. This constraint ensures that the robot operates safely within its allowable range without relying on these position sensors. Therefore, the system requirements can be formalized using the following A/G contract:

- **Assumption:**
  - math: $23.52 \leq Ui(t) \leq 24.48$
  - syntax: $t| = R\ ?\ d| = [23.52, 24.48]$
- **Guarantee:**
  - math.: $-2 \leq \phi(t) \leq 182$
  - syntax: $t| = R\ ?\ d| = [-2, 182]$

It is assumed that the supply voltage of the robot motor is regulated by a PWM signal generated by the controller. The control system consists of three main components: the robot, the controller, and the sensor. Fig. A.2 illustrates the system architecture and the proposed methodology.

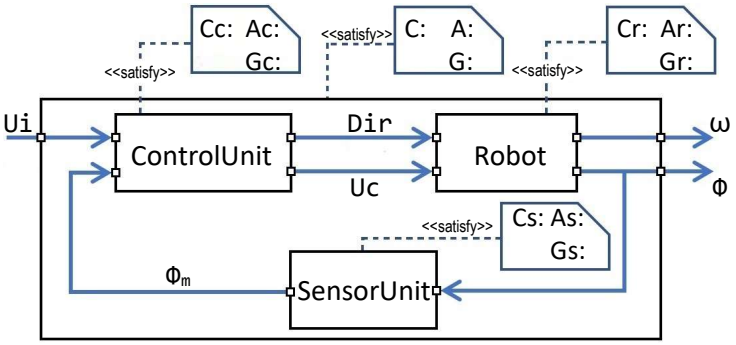


Figure A.2: Block diagram for the PnP controller design

Within the contract-based design framework, contracts are assigned to each component, namely the robot, the controller, and the sensor.

The controller design requirement specifies that the supply voltage U_c must be reduced to half of its nominal value (i.e., $U_c \leq 12.24$ V) before the robot reaches the stopping point, which is 15° prior to the final position. In other words, when the executed rotational angle satisfies $\phi \geq 165°$, the controller must reduce the supply voltage to decrease the robot's angular velocity.

The braking rotational speed ω_b is bounded within [0.3157, 0.3286] rad/s with load and [0.3178, 0.3308] rad/s without load. Based on these constraints, the average rotational speed $\bar{\omega}$ can be calculated according to the controller design requirements as follows:

$$\bar{\omega} = \frac{\omega_M \times 165° + \omega_b \times 15°}{180°} \tag{A.1}$$

The execution time t_e for the robot motion can now be determined for both cases: with load (from A to B) and without load (from B to A). It can be formally expressed as:

$$t_e = \frac{\phi}{\bar{\omega}} \tag{A.2}$$

Based on the above analysis, the bounded execution time t_e can be determined as [4.9709, 5.1756] s (with load) and [4.9552, 5.1586] s (without load).

Additionally, the switching time for adjusting the PWM signal, denoted as t_c, must be specified. This time point is defined as:

$$t_c \geq \frac{165° \times \pi}{180° \times \omega_M} \tag{A.3}$$

According to (A.3), the controller action time from the start of the robot motion is bounded within [4.5566, 4.7443] s for the case with load, and within [4.5423, 4.7287] s for the case without load.

To ensure proper deceleration, the robot must reduce its speed after reaching an executed rotational angle of $\phi \geq 165°$. Therefore, the monitoring interval for this behavior is defined as [4.7443, 4.9552] s.

A fixed switching time of $t_c = 4.9$ s is selected to guarantee that the executed rotational angle ϕ exceeds 165° before the control action is applied.

The controller assumes that the input supply voltage is bounded by $23.52 \leq U_i \leq 24.48$ V, and ensures that the controlled voltage satisfies the required behavior described above.

In summary, the controller must satisfy the following contract:

- **Assumption(A_c):**
  - math.: $23.52 \leq Ui(t) \leq 24.48$
  - syntax: $t| = R \ ? \ d| = [23.52, 24.48]$
- **Guarantee(G_c):**
  - math.: $\begin{cases} 0 \leq Uc(t < 4.9s) \leq 24.48 \\ 0 \leq Uc(t \geq 4.9s) \leq 12.24 \end{cases}$
  - syntax: $t| = (-\infty, 4.9) \ ? \ d| = [0, 24.48] \ : \ d| = [0, 12.24]$

The robot assumes that the controlled voltage U_c, provided by the controller, satisfies the desired behavior, specified as $U_c(t < 4.9, \mathrm{s}) \in [0, 24.48], \mathrm{V}$ and $U_c(t \geq 4.9, \mathrm{s}) \in [0, 12.24], \mathrm{V}$.

Due to the effects of inertia, the rotational speed cannot decrease instantaneously. However, it must be guaranteed that within 0.2 s after the controller action is initiated, the rotational speed is reduced to below 0.45 rad/s.

The robot contract can be formally defined as follows:

- **Assumption(A_r):**
  - math.: $\begin{cases} 0 \leq Uc(t < 4.9s) \leq 24.48 \\ 0 \leq Uc(t \geq 4.9s) \leq 12.24 \end{cases}$
  - syntax: $t| = (-\infty, 4.9) \ ? \ d| = [0, 24.48] \ : \ d| = [0, 12.24]$
- **Guarantee(G_r):**
  - math.: $\begin{cases} 0 \leq \omega(t < 5.1s) \leq 0.66 \\ 0 \leq \omega(t \geq 5.1s) \leq 0.45 \end{cases}$
  - syntax: $t| = (-\infty, 5.1) \ ? \ d| = [0, 0.66] \ : \ d| = [0, 0.45]$

The rotary position sensor is characterized by the resolution of a 16-bit rotary encoder. The resulting measurement error is given by $e_p = 360°/2^{16} = 0.0055°$. The sensor converts the actual physical position of the robot into a measured rotary position ϕ_m for the controller, which may deviate from the true position by up to 0.0055°. A contract for the rotary position sensor can then be formally specified as follows:

- **Assumption(A_s):**

  - math.: $-2 \leq \phi(t) \leq 182$
  - syntax: $t| = R\ ?\ d| = [-2, 182]$

- **Guarantee(G_s):**
  - math.: $-2 \leq \phi_m(t) \leq 182$
  - syntax: $t| = R\ ?\ d| = [-2, 182]$

A.3 Implementing and Evaluation

As shown in Fig. A.2, the robot's guarantee (G_r) must be compatible with the guarantee of the overall system (G). Furthermore, the guarantee G_r regarding the position ϕ must be compatible with the assumption of the sensor (A_s). The controller's assumptions (A_c) consist of the overall system assumptions (A) and the assumption about the measured position ϕ_m, which must be compatible with the sensor's guarantee (G_s). The controlled voltage Uc must be ensured by the controller (G_c) and be compatible with the robot's assumption (A_r). Fig. A.3 shows the function block network of the application model for the pick-and-place robot control system, including *TSBC*s implemented in *4diac*.

The implementation follows the principles illustrated in Fig. 4.4, and the system is built according to the structure shown in Fig. A.2. The testbed (or plant model) and the control system were implemented together within a single *4diac* application to facilitate evaluation. To test and evaluate the contract function blocks (FBs), we created a *Randn* FB named *SupplyVoltage*, which generates the supply voltage values for system execution. The *Sensor* FBs model the behavior of the limit switches, while the *E_SR* FB from the standard *4diac* library is used to represent the appearance of the workpiece. In Fig. A.3, the application of contracts at the system level is shown at the top, whereas the applied contracts at the subsystem (lower) level are depicted at the bottom.

The evaluation was conducted under the environmental assumptions introduced in Chapter 5, following the procedure outlined in Section 5.1. Across all experiments, the results demonstrate that the *TSBC* correctly responds to violation events. These outcomes confirm that the proposed requirement specification syntax and the functionality of the *TSBC* operate correctly within hierarchical, contract-based system applications.

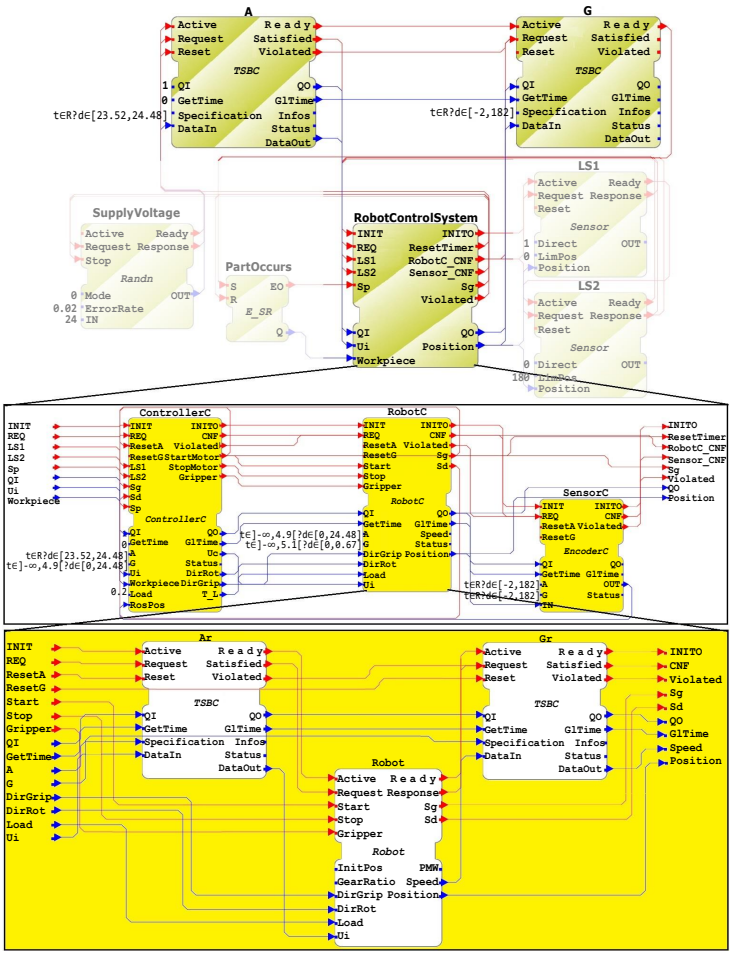


Figure A.3: The IEC 61499 system model with contract FBs

B

Demonstration of *TCFB*

B.1 System description

This section presents a simple use-case example of a pick-and-place (PnP) robot control system in a production line, as illustrated in Fig. B.1, to demonstrate the proposed timing contract methodology. The robot picks up a workpiece from the supply conveyor belt at the pick location (point A, corresponding to 0° in the robot workspace). It then transports the workpiece to the press machine at point B (90° in the workspace), where it must wait until the press machine completes the processing. Afterward, the robot moves the finished product to the transport conveyor belt at the place location (point C, corresponding to 180°). Once unloading at point C is complete, the robot immediately returns to point A.

The IEC 61499-compliant implementation of this use case will be constructed and presented step by step in the following section.

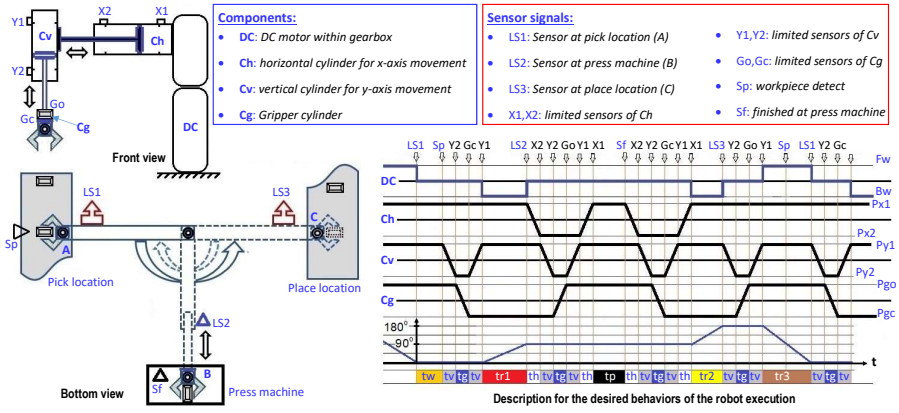


Figure B.1: Example of the PnP robot in a manufacturing line

B.2 Modeling

B.2.1 Plant model

The ***PnP*** manipulator is equipped with a DC motor and a gearbox for rotating the robot. It has three cylinders: *Ch* and *Cv*, which control horizontal and vertical motion, respectively, and *Cg*, which operates the gripper. The stroke limits of cylinders *Ch*, *Cv*, and *Cg* are monitored by three pairs of sensors: *X1* and *X2*, *Y1* and *Y2*, and *Gc* and *Go*, respectively. Fig. B.1 illustrates the mechanical configuration of the PnP manipulator modeled in *4diac*. The IEC 61499 function block model of the plant (*PlantFB*) is shown in Fig. B.2. Additionally, an adapter named *RobotInterface* (*RI*) was created to reduce the number of connections between *PlantFB* and *ControllerFB*. The robot's rotational movement is controlled by supplying voltage values to its DC motor. The label *Fw* denotes clockwise rotation, while the label *Bw* denotes counterclockwise rotation. The movement durations of the robot for the paths from *A* to *B*, *B* to *C*, and *C* to *A* are represented by *tr1*, *tr2*, and *tr3*, respectively. Similarly, the execution times of the cylinders for moving between their stroke limits are denoted by *th*, *tv*, and *tg*. The behavior of the robot's components, including signal occurrences and timing, is illustrated in the lower right corner of Fig. B.1.

B.2.2 Controller model

Fig. B.2 illustrates the controller model in 4*diac*, referred to as *ControllerFB*. The design requirements for this controller specify that it must provide the desired behavior shown on the right side of Fig. B.1. This is crucial, as the correctness of the robot's execution relies entirely on the commands (control values) issued by the controller. The controller's commands depend not only on the signals received from the sensors but also on the execution of other components or subsystems—for example, the availability of a workpiece at the feeding system. The activities within the system are detected and reported by a workpiece sensor denoted as *Sp*, while the sensor *Sf* monitors the waiting time for material processing at the press machine.

Suppose that this production line is composed of several subsystems developed by different teams (e.g., the feeding system, the press system, and the robot control system). In this configuration, the operation of the overall system emerges from the collaboration of activities across these subsystems. Consequently, the controller design must be based on certain assumptions regarding the outcomes and events of the relevant subsystems. Therefore, the correctness of the system operation depends not only on the correctness of the control algorithms but also on the satisfaction of the specified timing constraints for each component, each process, and the system as a whole.

In the contract-based controller design approach, the timing constraints of all components, as well as those of the overall system, must be satisfied. Therefore, instead of connecting sensor signals directly to the controller's request events, sensor and control signals are first monitored and evaluated by timing contract function blocks (*TCFB*s). Only when these signals satisfy the conditions specified in the

*TCFB*s can they trigger the execution of the control algorithms (see Fig. B.2).

B.2.3 Application model

To model the *Application* for the production line, in addition to the robot and controller models described above, we also created function blocks (FBs) to represent the behavior of position sensors, such as limit switches and optical sensors. In this work, we do not aim to cover all components of the system; rather, we focus solely on the PnP manipulator and its controller. The essential events for controller development are the arrival of a workpiece at the conveyor and the completion of material processing at the press machine. These events are assumed to be detected by sensors—for example, the sensor signal *Sp* indicates the arrival of a workpiece, and the sensor signal *Sf* indicates the completion of processing at the press machine.

In the IEC 61499-compliant application modeled in 4*diac*, the E_DELAY FB (named Sf) represents the signal *Sf*, while a newly created composite FB (named $SupplyConv$[1]) models the signal *Sp*. Additionally, the sensor signals at locations *A*, *B*, and *C* are represented by newly developed FBs in 4*diac*, named $LS1$, $LS2$, and $LS3$, respectively.

The component and element models (represented by FBs) are then arranged and connected through their respective interface events and data connections. The resulting FB network describes the application model, capturing the actual behaviors and properties of the manufacturing process, as illustrated in Fig. B.2. A detailed description of the IEC 61499 application model and all FBs used in the system is available at `https://github.com/offis/ETFA2020Demo` .

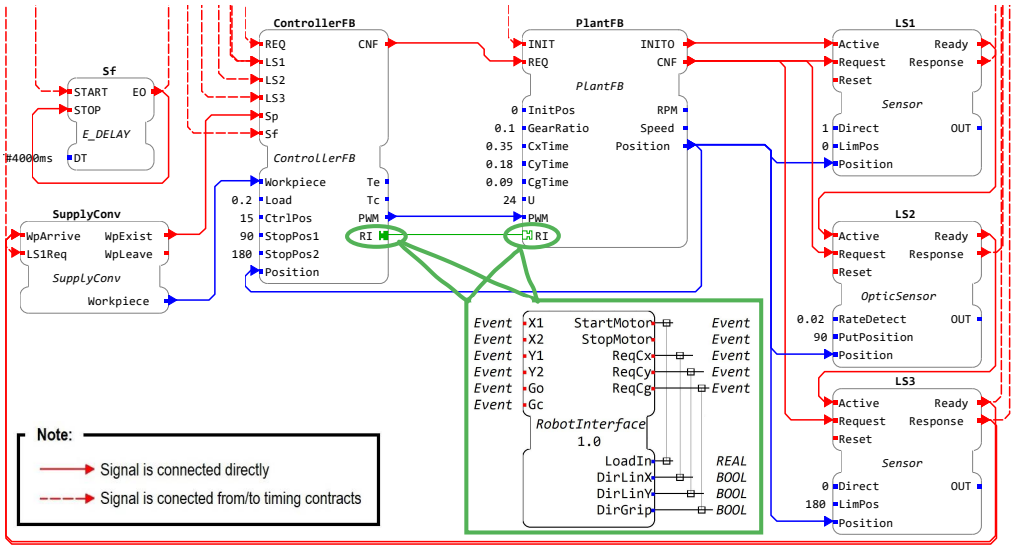


Figure B.2: The application model for a manufacturing line in IEC 61499 (signals ending at the top part are continued in Fig. B.3)

[1]This composite FB is constructed using available FBs from the standard 4*diac* library, including *E_CYCLE*, *E_DELAY*, and *E_RS*.

B.2.4 System analysis

As a reminder, the purpose of the demo in this section is to model the robot control system to ensure the safe operation of the production line. To achieve this goal, it is necessary not only to ensure the correct functional behavior of all components and the overall system but also to guarantee that these components satisfy the specified timing requirements. For instance, the execution time of each process in the system must remain within predefined time intervals. Within the context of a model-based design methodology, the results of the application simulation can be used to characterize timing behaviors, thereby enabling the formal specification of timing constraints for the operations and processes in the system.

The system in this demo is assumed to operate under the parameters and requirements described in the previous section **??**. For instance, the controller is required to reduce the controlled voltage *Uc* to half its value before each stop point at an angle of 15°. The supply voltage *Ui* is 24V with a maximum tolerance of 2%. The workpiece generates a load on the robot arm with a torque of *0.2Nm*, and it is assumed to either be already present or appear within 1s after the robot reaches position *A*. Other parameters are presented in Fig. B.2. The execution or process time intervals of the system components are measured during the simulation and are summarized in Table B.1.

Table B.1: The time interval of the execution or the process of components

Variable	Execution/Process	Time interval
tr1	robot rotation time: $A \rightarrow B$	$[3.1, 3.3]\ s$
tr2	robot rotation time: $B \rightarrow C$	$[3.1, 3.3]\ s$
tr3	robot rotation time: $C \rightarrow A$	$[5.8, 6.0]\ s$
th	working time of cylinder Cx	$[0.3, 0.4]\ s$
tv	working time of cylinder Cy	$[0.16, 0.2]\ s$
tg	working time of gripper Cg	$[0.08, 0.1]\ s$
tp	press machine execution time	$[2.9, 3.1]\ s$
tw	waiting time for a workpiece	$[0, 1]\ s$

B.2.5 Application of timing contracts

The system is developed under scenarios coordinated by different teams. The contract C_S specifies the requirements for the overall system. Additionally, we denote the contract for the supply workpiece system as *SW*, the contract for the press process as *PP*, and the contract for the robot execution as *ER*. In the context of contract-based development, each team is responsible for guaranteeing its respective subsystem contracts: C_{SW}, C_{PP}, and C_{ER}. The overall system must satisfy the following requirement:

$$C_S \vDash C_{SW} \otimes C_{PP} \otimes C_{ER}$$

In addition, the system development requirements may include various aspects such as behavior, timing, and security. Therefore, these requirements must be allocated to the respective development teams as separate contracts for different areas. For this demo, assume that the system requirements consist of a combination of behavioral aspects (specified by contract C_S^B) and timing aspects (specified by contract C_S^T). Consequently, the system design must satisfy the following contract:

$$C_S \vDash C_S^B \wedge C_S^T$$

In particular, the requirements of systems in this example for the timing aspects can be described by Fig. B.3 and formally denoted as:

$$C_S^T \preceq C_{SW}^T \otimes C_{PP}^T \otimes C_{ER}^T$$

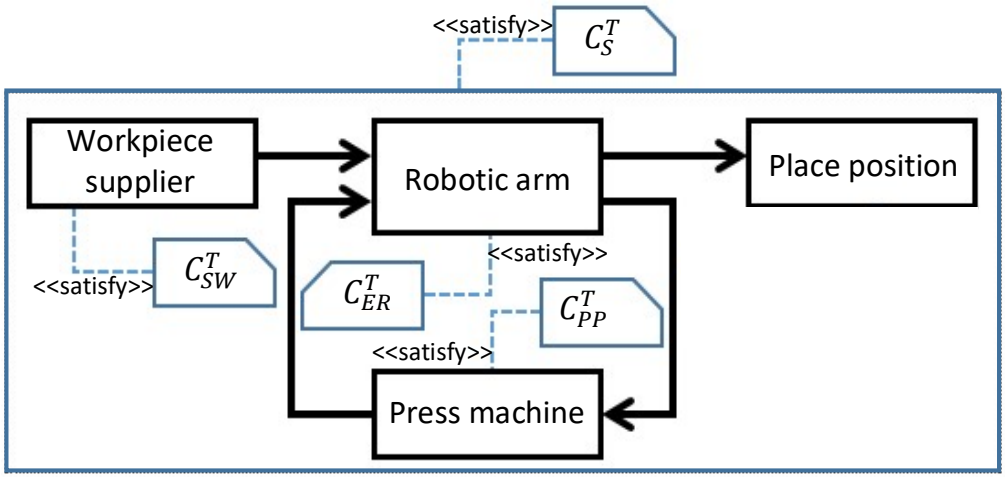


Figure B.3: Application of timing contracts to the ControllerFB subsystem

B.2.6 Formalization of timing phenomena and effects

As shown in the diagram of the system's operating components, the execution or process of each component can be constrained by a specified time interval. For instance, the rotational movement of the robot is limited by different durations *tr1*, *tr2*, and *tr3* for the movements $A \mapsto B$, $B \mapsto C$, and $C \mapsto A$, respectively. It is also evident that the robot's total operation time in a working cycle depends not only on the execution times of its individual elements but also on the waiting times for other processes, such as material supply and material processing. The timing behavior of components can be observed and extracted through the event ports of the FBs during the simulation of the application in 4*diac*. The recorded results of these processes can then be used to reason about the timing constraints of the component models or processes. Specifically, the execution time of a component or process can be calculated by measuring the time difference between the occurrence

of events. The reaction or response time of an FB (representing a component or process) is typically defined as the interval between the occurrence of the request event (REQ) and the confirmation/response event (CNF/RSP).

In this work, timing constraints can also be specified for repetitive signal occurrences. For example, the arrival of a workpiece at the pick location can be expressed as: *The workpiece (represented by the signal **Sp**) occurs within [10,20]s.* Additionally, causal relationships between two signals of a component or process can be expressed. This relates to the question: if a component is triggered by a request event at a specific time, how long afterward can a response or confirmation event occur? In this thesis, causal relationships between events are specified using forward and backward delay methods. For example, to constrain the time for a complete robot execution cycle, one can use either a forward delay specification, *"whenever the signal **LS1** occurs, then **LS3** will occur within [11,13]s"*, or a backward delay specification, *"whenever the signal **LS3** occurs, then **LS1** has occurred within [11,13]s"*. The rationale and usage of forward and backward delay specifications are introduced in Section 4.3.2.

B.2.7 Contract specification and consistency check

In this section, we define the contracts for the system and each component, as depicted in Fig. B.3. The timing constraints for the execution time of the system and each component can be determined by the occurrence of sensor signals in the best-case and worst-case scenarios for different periods, as follows:

- Overall system contract: The contract for the entire system is defined by the cycle time of the system operation, which can be specified as: $T_{LS1\rightarrow LS1} = tr1 + tr2 + tr3 + 6 \times tv + 4 \times th + 3 \times tg + tp + tw \Leftrightarrow T_{Sp\rightarrow LS3} \in [17.7, 20.3]\, s$. Furthermore, the system contract C_S^T can be observed by the repetitive occurrence of a signal *LS1* or *LS3*, and expressed by: "*signal LS1 or LS3 occurs within [17.7,20.3]s.* Because the robot operation is present in the entire cycle, since the beginning until the end of a system operation cycle, then we can apply contract for robot execution C_{ER}^T by the system contract C_S^T."

- The execution time of the robot when it moves from position A to position B is measured via the span from the occurrence of sensor signal *LS1* until the appearance of sensor signal *LS2*. It can be calculated as: $T_{LS1\rightarrow LS2} = tr1 + 2 \times tv + tg + tw \Leftrightarrow T_{LS1\rightarrow LS2} \in [3.5, 4.8]\, s$. This process can be formalized by the contract $C_{LS1\rightarrow LS2}^T$ as: *Whenever signal LS1 occurs (**A**), then signal LS2 occurs within the interval [3.5,3.8]s* (**G**).

- The execution time of the robot when it moves from position B to position C is measured in the span between the occurrence of sensor signals *Sf* and *LS3*, which can be determined as $T_{Sf\rightarrow LS3} = tr2 + 2 \times th + 2 \times tv + tg \Leftrightarrow T_{Sf\rightarrow LS3} \in [4.1, 4.6]\, s$. The timing contract $C_{Sf\rightarrow LS3}^T$ therefore formally specified as follows: *Whenever Sf signal occurs (**A**), then LS3 signal will occur within the interval [4.1,4.6]s* (**G**).

- The execution time of the robot when it moves from position C to position A is measured from the appearance of signal *LS3* until the appearance of signal *LS1*, which is calculated by: $T_{LS3\to LS1} = tr3 + 2 \times tv + tg \Leftrightarrow T_{LS3\to LS1} \in [6.2, 6.5]\, s$. Its timing contract $C^T_{LS3\to LS1}$ is given by: *Whenever signal LS3 occurs (**A**), then signal LS1 will be occurred within the interval [6.2,6.5]s* (**G**).

- The execution time of the press machine is measured by the span between the appearance of signal *LS2* and signal *Sf*, which is calculated by: $T_{LS2\to Sf} = tp + 2 \times th + 2 \times tv + tg \Leftrightarrow T_{LS2\to Sf} \in [3.9, 4.4]\, s$. In this case, we cannot use the *Reaction* constraint because the signal *LS2* will occur in the rotational movement path $A \mapsto B$ and $C \mapsto A$. This execution at the press machine C^T_{PP} can only constrained by the backward specification defined by *Age* constraint $C^T_{LS2\to Sf}$ as *Whenever signal Sf occurs (**A**), then signal LS2 has occurred within the interval [3.9,4.4]s* (**G**).

- The feeding system must meet the requirement that a workpiece needs to exist before the robot arrives. A workpiece can arrive at the pick location within 1s after the robot reaches point A. The contract for this feeding system C^T_{WP} can be defined as *Whenever signal LS1 occurs (**A**), then Sp will occur within the interval [0,1]s* (**G**).

As mentioned, the contract for the entire system is applied for the robot execution. This means: $C^T_{ER} \subseteq C^T_S$. Moreover, the contract for the robot execution is defined by: $C^T_{ER} = C^T_{LS1\to LS2} \cap C^T_{LS2\to Sf} \cap C^T_{Sf\to LS3} \cap C^T_{LS3\to LS1}$. On the other hand, we can recognize that: $C^T_{WP} \subseteq C^T_{LS1\to LS2}$ and $C^T_{PP} \subseteq C^T_{LS2\to Sf}$. The timeout values for all timing contract FBs are set at the value of 20.3s, to ensure the system operation will be stopped when any component execution time exceeds the maximum timing constraint for a normal operating cycle of the system.

To evaluate the assurance of safety using the timing contracts, we built three scenarios for the operation of the production line system as follows:

- Scenario 1: An obstacle blocks the robot's movement along its path.

- Scenario 2: The supply system stops feeding or its execution rate is too low.

- Scenario 3: The press machine stops working or its execution rate is too fast or too slow.

In the first scenario, a timing contract—$C^T_{LS1\to LS2}$, $C^T_{Sf\to LS3}$, or $C^T_{LS3\to LS1}$—will be violated corresponding to the movement paths $A \mapsto B$, $B \mapsto C$, or $C \mapsto A$, respectively. In the second scenario, the violation is detected by the contract C^T_{SW} or via its timeout constraint. In the third scenario, if the press machine has stopped, the timeout will trigger the *Violated* event of its TCFB. Otherwise, the execution is monitored by the contract $C^T_{LS2\to Sf}$.

Further tests were conducted with various cases of hidden contract violations. In all these hypothetical cases, the system successfully detected the error conditions through the monitoring mechanism.

Bibliography

[1] International Electrotechnical Commission, "Programmable controllers-part 3: Programming languages," *IEC 61131-3*, 1993.

[2] A. Zoitl and V. Vyatkin, "Iec 61499 architecture for distributed automation: The 'glass half full'view," *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 7–23, 2009.

[3] T. Strasser, A. Zoitl, J. H. Christensen, and C. Sünder, "Design and execution issues in iec 61499 distributed automation and control systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 41, no. 1, pp. 41–51, 2010.

[4] F. Cabadini, G. Montalbano, G. Kollegger, H. Mayer, V. Vytakin, J. Soldatos, O. Lazaro, and F. Cavadini, "Iec-61499 distributed automation for the next generation of manufacturing systems," in *The Digital Shopfloor: Industrial Automation in the Industry 4.0 Era–Performance Analysis and Applications*, pp. 103–127, River, 2019.

[5] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: A cyber-physical systems approach*. MIT press, 2016.

[6] B. Bordel, R. Alcarria, T. Robles, and D. Martín, "Cyber–physical systems: Extending pervasive sensing from control theory to the internet of things," *Pervasive and mobile computing*, vol. 40, pp. 156–184, 2017.

[7] Wikipedia contributors, "Cyber-physical system — Wikipedia, the free encyclopedia," 2023. [Online; accessed 19-July-2023].

[8] International Electrotechnical Commission, "International standard iec61499, function blocks, part 1-4," *International Electrotechnical Commission, Geneva, Switzerland, Tech. Rep. IEC*, 2005.

[9] V. Vyatkin, "Iec 61499 as enabler of distributed and intelligent automation: State-of-the-art review," *IEEE transactions on Industrial Informatics*, vol. 7, no. 4, pp. 768–781, 2011.

[10] M. Bozzano, A. Cimatti, C. Mattarei, and S. Tonetta, "Formal safety assessment via contract-based design," in *International Symposium on Automated Technology for Verification and Analysis*, pp. 81–97, Springer, 2014.

[11] P. Nuzzo, *Compositional design of cyber-physical systems using contracts*. PhD thesis, Ph. D. dissertation, EECS Department, University of California, Berkeley, 2015.

[12] Wikipedia contributors, "Industrial control system — Wikipedia, the free encyclopedia," 2022. [Online; accessed 19-July-2023].

[13] K. Stouffer, J. Falco, K. Scarfone, *et al.*, "Guide to industrial control systems (ics) security," *NIST special publication*, vol. 800, no. 82, pp. 16–16, 2011.

[14] J. Krüger, L. Wang, A. Verl, T. Bauernhansl, E. Carpanzano, S. Makris, J. Fleischer, G. Reinhart, J. Franke, and S. Pellegrinelli, "Innovative control of assembly systems and lines," *CIRP annals*, vol. 66, no. 2, pp. 707–730, 2017.

[15] R. Lewis, *Modelling control systems using IEC 61499: Applying function blocks to distributed systems*. No. 59, Iet, 2001.

[16] G. Black and V. Vyatkin, "Intelligent component-based automation of baggage handling systems with iec 61499," *IEEE Transactions on Automation Science and Engineering*, vol. 7, no. 2, pp. 337–351, 2009.

[17] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-based design for cyber-physical systems," *European journal of control*, vol. 18, no. 3, pp. 217–238, 2012.

[18] L. F. Osborne, J. Brummond, R. Hart, M. Zarean, S. M. Conger, *et al.*, "Clarus: Concept of operations," tech. rep., United States. Federal Highway Administration, 2005.

[19] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, pp. 40–51, Oct. 1992.

[20] M. Grabowski, B. Kaiser, and Y. Bai, "Systematic refinement of cps requirements using sysml, template language and contracts," *Modellierung 2018*, 2018.

[21] 4DIAC, "Framework for industrial automation and control." `https://www.eclipse.org/4diac/`.

[22] V. Vyatkin, H.-M. Hanisch, P. Starke, *et al.*, "Formalisms for verification of discrete control applications on example of iec 61499 function blocks," 2000.

[23] V. Vyatkin and I. S. of America, "Iec 61499 function blocks for embedded and distributed control systems design," 2007.

[24] J. H. Christensen, T. Strasser, A. Valentini, V. Vyatkin, A. Zoitl, J. Chouinard, H. Mayer, and A. Kopitar, "The iec 61499 function block standard: Software tools and runtime platforms," *ISA Automation Week*, vol. 2012, 2012.

[25] A. Zoitl and R. Lewis, *Modelling control systems using IEC 61499*, vol. 95. IET, 2014.

[26] Holobloc, "Function block development kit(fbdk)." `https://www.holobloc.com/fbdk8/index.htm`, 2009.

[27] J. Fischer and T. O. Boucher, "Workbook for designing distributed control applications using rockwell automation's holobloc prototyping software,"

[28] L. I. Pinto, C. D. Vasconcellos, R. S. U. Rosso, and G. H. Negri, "Icaru-fb: An iec 61499 compliant multiplatform software infrastructure," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 3, pp. 1074–1083, 2016.

[29] T. Strasser, M. Rooker, G. Ebenhofer, A. Zoitl, C. Sunder, A. Valentini, and A. Martel, "Framework for distributed industrial automation and control (4diac)," in *2008 6th IEEE International Conference on Industrial Informatics*, pp. 283–288, IEEE, 2008.

[30] A. Zoitl, T. Strasser, and G. Ebenhofer, "Developing modular reusable iec 61499 control applications with 4diac," in *Industrial Informatics (INDIN), 2013 11th IEEE International Conference on*, pp. 358–363, IEEE, 2013.

[31] ISaGRAF, "Isagraf development toolkit." `https://www.rockwellautomation.com/en-us/support/documentation/technical-data/isagraf_20190326-0743.html#`.

[32] T. Hussain and G. Frey, "Developing iec 61499 compliant distributed systems with network enabled controllers," in *IEEE Conference on Robotics, Automation and Mechatronics, 2004.*, vol. 1, pp. 507–512, IEEE, 2004.

[33] NxtControl, "Nxtstudio." `https://www.nxtcontrol.com`.

[34] V. Vyatkin and J. Chouinard, "On comparisons of the isagraf implementation of iec 61499 with fbdk and other implementations," in *2008 6th IEEE International Conference on Industrial Informatics*, pp. 289–294, IEEE, 2008.

[35] S. Preuße, D. Missal, C. Gerber, M. Hirsch, and H.-M. Hanisch, "On the use of model-based iec 61499 controller design," *Int. J. of Discrete Event Control Systems (IJDESC)*, 2010.

[36] L. Prenzel, A. Zoitl, and J. Provost, "Iec 61499 runtime environments: A state of the art comparison," in *17th International Conference on Computer Aided Systems Theory (EUROCAST 2019)*, 2019.

[37] K. C. Thramboulidis, "Using uml in control and automation: a model driven approach," in *2nd IEEE International Conference on Industrial Informatics, 2004. INDIN'04. 2004*, pp. 587–593, IEEE, 2004.

[38] A. Malik, P. S. Roop, N. Allen, and T. Steger, "Emulation of cyber-physical systems using iec-61499," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 1, pp. 380–389, 2017.

[39] Online:, "Iec 61499: The new standard in automation." `https://iec61499.de/index.htm`.

[40] C.-H. Yang, V. Vyatkin, and C. Pang, "Model-driven development of control software for distributed automation: a survey and an approach," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, no. 3, pp. 292–305, 2014.

[41] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni Vincentelli, W. Damm, T. Henzinger, and K. Larsen, "Contracts for systems design: Theory," Research Report N°. 8759, Project-Teams Hycomes, July 2015.

[42] Y. Glouche, P. Le Guernic, J.-P. Talpin, and T. Gautier, "A boolean algebra of contracts for assume-guarantee reasoning," *Electronic Notes in Theoretical Computer Science*, vol. 263, pp. 111–127, 2010.

[43] S. Preuße and H.-M. Hanisch, "Specification and verification of technical plant behavior with symbolic timing diagrams," in *2008 3rd International Design and Test Workshop*, pp. 313–318, IEEE, 2008.

[44] S. Preuße and H.-M. Hanisch, "Specification of technical plant behavior with a safety-oriented technical language," in *2009 7th IEEE International Conference on Industrial Informatics*, pp. 632–637, IEEE, 2009.

[45] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand, "Using contract-based component specifications for virtual integration testing and architecture design," in *2011 Design, Automation & Test in Europe*, pp. 1–6, IEEE, 2011.

[46] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57, ieee, 1977.

[47] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-time systems*, vol. 2, no. 4, pp. 255–299, 1990.

[48] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pp. 152–166, Springer, 2004.

[49] P. Battram, B. Kaiser, and R. Weber, "A modular safety assurance method considering multi-aspect contracts during cyber physical system design.," in *REFSQ Workshops*, pp. 185–197, 2015.

[50] A. Cimatti, M. Roveri, and S. Tonetta, "Hreltl: A temporal logic for hybrid systems," *Information and Computation*, vol. 245, pp. 54–71, 2015.

[51] A. Cimatti, M. Roveri, and S. Tonetta, "Requirements validation for hybrid systems," in *International Conference on Computer Aided Verification*, pp. 188–203, Springer, 2009.

[52] A. Cimatti, M. Roveri, A. Susi, and S. Tonetta, "Formalizing requirements with object models and temporal constraints," *Software & Systems Modeling*, vol. 10, no. 2, pp. 147–160, 2011.

[53] A. Cimatti, M. Roveri, A. Susi, and S. Tonetta, "Validation of requirements for hybrid systems: A formal approach," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 4, pp. 1–34, 2013.

[54] OCRA, "Othello contracts refinement analysis(ocra)." `https://ocra.fbk.eu/`.

[55] NuSMV, "Nusmv: a new symbolic model checker." `https://nusmv.fbk.eu/`.

[56] CHASE, "Contract-based heterogeneous analysis and system exploration(chase)." `https://chase-cps.github.io/chase/doc/md/tools.html`.

[57] P. Nuzzo, M. Lora, Y. A. Feldman, and A. L. Sangiovanni-Vincentelli, "Chase: Contract-based requirement engineering for cyber-physical system design," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 839–844, IEEE, 2018.

[58] S. Ghosh, D. Sadigh, P. Nuzzo, V. Raman, A. Donzé, A. L. Sangiovanni-Vincentelli, S. S. Sastry, and S. A. Seshia, "Diagnosis and repair for synthesis from signal temporal logic specifications," in *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, pp. 31–40, 2016.

[59] J. Li, P. Nuzzo, A. Sangiovanni-Vincentelli, Y. Xi, and D. Li, "Stochastic contracts for cyber-physical system design under probabilistic requirements," in *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pp. 5–14, 2017.

[60] W. Damm, G. Ehmen, K. Grüttner, P. Ittershagen, B. Koopmann, F. Poppen, and I. Stierand, "Multi-layer time coherency in the development of adas/ad systems: Design approach and tooling," in *Proceedings of the Workshop on Design Automation for CPS and IoT*, pp. 20–30, 2019.

[61] E. Böde, W. Damm, G. Ehmen, M. Fränzle, K. Grüttner, P. Ittershagen, B. Josko, B. Koopmann, F. Poppen, M. Siegel, *et al.*, *MULTIC-tooling*. FAT Forschungsvereinigung Automobiltechnik, 2019.

[62] L. Wang, M. Törngren, and M. Onori, "Current status and advancement of cyber-physical systems in manufacturing," *Journal of manufacturing systems*, vol. 37, pp. 517–527, 2015.

[63] A. W. Colombo, S. Karnouskos, O. Kaynak, Y. Shi, and S. Yin, "Industrial cyberphysical systems: A backbone of the fourth industrial revolution," *IEEE Industrial Electronics Magazine*, vol. 11, no. 1, pp. 6–16, 2017.

[64] R. Baheti and H. Gill, "Cyber-physical systems," *The impact of control technology*, vol. 12, no. 1, pp. 161–166, 2011.

[65] J. C. Jensen, D. H. Chang, and E. A. Lee, "A model-based design methodology for cyber-physical systems," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*, pp. 1666–1671, IEEE, 2011.

[66] P. Derler, E. A. Lee, and A. L. Sangiovanni-Vincentelli, "Addressing modeling challenges in cyber-physical systems," tech. rep., CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE, 2011.

[67] I. Quadri, A. Bagnato, E. Brosse, and A. Sadovykh, "Modeling methodologies for cyber-physical systems: research field study on inherent and future challenges," *Ada User Journal*, vol. 36, no. 4, pp. 246–253, 2015.

[68] P. J. Mosterman and J. Zander, "Cyber-physical systems challenges: a needs analysis for collaborating embedded software systems," *Software & systems modeling*, vol. 15, no. 1, pp. 5–16, 2016.

[69] J. Fitzgerald, C. Gamble, P. G. Larsen, K. Pierce, and J. Woodcock, "Cyber-physical systems design: formal foundations, methods and integrated tool chains," in *2015 IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering*, pp. 40–46, IEEE, 2015.

[70] J. Fitzgerald, C. Gamble, R. Payne, P. G. Larsen, S. Basagiannis, and A. E.-D. Mady, "Collaborative model-based systems engineering for cyber-physical systems, with a building automation case study," in *INCOSE International Symposium*, vol. 26, pp. 817–832, Wiley Online Library, 2016.

[71] S. A. Seshia, S. Hu, W. Li, and Q. Zhu, "Design automation of cyber-physical systems: Challenges, advances, and opportunities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 9, pp. 1421–1434, 2016.

[72] M. Neghina, C.-B. Zamfirescu, and K. Pierce, "Early-stage analysis of cyber-physical production systems through collaborative modelling," *Software and Systems Modeling*, vol. 19, no. 3, pp. 581–600, 2020.

[73] R. Al Ali, M. Amrani, A. Barisic, F. Barros, D. Blouin, H. Giese, M. Goulao, M. Iacono, N. Eva, H. Vangheluwe, *et al.*, *State-of-the-art on Current Formalisms used in Cyber-Physical Systems Development*. PhD thesis, COST European Cooperation in Science and Technology, 2017.

[74] M. Amrani, D. Blouin, R. Heinrich, A. Rensink, H. Vangheluwe, and A. Wortmann, "Multi-paradigm modelling for cyber–physical systems: a descriptive framework," *Software and Systems Modeling*, vol. 20, no. 3, pp. 611–639, 2021.

[75] A. Barišić, I. Ruchkin, D. Savić, M. A. Mohamed, R. Al-Ali, L. W. Li, H. Mkaouar, R. Eslampanah, M. Challenger, D. Blouin, *et al.*, "Multi-paradigm modeling for cyber–physical systems: A systematic mapping review," *Journal of Systems and Software*, vol. 183, p. 111081, 2022.

[76] 20-Sim, "Software for modeling complex physics." `https://www.20sim.com/`.

[77] P. Derler, E. A. Lee, and A. S. Vincentelli, "Modeling cyber-physical systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 13–28, 2011.

[78] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet, "Hybrid systems modeling challenges caused by cyber-physical systems," *Cyber-Physical Systems (CPS) Foundations and Challenges. Available on-line: http://people. rennes. inria. fr/Albert. Benveniste/pub/NIST2012. pdf*, 2013.

[79] R. G. Sanfelice *et al.*, "Analysis and design of cyber-physical systems. a hybrid control systems approach," in *Cyber-physical systems: From theory to practice*, pp. 1–29, CRC Press Boca Raton, FL, USA, 2016.

[80] A. Tolk, E. H. Page, and S. Mittal, "Hybrid simulation for cyber physical systems: state of the art and a literature review.," *SpringSim (ANSS)*, pp. 10–1, 2018.

[81] R. Alur, C. A. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theor. Comput. Sci.*, vol. 138, pp. 3–34, 1995.

[82] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, pp. 183–235, 1994.

[83] Ptolemy II. `https://ptolemy.berkeley.edu/ptolemyII/index.htm`.

[84] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "Spaceex: Scalable verification of hybrid systems," in *CAV*, 2011.

[85] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, pp. 134–152, 1997.

[86] J. M. Molina, M. Damm, J. Haase, E. Holleis, and C. Grimm, "Model based design of distributed embedded cyber physical systems," in *Models, methods, and tools for complex chip design*, pp. 127–143, Springer, 2014.

[87] O. S. Initiative *et al.*, "Systemc 2.1 language reference manual," *http://www.systemc.org*, 2003.

[88] D. Henriksson and H. Elmqvist, "Cyber-physical systems modeling and simulation with modelica," in *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*, no. 063, pp. 502–509, Linköping University Electronic Press, 2011.

[89] A. T. Al-Hammouri, "A comprehensive co-simulation platform for cyber-physical systems," *Computer Communications*, vol. 36, no. 1, pp. 8–19, 2012.

[90] P. G. Larsen, J. Fitzgerald, J. Woodcock, and T. Lecomte, "Collaborative modeling and simulation for cyber-physical systems," in *Trustworthy Cyber-Physical Systems Engineering*, pp. 203–224, Chapman and Hall/CRC, 2016.

[91] INTO-CPS, "Integrated tool chain for model-based design of cyber-physical systems(into-cps)." `https://projects.au.dk/into-cps/`.

[92] P. G. Larsen, J. S. Fitzgerald, J. Woodcock, P. A. Fritzson, J. Brauer, C. R. Kleijn, T. Lecomte, M. Pfeil, O. Green, S. Basagiannis, and A. Sadovykh, "Integrated tool chain for model-based design of cyber-physical systems: The into-cps project," *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, pp. 1–6, 2016.

[93] S. Bogomolov, M. Greitschus, P. G. Jensen, K. G. Larsen, M. Mikučionis, T. Strump, and S. Tripakis, "Co-simulation of hybrid systems with spaceex and uppaal," in *11th International Modelica Conference*, pp. 159–169, Linköping University Electronic Press, 2015.

[94] A. Suzuki, K. Masutomi, I. Ono, H. Ishii, and T. Onoda, "Cps-sim: Co-simulation for cyber-physical systems with accurate time synchronization," *IFAC-PapersOnLine*, vol. 51, no. 23, pp. 70–75, 2018.

[95] MathWorks, "Matlab/simulink." `www.mathworks.com`.

[96] QualNet, "Scalable network technologies." `web.scalable-networks.com/qualnet-network-simulator`.

[97] OMNeT++, "Discrete event simulator." `www.omnetpp.org`.

[98] T. Nägele and J. Hooman, "Co-simulation of cyber-physical systems using hla," in *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 1–6, IEEE, 2017.

[99] T. Nägele, T. Broenink, J. Hooman, and J. Broenink, "Early analysis of cyber-physical systems using co-simulation and multi-level modelling," in *2019 IEEE International Conference on Industrial Cyber Physical Systems (ICPS)*, pp. 133–138, IEEE, 2019.

[100] O. Florescu, J. Voeten, B. Theelen, M. Geilen, and H. Corporaal, "Parallel object-oriented specification language," in *ARTIST Survey of Programming Languages*, ARTIST2, Network of Excellence on Embedded System Design, 2008.

[101] H. A. Pearce, M. M. Kuo, N. Allen, P. S. Roop, and A. Malik, "Simulation of cyber-physical systems using iec61499," in *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pp. 136–145, 2017.

[102] H. Pearce, *Model-driven Engineering for Safety and Security in Industry 4.0*. PhD thesis, University of Auckland, 2020.

[103] I. Hegny, M. Wenger, and A. Zoitl, "IEC 61499 based simulation framework for model-driven production systems development," in *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pp. 1–8, IEEE, 2010.

[104] A. Parant, D. Zander, F. Gellot, and A. Philippot, "Iec 61499 control architectures evaluation for automation software development," *IFAC-PapersOnLine*, vol. 56, no. 2, pp. 3660–3665, 2023.

[105] D. Ivanova, G. Frey, and I. Batchkova, "Intelligent component based batch control using iec61499 and ansi/isa s88," in *2008 4th International IEEE Conference Intelligent Systems*, vol. 1, pp. 4–44, IEEE, 2008.

[106] M. V. García, E. Irisarri, F. Pérez, E. Estévez, and M. Marcos, "An open CPPS automation architecture based on IEC-61499 over OPC-UA for flexible manufacturing in oil&gas industry," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 1231–1238, 2017.

[107] A. A. Santos and A. F. da Silva, "Simulation and control of a cyber-physical system under iec 61499 standard," *Procedia Manufacturing*, vol. 55, pp. 72–79, 2021.

[108] A. Zoitl and H. Prähofer, "Guidelines and patterns for building hierarchical automation solutions in the IEC 61499 modeling language," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 4, pp. 2387–2396, 2013.

[109] A. Zoitl and H. Prähofer, "Building hierarchical automation solutions in the IEC 61499 modeling language," in *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pp. 557–564, IEEE, 2011.

[110] V. Dubinin, V. Vyatkin, and T. Pfeiffer, "Engineering of validatable automation systems based on an extension of uml combined with function blocks of iec 61499," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pp. 3996–4001, IEEE, 2005.

[111] S. Panjaitan and G. Frey, "Functional design for iec 61499 distributed control systems using uml activity diagrams," in *International Conference Instrumentation, Communication and Information Technology (ICICI)*, pp. 64–70, 2005.

[112] I. Antonova and I. Batchkova, "Development of multi-agent control systems using uml/sysml," in *2008 4th International IEEE Conference Intelligent Systems*, vol. 1, pp. 6–26, IEEE, 2008.

[113] A. Parant, F. Gellot, P. Alexandre, and V. Carré-Ménétrier, "Model-based engineering for designing cyber-physical systems control architecture and improving adaptability from requirements," in *11th Workshop on Service Oriented, Holonic and Multi-Agent Manufacturing Systems for Industry of the Future (SOHOMA 2021)*, 2021.

[114] A. Parant, F. Gellot, A. Philippot, and V. Carré-Ménétrier, "Hierarchical intelligent component-based development for the design of cyber-physical control architecture," in *2021 5th International Conference on Control and Fault-Tolerant Systems (SysTol)*, pp. 348–353, IEEE, 2021.

[115] OMG-UML, "Unified modeling language." `https://www.omg.org/spec/UML`.

[116] OMG-SysML, "The omg systems modeling language." `https://omgsysml.org/`.

[117] E. A. Lee, "Cyber physical systems: Design challenges," in *11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pp. 363–369, IEEE, 2008.

[118] L. Ferrarini and C. Veber, "Implementation approaches for the execution model of iec 61499 applications," in *2nd IEEE International Conference on Industrial Informatics, 2004. INDIN'04. 2004*, pp. 612–617, IEEE, 2004.

[119] A. Zoitl, *Real-time Execution for IEC 61499*. ISA, 2008.

[120] C. Sunder, H. Rofner, V. Vyatkin, and B. Favre-Bulle, "Formal description of an iec 61499 runtime environment with real-time constraints," in *2007 5th IEEE International Conference on Industrial Informatics*, vol. 2, pp. 853–859, IEEE, 2007.

[121] L. Prenzel, S. Hofmann, and S. Steinhorst, "Real-time dynamic reconfiguration for iec 61499," in *2022 IEEE 5th International Conference on Industrial Cyber-Physical Systems (ICPS)*, pp. 1–6, IEEE, 2022.

[122] P. Nuzzo, A. L. Sangiovanni-Vincentelli, D. Bresolin, L. Geretti, and T. Villa, "A platform-based design methodology with contracts and related tools for the design of cyber-physical systems," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2104–2132, 2015.

[123] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho, "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems," in *Hybrid systems*, pp. 209–229, Springer, 1992.

[124] L. Benvenuti, A. Ferrari, L. Mangeruca, E. Mazzi, R. Passerone, and C. Sofronis, "A contract-based formalism for the specification of heterogeneous systems," in *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pp. 142–147, IEEE, 2008.

[125] L. Benvenuti, A. Ferrari, E. Mazzi, and A. Vincentelli, "Contract-based design for computation and verification of a closed-loop hybrid system," in *International Workshop on Hybrid Systems: Computation and Control*, pp. 58–71, Springer, 2008.

[126] A. Balluchi, A. Casagrande, P. Collins, A. Ferrari, T. Villa, and A. L. Sangiovanni-Vincentelli, "Ariadne: a framework for reachability analysis of hybrid automata," in *In: Proceedings of the International Syposium on Mathematical Theory of Networks and Systems.(2006*, Citeseer, 2006.

[127] M. Toerngren, S. Tripakis, P. Derler, and E. A. Lee, "Design contracts for cyber-physical systems: making timing assumptions explicit," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-191*, 2012.

[128] E. S. Kim, S. Sadraddini, C. Belta, M. Arcak, and S. A. Seshia, "Dynamic contracts for distributed temporal logic control of traffic networks," in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pp. 3640–3645, IEEE, 2017.

[129] A. Cimatti and S. Tonetta, "A temporal logics approach to contract-based design," in *2016 Architecture-Centric Virtual Integration (ACVI)*, pp. 1–3, IEEE, 2016.

[130] P. Nuzzo, J. B. Finn, A. Iannopollo, and A. L. Sangiovanni-Vincentelli, "Contract-based design of control protocols for safety-critical cyber-physical systems," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pp. 1–4, IEEE, 2014.

[131] H. Prähofer and A. Zoitl, "Verification of hierarchical IEC 61499 component systems with behavioral event contracts," in *Industrial Informatics (INDIN), 2013 11th IEEE International Conference on*, pp. 578–585, IEEE, 2013.

[132] P. Lindgren, M. Lindner, D. Pereira, and L. M. Pinho, "A formal perspective on IEC 61499 execution control chart semantics," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 3, pp. 293–300, IEEE, 2015.

[133] D. D. Tran, J. Walter, K. Grüttner, and F. Oppenheimer, "Towards contract based assertions in iec 61499 applications," in *Forum on specification & Design Languages 2018 (FDL 2018)*, 2018.

[134] D. D. Tran, J. Walter, K. Grüttner, and F. Oppenheimer, "Towards time-sensitive behavioral contract monitors for iec 61499 function blocks," in *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*, vol. 1, pp. 27–34, IEEE, 2020.

[135] D. D. Tran, K. Grüttner, F. Oppenheimer, and W. Nebel, "Timing contracts and monitors for safety relevant controller design in iec 61499," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, pp. 156–163, IEEE, 2020.

[136] D. Kempf, "Implementation of hardware monitors based on time-sensitive behavioral contracts in fpgas," Master's thesis, Carl von Ossietzky University of Oldenburg, 2022.