TUM School of Computation, Information and Technology
Technische Universität München

Dynamic Adaptation of Industrial Control Systems: A Roadmap to Resilience

Laurin Alban Prenzel

Vollständiger Abdruck der von der School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz:

Prof. Dr. Ing. Robert Wille

Prüfende der Dissertation:

1. Prof. Dr. Sebastian Steinhorst
2. Prof. Dr. Alois Zoitl

Die Dissertation wurde am 12.02.2024 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 05.09.2024 angenommen.

Acknowledgements

I express my deepest gratitude to Prof. Dr. Sebastian Steinhorst for his unwavering support and mentorship. Prof. Steinhorst provided me with the opportunity to continue my work when there was a high chance of failure and allowed me to resiliently recover. He fostered an excellent work environment and I could always count on supportive and sympathetic feedback whenever doubts arose. I am equally thankful to Prof. Dr. Alois Zoitl, whose extraordinary support has been a cornerstone of my journey since during my master's degree in 2017. His continuous encouragement and exceptional mentorship have been essential in this endeavor. My sincere thanks also go to Prof. Dr. Julien Provost for courageously enabling me to commence my research journey. His invaluable guidance and support set me on a path toward success. I also extend my gratitude to Prof. Dr. Robert Wille for leading the committee of my dissertation.

This journey would have been a solitary one without the support of my colleagues. At the Associate Professorship of Embedded Systems and Internet of Things, my gratitude goes to Andreas, Ege, Emanuel, Fady, Jan, Jens, Julian, Mohammad, Nitin, Philipp, Rubi, Selman, and Waltraud for their collaborative spirit and enriching discussions. Similarly, at the Assistant Professorship for Safe Embedded Systems, my thanks go to Canlong, Claudius, and Nancy for their invaluable insights and support. A special mention to Bianca from JKU Linz for her collaborative efforts in our joint work.

Finally, I would like to extend my heartfelt thanks to my friends and family for their endless encouragement and support. To my parents, for their unwavering belief in me; to my siblings, for being inspiring role models in their unique ways; and to Ladan, for her immeasurable patience and loving care throughout this journey.

Abstract

Industrial automation systems have been subject to increasing uncertainty over the last years with frequent and far-reaching disruptions to global supply chains. Production systems of the future must show higher agility, adaptability, and autonomy to react quickly to anticipated and unanticipated events. Although agile and adaptable production systems have been discussed for decades, current systems are far from autonomous. In contrast to information technology (IT), operational technology (OT) has not progressed as much in autonomy and resilience. Observing the growing complexity in industrial control systems (ICS), mounting labor costs, and shortage of skilled labor, why are ICS behind in automation and autonomy?

This dissertation attempts to narrow this gap in adaptability and agility between IT and OT. To this end, a resilient self-adaptive architecture is proposed: The WATERBEAR architecture, which merges self-adaptive concepts with domain-specific views from ICS. The contributions of this dissertation appear in three stages. First, the adaptable foundation is investigated by implementing a reconfigurable runtime environment (RTE) for industrial control software based on the IEC 61499 standard and comparing this middleware to existing developments. The results indicate that IT developments are highly relevant to the OT domain and that the key characteristics of dynamic adaptation are simplicity and correctness. Second, it became clear that to simplify the procedure, it must be automated and correct by design. Paramount is the consideration of consistency requirements during the adaptation which are analyzed in this work. The consequence is a need for clear execution semantics and a way to express the system behavior in a machine-readable format to guarantee the consistency of these behaviors. Third, the timeliness of the dynamic adaptation is reviewed, which exposed a gap between the real-time theory and practice in ICS. While it is shown that dynamic adaptation can be performed in real-time, this requires suitable real-time models that are not used in practice.

Finally, resilience measures are used to quantify the impact of dynamic adaptation on the resilience of an ICS. The integration into the WATERBEAR architecture highlights the possibility of achieving resilient and autonomous modes of operation with today's technologies. While the path to resilience is a never-ending journey, it is clear that advancements can be made, and must be made to prepare for an uncertain future.

Contents

Chapter 1

Agile Industrial Control Architectures

Contents

The past years have shown how inherent uncertainties can upend the world, disrupt global markets, and destabilize production networks. The COVID-19 pandemic, the 2021 Suez Canal obstruction, and the 2022 energy crisis revealed the fragility of globalization and global supply chains [10]. Looking ahead, climate change is expected to lead to more frequent extreme weather events, disrupting all parts of society [11]. For production systems, this uncertainty manifests in disrupted supply chains, surging demand for specific products (such as medical supplies), and a shortage of skilled labor. Further, the disruptions to supply chains indicate a lack of resilience against unanticipated events [12]. For instance, the manufacturing industry was largely unprepared for the shortage of spare parts like programmable logic controllers (PLCs). Flexibility becomes necessary due to changes in demand [13]. The continued shortage in skilled labor must lead to efficient use of human work, e.g. through further automation [14]. If these challenges are not addressed, future production systems are not ready for the reality they must operate in. In contrast, future production systems must exhibit higher agility, adaptability, and autonomy to react quickly and decisively to anticipated and unanticipated events.

The concepts of adaptability and agility have been discussed since the 1990s [15]. Researchers have been studying flexible and reconfigurable manufacturing systems for decades, most importantly to quickly adjust production systems to market demands or to rapidly integrate new functions [16]. Yet, while developing in that direction, current ICS, the systems that control most of the world's production systems, have not achieved adaptability or agility, and are far from autonomous. ICS are considered to be part of OT, in contrast to IT. The need to reduce costs, increase availability, and improve system security has rapidly driven IT evolution over the years. For instance, the roles of operations and development have merged towards a *DevOps* role with the introduction of microservices and cloud architectures, and this role may further evolve towards *NoOps* or serverless architectures, where most of the operation tasks are automated [17, 18]. Similarly, the *vision of autonomic computation* imagines a world of self-managing computation systems [19], which can modify themselves according to their environment. An IBM white paper mentions increasing labor costs for maintaining IT systems as a reason driving the push towards autonomous behaviors [20]. Two examples of positive effects are mentioned as *selective process automation* and *reduced time and skill requirements*. Observing the growing complexity in ICS, mounting labor costs, and shortage of skilled labor in the highly-specialized automation domain, why are ICS lagging in automation and autonomy?

Key answers to this question include the significant need for safety and the inertia of highly cost-efficient equipment. In contrast to most IT systems, ICS pose strict requirements on process and environmental safety. Unlike web services or database systems, ICS are usually safety-critical hard real-time systems. As a consequence, these industries are also known to be reluctant to apply new technologies that may endanger the availability or system integrity [21]. Furthermore, manufacturers already struggle to oversee a zoo of legacy devices and machines. New technologies must be introduced carefully to remain compatible with existing systems.

This dissertation investigates this divergence between IT and OT considering adaptability and agility and attempts to narrow the gap or at least provide a sensible path for OT systems to start this journey. While a major pivot towards new system architectures is unlikely and undesirable, existing architectures and standards can indeed be extended in the right direction. And, as it will turn out, there exists a plethora of previous works and basic architectures in research to build upon.

This chapter introduces the state of the art in agility and adaptability of IT and

summarizes the challenges in this field. Following this, a resilient, self-adaptive ICS architecture, the WATERBEAR architecture, is proposed, which bridges the gap between IT and OT systems. This dissertation identifies three current and urgent challenges, addressing them using this architecture as the long-term target.

1.1 Agility and Adaptability in Software Engineering

Traditionally, agility and adaptability have played a crucial role in software engineering. In the software engineering domain, requirements change frequently, and agility enables risk management to react to respond to these risks before significant issues arise. On a broader scale, agility and adaptability are necessary means to transition towards resilient and autonomous behaviors. These are not novel developments. Gould [22], for instance, defines agility as "the ability of an enterprise to thrive in an environment of rapid and unpredictable change". Adaptability, and its extension self-adaptability, have been discussed in research for decades, nowadays often concerning cyber-physical systems (CPS) [23]. This dissertation specifically addresses technical adaptability, i.e., the embedded ability of a technical system to be adapted or to adapt itself.

Although adaptability seems to be a critical characteristic, making a system adaptable is not a straightforward task. Before discussing the low-level and high-level challenges in achieving (self-) adaptive behaviors, it is thus important to consider the history, the diverse motivations to adapt, and the types of adaptation. These will be the topic of the following section.

1.1.1 Requirements and Types of Adaptation

Adaptability is the ability of a system to adjust to changing requirements or environments. This broad definition allows for a broad selection of reasons to adapt, and as a consequence, numerous types of adaptation. After a brief historical review of adaptation, reasons, and types of adaptation are introduced.

Historical Overview

There are several names for the concept of online adaptation: Dynamic software updating, "on the fly" program modification, online version change, hot-swapping, dynamic software maintenance, or software reconfiguration [24]. *Online Version Change* is not a new problem [25]. Yet, while some approach it by looking at low-level C implementations, others have focused on managing the change within software architectures [26]. Within the context of this dissertation, the term *adaptation* is used interchangeably for any type of online change. *Self-adaptation* is used to denote an adaptation that a system can effect on itself.

Miedes and Muñoz-Escoi [27] provide a comprehensive history of important works in online adaptation, starting from 1976. There are numerous surveys on the topic of dynamic adaptation as well [24, 28, 29].

Going a step further, *self-adaptation* implies that a system is not only able to be adapted but empowered to perform this adaptation by itself [30]. The transition from simple adaptability towards self-adaptation is condensed by Weyns [29] into six waves:

1. **Automating tasks:** Automating management problems such as installation, configuration, operation, or maintenance.
2. **Architecture-based adaptation:** Applying basic design principles, e.g. abstraction and separation of concerns.
3. **Runtime models:** Using runtime models to provide up-to-date information about the system and to effect change on the model, rather than the system.
4. **Goal-driven adaptation:** Transitioning from strict requirements to more relaxed goals.
5. **Guarantees under uncertainties:** Taming the uncertainty by providing guarantees during the self-adaptation processes.
6. **Control-based adaptation:** Employing mathematically founded control theory to self-adaptation.

The waves continuously increase the sophistication, but also the abstraction level on which the adaptation is implemented. Thus, there is a shift from automation towards reconfigurable models and middlewares, towards managing uncertainty with goals and guarantees. The contributions of this dissertation mainly focus on waves 1 to 4, in which at least the system goals are clearly defined, but are not limited to mere automation of management tasks.

Software adaptation and resilience can be closely related. Gama, Rudametkin, and Donsez [31] mention this connection between software evolution and resilience. Specifically, *instantaneous repair* is the keyword used to describe resilient behavior, which can be achieved through software adaptation. The term evolvability is mentioned as a crucial property for systems to be resilient, and which goes beyond the concepts of *availability*.

Although the idea of online adaptation is not new, it is not commonly used. Ilvonen, Ihantola, and Mikkonen [32] review how well dynamic software updating is represented in practice and education, particularly software engineering education. A key takeaway is that there is a need for a holistic approach on the one hand, and a need to include currently feasible approaches into education curriculums on the other hand. Still, today, software change, and particularly online software change is not a first-class concept in most IT systems, despite extensive developments in the area of DevOps [33]. There is a need to incorporate formal models and verification into agile development processes in the manner that self-adaptive architectures envision it.

The field of adaptation and self-adaptation has been evolving continuously over the previous decades, and it will evolve further. There is no one-size-fits-all solution. The *six waves of engineering self-adaptive systems* ([29]) indicate a direction for the future, yet current systems are often stuck in the first waves. This slow progression is partly due to the need for a deeper understanding when transitioning between waves. For instance, automating management tasks requires an understanding of why these tasks are needed and what parts of them can be automated. Runtime models require a deep understanding of the process. Goals and guarantees require consciousness of the high-level goals and guarantees and their boundary conditions. Finally, incorporating (self-) adaptability into a system comes at a cost—if a non-adaptable system can perform the tasks, why invest in adaptability?

There are plenty of reasons to provide adaptability, and there must be reasons not to. In the section that follows, it will be shown that adaptation can be implemented for many reasons, and this must lead to a spectrum of solutions.

Types of Adaptation

There are numerous reasons to adapt a (software) system. The ISO/IEC 14764 [34], for instance, mentions five types of maintenance for software systems which indicate reasons for online adaptation:

- **Corrective maintenance:** Modification of a software product performed after delivery to correct discovered problems.
- **Preventive maintenance:** Modification of a software product after delivery to correct latent faults in the software product before they occur in the live system.
- **Adaptive maintenance:** Modification of a software product, performed after delivery, to keep a software product usable in a changed or changing environment.
- **Additive maintenance:** Modification of a software product performed after delivery to add functionality or features to enhance the usage of the product
- **Perfective maintenance:** Modification of a software product to provide enhancements for users, improvements of information for users, and recording to improve software performance, maintainability, or other software attributes.

These types already indicate a temporal component: Corrective maintenance occurs after a failure has occurred, whereas preventive maintenance must happen before. Furthermore, there is a functional dimension: Adaptive maintenance handles changes in the environment, additive maintenance deals with increased feature scope, and perfective maintenance addresses non-functional improvements.

Chapin et al. [35], on the other hand, classify twelve types of software evolution and maintenance along three criteria: Whether or not the software was changed, whether or not the source code was changed, and whether or not a function was changed. This results in four clusters (ordered by increasing impact): Support interface maintenance, documentation maintenance, software properties maintenance, or business rules maintenance. This classification is thus mainly concerned with the why and the scope of the change, i.e., if it is limited to documentation, or if it touches the business logic. Buckley et al. [36] provide a taxonomy of software change and identifies fifteen dimensions over which changes can be compared, complementing the previous works by focusing on the how, when, what, and where of software change. Consequently, there is not one single reason to adapt a system dynamically but many.

As a result of this multitude of reasons, there have been several implementations of dynamic adaptation. Seifzadeh, Abolhassani, and Moshkenani [24] classify software adaptation by capabilities, constraints, and techniques. In terms of capabilities/constraints, some metrics are scope, level of abstraction, ability to update previously started code, simplicity, consistency, wait time to update and predictability, update duration, code cleanup, performance overhead, supported changes, program-

ming language, and multithreading support. Regarding techniques the following metrics are chosen: Unit of update, upgrading dependents of an updated module, update atomicity, state transformation, type-safety, time of update, updating active (non-quiescent) code, and level of code differencing. This large array of metrics indicates that most dynamic adaptation frameworks differ significantly and not all applications require the same sophistication. Specifically, the implementation characteristics will mainly depend on the type of modification. An adaptation that, according to Chapin et al. [35], modifies the software properties without touching the business rules may not require a state transformation at all.

While online software change or software evolution allows the software to be changed, this leads to an open loop in which the programmer must implement the changes manually. Self-adaptive software closes this loop and uses feedback from the self and the context [37]. Self-adaptation is considered to be on a higher abstraction level (*General Level*) compared to, for example, self-configuration (*Major Level*) or self-awareness (*Primitive Level*) [37]. Salehie and Tahvildari [37] identify different paradigms of self-adaptation: *Closed/Open Adaptation* refers to either a limited or unlimited number of adaptive actions available during runtime. *Model-Based/Free Adaptation* is either based on a fixed model or allows adaptation independent of a model. *Specific/Generic Adaptation* distinguishes if only specific domains/applications can be adapted, or if the mechanism is suitable for other domains as well.

Self-adaptation can be an internal or an external process. In an internal process, the software truly adapts *itself*, i.e., there is no external entity to trigger the change. In an external process, the *adaptable software* is adapted by an *adaptation engine* [37]. External adaptation is often achieved through the use of a middleware that provides support for configurability and reconfigurability [38]. Consequently, self-adaptation can be split into multiple tasks and phases. For example, Blair et al. [38] mentions four phases: Identification of a need, computing the reconfiguration, detecting a safe state, and adaptation execution. In the *vision of autonomic computing* [19], slightly different phases were identified: Monitoring, Analysis, Planning, and Execution. These phases make up the MAPE-K self-adaptation model.

Reasons for adaptation and self-adaptation are abundant and the need for adaptability and agility in future systems cannot be discounted. In practice, (self-) adaptation must be integrated carefully into the system architecture and the cost of adaptability must be recognized. For a simple architecture, an open-loop self-configurable architecture may suffice. For highly complex systems with high avail-

ability requirements and long life cycles, such as specific forms of ICS, a sophisticated self-adaptation framework is highly desirable.

1.1.2 Adaptation Challenges

A broad spectrum of reasons to adapt leads to a broad spectrum of implementations, which consequently leads to a broad spectrum of challenges that must be tackled. This section summarizes challenges from both the dynamic adaptation and the self-adaptation perspective. The specific challenges addressed in this dissertation are pointed out in Section 1.3.

Dynamic Adaptation Challenges

Self-adaptation on a high abstraction level requires adaptability under the hood. As seen in the previous section, self-adaptation can be achieved by separating the adaptable software from the adaptation engine [37]. This separation of concern is, for most applications, preferable. If this adaptation should take place on the fly, this is referred to as *Dynamic Adaptation*. General challenges of dynamic adaptation are mentioned by Mlinarić [39] as availability, correctness, flexibility (changeability), performance, and simplicity (usability). These goals are stated as potentially contradictory, thus there has to be a balance between the requirements. For example, simplicity may conflict with flexibility or performance. Other goals, such as availability and correctness should not be compromised. A major challenge in developing an adaptation framework is thus finding the right balance between the requirements.

Temporally, the challenges can be split into the phases *before*, *during*, and *after* the adaptation [31]: For instance, before an adaptation, verification is necessary and compatibility must be checked. During the adaptation, the service must not be interrupted, and the component state must be maintained. After the adaptation, inconsistencies must be resolved, such as dead code / dangling objects, or non-terminated processes. Additional challenges arise with the update cost and the update duration.

Looking closer at the *during* phase of the adaptation, a more detailed analysis is possible. Lounas, Mezghiche, and Lanet [40] mention the following challenges: Code update, data update, update timing, and correctness. These challenges are

very subjective to the application area. For example, the meaning of update timing and correctness crucially depends on the system. Going down to the component or code level, more problems arise. Feng et al. [41] formalize three important issues in dynamic adaptation of component-based architectures:

Referential Transparency Problem Components may not have a full understanding of the references they hold to other components and thus may be unable to update them accordingly during a change.

State Transfer Problem The state or history of a component must be updated in a way that guarantees consistent behavior.

Mutual Referential Problem Dependencies between components must be resolved, which may require the concurrent change of multiple components.

Further problems are the increased complexity between swapping a component and swapping an object, the processing of requests issued during the evolution process, and the processing of unfinished requests or transactions [42]. On the binary level, one must deal with issues such as rewriting of binary code, programming language issues, and version coexistence [27].

These issues highlight how crucial information about the system is during an adaptation. Decomposition of an architecture is insufficient, if the references and connections between components are not documented or known. Updating a state within a component requires the awareness that there is a state, and what this state is composed of. Coming back to the waves of self-adaptation as defined by Weyns [29], architecture-based adaptation is crucial to handle the referential transparency problem, whereas runtime models can assist in dealing with state transformations or processing unfinished requests. Yet, while self-adaptation addresses some of these issues, it introduces new challenges as well.

Self-Adaptation Challenges

The possibility of dynamic adaptation invites the use of self-adaptation, where the system can adapt itself. Although the idea is simple, the implementation in practice is not. The following paragraphs show three levels on which challenges in self-adaptation can be observed, and how the perspective has evolved over the years.

In their 2009 paper, Salehie and Tahvildari [37] identify research challenges in four distinct areas of self-adaptation: Engineering, self-* properties, adaptation processes, and interaction challenges. Engineering challenges are mainly concerned

with the requirements, design, implementation, and testing of self-adaptive systems. Self-* property challenges refer, on the one hand, to the development of individual, less-noticed properties, and, on the other hand, to multi-property systems. The challenges of the adaptation processes can be assigned to the phases of the MAPE cycle, and interaction challenges address the difficulties in designing effective policies, establishing trust, and achieving interoperability.

In 2013, Lemos et al. [43] wrote a paper on a renewed research roadmap. This time, the challenges were split into design space, processes, decentralization, and run-time verification & validation. Specifically, the design space challenges refer to the issue of defining, representing, and observing the solution or decision space over which self-adaptation is performed. *Process* challenges describe the changes to software development processes required in self-adaptive systems, e.g. shifting design-time activities to run-time or requiring continuous validation. Decentralization is required for complex architectures, yet it also must guarantee system-wide quality goals and requires further coordination between participants. Finally, validation & verification (V&V) in a self-adaptive system must handle changing requirements due to evolution and the complexity of run-time techniques in contrast to design-time V&V. These challenges indicate a shift from implementation issues towards software engineering issues.

In the more recent work (2019), Weyns [29] proposes challenges in the current and future waves of engineering self-adaptive systems. In current waves, the identified challenges are adaptation in decentralized settings, dealing with changing goals, domain-specific modeling languages, dealing with complex types of uncertainties, empirical evidence for the value of self-adaptation, and aligning with emerging technologies. In future waves, there are further challenges ahead: Exploiting artificial intelligence, coordinating adaptation with blockchain technology, dealing with unanticipated change, control theory as a scientific foundation for self-adaptation, and multidisciplinarity. These challenges indicate an increased abstraction level of the self-adaptation topic and the maturity of the domain.

Conclusion

As explained earlier, there are many reasons to perform (self-) adaptation which has led to numerous implementations and proposed solutions. There is a clear difference between dynamic adaptation and self-adaptation challenges. In dynamic adaptation,

many challenges have been known for decades, and it has been accepted that there is no one-size-fits-all solution. For instance, guaranteeing integrity is highly dependent on the application. In self-adaptation, there is an evolution of challenges from the implementation level to higher abstraction levels. The underlying adaptability is expected, although for many systems, guaranteeing integrity is not trivial. The next section proposes a self-adaptive architecture for ICS before the specific challenges of this dissertation are presented in Section 1.3.

1.2 An Architecture for Self-Adaptive Industrial Control

The previous section has summarized the general requirements, types, and challenges of dynamic adaptation in the software engineering domain. Of these requirements, some are more, and some are less relevant for ICS. Also, not all types of dynamic adaptation are feasible for ICS, and not all challenges can be addressed within the scope of this dissertation.

This section introduces the most relevant self-adaptive architectures as they have been proposed in research. Elaborating on these architectures, a novel architecture is proposed that extends currently feasible approaches and addresses practically relevant challenges in dynamic adaptation of ICS to limit the scope of this dissertation.

1.2.1 Self-Adaptive Architectures

Multiple architectures and implementations have been shown in research to achieve self-adaptive or autonomous behaviors. The most influential self-adaptation model is MAPE-K, which can be implemented in different design patterns. Consequently, this model and its extensions are chosen in this dissertation as the foundation for an application of this model to ICS. This section introduces the aforementioned models, while the next section explains the resulting ICS architecture.

MAPE-K

The MAPE-K self-adaptation model ([19, 44]) is one of the most influential concepts of self-adaptation to-date [45]. It is based on a feedback loop of four phases: Initially

coined *Collect*, *Analyze*, *Decide*, *Act*, the phases are later referred to as *Monitor*, *Analyze*, *Plan*, and *Execute*:

Knowledge The knowledge is the collection of all data, states, or goals of the system. Usually depicted as an abstract cloud, the knowledge is shared between the different phases and represents the central knowledge base.

Monitor During the monitoring phase, data is collected and gathered in the knowledge. This could be new information from sensors, changed system states, or new system goals.

Analyze The analysis phase scrutinizes the knowledge to identify if an adaptation could be necessary. The MAPE-K model does not limit the type of algorithm used to perform this analysis.

Planning During the planning phase, the adaptation is prepared, and the necessary actions are gathered. The actions are assembled into an adaptation workflow, i.e., a recipe of how to adapt the system into the desired shape.

Execution Once the decision to adapt is made and the adaptation workflow is composed, the adaptation can be executed.

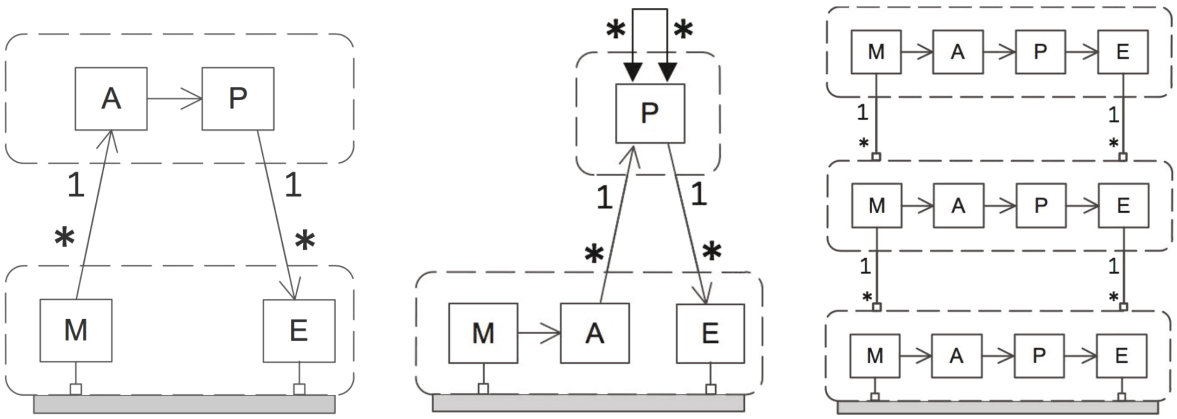
Decomposing the process of self-adaptation into four distinct actions over one shared knowledgebase permits the use of sophisticated architectures with clearly allotted tasks and responsibilities, as seen in the following section.

Distributed MAPE-K

The decomposition of the self-adaptation process into the four MAPE phases allows the decentralization and distribution of the feedback loop [46]. In practice, more than one feedback loop may be necessary to implement self-adaptation in a complex system. Multiple patterns to achieve this distribution or decentralization are outlined in [46], and summarized here:

Master/Slave Pattern[1] This centralized pattern allows distributed monitoring and execution phases, yet centralizes the analysis and planning phase on a single (master) peer (Figure 1.1a). A single point of failure is introduced, and there may be significant communication overhead, however, the implementation is simple and a centralized analysis and decision-making process can efficiently utilize all information available.

[1]There are terminology concerns regarding this naming convention. An alternative name could be *Controller / Target*.



(a) Master/Slave **(b)** Regional Planner **(c)** Hierarchical Control

Figure 1.1: Distributed design pattern for MAPE-K self-adaptive systems as presented in [46] that allow the distribution of responsibilities.

Regional Planning Pattern In some scenarios, it may be necessary to centralize the adaptation planning phase, yet the analysis of information can be done locally (Figure 1.1b). For example, the (re-) allocation of resources in a cloud setting may be decided locally, while the planning is done centrally. Local monitoring and analysis can lead to a smaller communication overhead.

Hierarchical Control Pattern A single self-adaptation layer of may not be sufficient in complex architectures. In this case, a hierarchical decomposition of the adaptation process can allow higher levels to focus on higher-level goals, while low-level goals are considered on the bottom layers (Figure 1.1c). This can add complexity, yet may also allow the self-adaptation of complex systems.

The basic MAPE architectures struggle with providing self-adaptation for complex technical systems. The MAPE-K model can partially hide this by assuming a shared *knowledge* component, however, sharing information comes at a cost. The distributed design patterns demonstrate the ability to transparently distribute the responsibilities of self-adaptation for complex architectures. The ability to distribute tasks is a key reason to split the self-adaptation loop into distinct phases. Distributing these tasks to multiple entities also provides clear responsibilities and a clear distinction between consumers and producers.

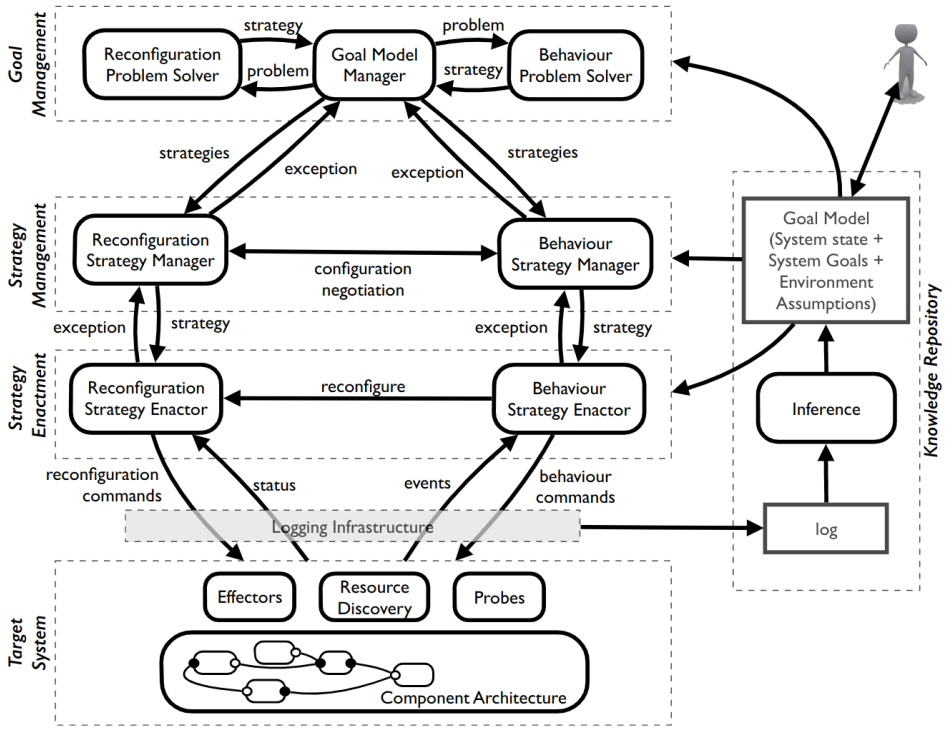


Figure 1.2: The MORPH reference architecture as presented by Braberman et al. [47], which clearly distinguishes between configuration and behavior changes.

The MORPH Reference Architecture

The MORPH reference architecture [47] combines previously proposed architectures for self-adaptation with multi-layered, hierarchical MAPE-K loops. This architecture explicitly distinguishes between changes to the configuration and changes to the behavior. The architecture is depicted in Figure 1.2 and consists of five layers. In this manner, the responsibilities are clearly distinguished, and the interactions and dependencies between the layers are explicit. The five layers are:

Target System The target system contains the operationally necessary functionalities. It performs monitoring tasks and reports its status and relevant events to higher layers. In return, it receives commands from higher layers to change its configuration or behavior.

Knowledge Repository The knowledge repository logs and stores information about the goals, system, and environment and makes it available to the other layers. It may contain a runtime model that it keeps up to date.

Strategy Enactment This layer decides if a change in the target system is neces-

sary within the current strategy, sends commands to the target system, and reports errors or exceptions to the higher layer. It does not decide on the strategy but merely receives it.

Strategy Management This layer manages and selects strategies. It receives multiple strategies, of which an appropriate one is propagated to the strategy enactors. When an exception occurs, it may decide to switch to a different strategy. However, this layer does not create new strategies.

Goal Management This layer performs the computationally expensive tasks of generating strategies based on the information in the knowledge repository and returns them to the strategy management layer. It does not decide which strategy to choose and does not enact the strategy.

This architecture further decomposes the tasks and responsibilities of a self-adaptive or autonomous architecture and describes the interactions between the actors. This decomposition is a necessary task to handle the complexity and provide transparency of the interactions. It is also reminiscent of the six waves of self-adaptation as proposed by Weyns [29]: Architecture-based adaptation takes place within the target system, runtime models are maintained within the knowledge repository, and goals and guarantees must be handled by the goal management layer. The architecture further distinguishes between two types of adaptations: Configuration changes and behavior changes. This is reminiscent of the works of Chapin et al. [35] and hints at the fact that different adaptations require different methods of enactment and guarantees.

Conclusion on Self-Adaptive Architectures

Current self-adaptive architectures have shown that splitting the responsibility into separate tasks, as in the MAPE-K model, is highly beneficial. The individual tasks can be distributed, and implementation becomes much more efficient. Instead of building a single *self-adaptive system*, multiple subsystems can be implemented that, together, yield a *self-adaptive architecture*. Continued hierarchical decomposition between strategy generation, management, and enactment provides the ability to quickly debug issues. Was the fault in the selection of a strategy, or was there a lack of a feasible strategy? Was this due to a lack in the generation process, or was there insufficient or wrong data in the knowledge repository? Merging the layers into one complicates the procedure and erases the transparency that is especially needed for

safety-critical systems.

This basic understanding is applied to ICS in the following section, where a novel architecture for self-adaptive ICS is proposed.

1.2.2

Self-Adaptation in ICS

So far this chapter has offered a motivation why self-adaptation is needed for ICS and introduced the transition from self-adaptive systems to self-adaptive architectures with delegated tasks and responsibilities. This section is about how self-adaptive architectures can be applied to ICS.

Two points should be emphasized: First, the fact that ICS are often safety-critical and hard real-time systems, the burden of guaranteeing the integrity of the adaptation process is much higher than for many IT systems. Secondly, the control logic in many ICS is comparatively simpler than for general IT systems. This led to the development of domain-specific programming languages with stripped-down features compared to general-purpose programming languages. Considering these domain-specific languages is important to address concerns about the adoption of new techniques.

This section first describes the intended behavior of the ICS in a self-adaptation life cycle, before describing a potential self-adaptive architecture for ICS. Finally, the scope of this dissertation is defined.

Self-Adaptive Behavior in ICS

The dependability of the industrial process is the most important task of ICS. Current ICS achieve this through redundancy, determinism, and robustness. However, agility, adaptability, and autonomy can provide new levels of dependability and flexibility. This is why this dissertation is concerned with introducing these abilities to ICS Self-adaptation is a valid path toward these abilities.

Industrial control systems (ICS) are safety-critical real-time systems. As such, they must satisfy strict requirements and must undergo extensive validation & verification (V&V). Iber et al. [48] detail the potentials of self-adaptation in ICS, specifically for typical IEC 61131-3 POUs / tasks. A key insight is that the adaptation may disturb the execution, thus the advantage of an adaptation must be weighed against the risks of a delay. This is particularly important for traditional ICS with computationally

heavy, monolithic tasks.

In this manner, one of the main goals of self-adaptation is to provide a type of fault tolerance to improve the dependability against unanticipated changes. Faults are inevitable in any technical system and can be the result of wear and tear, external influence, or even bugs in the software. While faults cannot be prevented, the goal of fault tolerance is to prevent a resulting error or failure. In many scenarios, the error or failure can be prevented by deliberately changing the system to work around the fault. Depending on whether or not the fault is fixed before or after a failure, this would correspond to either *preventive* or *corrective maintenance*, according to ISO/IEC/IEEE [34]. Self-adaptation could also be used to, for instance, perform *perfective maintenance*. These types of adaptations would not be as critical for a safety-critical system, since they do not directly intend to correct or prevent a failure.

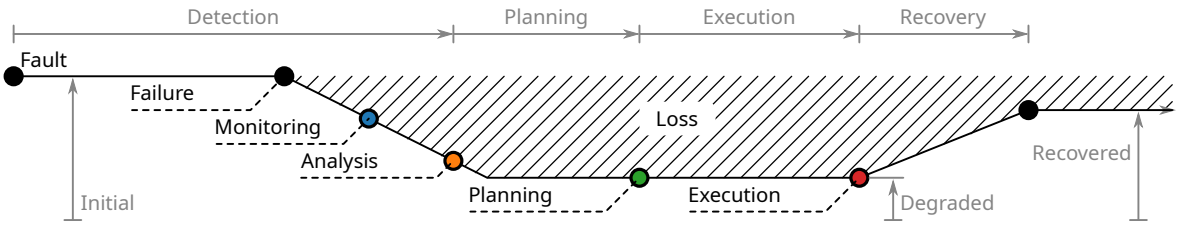


Figure 1.3: The resilience graph allows the quantification of a resilience loss over time. The MAPE-K model can be integrated into this graph to highlight how each phase affects the resilience loss.

Concerning the MAPE feedback loop, a potential reaction to a fault is depicted in Figure 1.3. In this particular case, the fault detection only finishes after the failure, yet this is not always the case. Nevertheless, detection (monitoring & analysis) must identify that there is a fault before a plan can be made to repair it. Once the adaptation plan is made, it can be executed, and the system can be returned to a recovered state. The goal of self-adaptation, within this figure, is to minimize the area underneath the graph, i.e., the loss incurred by the fault/failure. In the most optimal scenario, the failure can be prevented.

This optimization problem sets the scene for how self-adaptation should be integrated into ICS. The following section presents an architecture that implements this behavior and integrates the existing works on dynamic reconfiguration of component-based architectures, particularly in the domain of ICS.

1.2.3

The WATERBEAR Architecture[2]

Having seen the multi-layered self-adaptive architectures in research, this state of the art can now be merged with current ICS architectures that enable dynamic reconfiguration. The result is the *WATERBEAR* architecture, named after the most resilient creature on earth. This architecture combines the layered decomposition of the MORPH architecture ([47]) with the reconfiguration model of the IEC 61499 standard as defined by Zoitl [50], which itself builds on the works of Kramer and Magee [51, 52]. The blended model is depicted in Figure 1.4.

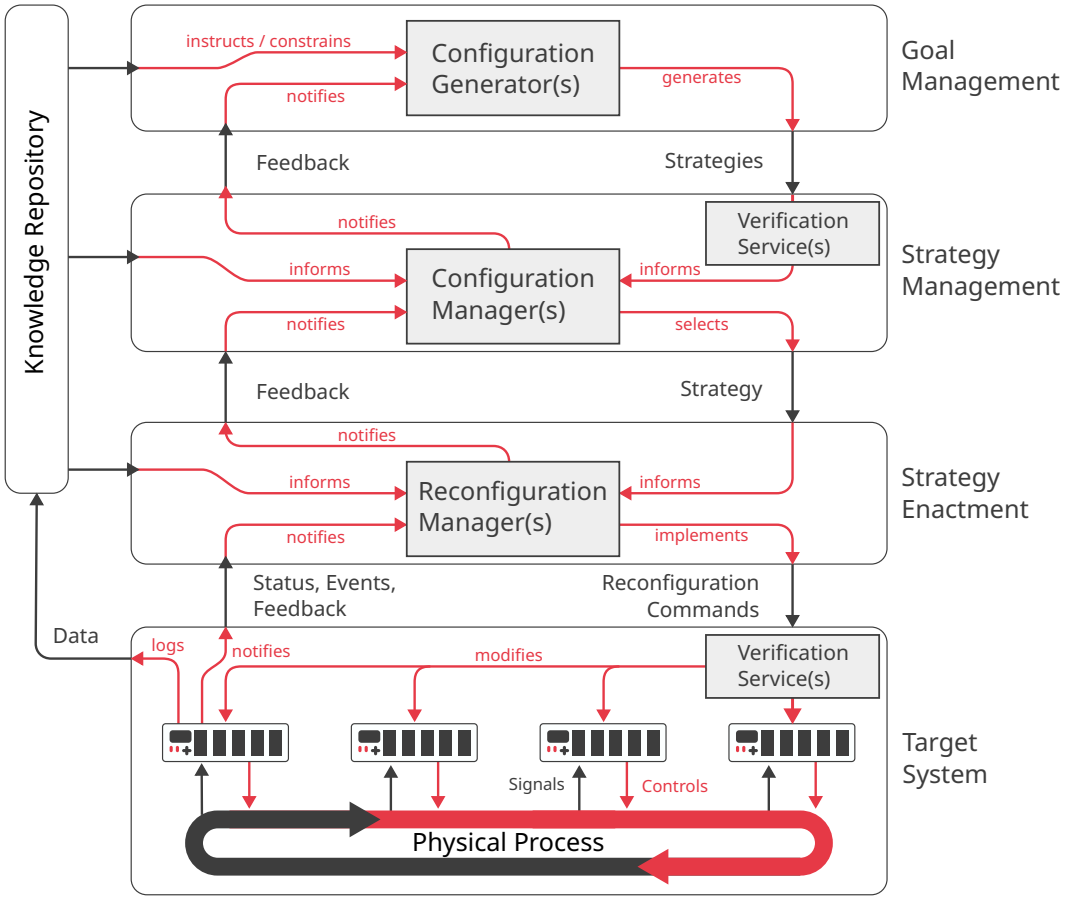


Figure 1.4: The *WATERBEAR* architecture blends the MORPH architecture [47] with the existing ICS architectures to fulfill the requirements of reconfigurable industrial control software.

[2]The *water bear*, also known as *tardigrade* or *moss piglet*, is considered to be one of the most resilient creatures on earth [49]. Destined to outlive humanity, just like the WATERBEAR architecture.

In contrast to the MORPH architecture, behavior and configuration changes are treated equally and use the same reconfiguration channels in the WATERBEAR architecture. This is a simplification, since ICS are typically resource-constrained and require extensive V&V. Implementing two separate interfaces may require additional resources and two separate validation mechanisms. Principally, using a separate channel for minor behavior changes may be feasible, yet out of the scope of this dissertation.

The physical process and the ICS controlling it are located in the *target system* layer, the *reconfiguration manager* is located in the *strategy enactment* layer, the *configuration manager* is located in the *strategy management* layer, and the *configuration generator* is presented within the *goal management* layer. The individual layers and their responsibilities are detailed in the following:

Target System The target system is the ICS, which logs data into the knowledge repository and feeds back status messages or important events to the strategy enactment layer. It receives reconfiguration commands from the strategy enactment layer as intended by the IEC 61499 reconfiguration model [50].

Strategy Enactment The reconfiguration manager is triggered by status changes or events from the target system. In response, it uses the information in the knowledge repository to implement a reconfiguration in the form of reconfiguration commands in correspondence to a specific strategy that was provided by the strategy management layer. In the scope of the IEC 61499 standard, the strategy could be an IEC 61499 system configuration or a corresponding change specification.

Strategy Management The configuration manager is triggered by the reconfiguration manager in case the strategy is not sufficient for the current scenario. It uses the information in the knowledge repository to select a suitable strategy from a set of strategies provided by the goal management layer.

Goal Management The configuration generator is triggered by the configuration manager if all currently available strategies are insufficient to handle the current scenario. Then, the configuration generator must use the information in the knowledge repository to generate new strategies that propagate down to the strategy management layer.

Knowledge Repository The information generated by the ICS, as well as the initial requirements and goals, must be stored and made available to all partici-

pants. This abstract knowledge repository serves this purpose.

Each layer may contain more than one generator or manager service. For example, a pool of configuration generator services may be used to find as many feasible strategies as possible. Similarly, there may be multiple configuration managers which can decentrally decide on the most feasible strategy to be implemented. To ensure the safety of the architecture, most importantly if a decentralized approach is utilized, there may be a need for verification services that confirm the consistency, integrity, and safety of the strategies and the resulting reconfiguration commands. This verification is only necessary when new strategies are formed, or when a strategy is implemented in reconfiguration commands.

The MORPH architecture builds upon the MAPE-K feedback loop models. Thus, these phases also re-emerge in the WATERBEAR architecture. The *Monitoring* phase is present in the target system where data is logged to the *Knowledge* in the knowledge repository and status and events are generated and sent up the chain of command towards the strategy enactment layer. *Analysis* and *Planning* are implemented in the reconfiguration managers, configuration managers, and configuration generators in a hierarchical manner [46]. The reconfiguration manager, for instance, analyzes the information in the knowledge repository and the current status to decide if reconfiguration is necessary, and if it is, plans the procedure. The configuration manager, in turn, analyzes the knowledge to select the most suitable strategy for the current situation.

Due to the multi-layered decomposition of tasks and responsibilities, the implementation is simplified, and the individual challenges are more easily addressed and solved. The following section describes the challenges in achieving a self-adaptive architecture, such as the WATERBEAR architecture, before the scope of this dissertation is specified.

1.3 Challenges for the WATERBEAR architecture

The general challenges of (self-) adaptive architectures and systems were discussed in Section 1.1.2. The WATERBEAR architecture provides a concept that promises superior adaptability, agility, and resilience for ICS. Yet, it is only a conceptual architecture that needs refinement and implementation effort to put into practice.

Industrial automation systems as a whole are typically highly customized for

their application. While there is some overlap in models, languages, and technologies, there is a significant difference between, for example, discrete manufacturing and process control systems. This difference must also materialize in the WATERBEAR architecture. The goal management or configuration generation, for instance, must integrate domain-specific information and models. The risk assessment influences the possible levels of adaptability and what assurance measures must be taken along the way, which must be reflected on the strategy management layer.

This dissertation concentrates on the layers necessary to achieve self-adaptation capabilities, which are use-case independent. These layers are the lower layers: The target system layer and the strategy enactment layer. Other layers, such as the strategy management or the knowledge repository, are only addressed as needed. This scope is illustrated in Figure 1.5. Generally, the scope includes *how* the system can (self-) adapt, and excludes the application-specific questions of *what* or *when* to adapt. For instance, the aim is not to investigate specifically how to adapt a PID controller for a process control system, but how to facilitate adaptability and self-adaptability for general ICS applications. In the same manner, the triggering of the adaptation is not addressed, because this is highly application-specific and there is much room for interpretation. Following, the layers are discussed individually:

Target System The state of the art in reconfigurable industrial control software is applied and extended to investigate how, on the control level, the software can be modified in real-time using a middleware or runtime environment (RTE). This requires the handling of the reconfiguration commands from the higher layers. Logging of data is limited to the information necessary to ensure a safe adaptation procedure. Here, dynamic adaptation challenges arise, such as the state transfer problem [41].

Strategy Enactment The strategy enactment layer must decide how the intended change can be implemented in reconfiguration commands. These commands must consider the requirements of consistency and timing. The decision-making processes that decide when an adaptation is necessary are excluded, because they do not influence how the adaptation is implemented. Here, dependencies must be resolved and correctness and consistency are key [40, 41].

Strategy Management The strategy management layer plays a crucial role in selecting a strategy while engaging with the knowledge repository and lower layers. Within this dissertation, it is assumed that a strategy already exists,

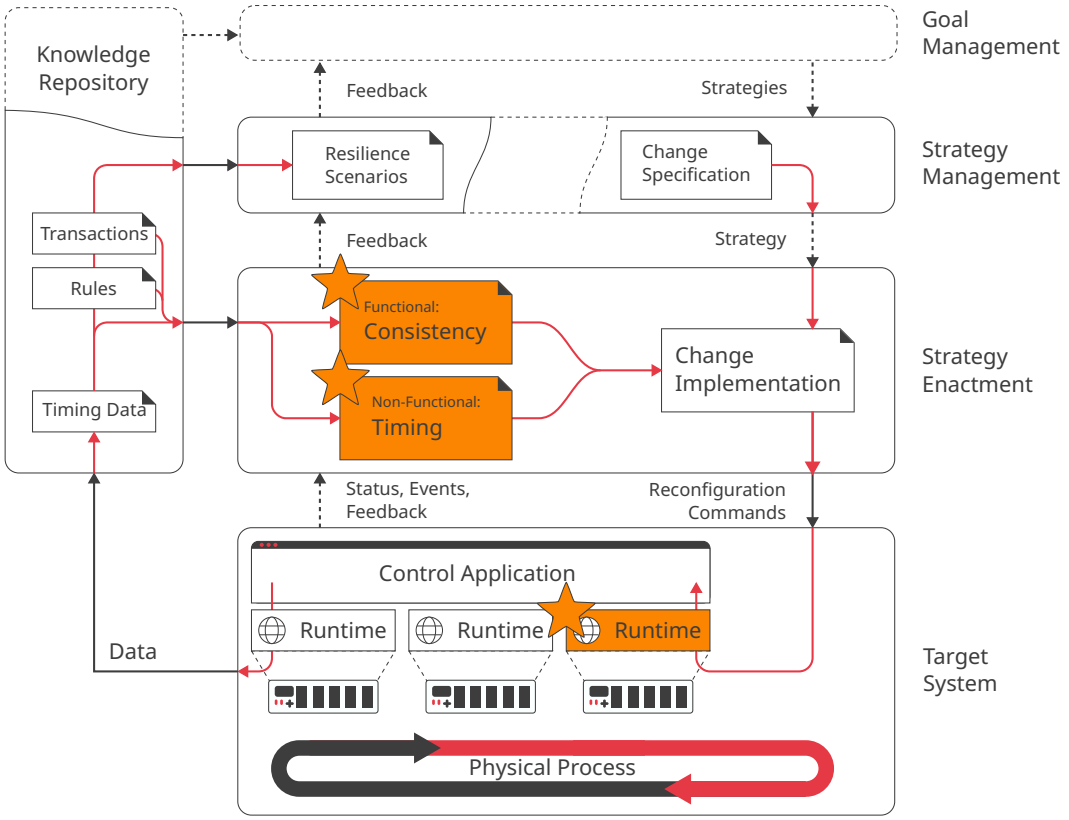


Figure 1.5: The scope of this dissertation focuses on the adaptation of the ICS, and as a consequence concentrates on the target system and strategy enactment layers. The three key cornerstones are the adaptability of the runtime environment, the functional consistency, and the non-functional timing requirements.

because the selection process is highly application- and domain-specific. Recognizing the diverse nature of strategies, it must be acknowledged that there is no optimal selection strategy for all use cases. This dissertation takes an agnostic approach here to provide a flexible self-adaptation framework that can be tailored to fit most applications.

Goal Management For similar reasons, the goal management layer is largely excluded in this dissertation, primarily due to its application- and domain-specific nature. A possible solution is sketched in the final chapters to demonstrate the full architecture and to underscore the flexibility and adaptability of the architecture to diverse applications.

Knowledge Repository The knowledge repository is partially relevant to the dissertation. Some information, e.g., system requirements, basic behaviors, and timing data are commonly available or feasible to gather in current systems.

These models are considered in this dissertation as necessary input. An interested reader could find plenty of inspiration on how this could be extended within the domain of the *digital twin*.

Consequently, this dissertation focuses on the core feature of the WATERBEAR architecture: (Self-) adaptability. To create the highest impact, the argument can be made that of all the challenges proposed by Mlinarić [39], correctness and simplicity are the most urgent for the advancement of ICS. Correctness is critical to comply with the stringent safety requirements of ICS. Simplicity reduces the threshold to application, which can lead to multiplier effects if adaptation is more frequently used. Once a simple and correct process is established, its flexibility and performance can be improved further. Regarding self-adaptation, the use of domain-specific languages and gathering empirical evidence for the value of self-adaptation can be seen as most critical [29]. Use in ICS must take into account domain-specific languages, and empirical evidence can substantiate the need for self-adaptation. Without clear cost benefits, (self-) adaptation will not reach the domain of ICS.

Summary

This dissertation offers a roadmap to achieve a basic level of acceptance of (self-) adaptation in ICS, which promotes resilient and autonomous behaviors. As seen previously, there are open challenges on all architecture levels. The foundation, however, consists of the lower levels and specifically of empowering the target system to be adaptable. Consequently, this is the key area of interest for this dissertation, and the three main challenges addressed in this dissertation are:

Adaptable Runtime Environment The development of an adaptable middleware that abstracts the safety-critical control logic from the low-level execution semantics. This is achieved by extending existing domain-specific models and reusing existing technologies from the IT domain.

Consistency during Adaptation As safety-critical systems, ICS are subjected to strict safety requirements, and their behavior must be predictable. A change in the implementation must only lead to intended behavior changes. This topic is approached by applying formal consistency conditions to the adaptation process. Automating the procedure increases confidence in the correctness and reduces the manual effort.

Timeliness during Adaptation As real-time systems, ICS must satisfy real-time

constraints. This also applies during the adaptation, when the behavior will be temporarily disrupted. The system integrity is only guaranteed if real-time constraints are considered before, during, and after the adaptation. Automatically considering the timing and schedulability further reduces the threshold for application.

These challenges are explained in detail in the following sections, and solutions are addressed in the chapters to come.

1.3.1 Adaptable Runtime Environment

As was identified in the past, adaptation, and, in particular, self-adaptation, rely on higher-level abstractions than low-level dynamic software updating. To achieve this, a reconfigurable or adaptable middleware can be used that separates the control logic from the underlying implementation. This separation serves two main purposes:

- It provides re-usability for the necessary high-level services. For example, most adaptations rely on the addition or removal of functionality. The low-level instructions can be assembled into services that can be reused in diverse use cases.
- It simplifies the implementation of an adaptation because the focus can be put on the adaptation logic instead of the control logic. In traditional source-code level updating, the logic and modification are deeply intertwined. This hampers the automatic handling of the adaptation and complicates the manual handling as well.

Current domain-specific runtime environments do not fully support dynamic adaptation or lack high-level abstractions necessary for an application in ICS. While rudimentary support for dynamic adaptation exists, it is rarely applied. To achieve closed-loop self-adaptation, however, the ability to adapt is essential.

Thus, in Chapter 2, the current state of the art in adaptable runtime environments for ICS is investigated, analyzed, and compared to middlewares of other domains in which online-change is already established. Further, a runtime environment with a focus on dynamic adaptation is implemented and evaluated concerning its ability to satisfy real-time constraints.

1.3.2

Consistency during Adaptation

ICS oversee physical processes. Many of these processes can cause harm or damage, and, thus, ICS are often subject to stringent safety constraints. The aversion to change and modification in industrial systems is very well motivated: The execution should be consistent and reliable. Dynamic adaptation represents a type of intended change. Nevertheless, the behavior must remain predictable and consistent.

There are many ways to achieve consistency during an adaptation. Stopping the plant, returning it to an initial state, and restarting the process, for example, is a way to achieve consistency. It does, however, restrict the availability, productivity, and agility. Thus, dynamic adaptation must provide the same guarantees, but without the downtime.

This challenge is further developed in Chapter 3. Specifically, after introducing a selection of existing consistency conditions, a mechanism to resolve the dependencies between components is proposed. This mechanism does come with its disadvantages, yet it builds on existing models and provides a key component to the propagation of dynamic adaptation: simplicity. The automation of the adaptation procedure reduces the friction currently encountered when evaluating whether or not dynamic adaptation is production-ready.

1.3.3

Timeliness during Adaptation

While functional consistency is important, it is not the only requirement when adapting ICS, as ICS are usually real-time systems. As the name implies, dynamic adaptation must be performed *dynamically*, i.e., while the system is and remains running. Thus, the real-time guarantees of the system must be preserved.

This leads to two problems that must be addressed. First, there needs to be an expressive real-time model of the system under adaptation, i.e., the ICS. Second, the disruption of the adaptation concerning this real-time model must be quantified. Deciding if an adaptation is feasible or not requires the existence of a schedulability condition. Not every adaptation can be feasible in real-time. Thus, identifying a workable adaptation that satisfies both the consistency and timing requirements is crucial.

The topic of real-time dynamic adaptation is addressed in Chapter 4. After summarizing the state of the art in real-time scheduling for ICS, a schedulability condition for dynamic adaptation is found that extends existing real-time models. Then, expected adaptation times for ICS are extrapolated from measurements.

1.4 Conclusion

This chapter motivated the need for agility and self-adaptability in ICS. Based on the current challenges in dynamic adaptation and self-adaptation, it derived the three most pressing challenges that currently limit the adoption of dynamic adaptation and self-adaptation in industrial architectures. These three challenges are addressed in Chapters 2, 3, and 4. Chapter 5 assembles these pieces and shows the impact of dynamic adaptation and self-adaptation on the resilience of technical systems, and further proposes a decentralized implementation of the WATERBEAR architecture. Finally, Chapter 6 summarizes the contributions of this dissertation and highlights limitations and future research directions.

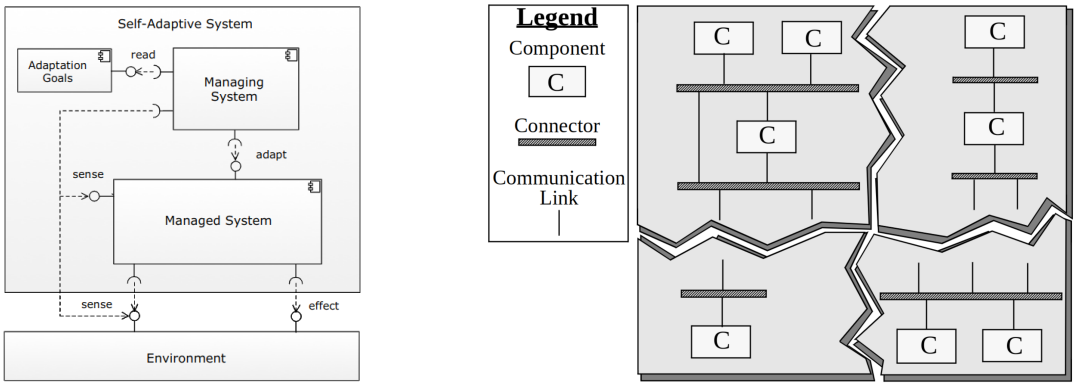
Chapter 2

Adaptable Industrial Runtime Environments

Contents

Self-adaptive architectures, such as the WATERBEAR architecture (introduced in Chapter 1), require adaptability. Once there is adaptability, the loop can be closed and open-loop adaptation can be transformed into closed-loop adaptation [37]. An example of this transition is established in the MAPE feedback loops [19, 44]. The *execution* phase of the MAPE model (see Section 1.2.1) is critical because this is where the adaptation is executed. Commonly, this adaptability is implemented in a reconfigurable middleware that actively assists the adaptation. Two examples are given in

Figure 2.1. Figure 2.1a demonstrates the benefits of splitting the managed from the managing system to achieve a separation of concerns. Instead of building one system that adapts itself, it may be simpler to implement one system that plans the change and one that is changeable. Regarding the latter, the component-based architecture in Figure 2.1b promotes adaptation by integrating connectors between components that simplify the replacement of components. Consequently, a reconfigurable middleware is crucial for the implementation of the WATERBEAR architecture in the domain of ICS to provide the necessary adaptability.



(a) Concept of a self-adaptive system from Weyns [53]

(b) Component and connector architecture for facilitated reconfiguration from Oreizy and Taylor [54]

Figure 2.1: Adaptation and reconfiguration concept architectures

ICS are using specialized hardware and domain-specific programming languages. For example, the IEC 61131-3 standard defines the most common programming languages for use in PLCs. The IEC 61499 standard extends these models towards distributed and reconfigurable control systems, yet has struggled with adoption [55]. Recently, there has been a push from both industry and academia for the continued integration of IEC 61499 models [56, 57]. Despite these efforts, IEC 61499 standard remains under scrutiny for its ambiguous execution semantics and the numerous diverging RTEs that, instead of promoting portability, inhibit it. Further, while there is an interface for reconfiguration, it is rarely used and at the very least struggles with the adaptation challenges of *simplicity* and *correctness*, since reconfiguration is a manual process that can easily lead to inconsistent behavior.

Certainly, after years of independent development, there is a need to reconnect the domain-specific models and languages back to the developments in general-purpose languages. This can effectively narrow the gap between IT and OT. One general-purpose language that provides some of the needed functionality and is

well-adopted in practice is Erlang. Originating in the telecommunication industry, it provides dependability and adaptability to high-availability communication systems. In this chapter, an IEC 61499 RTE is implemented in Erlang to, on the one hand, investigate Erlang's fitness for safety-critical real-time systems, and, on the other hand, to review Erlang's dynamic adaptation model for integration in ICS. For this purpose, a slice of the WATERBEAR architecture is implemented (Figure 2.2). Specifically, the target system layer is implemented with the ability to apply the *change implementation* that originates in the strategy enactment layer. The implementation allows the comparison between existing IEC 61499 RTEs and Erlang concerning their adaptation capabilities.

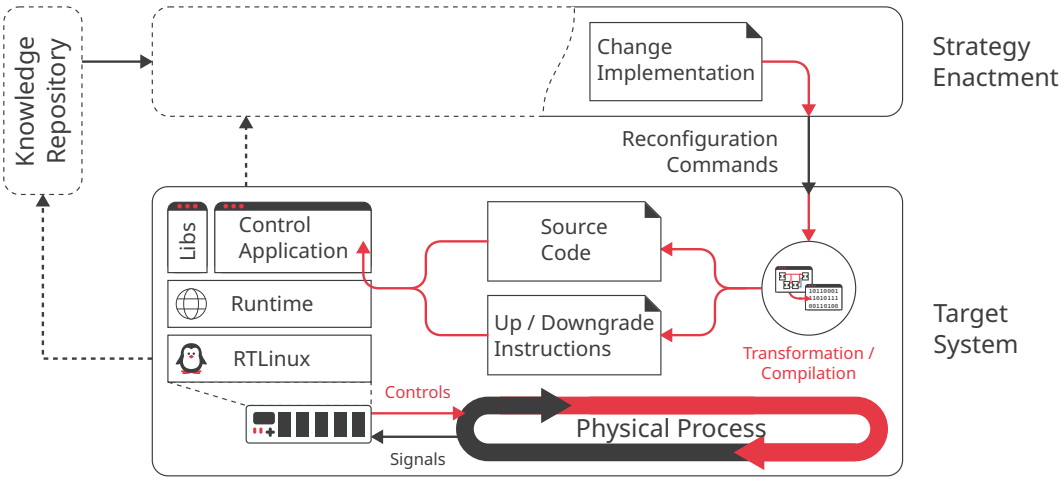


Figure 2.2: This slice of the WATERBEAR architecture focuses on the target system layer, which must use the change implementation from the strategy enactment layer to modify itself.

This chapter first introduces the background regarding general industrial control system (ICS) software, and in particular the IEC 61499 standard and its dynamic reconfiguration capabilities in Section 2.1. In the same section, the programming language Erlang is introduced which already has the desired dynamic adaptation features. Following this, a synthetic performance analysis is shown that evaluates the real-time performance of Erlang and the Erlang Runtime System (ERTS) (Section 2.2). This leads to the implementation and evaluation of an IEC 61499 RTE in Erlang in Section 2.3. The key findings are summarized in Section 2.4.

2.1

Industrial Control Software

The work in this chapter builds upon previous works in the domain of industrial control software. Compared to general software engineering, the domain-specific languages used in this field are particularly tailored towards real-time performance and the control of physical processes. After briefly introducing the elementary concepts of PLCs and the most common programming languages of the IEC 61131-3, Section 2.1.1 presents the fundamentals of the IEC 61499 standard for distributed and reconfigurable industrial control systems. Following this, reconfiguration capabilities of the IEC 61499 models (Section 2.1.3) and Erlang (Section 2.1.4) are established, which will be merged in the following section.

Programmable Logic Controllers

ICS are commonly controlled using PLCs, which replaced the relay control commonly used in the 1960s. The main reasons for this change were the lack of flexibility and the difficulty of troubleshooting [58]. Maintaining an electrical system was more cumbersome than maintaining software, and this advantage extends until today, when many of these systems can be controlled, monitored, and maintained remotely. PLCs are traditionally programmed using domain-specific programming languages, however, general-purpose programming languages are becoming more and more common. The use of domain-specific languages allows the use by engineers without a programming background and facilitates the implementation of control logic in a graphical manner.

Programming Languages: The IEC 61131-3 Standard

After the introduction of PLCs, the main difficulty was standardization between the different vendors and industries, since most vendors used customized languages with slightly different syntax and semantics. Generally, standardization facilitates reuse and portability and simplifies training and the availability of qualified personnel. The IEC 61131 standard is the first standard to reach international and industrial acceptance [59]. There are ten parts to the IEC 61131 standard:

- **Part 1: General information:** This introductory chapter provides definitions,

terms, and basic concepts that serve as the foundation for the subsequent parts.

- **Part 2: Equipment requirements and tests:** This chapter established the requirements and tests for programmable controllers and their peripherals, for instance, service conditions, functional requirements, functional type tests, and verification, or electromagnetic compatibility (EMC) requirements.
- **Part 3: Programming languages:** Since PLCs can be programmed in multiple languages, this part defines the three graphical and two textual programming language standards.
- **Part 4: User guidelines:** This technical report mainly caters to PLC end-users and introduces them to the IEC 61131 series and how to select or specify PLC equipment.
- **Part 5: Messaging service specification:** This part specifies how devices can communicate to a PLC and how PLCs can communicate with other devices.
- **Part 6: Functional Safety:** This part describes the life-cycle, requirements, and evaluation methods for a PLC to be used in a safety context, i.e., as a functional safety PLC.
- **Part 7: Fuzzy Control Programming:** This part provides a common understanding for manufacturers and end-users to integrate fuzzy control applications in the languages of the IEC 61131-3.
- **Part 8: Guidelines for the application and implementation of programming languages:** This technical report provides guidelines for the application of IEC 61131-3 and for the implementation of IEC 61131-3 languages for PLCs.
- **Part 9: Single-drop digital communication interface for small sensors and actuators (SDCI):** This part provides an extension to the IEC 61131-2 standard by specifying a single-drop digital communication interface.
- **Part 10: PLC open XML Exchange Format:** This part defines an XML-based exchange format for the export and import of IEC 61131-3 applications and projects, independent of vendors.

Looking closer at IEC 61131-3, this standard specifies the domain-specific modeling languages to be used to program PLCs. In no particular order, these languages are:

- **Sequential function chart (SFC):** A graphical programming language based on GRAFCET which itself is based on Petri nets and uses steps and transitions to structure the control flow.
- **Ladder diagram (LD):** Originally used to document the design and construction

of relay racks in manufacturing and process control, this graphical language represents a program based on circuit diagrams, which resemble ladders.

- **Function block diagram (FBD):** The third graphical language uses blocks with input and output variables to decompose the control logic. The blocks can be connected in countless configurations.
- **Instruction list (IL):** This low-level textual language resembles assembly code and uses jump instructions and function calls to achieve control flow.
- **Structured text (ST):** This high-level textual language syntactically resembles Pascal and provides the common statements and instructions found in most high-level programming languages, e.g. iterative loops, conditions, and functions.

Despite the rise of high-level general-purpose programming languages, these five domain-specific languages are still dominating the control domain [60]. The programs are structured in program organization units (POUs), such as programs, function blocks, or functions, which represent hierarchical blocks that can be used to encapsulate functionality. Each POU has a declaration part, where local variables can be declared, and a code part that contains the instructions for the POU [59]. PLC programs are most commonly executed cyclically using a *scan cycle*. This cycle consists of three phases:

1. Scanning the inputs
2. Executing the control logic
3. Updating the outputs

This simple execution simplifies the preservation of real-time constraints. This can be supervised by a watchdog that raises an alarm if a program exceeds its cycle time. The time-triggered execution facilitates the verification of the schedulability of the real-time behavior, yet it may present a disadvantage concerning utilization [61].

Many PLCs currently offer *online change* features. These features usually merely download the changed application and replace the old application. This type of adaptation works only on the syntax level but ignores the semantics of the change and can be considered a type of code replacement. Important challenges, such as consistency or timeliness are not addressed (see Chapter 1). These features provide a benefit for many applications but rely on the responsibility of the engineer applying the adaptation and lack V&V. Further, some PLC vendors offer solutions for "bumpfree switching", either for systems that must switch between automatic and manual

modes, or for high-availability solutions in which the failure of one CPU must be tolerated. In this case, a second CPU is running redundantly and synchronized via PROFINET or via optical fiber to achieve switchover times between 50 and 300 ms [3]. These switchover times are in the same range or longer as typical deadlines and could thus interrupt the real-time behavior of an application. This type of "fail-over" is suitable for emergencies, however, it still only considers changes to a single PLC and can not change a larger system.

The IEC 61131 defines the current state of practice in industrial control with PLCs. This state is first and foremost focused on robustness, stability, and cost-efficiency. While there are extensions to this standard, e.g., through object orientation ([62]), the main obstacle is handling legacy systems and dealing with the portability of legacy systems. This leads to simple execution semantics and limited reconfiguration support.

2.1.1 The IEC 61499 Standard[4]

The IEC 61499 standard defines an architecture for distributed control systems built on top of the languages defined in IEC 61131-3. It was introduced as a possible successor of the IEC 61131-3 standard for industrial control systems. Despite several architectural advantages, the standard is yet to be widely accepted by the industry. The distributed and event-driven concept allows for more flexible systems to tackle the upcoming challenges of the next decades such as the *internet of things* or *industry 4.0*.

Flexibility, reconfigurability, and distribution are some characteristics associated with this standard [55]. The main advantage is the encapsulation of independent functionality in function blocks without global states. This feature facilitates the reuse of function blocks as modules for many applications on different platforms. It allows the modification of the function block network without causing unexpected issues with seemingly unrelated subsystems [63] and it enables the dissemination of function

[3] https://www.siemens.com/de/de/produkte/automatisierung/systeme/industrie/sps/simatic-s7-1500/redundante-und-hochverfuegbare-cpus.html

[4] Major parts of this section were published in L. Prenzel and J. Provost. "Implementation and Evaluation of IEC 61499 Basic Function Blocks in Erlang". In: *International Conference on Emerging Technologies and Factory Automation*. Torino, Italy: IEEE, 2018 and L. Prenzel and J. Provost. "FBBeam: An Erlang-based IEC 61499 Implementation". In: *International Conference on Industrial Informatics*. IEEE, 2019.

blocks over large networks and resources, thus permitting real, physical distribution. In addition, the model-based approach lends itself to formal verification [64, 65]. On the other hand, several design and execution ambiguities prevent the full application of the IEC 61499 standard. [66].

The IEC 61499 standard currently has three parts: The first part describes the architecture for distributed systems. The second part introduces requirements for software tools, e.g., a Document Type Definition that describes an XML exchange format. Part three is currently withdrawn, and part four consists of compliance profiles for systems, devices, and software tools. Apart from the standard itself, Zoitl and Lewis [67] offer an extensive overview of the concepts of the IEC 61499 standard.

The standard defines models to specify a solution for a control problem. The main element is the function block (FB), which serves as a software component encapsulating functionality and data. It can be composed in large networks by linking the event and data connections. The set of function blocks describing the solution to a control problem is bundled in an application. The individual function blocks may be distributed over multiple resources and devices with the corresponding models.

The basic function block allows the manual implementation of custom algorithms. The execution of the algorithms is conducted by the Execution Control Chart (ECC). Triggered by incoming events, the ECC requests the execution of an algorithm and issues outgoing events. A schematic view of the basic function block is presented in Figure 2.3. The inputs are typically on the left, while the outputs are on the right side. Events are connected to the top part of the block, and data links are connected to the bottom part.

The IEC 61499 standard specifies multiple function block types that can be mixed in arbitrary combinations and provide a common interface. This allows the replacement of one FB by another, independent of their implementation. They also allow the hierarchical structuring of the control application and the introduction of I/O functionality. Compared to textual languages, these applications can be represented graphically which simplifies debugging and quick understanding, similar to the other programming languages of the IEC 61131-3 standard.

- **Basic FBs** implement a state machine that defines under what circumstances algorithms should execute and output events should be sent.
- **Service Interface FBs** may be used to implement I/O functionality.
- **Subapplications** contain a Function Block network and can be used to structure an Application hierarchically.

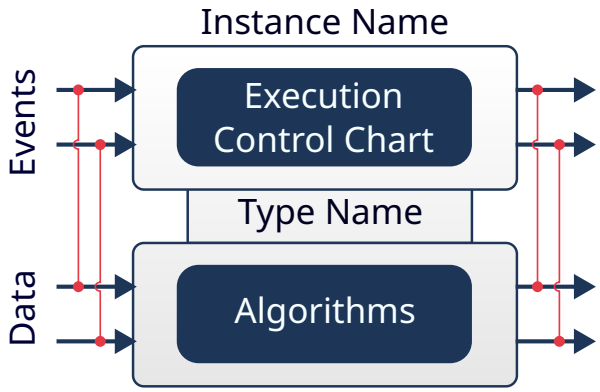


Figure 2.3: Events (top part of the function block) trigger the execution control chart (ECC) which controls the execution of the algorithms and the emission of output events. Data is updated when the events connected by the with-qualifiers are triggered.

- **Composite FBs** are similar to subapplications in that they contain a function block network, however, composite FBs have a different execution behavior.

In addition to the FB models, models of higher abstraction levels are used to describe the system and the control solution: The *Application* model contains a network of Function Blocks to solve a control problem. The *System* model contains devices and resources that the *Applications* are mapped to by the *Distribution* model.

There exist multiple possible implementations of the function block network. An overview is given by Ferrarini and Veber [68] and more recent developments are summarized by Vyatkin [69]. Common execution modes are sequential, cyclic, and parallel. This ambiguity in the execution semantics is further discussed in the following section.

2.1.2

Execution Semantics of the IEC 61499 Standard[5]

Since the introduction of the IEC 61499 standard, there has been a discussion about its execution semantics and possible ambiguities [66]. Most notably, [68] classified different execution semantics on a theoretical level. Thus, for researchers and commercial users of the standard, it is important to know the available execution semantics and

[5]Major parts of this section were published in L. Prenzel, A. Zoitl, and J. Provost. "IEC 61499 Runtime Environments: A State of the Art Comparison". In: *International Conference on Computer Aided Systems Theory*. Springer, 2019, pp. 453–460.

the most prevalent solutions. The different RTEs may be compared on different levels. There are *organizational characteristics*, such as the license of the project (open or closed source), the status (commercial, research, or inactive), or the programming language used for the implementation. The *execution semantics* may be described by the trigger mechanism (cyclic or event-based) and the execution model of the RTE. Finally, RTEs may be distinguished by the features they offer, such as real-time performance, multitasking, or dynamic reconfiguration.

Execution Models

The IEC 61499 standard does not strictly define the execution semantics of its models. This has led to several papers outlining these ambiguities [66, 70, 71]. Currently, there is no consistent framework to describe the execution semantics of an IEC 61499 implementation, but two different views are discussed here. Ferrarini and Veber [68] use the factors *Multitasking* and *Scan order* to describe seven groups of possible implementation approaches (see Table 2.1). The first factor describes if the order in which FBs are scanned is fixed or not fixed. The second factor characterizes whether multitasking is used, and if so, how it is controlled. This leads to a total number of eight combinations, but Ferrarini and Veber [68] exclude the case of a fixed scan order and not controlled multitasking.

		Multitasking Implementation			
		Not used	Not controlled	Controlled, time slice	Controlled, FB slice
Scan Order	Not fixed	A0	A1	A2	A3
	Fixed	A4	x	A5	A6

Table 2.1: Classification of possible implementation approaches according to Ferrarini and Veber [68] along the dimensions *Scan Order* and *Multitasking Implementation*.

In addition to this classification, many publications have introduced names for the most common implementation. The earliest model is arguably NPMTR (*Non-Preemptive Multi-threaded Resource*), which is employed in FBDK and mentioned already in 2006 [72]. At a similar time, a sequential model was discussed by Zoitl et al. [73] and Cengic, Ljungkrantz, and Akesson [70]. This model was later termed *Buffered Sequential Execution Model* (BSEM) [74]. Finally, Cengic and Akesson [71]

	Organizational			Execution			Features		
Name	Open / Closed Src	Research / Commercial	Language	Cyclic / Event	Execution Model	Ferrarini Model	Real-Time	Multitasking	Reconfiguration
Archimedes		R	Java C++	E	NPMTR	A1	Hard	Yes	Yes
FBBeam	O	R	Erlang	E	PMTR	A2	Soft	Yes	Yes
FBDK	C	R	Java	E	NPMTR	A1		Partly	Partly
4diac FORTE	O	R	C++	E	PMTR	A1	Hard	Yes	Yes
Fuber	O	R	Java	E	BSEM	A0		Yes	Yes
ICARU_FB	O	R	C	C	CBEM	A4	Hard	No	Yes
ISaGRAF	C	C	IEC 61131-3	C	CBEM	A4	Hard		
nxtIECRT	C	C	C++	E	PMTR	A1	Hard	Yes	Yes
RTFM-RT		R	C	E	PMTR	A1	Hard	Yes	

Table 2.2: Comparison of key characteristics of IEC 61499 RTEs found in research and industry.

termed a third model, named *Cyclic Buffered Execution Model* (CBEM).

Comparison of Semantics

As mentioned before, the IEC 61499 standard does not strictly define its execution semantics and thus different implementations are possible. This section presents a collection of RTEs that have been implemented since the inception of the standard. An overview of the comparison is displayed in Table 2.2. A total of nine different RTEs are compared based on information that was available from websites and publications.

In addition to the three execution models already introduced in the literature, an additional model (*PMTR*) was added. *NPMTR* describes non-preemptible multitasking resources. This explicitly excludes the preemptible multitasking resources, that nevertheless do not fall into the categories of buffered sequential or cyclic execution semantics. Thus, the name *PMTR* was chosen, to indicate the set of preemptible multitasking resources.

The assignment and collection were performed to the best of the author's knowl-

edge. Where no reliable data was found, and the clues were inconclusive, the field was left blank. Following, the nine RTEs are shortly presented.

Archimedes There are three different RTEs using similar execution semantics: RTSJ-AXE [75], RTAI-AXE [76], and Luciol-AXE [77]. They are implemented in Java and C++ and allow both reconfiguration and multitasking. FBs may be implemented as independent tasks/threads, or combined in function block containers.

FBBeam In this Erlang-based IEC 61499 RTE, every FB is implemented as a process, and scheduling is left to the Erlang Virtual Machine. Erlang processes do not share memory, and messages between processes are sent asynchronously. Because of the fair round-robin scheduling, only soft real-time performance can be guaranteed. Erlang includes sophisticated frameworks for distribution, dynamic reconfiguration, debugging, and monitoring of distributed, highly concurrent systems [4]. Its execution model may be best described by *PMTR*, since FBs may be preempted.[6]

FBDK FBRT The FBDK (Function Block Development Kit) and the accompanying FBRT (Function Block Runtime Environment) allow the modeling and execution of IEC 61499 systems in a Java-based RTE [78]. FBs are compiled into Java classes and scheduled in a depth-first manner. Instead of emitting events, the FBRT uses method calls to communicate between FBs [79]. The execution model of the FBRT was referred to as Non-Preemptive Multi-threaded Resource (NPMTR) [72].

4diac FORTE 4diac FORTE is the RTE provided by the Eclipse 4diac open source project [80, 81]. The implementation is based on C++ and uses the Event Chain concept described by Zoitl [50] to achieve deterministic real-time performance by allowing the introduction of real-time constraints for event chains. Execution of an event chain may preempt execution of other event chains, thus the execution model *PMTR* seems the most appropriate.

Fuber Fuber was build to investigate the different execution semantics of the IEC 61499 standard [70]. It executes in two threads: One for the execution of the execution control chart (ECC), and one for the scheduling of algorithms. FBs and algorithms are assigned to FIFO queues and algorithms are interpreted

[6] This RTE will be discussed in more detail in Section 2.3 and is mentioned here due to the non-chronological ordering of publications in this dissertation.

on the fly instead of static compilation, thus allowing modification of the algorithm code during the execution. As the focus of this implementation is research on the execution semantics, real-time performance is not considered. Fuber employs the Buffered Sequential Execution Model (BSEM), where FBs are put in a FIFO ready queue [74].

ICARU_FB ICARU_FB is a RTE for lightweight embedded systems, e.g. 8-bit Arduino boards. The IEC 61499 model is converted into C code. FBs are implemented as objects and events are passed directly to a variable in the destination FB object [82]. Since the execution is cyclic, and the FBs are scanned in a fixed order, the most appropriate execution model for this RTE is CBEM and A4. Hard real-time performance may be achieved and dynamic reconfiguration is available.

ISaGRAF ISaGRAF was the first commercial IEC 61499 implementation [83]. IEC 61499 FBs are compiled to IEC 61131-3 code that may be executed on traditional IEC 61131-3 devices. Because of the IEC 61131-3 base, the execution is cyclic instead of event-triggered. Its execution model is referred to as Cyclic-Buffered Execution Model (CBEM) [71].

nxtControl nxtIECRT According to [83], the solution provided by nxtControl, nxtIECRT, is based on the open source RTE 4diac FORTE. Thus, the execution semantics should mostly be identical. The nxtIECRT RTE is a hybrid runtime system, that may execute both IEC 61131-3 and IEC 61499 systems [84]. Furthermore, nxtIECRT provides extensive features for changing control applications during system operation.

RTFM-RT RTFM-RT is an IEC 61499 RTE built on the RTFM core language [85]. It uses the event chain concept and implements them as synchronous task chains [86]. The RTE is mostly built for real-time research. Threads of execution are preemptible and multitasking is possible, thus the model *PMTR* was assigned.

Discussion

Table 2.2 summarizes the findings of this section. Up until now, the IEC 61499 standard has been implemented numerous times with different execution semantics. Most RTEs are open source and research projects, but there are at least two

commercially available IEC 61499 RTEs. Both of them do not only implement the IEC 61499 model but support also the languages of the IEC 61131-3 standard. The implementation languages vary but are mostly focused on Java and C / C++.

All RTEs except for two employ an event-triggered execution. Using the classification introduced by Ferrarini and Veber [68], most RTEs employ the semantics A0, A1, or A2, where no fixed scan order exists. ISaGRAF and ICARU_FB are the only implementations with a fixed scan order, falling into the category A4. To the knowledge of the authors, the categories A3, A5, and A6 are currently not used, i.e., there are no RTEs with a fixed scan order and multitasking, or RTEs using FB slice multitasking. For categories A5 and A6, this may be because a fixed scan order with multitasking can be contradictory, since a multitasking implementation by itself may disturb a fixed scan order. If the next FB in the fixed scan order must wait for the previous FB to finish, multitasking is not possible. If it does not have to wait for the previous FB to finish, this would disturb the determinism of a fixed scan order implementation, since the previous FB might want to send events to the next FB in the scan order. For A2 and A3, only one implementation currently exists, that uses a fair scheduler with time slice preemption. Most other implementations do not prescribe the scan order, and either do not use multitasking or do not control it.

Since the standard is aimed at industrial process measurement and control systems, most implementations claim to offer hard real-time performance. Multitasking is available in some RTEs but not all. Although dynamic reconfiguration has been the topic of multiple research papers, and many RTEs seem to support it, information about the usability or performance of the reconfiguration process is rare.

Conclusion

This section summarized the developments concerning IEC 61499 RTEs for users and researchers alike interested in working with standard or wanting to implement their own RTE. Since its introduction, the standard has been implemented multiple times. Despite the ambiguities of the execution semantics, there exist both commercial and research RTEs that may be used to control physical systems.

From a theoretic perspective, the existing and possible execution models call for a deeper investigation. The current classification frameworks help distinguish fundamental differences between the RTEs but fail to describe the different execution models of the standard precisely. Given that the execution semantics of the standard

have room for interpretation, it is even more important to differentiate between the implementations.

Given the availability of lightweight, multitasking embedded systems that require real-time performance, the IEC 61499 standard may offer suitable models for this application. In this regard, deterministic real-time scheduling of multitasking IEC 61499 systems may require further investigation.

Although the topic of Dynamic Reconfiguration has been addressed from a modeling perspective, and many RTEs claim to allow dynamic reconfiguration, examples of dynamic reconfiguration with the IEC 61499 models are rare. Most RTEs focus on the execution semantics, whereas the frameworks for deployment, distribution, configuration, and reconfiguration are also key selling points of the IEC 61499 standard.

2.1.3 Dynamic Reconfiguration of IEC 61499 Standard

Dynamic reconfiguration was integrated into the IEC 61499 standard from the very beginning since adaptability and reconfigurability were key requirements during the development [50]. A detailed description of a real-time execution model for the IEC 61499 standard, which also supported dynamic reconfiguration was developed by Zoitl [50]. However, while the model was implemented in current RTEs and the real-time execution is at least partially available, dynamic reconfiguration is currently rarely applied.

Reconfiguration Services

The IEC 61499 standard includes a set of reconfiguration services to be used for dynamic reconfiguration. These services are provided by the RTE and can also be accessed via special service interface function blocks, which allow the design of the reconfiguration logic in the same language as the control logic.

An exhaustive list of reconfiguration services is given by Zoitl [50]. The services are summarized in Table 2.3. Some services are implemented in current RTEs such as 4diac FORTE [81]. Generally, these services can be split into two categories: Services to interact with FBs and the FB network (i.e., CREATE, DELETE, READ, WRITE, START, STOP, KILL, RESET), and services to communicate with the RTE

(i.e., QUERY).

Service	Description
CREATE	Allows the creation of resources, function blocks, connections, or types.
DELETE	Allows the deletion of resources, function blocks, connections, or types.
WRITE	Enables the writing of parameters, data inputs, data outputs, and internal variables.
START	Sets the state of the function block to *Running*.
STOP	Sets the state of the function block to *Stopped*
KILL	Interrupts the execution and sets the state to *Killed*.
RESET	Resets a stopped or killed function block back to the initial state.
READ	Read data inputs, data outputs, and internal variables of function blocks.
QUERY	Request information from the RTE regarding resources, function blocks, function block states, connections, and loaded types.

Table 2.3: IEC 61499 reconfiguration services according to [87] and [50].

Theoretically, these services should suffice to achieve any higher level reconfiguration task [50]. For convenience, it may be desirable to combine multiple of these low-level services into high-level services, e.g., for updating a FB.

Reconfiguration Control Applications

The topic of dynamic reconfiguration of IEC 61499 models has been discussed for a long time [88, 89]. It is facilitated by the event-triggered execution and the fractured system state which allows the application to be exchanged efficiently without halting the entire application, and complex changes are possible. Typically, the reconfiguration services are wrapped inside a reconfiguration control application (RCA) which modifies the concurrently running control application in real-time. The process of reconfiguration can thus be split into five phases: The upload of the RCA, the initial RINIT phase, the critical RECONF phase, the RDINIT phase, and the final cleanup of the RCA [90].

A picture of an RCA (or Evolution Control Application) is shown in Figure 2.4. As can be seen, the RCA is larger than the control application, and the RCA does not contain any fail-over or recovery mechanisms. As RCAs are usually implemented manually, this obstructs the use of dynamic reconfiguration in practice. Further, the

large number of necessary connections means that the implementation is error-prone and cumbersome.

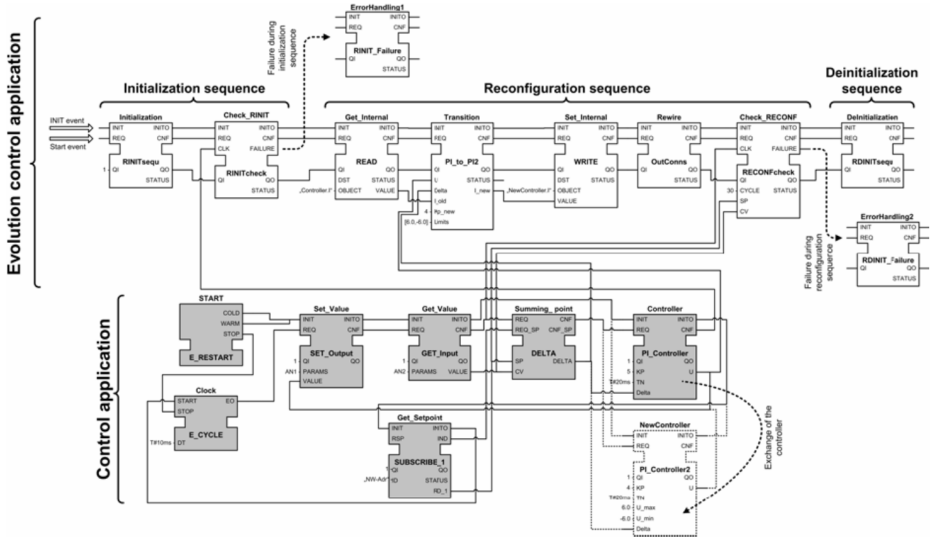


Figure 2.4: An example RCA as given by Sünder, Vyatkin, and Zoitl [90].

While the IEC 61499 standard provides the necessary means to achieve dynamic adaptation, in its current form, it is far from simple or even feasible. The cost of implementing the necessary RCA and the needed validation dwarf the benefit of adaptability. Further, there are no guarantees of consistency or real-time behavior, thus this lies within the responsibilities of the developer or must be checked during validation.

2.1.4

Erlang and the Erlang Runtime System

The previous sections have introduced the fundamentals of industrial control software. Being rather traditional, the adoption of new technologies has been slow. The IEC 61499 standard is slowly gaining traction, yet dynamic reconfiguration support is not a first-class citizen within the existing RTEs. Most importantly, even once the IEC 61499 models would see more use in practice, the dynamic reconfiguration support that currently exists is rudimentary and a purely manual process. In contrast, other domains already have decades of experience with dynamic adaptation and the surrounding ecosystem and tooling.

This section introduces Erlang, a general-purpose programming language, in-

tended for the telecommunication industry, that features dynamic adaptation, or hot code loading, to achieve high availability. While some sources claim an availability of "nine nines" (99.9999999%), this statistic has been critized [91].

Erlang[7]

Erlang is an open-source functional programming language with roots in the telecommunication industry. It was developed for distributed, highly available, and concurrent systems. Over time, there have been many changes and improvements, that lead Erlang to the point where it currently is. With the introduction of Elixir[8] in 2012, there is at least one other language with wider adoption using the underlying virtual machine of Erlang. Since its inception in 1986, many books have been written about Erlang that provide great resources on the language, semantics, and application [92–96]. This section provides an elementary introduction to the background necessary to understand the methodology of this dissertation. Erlang is relatively fast to learn and allows beginners to quickly delve into concurrent and fail-operational system design. From the many available resources, [96] provides a great starting point.

The Erlang ecosystem can be split into three components that all contribute to the advantages of using Erlang in the first place. These components are:

- The functional programming language, which allows for predictable syntax that is largely side effect-free,
- the Open Telecom Platform (OTP), a collection of high-level behaviors and templates that simplify the implementation of concurrently and fault-tolerance, and
- the Erlang Runtime System (ERTS), the runtime system that provides the necessary functionality to run Erlang systems and builds on top of the BEAM virtual machine.

Erlang Programming Language

Erlang is a functional programming language, in contrast to imperative languages such as C. This comes with several differences that can have advantages or may

[7] Major parts of this section were published in L. Prenzel and J. Provost. "FBBeam: An Erlang-based IEC 61499 Implementation". In: *International Conference on Industrial Informatics*. IEEE, 2019.

[8] https://elixir-lang.org/

constitute a disadvantage in some cases. Three characteristics of Erlang are presented here: Immutable variables, functions, and the actor model.

Immutable Variables Variables in Erlang are immutable. Once assigned, the value of a variable cannot change within its scope. This provides predictable behavior of code and prevents unexpected side effects during execution. On the other hand, immutable variables prevent the use of for-loops. Instead, iteration is solved by recursive function calls.

```
Eshell V12.3  (abort with ^G)
1> Var = 1.
1
2> Var.
1
3> Var = 2.
** exception error: no match of right hand side value 2
```

Listing 1: After assigning the variable `Var`, the value can be returned, but it cannot be changed again, since variables in Erlang are immutable.

Functions As a functional programming language, all code in Erlang is organized into functions. Functions have a name, receive some inputs, perform computations, and return an answer. An example function is given in Listing 2.

```
countdown(0) ->
    ok;
countdown(N) when N > 0 ->
    countdown(N-1);
countdown(_) ->
    ok.
```

Listing 2: An example *countdown* function that counts down until 0 and then returns `ok`.

This single function uses pattern matching to differentiate between different inputs, i.e., `0`, `N when N>0`, or the anonymous variable _. Depending on which input is received, a different function is called. Since variables are immutable, their values cannot be changed once they are assigned. However, a different variable can be returned that has updated the value of the variable. In this function, counting down is achieved by the recursive call of the function until the stop condition is reached. A tremendous advantage of this functional programming paradigm is that the function

is essentially side effect free, i.e., every function clause can be tested individually, and the resulting behavior only depends on the function inputs.

Actor Model All Erlang code runs in processes. For instance, when an Erlang shell is started, the shell runs in an independent process to execute commands or functions. When an exception occurs, only the process is terminated but the virtual machine survives. A new process can be created using the `spawn(Module, Name, Args)` function, as seen in Listing 3, where a process is spawned executing the `countdown(N)` function. The `start()` function immediately returns the process identifier (PID) of the newly spawned process and does not have to wait for the `countdown(1000)` call to finish.

```
1 start() ->
2     Pid = spawn(?MODULE, countdown, [1000]),
3     Pid.
```

Listing 3: The `spawn(Module, Name, Args)` function creates a new process and returns the PID of the new process.

The following section introduces the Erlang Runtime System (ERTS) and discusses the process behavior in more detail.

Erlang Runtime System (ERTS)

The Erlang Runtime System (ERTS) is the runtime system that contains all the necessary functionality to run Erlang systems. It is not the same as the BEAM, which is the virtual machine that executes the user code within the ERTS.

Processes Code is executed in processes, which themselves are blocks of memory encapsulating the state and protecting the virtual machine from errors. The virtual machine is optimized for highly concurrent and available systems. Computation time is allocated fairly and dynamically over the currently executable processes. Executable processes receive a time slice measured by a reduction count (number of function calls) of 4000 before they are preempted.

The execution order of processes is defined with priorities. Processes are put into run queues according to their priority when they are ready to execute (see Figure 2.5). There are three run queues per scheduler: *Max* and *high* priorities have separate run queues, while *normal* and *low* priority processes share a run queue. Processes are taken from the run queue in a FIFO manner, except for low-priority processes, which have to reach the top of the run queue eight times before they are executed. The max

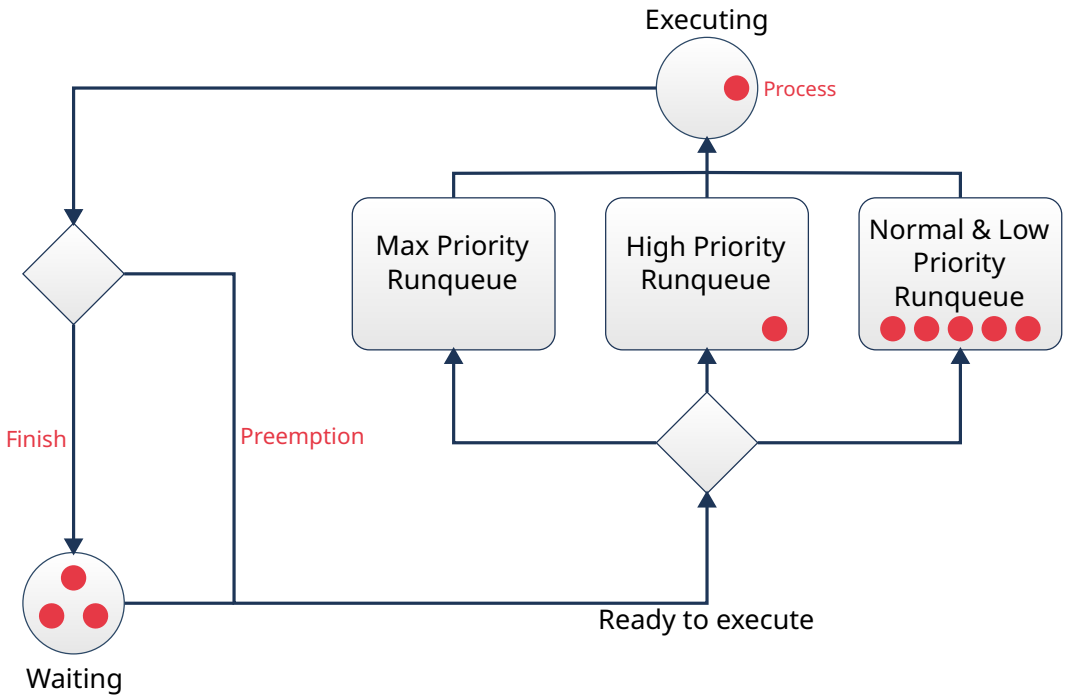


Figure 2.5: Erlang processes transition through the run queues to receive execution time depending on their priorities.

run queue precedes all other run queues when processes are rescheduled, and is reserved for system processes. The high priority is available to the user, although it should be used carefully, because it may lead to blocking and process starvation.

A process is only interrupted if its *reduction count* is consumed; although it may finish early. The reduction count is a variable that measures the computation work that was performed. A high-priority process is only executed after the currently running process has yielded—in the worst case after the consumption of the full reduction count. Natively-implemented functions (NIFs), which are coded in C and called from an Erlang process, are not preempted by default, and can thus lead to the blocking of the scheduler.

Erlang may use more than one scheduler to allow multitasking. During startup, one scheduler may be spawned for every available CPU thread, and the load is distributed dynamically. Every scheduler has its own set of run queues and assigned processes. There are two load distribution paradigms built into the ERTS: *Load balancing* and *load compaction*. Load balancing balances the load evenly over all available schedulers. Load compaction (default) fully utilizes the smallest number of schedulers to allow hibernation of idle schedulers. Processes are transferred

between schedulers based on *task-stealing* and *task-migration*: Idle schedulers steal processes from busy schedulers, and periodically, all schedulers redistribute their work according to a migration plan [97].

Garbage Collection The virtual machine performs garbage collection per process. This permits high responsiveness of the system since there are no long periods of garbage collection that block the entire virtual machine. Generally, Erlang produces a lot of garbage due to its immutable variables that are frequently discarded. There are *full-sweep* (major) and *generational* (minor) garbage collection runs [98]. In a full-sweep run, both the old and new heap are garbage collected. In a generational run, only the new heap is garbage collected. This follows the idea that an object that has survived several garbage collection runs will most likely survive for much longer before being discarded [97].

Open Telecom Platform (OTP)

The Open Telecom Platform (OTP) is a collection of applications and behaviors that facilitate the implementation of common system architectures. The most important concepts are:

Packaging All modules can be packaged into applications and releases that enable the version management and deployment of the implemented software. This also allows the management of dependencies between applications and facilitates the administration of distributed applications with dozens or hundreds of nodes running in parallel.

Behaviors Common process behaviors are abstracted into reusable templates. For instance, there is a behavior for the implementation of a server in a client-server pattern. Consequently, only the relevant logic inside the server must be implemented, whereas the available communication patterns for synchronous and asynchronous messaging can be reused.

Supervisors Similarly, there are patterns for the creation of supervisor processes, whose job is the management of other processes. These supervisors receive messages when a child process terminates, and can immediately start a new child. Similarly, they are used to start all child processes during startup. When a non-recoverable error occurs, the supervisor can terminate itself and notify the supervisor above it in the supervision tree. In this manner, a hierarchical fault-tolerant supervision tree can be created, that can tolerate multiple layers

of failures before the application fails. Still, an application failure does not crash the virtual machine.

This ecosystem allows the implementation of highly reliable systems with multiple layers of redundancy and fault tolerance. In addition, the actor model allows the monitoring and debugging of individual processes that provide transparency as to why a failure occurs.

Erlang Execution Model[9]

The internals of the virtual machine are sparsely documented and subject to changes and optimizations. The most thorough resource is [97]. Parts of the scheduling were already explained in the previous pages. This section focuses on the scheduling semantics from the point of view of a process.

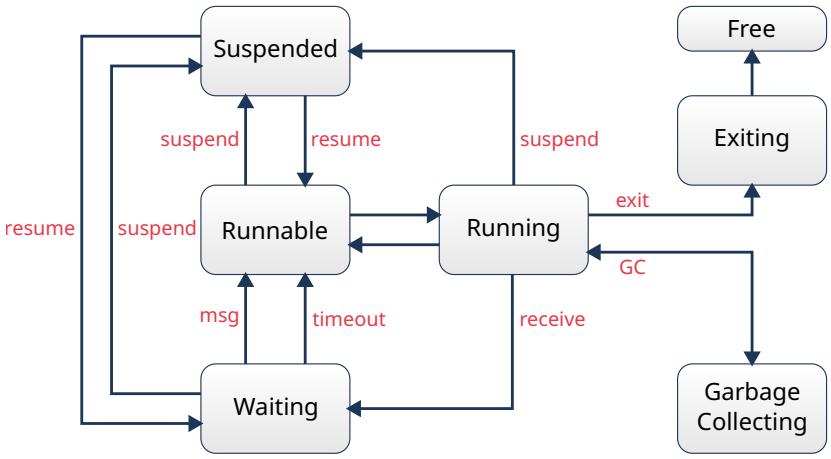


Figure 2.6: State machine depicting the operation of an Erlang process [97]

Figure 2.6 shows the state machine of an Erlang process. The blue elements represent the typical cycle during execution. Processes are waiting, become runnable because of a message or a timeout, and are eventually scheduled. Messages are stored in a mailbox inside the recipient's heap, or, if the recipient is locked, in a separate heap fragment.

The processes are scheduled based on run queues, which they enter once they are *runnable*. The behavior of the run queues is described in the previous section. Processes are taken from their run queues in a first-in-first-out order. The process may

[9]Major parts of this section were published in L. Prenzel and J. Provost. "FBBeam: An Erlang-based IEC 61499 Implementation". In: *International Conference on Industrial Informatics*. IEEE, 2019.

run for 4000 reductions before it is preempted, and this count includes garbage collection. Generational garbage collection per process is triggered when the combined heap and stack size exceeds the current limit for this process. Either sufficient garbage is cleared, or the limit is increased. Priorities affect the order in which processes are taken from the run queues. A currently running process cannot be interrupted until it has reached its reduction count or has yielded.

This type of execution is optimized for telecommunication devices. Processes communicate asynchronously and receive a fair amount of execution time, which leaves the system responsive even under high load. This behavior was observed in [2], where the reaction time of an IEC 61499 FB in Erlang under high load was analyzed. Depending on the computational effort of handling a message, a process may consume multiple messages during one scheduling without context switches.

Formally, the execution of the ERTS follows a combination of round-robin, priority, and first-come-first-serve scheduling. On the highest abstraction level, priorities define the order of execution. These priorities may lead to the starvation of lower-priority processes, which is commonly undesirable. Within each priority queue, processes are scheduled in a first-come-first-serve manner. Only currently runnable processes enter the queue, while blocked processes will only enter once they are runnable. In this way, numerous processes can be executed "concurrently", e.g., there can be a million processes at a time of which only hundreds are runnable at a given moment, and only one is executed at once. On the lowest level, processes are preempted if they run for too long and deferred to the back of the run queue. This prevents blocking and enables *fair* scheduling, where every process can receive an equal amount of time to execute.

Hot-Code Reloading with OTP

The documentation for the hot-code reloading procedure can be found in the Erlang *System Architecture Support Libraries (SASL) Reference Manual* [99]. Further examples of how to use hot code reloading in practice are found in [96].

Generally, the procedure relies on the ability of the virtual machine to load new code during runtime to replace existing modules. Then, processes running a function in these modules are simply switched over to the function in the new module. In more complex cases, two versions of the same module can coexist within the same virtual machine, and processes switch over in specific states. Since the

process state is explicitly stored in the function arguments, transitioning to a new version is as simple as executing the new function with the old arguments. This basic functionality is wrapped and extended into a sophisticated framework that facilitates the implementation of adaptations. This also takes care of dependencies between processes, modules, and applications and effectively supports the end user in the implementation.

Appup and Relup

Apart from the elementary, manual upgrade of processes, Erlang permits the up- and downgrade of applications and releases in a structured manner. For applications, this process is termed *Appup*, and for releases *Relup*. Both processes rely on a textual description of how the application/release can be upgraded or downgraded. The syntax for an Appup is shown in Listing 4.

```
1 {Vsn,
2     [{UpFromVsn, Instructions}, ...],
3     [{DownToVsn, Instructions}, ...]}.
```

Listing 4: Erlang up- and downgrades are specified in Appup files as lists of instructions.

This single Erlang term contains the information on how the version `Vsn` can be reached. There can be a list of instructions (`Instructions`) for every version from which it can be reached by upgrade (`UpFromVsn`) and one list for every version it can be left into by downgrade (`DownToVsn`). Thus, whenever a new application is deployed, this application specifies how it can be reached (by upgrade) or left to a previous version (by downgrade) explicitly. This facilitates the handling of multiple parallel versions in large systems with dozens or hundreds of nodes.

The `Instructions` list is a list of high- or low-level instructions that describe the up-or downgrade procedure. These instructions are presented in the following paragraphs.

Low-level instructions in the up- or downgrade instructions inform the ERTS how to perform the modification. They represent the underlying services needed to change the application as desired.

`{load_object_code, {App, Vsn, [Mod]}}`

Read all modules `Mod` as a binary without loading the module. This should be placed first to shorten the suspend-load-resume cycle.

`point_of_no_return`
: Indicate the point of no return, after which a recovery is no longer possible.

`{load, {Mod, PrePurge, PostPurge}}`
: Load the module

`{remove, {Mod, PrePurge, PostPurge}}`
: Mark the current module version as *old*.

`{purge, [Mod]}`
: Remove all *old* modules and kill processes running *old* code.

`{suspend, [Mod | {Mod, Timeout}]}`
: Suspend all processes running module `Mod`.

`{resume, [Mod]}`
: Resume all processes running module `Mod`.

`{code_change, [{Mod, Extra}]}`
: Send a `code_change` message to all processes running module `Mod`.

`{stop, [Mod]}`
: Stop all processes using module `Mod` through their supervisor.

`{start, [Mod]}`
: Start all stopped processes using module `Mod` through the supervisor.

`{sync_nodes, Id, [Node]}`
: Synchronize with other nodes `Node`.

`{apply, {M, F, A}}`
: Apply a function.

`restart_new_emulator`
: Restart the emulator in case a system application is upgraded.

Commonly, when developing an Appup, high-level services are used that use the low-level services underneath. For example, a synchronous code replacement will suspend, update, and resume all processes running a specific module. Instead of implementing the low-level services individually, a high-level service can be used.

`{update, Mod, Options[...]}`
: Synchronized code replacement for a single module or supervisor `Mod` using suspension. Suspends all processes running `Mod`.

`{load_module, Mod, Options[...]}`
: Simple code replacement for a single module `Mod`.

`{add_module, Mod, Options[...]}`
: Loading a module `Mod` for the first time.

`{delete_module, Mod, Options[...]}`
: Deleting a module `Mod`.

`{add_application, Application, Options[...]}`
: Adding another application `Application` that the application depends on.

`{remove_application, Application, Options[...]}`
: Removing another application `Application` that the application depends on.

`{restart_application, Application, Options[...]}`
: Removing another application `Application` that the application depends on.

If during the up- or downgrade an error/exception occurs, it may be possible to recover the operation. In Erlang, the `point_of_no_return` instruction indicates the time after which recovery is no longer feasible. Typically, it is placed after the `load_object_code` instructions, i.e., after the new code has been added, but no functional change has occurred [99]

2.1.5

Conclusion

This section introduced Erlang and the surrounding ecosystem. The virtual machine and its semantics were presented, and how dynamic adaptation can be performed was outlined. From this point, the difference to, for instance, the IEC 61499 models is already evident: Most effort in Erlang is spent on how to achieve reliability, transparency, and fault-tolerance practically and efficiently. The use of the virtual machine and supervision trees allows failures to be tolerated and means that there is always a backup plan available. Whereas most industrial control software specifies that there shall be no errors, Erlang accepts that bugs are inevitable and must be tolerated to achieve maximum availability of the system.

2.2

Synthetic Performance Evaluation[10]

As the previous section outlined, there are existing IEC 61499 RTEs, some of them implemented in languages like C. In contrast, Erlang provides a virtual machine, implemented in C, to be used in reliable and highly concurrent soft real-time applications. Thus, why not implement an IEC 61499 RTE in Erlang? The most obvious reason is that Erlang only offers soft real-time performance and cannot guarantee the hard real-time requirements. Further, it is not clear if an IEC 61499 implementation can be achieved efficiently, and if the syntax and the semantics of the IEC 61499 models are compatible with Erlang.

An obvious solution is to try it out and implement the key component of the IEC 61499 models, the basic function block, in Erlang. During this implementation, the syntax and semantic differences can be unveiled and finally, the real-time performance of this implementation can be investigated.

The section analyzes the syntax and semantics of the IEC 61499 basic function block and presents a prototypical implementation in Erlang. This implementation is evaluated concerning its real-time performance in a synthetic benchmark to understand the advantages and disadvantages.

2.2.1

Function Block Syntax and Semantics

The general execution semantics of the IEC 61499 standard were described in Section 2.1.2. After introducing the fundamentals of Erlang, this section describes the syntax and semantics of the IEC 61499 standard relevant for implementation in Erlang.

IEC 61499 Basic Function Block Syntax

A basic function block is defined by its type and by the instance properties. The type is defined separately from the instance definition, which can be found in the FB

[10]Major parts of this section were published in L. Prenzel and J. Provost. "Implementation and Evaluation of IEC 61499 Basic Function Blocks in Erlang". In: *International Conference on Emerging Technologies and Factory Automation*. Torino, Italy: IEEE, 2018.

network. The FB instance is defined by the following properties:

- The name of the type and an unambiguous instance name
- Connections to and from the event inputs and outputs to and from other FBs
- Connections to and from the data inputs and outputs to and from other FBs

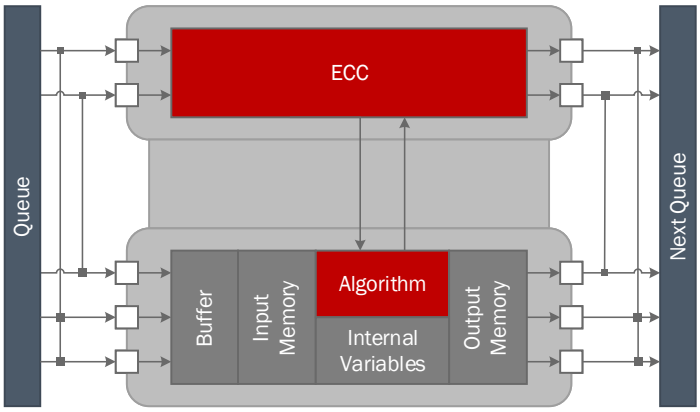


Figure 2.7: IEC 61499 Function Block Layout

The general contents of the basic function block type are visualized in Figure 2.7. The characterizing elements are:

- An execution control chart (ECC), that controls the event-triggered execution
- A set of algorithms, triggered by the ECC
- A set of internal, input, and output variables, that hold the internal state of the FB
- A set of *with* qualifiers that define when data inputs are sampled and when data outputs are sent

The ECC specifies the relationship between input events, algorithms, and output events in the form of a Moore-type finite state machine. At the occurrence of an input event, the available transitions are evaluated. When a new state is entered, the corresponding algorithms are executed and outgoing events are sent. The transitions may have guards using the available variables. The algorithms have access to the variables and can calculate new output and internal variables. The IEC 61499 standard does not strictly specify the language for the algorithms. The most common choice is the use of IEC 61131-3 languages, most importantly structured text. This imperative programming language is well-suited for simple algorithms. In the scope of the

IEC 61499 models, algorithms should be brief and terminate quickly. Long-running algorithms block the execution of other FBs and conflict with the event-triggered execution. The set of variables holds the input-, output-, and internal variables. On the occurrence of an input event, input variables connected to this event by the *with*-qualifier are updated. When output events are sent, the corresponding data outputs are propagated to the connected function blocks. Internal variables are only updated by the algorithms.

IEC 61499 Basic Function Block Semantics

Within the IEC 61499 application, the FB is embedded in a FB network. The execution of the FB is decided by the network and whatever execution semantics the RTE is using to schedule the transmission of events, as seen in Section 2.1.2.

Once the event has been scheduled to arrive at the FB, first, the with-qualifiers have to be used to update the input data connected to the event. Then, the ECC can be triggered, which decides about the execution of algorithms and the emission of outputs. The algorithms may change output variables, even if they are not sent yet. Thus, they also must be buffered. Only when the corresponding output event is sent, is the data from the output buffer transmitted.

2.2.2 IEC 61499 Basic Function Block Implementation

An approach to automatically generate an Erlang system from an IEC 61499 application is described in [1]. This approach served as a proof of concept for updating an automatically generated IEC 61499 application, but not all features of the basic function block were supported. This section focuses on the full implementation and evaluation of a basic function block with data connections and functional algorithms.

Kruger and Basson [100] show an implementation of a resource holon in Erlang/OTP, although unrelated to the IEC 61499 standard, and conclude that Erlang is well suited due to its modularity, scalability, customizability, maintainability, and robustness characteristics.

Since Erlang is a functional programming language, the function block is implemented as a set of functions. To simplify this, the OTP behavior `gen_statem` for finite state machines is used. The implementation makes use of *records* (a key-value

construct) and is fully specified with type hints, allowing the use of the dialyzer to check the type safety [101]. Displaying the full implementation would be out of the scope of this section, but several code snippets are shown to illustrate the implementation.

There are two types of functions to be implemented: Generic functions that are the same for all possible basic function blocks (such as `handle_event`), and functions that are specific to a certain FB type. To initialize the instance of a FB type, a set of variables is passed to it during startup.

Generic Functions

The `gen_statem` behavior expects generic functions for initialization, startup, termination, update, and message handling. Termination and update functions can be used for possible fault tolerance mechanisms or dynamic software updating.

```
% Handling data messages to be stored in the buffer.
handle_event(cast, #msg{type=data, name=Name, value=Value},
             State, {ID, CON}) ->
    NewID = upd_buf(Name, Value, ID),
    {next_state, State, {NewID, CON}};
% Handling events that trigger the ECC.
handle_event(cast, #msg{type=event, name=Event},
             State, {ID, CON}) ->
    NewIM = sample_inputs(Event, ID#id.buf, ID#id.im),
    UpdID = makeID(ID#id.name, ID#id.buf, NewIM,
                   ID#id.iv, ID#id.om),
    {NewState, NewID} = ecc(Event, State, UpdID, CON),
    erlang:garbage_collect(), %OPTIONAL
    {next_state, NewState, {NewID, CON}}.
```

Listing 5: The FB process can receive two types of messages: Data and events.

The functions for handling messages are depicted in Listing 5. Mainly, two types of messages are expected: Data messages and event messages. Data messages will update the FB internal buffer. Event messages trigger the sampling of inputs and the ECC.

The ECC function is shown in Listing 6. This recursive function triggers the internal state machine and reacts according to whether or not a transition was taken. If a transition is taken, algorithms are executed, data and events are sent, and the ECC is triggered again. If no transition is performed, the recursive call ends and

```
ecc(Event, State, ID, CON) ->
    case statemachine(Event, State, ID) of
% No transition was taken.
        no_transition -> {State, ID};
% A transition was taken, but the event was not used.
        {no_event, NewState} ->
            {NewIV, NewOM} = alg(NewState, ID),
            send_do(NewState, NewOM, CON#con.do),
            send_eo(NewState, CON#con.eo),
            NewID = makeID(ID#id.name, ID#id.buf, ID#id.im,
                           NewIV, NewOM),
            ecc(Event, NewState, NewID, CON);
% A transition was taken and the event was used.
        {event, NewState} ->
            {NewIV, NewOM} = alg(NewState, ID),
            send_do(NewState, NewOM, CON#con.do),
            send_eo(NewState, CON#con.eo),
            NewID = makeID(ID#id.name, ID#id.buf, ID#id.im,
                           NewIV, NewOM),
            ecc(no_event, NewState, NewID, CON)
    end.
```

Listing 6: This Erlang implementation of an ECC calls itself recursively until no more transition can be taken.

returns the internal FB state.

Figure 2.8 displays the structure of the `handle_event` function that is called whenever a new message is picked from the mailbox. Depending on the type of message, different sequences are followed. For data messages, the buffer is updated, and the function returns. For event messages, the ECC is called after the inputs are sampled. The ECC itself first calls the state machine to find executable transitions. If no transition can be triggered, the ECC returns. If a new state is entered, the corresponding action for this state is performed, i.e., algorithms are executed and data and events are distributed to other function blocks. Finally, the ECC is called again to find available transitions without events. This recursive call will run until no transition can be fired anymore. Consequently, by design, the ECC should not contain live locks and must terminate eventually.

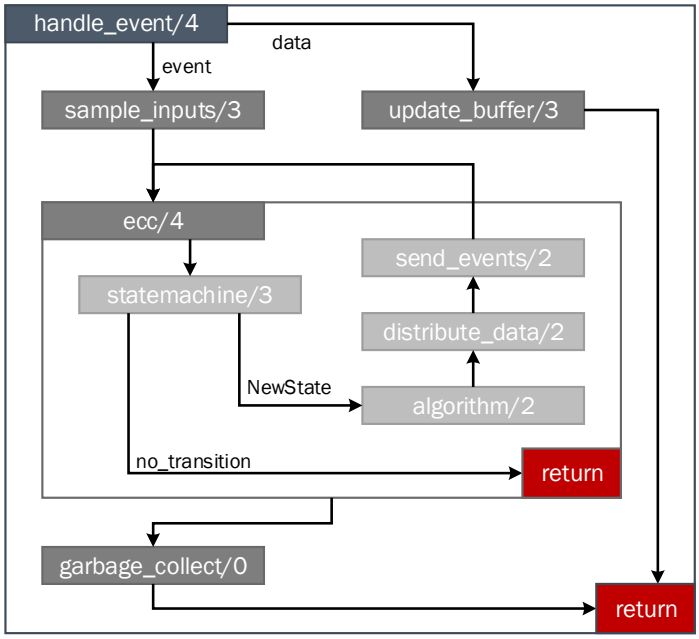


Figure 2.8: IEC 61499 Function Block Timing

Function Block Specific Functions

While the `handle_event` and `ecc` functions are the same for every basic function block type, the functions to sample the inputs, update the buffers, call the state machine, execute the algorithms, distribute the data, and send the events are specific to the FB type. That means they have to be generated from the FB type definition. Their implementation in Erlang is straightforward, as can be seen in the examples provided in Listing 7.

```
% Statemachine function that decides whether a transition was
% executed, and if the transition used the event.
statemachine('REQ', 'START', _ID) -> {event, 'State'};
statemachine(_, 'State', _ID) -> {no_event, 'START'};
statemachine(_Event, _State, _ID) -> no_transition.

% Algorithm function that executes the algorithms in a given
% state.
alg('State', #id{im=IM, iv=IV0, om=OM0}) ->
    {IV1, OM1}=alg_ALG(IM, IV0, OM0),
    {IV1, OM1};
alg(_State, #id{im=_,iv=IV, om=OM}) -> {IV, OM}.
```

Listing 7: Statemachine and algorithm implementation in Erlang

Every transition is defined by the event, the current state, and a guard on the internal data. This `statemachine` function then returns the corresponding state or `no_transition`. The last clause is the catch-all clause if no transition can be taken in the ECC. The `alg` function executes the algorithms in a particular state. Every algorithm must be defined in a separate function, e.g. `alg_ALG(IM, IV0, OM0)`. The algorithm returns the updated internal variables `IV` and output variables `OM`.

Initialization

```
1  instance_args('InstanceName') ->
2      InstName = 'InstanceName',
3      InitV = #initV{'QI'=0, 'QO'=0},
4      EO = #eo{'CNF' = #conx{evN='REQ', tarN='TargetFB'}},
5      DO = #do{'QO' = #conx{evN='QI', tarN='TArgetFB'}},
6      CON = #con{eo=EO, do=DO},
7      InitD = #initD{initV=InitV, con=CON},
8      {InstName, InitD}.
```

Listing 8: Function that initializes the internal state of the FB instance.

The Erlang process is part of a supervision tree and is started by the responsible supervisor. The supervisor spawns the process from the Erlang module defining the function block type with an additional set of arguments. Those arguments are used to initialize the process and later, the function block. The first argument is the instance name, which is used to register the process. This is more efficient than using process identifiers, as the function block instance, by design, must have an unambiguous name. In addition, there is a set of initialization values and a connection table. The initialization values describe constant inputs of the function block and the connection table describes for every outgoing event and data connection the name of the receiver and what name is expected.

Language of the algorithm

The IEC 61499 standard does not strictly define the language of the algorithm. For an implementation in Erlang, multiple options are available. The most forward approach, and the one used for the evaluation in this work, is to implement the algorithm directly in Erlang.

The most convenient approach would be to convert the algorithms to C code and call them as natively implemented functions in Erlang. The performance is generally even better than Erlang code, but long algorithms may lead to blocking, as they cannot be interrupted by the scheduler.

The third approach is the automatic conversion from one language (IEC 61131-3, C, Ladder Logic, ...) to a language more compatible with Erlang, i.e., Erlang itself or Elixir. This is currently being investigated.

2.2.3

Performance Evaluation

The previous section described the implementation of the IEC 61499 basic function block in Erlang. As initially stated, this section aims to evaluate the real-time performance of the Erlang implementation of the IEC 61499 basic function block. Since the worst-case execution time in Erlang is unbounded due to its non-deterministic nature, only an empirical analysis of the distribution of the reaction time can be produced.

Methodology

Real-time performance is a topic intensively studied in literature. Wilhelm et al. [102] creates an important overview of how to estimate the worst case execution time (WCET) of a system. More related to the IEC 61499 standard, Zoitl [50] presents a model for the real-time execution of this standard.

The result of interest in this chapter is the reaction time of a function block and what parameters it depends on. In combination with a control flow analysis, this result may be used to determine the reaction time of a function block event chain, similar to the description by Zoitl [50], although empirical. To measure the reaction time, a basic function block is implemented as seen in the previous section, and equipped with time-measuring capabilities. In this case, the function block will return a set of timestamps to the requesting process. The full evaluation setup is described in the next section.

In total, eight values are collected per measurement. Those are a counter, the current system time, the monotonic process reduction count, and five durations of the function block execution:

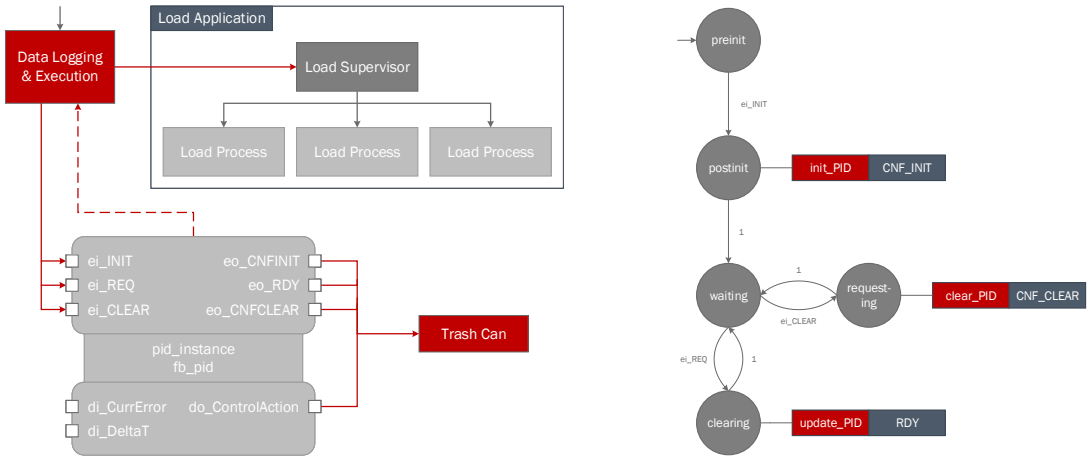
1. T1: Event send time to the FB

2. T2: Data sampling time
3. T3: Execution control chart execution time
4. T4: Garbage collection time
5. T5: Event send time from the FB

T1 is the time it takes to send a message from a high-priority process to a normal-priority process. T2 is the time to update the input memory from the buffer according to the event. T3 is the time to execute the ECC. T4 is the time to perform the garbage collection. T5 is the corresponding counterpart to T1, i.e., sending a message from a normal priority process to a high priority process.

Evaluation Setup

To make realistic measurements, the function block has to be embedded in an environment, as depicted in Figure 2.9a. There is a *high* priority process in charge of orchestrating the tests (Data Logging & Execution). It starts the load, requests the function block execution, and stores the data. The outputs from the function block are sent to a process serving as a trash can. Additionally, there is an application to apply additional load to the ERTS. This application allows the spawning of processes that repeatedly perform an expensive computation, thus filling the run queue.



(a) IEC 61499 Function Block Timing **(b)** IEC 61499 PID ECC

Figure 2.9: Experimental Setup

The implemented function block, implemented as a *normal*-priority process, acts as a PID controller, calculating an algorithm when requested and distributing its results to other processes. Its interface is shown in Figure 2.9a and the ECC is

depicted in Figure 2.9b. The dashed connection from the top of the function block interface stands for the additional time-measuring output.

To achieve realistic and consistent execution, the function block is triggered every 25 ms. This value was chosen as a compromise between the highest possible resolution (frequent measurement), economical generation of data, and avoidance of feedback from the measurement. This execution is achieved by measuring the total execution time and waiting for the remaining time. In case the execution takes longer than 25 ms the next cycle is started immediately after the previous cycle.

The additional load fills the run queue of the scheduler, thus causing the function block process to compete with other, *normal* priority processes. The load processes are always executed for the full amount of reductions, in this case 4000, before they are interrupted.

Tests

In a realistic setup, the function block would be part of a larger network. This concurrent execution causes the run queue of the scheduler to fill with processes. In normal systems, this additional load for an event-based system depends on the input events and their frequency and is highly fluctuating.

In this benchmark, to achieve the most deterministic result, the load processes are executed for the full amount of reductions and are continuously requested. Five test cases are executed with varying numbers of concurrent processes, starting with two and moving up to 32 in discrete steps. Lower numbers of processes show large fluctuations, whereas more processes would exceed the cycle time. This number resembles the number of processes waiting in the run queue before the function block.

The test platform for this test is a Raspberry Pi 3 Model B with Raspbian Jessie and Erlang 20. The only modification of the Raspberry Pi is to manually disable CPU throttling. Only one scheduler is spawned to prevent work stealing between the schedulers. The ERTS is started with a *nice* value of -20 to prevent interruptions as much as possible.

Each test is performed for 7,200,000 executions, which is equivalent to 50 hours, or 250 hours in total, for all 5 test cases.

2.2.4

Results

The performance evaluation yielded 2 main results:

- Table 2.4 comparing the maxima, minima, mean values, and standard deviations of the individual test cases for every measured variable.
- A scatter plot (Figure 2.10), visualizing the temporal distribution of the total time data. The tests were performed consecutively, but they are overlaid in the scatter plot to allow an easier comparison.

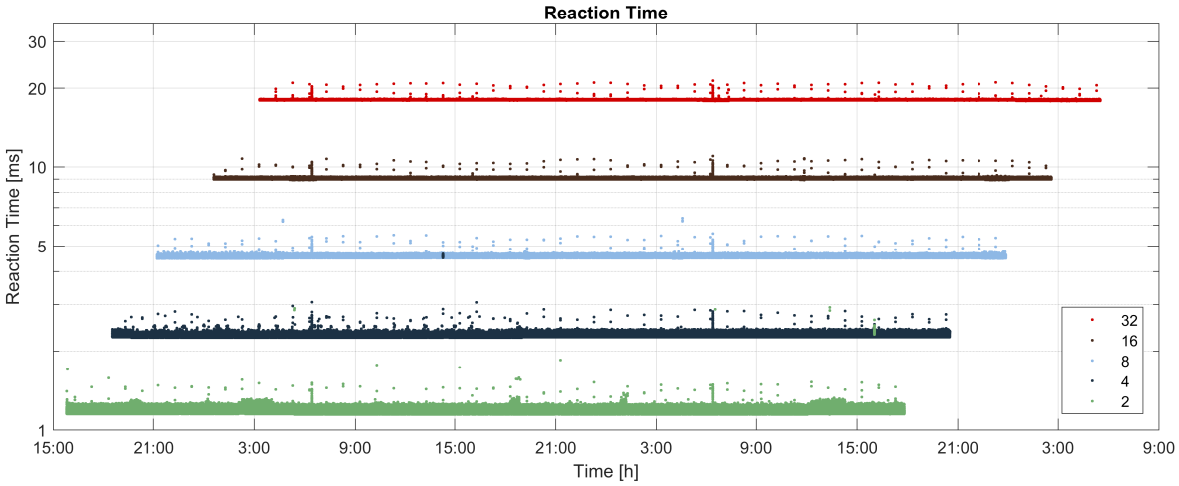


Figure 2.10: Temporal distribution of reaction time depending on number of concurrent processes.

Table 2.4 shows the relevance of the contributing durations. The *Send Time 1*, which represents the time it takes for the process to receive a time slice, is dominating in all test cases. Second is the *Execution Control Chart Time*. The manually enforced garbage collection takes a considerable chunk of the execution time of the function block. The sampling of the buffer values is nearly negligible. The *Send Time 2* can serve as a reference value for the reactivity of the scheduler, as in this case, the recipient of the message runs with a higher priority and will be scheduled immediately.

The scatter plot (Figure 2.10) reveals periodic interruptions every hour and once every 24 hours (around 6:25 AM). For 32 processes, a faint sinusoidal shape of the disturbances can be perceived. The more remarkable outliers for 2 and 4 additional processes all resemble the distribution of the next higher test case.

	# Proc.	Max [ms]	Min [ms]	Mean [ms]	StdDev
Total Time	2	2.9252	1.1452	1.1653	0.0131
	4	4.6620	2.2497	2.2844	0.0181
	8	6.3932	4.4769	4.5339	0.0195
	16	11.0155	8.9183	9.0264	0.0225
	32	21.3346	17.8875	18.0100	0.0340
Send Time 1	2	2.8468	1.1171	1.1358	0.0116
	4	4.6028	2.2222	2.2554	0.0168
	8	6.3147	4.4491	4.5045	0.0181
	16	10.9293	8.8894	8.9968	0.0209
	32	**21.2492**	17.8592	17.9803	0.0323
Sample Time	2	**0.0762**	0.0011	0.0013	0.0001
	4	**0.0766**	0.0011	0.0013	0.0002
	8	**0.1092**	0.0011	0.0013	0.0002
	16	0.0071	0.0011	0.0013	0.0001
	32	0.0065	0.0011	0.0013	0.0001
ECC Time	2	0.1241	0.0182	0.0206	0.0032
	4	**0.1940**	0.0180	0.0204	0.0033
	8	**0.2212**	0.0182	0.0206	0.0035
	16	0.1111	0.0183	0.0208	0.0037
	32	0.1072	0.0185	0.0209	0.0040
GC Time	2	**0.1031**	0.0067	0.0076	0.0006
	4	**0.1081**	0.0065	0.0074	0.0008
	8	**0.1269**	0.0065	0.0074	0.0008
	16	0.0192	0.0066	0.0075	0.0006
	32	0.0590	0.0066	0.0075	0.0006
Send Time 2	2	0.1204	0.0127	0.0180	0.0049
	4	**1.2427**	0.0127	0.0182	0.0050
	8	0.1294	0.0127	0.0164	0.0037
	16	0.1048	0.0128	0.0183	0.0037
	32	0.1927	0.0128	0.0186	0.0054

Table 2.4: Tabular performance analysis results

In addition, the process consistently required 91 reductions for the execution in every cycle. This includes the algorithm itself and all overhead introduced by the function block implementation.

Discussion

The results present the performance of an asynchronous, parallel function block implementation on a simple, commonly available hardware platform with a fair round-robin scheduler.

It is important to note the limitations of the evaluation. Only one scheduler was used, thus the Raspberry Pi platform was only able to use one core for Erlang. This also implies that the operating system can run other tasks in parallel on other cores. If four schedulers were spawned, parts of the performance would potentially improve

by 300 %, although more interruptions by the OS could be expected. The current operating system is not optimized for real-time performance.

The *Send Time 1* and consequently the *Total Time* scale linearly with the number of additional processes in the queue. This is very distinct for the mean and minimum. For two and four extra processes, the maximum is distorted due to the outliers visible in the scatter plot. This result follows the characteristics of the scheduler. A longer run queue causes a longer wait time to be executed. *Send Time 2* is independent of the number of additional processes because the high-priority process will get executed immediately. Intuitively, setting many processes to a higher priority will diminish the advantage. Both the garbage collection and ECC show notable outliers worth further investigation. The cyclic interruptions visible in the scatter plot are most likely due to the operating system.

The results allow two separate interpretations of the maximum number of processes executable within the 25 ms deadline. The current framework uses 32 concurrent processes as a maximum load. In a worst-case scenario, where 32 processes are busy blocking the scheduler, a real-time task may still be executed within 25 ms. This corresponds to the fairness property of the scheduler, as long-running processes will eventually be preempted. On the other hand, the current FB implementation required under 100 reductions for an ECC execution. Thus, preemption is rather unlikely during the execution of a single FB. In a realistic setup, where each FB process will use much less than 4000 reductions, a much larger number of processes can fit inside the 25 ms window. Assuming 500 reductions per function block and 4 schedulers, 1024 individual function blocks may be executed while consistently keeping the 25 ms deadline.

Conclusion

The aim of this section was the demonstration of an asynchronous, multitasking IEC 61499 Basic Function Block implementation in Erlang and its real-time performance evaluation. Erlang and the IEC 61499 models share many similarities since they are both intended for distributed, concurrent, and event-triggered applications. This simplifies the implementation of the IEC 61499 models.

Using Erlang as an implementation language comes with many benefits, e.g., the native support for distribution, concurrency, and event-based execution, as well as the functional paradigm which is well-suited for safety and traceability. Erlang also

allows dynamic software updating, i.e., updating running applications, which will be investigated in further projects.

On the other hand, Erlang's concurrency introduces overhead into the system, and it cannot guarantee hard real-time constraints due to the fair scheduler and non-deterministic garbage collection. Erlang's flexibility and scalability are not necessarily beneficial to an IEC 61499 implementation.

This section presented a feasible IEC 61499 basic function block implementation in Erlang. On a simple single-board computer with a soft real-time operating system, the real-time performance was consistent. Minor cyclic interruptions, most likely due to the operating system, were observed. The garbage collection was triggered manually to prevent unanticipated delays. The implementation of the function block is slim enough to not be interrupted by the scheduler, although this means the scheduler of the ERTS behaves more like a cooperative round-robin scheduler. The reaction time of the FB thus depends largely on the number of concurrent processes in the run queue. The upper bound, when every process uses as many reductions as possible until it is interrupted, was shown in this section. In this scenario, 32 additional load processes could be executed while still keeping a 25 ms deadline.

2.3 IEC 61499 Runtime Environment using Erlang[11]

The previous section analyzed a basic function block implementation in Erlang, which was shown to be generally possible and feasible. However, individual FBs do not allow a further analysis of the execution semantics and how well the techniques and concepts of Erlang can be mapped to the IEC 61499 models.

This section describes the IEC 61499 implementation of FBBeam. The solution builds on the experiences gained from the prototypical implementation in the previous section. FBBeam allows the automatic code generation from an IEC 61499 model, defined by the XML exchange format, into Erlang source code that may be executed directly within the Erlang Runtime System. Further, it leverages the prominent features of Erlang for use in industrial control systems, i.e., design for fault-tolerance and reliability as well as the ability to dynamically adapt.

[11] Major parts of this section were published in L. Prenzel and J. Provost. "FBBeam: An Erlang-based IEC 61499 Implementation". In: *International Conference on Industrial Informatics*. IEEE, 2019.

2.3.1

FBBeam Execution Semantics

As seen in Section 2.1.2, the IEC 61499 models can be implemented in different ways. In FBBeam, the implementation from Section 2.2 is extended. FBs are implemented as dedicated processes and messages are sent autonomously and directly to the recipient process. Process scheduling is handled by the Erlang virtual machine. As a result, no additional event handlers are necessary and processes behave the same, independently of how they may be distributed. All processes share the same priority, thus executable FBs get their fair share of execution time in a FIFO order. As shown in Section 2.2, a usual execution cycle of a FB consumes much less than the available 4000 reductions. If multiple messages, i.e., new data or events, are in the mailbox, the FB will consume all of them until it could be preempted, yet this is unlikely given that FBs are usually short-running.

The implementation of FBBeam corresponds to a multitasking implementation with time slices without a fixed FB scan order. Regarding the semantics defined by Vyatkin [69], the implementation follows an asynchronous, parallel execution. Wherever further choices about the execution semantics were necessary, the choice facilitating an implementation in Erlang was made.

In addition to the FB processes, additional processes are needed as supervisors. These supervisors start and monitor a set of processes, and restart them if they terminate unexpectedly. As long as no crashes occur, these supervisors do not add additional load and do not influence the execution behavior.

2.3.2

Compilation

The code generation is performed in three steps. The compiler, generating the Erlang source code from the IEC 61499 XML documents, is implemented in Python 3.

1. The IEC 61499 XML exchange format files are read, and parsed, and an internal model of the IEC 61499 system is created. Python objects collect all information regarding FB instances, connections, subapplications, and applications.
2. The internal IEC 61499 model is transformed into an internal representation of the Erlang source code. FB instance information is gathered in the correspond-

ing type modules, and the information necessary to generate supervisors is extracted from the application and subapplications.

3. Finally, the internal representation is inserted into templates for the Erlang source code modules. For Service Interface FBs, the source code is taken from a prepared library, and the functions defining the instances are appended.

The general code generation follows the scheme from Section 2.2. The main extension is the integration of FB networks as applications and subapplications and the implementation of service interface FBs in a library. These two aspects are discussed in the following sections.

Service Interface FB

IEC 61499 systems consist of either library FBs or custom-built FBs. Wherever possible, reuse if recommended. Most functionality can be built using the basic FB, the simple FB, or by combining multiple FBs into a subapplication or composite FB. There are two reasons to take another route: Either, the functionality requires access to functions that are not available in the IEC 61499 standard, or an implementation in the previously mentioned ways would be inefficient. In this case, a service interface FB can be implemented instead. These FBs provide an interface to an underlying service, e.g. written in C. Thus, these service interface FBs must be supplied by the RTE and cannot be generated from the XML files directly.

In FBBeam, the service interface FBs must be supplied separately as an Erlang implementation. Generally, arbitrary Erlang implementations are possible. For example, the Modbus interface consists of two separate Erlang processes: One process to provide the interface, and a server to handle the TCP connection. In this manner, multiple service interface FBs could connect to the same server without requiring a second TCP socket. Further, if the connection is broken, the interface survives and can reopen the connection. This provides fault tolerance to the implementation.

Supervision Trees

In Erlang, processes must be part of a supervision tree to be part of an application. There can be multiple layers of supervisors that connect to the top-level application supervisor. A supervisor specification is depicted in Listing 9.

In this specification, a single child is specified to be started during the startup of the supervisor. The strategy `one_for_one` specifies that when a child fails, only this

```
-module(SupervisorName).
-behaviour(supervisor).

-export([start_link/0]).
-export([init/1]).

start_link() ->
    supervisor:start_link({local, ?MODULE}, ?MODULE, []).

init(_Args) ->
    SupFlags = #{strategy => one_for_one, intensity => 1,
                 period => 5},
    ChildSpecs = [
        #{id => 'InstName',
          start => {'TypeName', start_link,
                    ['TypeName':instance_args('InstName')]},
          restart => permanent,
          shutdown => 5000,
          type => worker,
          modules => ['TypeName']}
        ],
    {ok, {SupFlags, ChildSpecs}}.
```

Listing 9: Supervisor implementation in Erlang with a single child specification.

child is restarted. The `intensity` and `period` define how often a process may fail during the period. The given values indicate that if a process terminates twice within five seconds, the supervisor itself terminates.

2.3.3 Current Limitations

The current implementation of FBBeam is for research purposes that focus on dynamic reconfiguration and real-time performance. The main intention is to introduce new ideas, concepts, and techniques into a domain that while being interested in concepts of reliability and adaptability, has not shown too much innovation in this direction.

Thus, the features that are implemented either share a common foundation, such as the FB model which conveniently overlaps with the actor model of Erlang, or are native to Erlang but appealing to the industrial control domain, such as dynamic adaptation. Specifically, of the IEC 61499 models, currently, only the System, the

Application, and some of the FB models are used. From the FB models, the Basic FB, the Subapplication, and the Service Interface FB are supported. A library of Service Interface FBs is being extended continuously and currently contains an interface for Modbus TCP. Adapters are not supported yet and the IEC 61499 distribution models are currently not used, mainly because they are largely incompatible with the distribution features of Erlang. The concept of distribution and internode communication is natively available for Erlang, but those features must be mapped to the IEC 61499 distribution models.

Within Basic FBs, the translation of algorithms is an open field for research. A converter from IEC 61131-3 ST code to Erlang has previously been developed but is not yet integrated, since the conversion suffers from the conceptual differences between ST and Erlang. Alternatively, ST code could be compiled into C code, which can be executed inside the Erlang Runtime System, although this circumvents all safeguards that make Erlang fault-tolerant. Currently, algorithms may be defined in Erlang itself. Since FBBeam is mainly intended for research, this is not a critical issue.

2.3.4

Evaluation & Case Study[12]

To demonstrate the multitasking capability of the Erlang Runtime System, a case study of a physical simulation is implemented. Figure 2.11 displays the application model of the simulation. A PID controller is connected to a FB network simulating a physical process with a random disturbance. This simulation is executed continuously and the number of executions reached over 60 seconds is counted.

The application is modified to run up to 32 concurrent simulations with 32 PID FBs in parallel, and the simulation is run with between 1 and 4 Erlang schedulers. This experiment is repeated for 120 runs and the results are displayed in Figure 2.12. In total, the simulation was run for over 256 hours.

The case study indicates, that without any further optimizations, Erlang can distribute the workload efficiently and automatically over multiple CPU cores. With four schedulers, the application has to compete with the operating system for resources, thus the performance can only exhibit linear behavior up to 3 schedulers.

[12]Major parts of this section were published in L. Prenzel and J. Provost. "FBBeam: An Erlang-based IEC 61499 Implementation". In: *International Conference on Industrial Informatics*. IEEE, 2019.

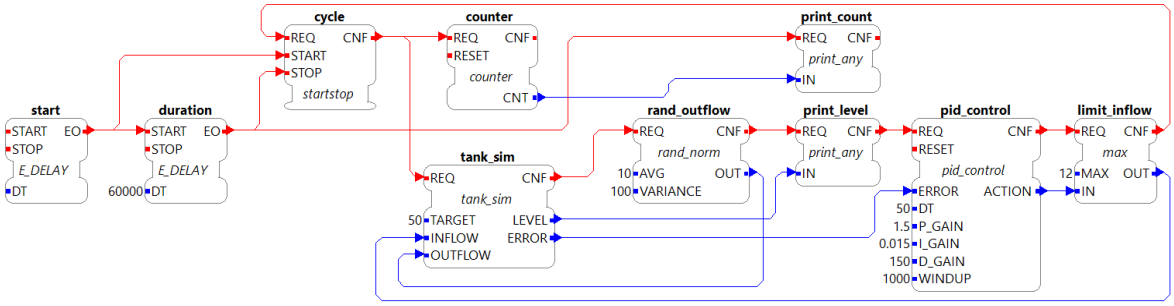


Figure 2.11: IEC 61499 Application model of a tank simulation case study. The simulation is executed continuously for 60 seconds, after which the total number of executions is displayed.

Full utilization of the additional schedulers requires a larger number of concurrent simulations.

Discussion

Reusing proven technology for the implementation of the IEC 61499 standard has multiple advantages. The Erlang Runtime System can support a large number of concurrent processes. It is not limited to the number of operating system threads and allows fast context switches. As illustrated by the case study, Erlang can run large numbers of FBs concurrently, and can efficiently employ the available CPU cores. The load is distributed dynamically and processes receive equal opportunities to handle their messages/events. When more than one scheduler is used, performance may scale almost linearly.

If real-time constraints or event rates were available, it could be possible to find a better scheduling mechanism, such as *earliest deadline first* or *rate monotonic scheduling*. Erlang is intended for dynamic soft real-time systems, not for static or cyclic hard real-time applications. Since static, cyclic hard real-time implementations of the IEC 61499 standard already exist [79], FBBeam shows how a dynamic, event-triggered, and scalable multitasking IEC 61499 implementation may look like.

In addition, an IEC 61499 implementation in Erlang allows the reuse of a proven and growing ecosystem. Recently, [103] demonstrated that there is a need to move the IEC 61499 ecosystem forward in terms of fault tolerance. FBBeam opens opportunities for investigations into how existing frameworks of Erlang may be adapted for the IEC 61499 models, for example:

- Unit & system testing

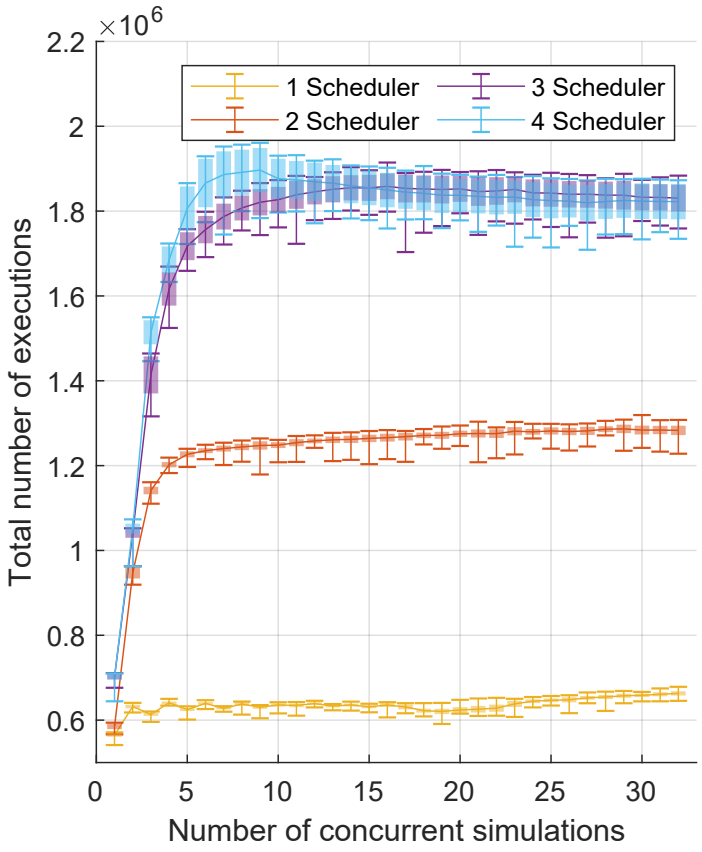


Figure 2.12: Event chain executions for 1–4 scheduler and 1–32 concurrent event chains for 60 seconds. Error bars indicate minimum and maximum values in 120 runs, the continuous line connects the average values.

- Distribution, deployment, and monitoring
- Dynamic reconfiguration
- Function Block error handling and fault-recovery

Dynamic Reconfiguration Support

As part of the implementation of FBBeam, Erlang's dynamic reconfiguration features were used to reconfigure IEC 61499 applications. Since the *hot code loading* process of Erlang differs significantly from the IEC 61499 dynamic reconfiguration models, a direct mapping from one to the other was not possible.

The implementation did, however, show how convenient and simple the adaptation process in Erlang is. Although the documentation is not extensive, there are enough resources on how to adapt Erlang systems. Generally, the usage of separate

up- and downgrade specifications facilitated the implementation. The process is clear: Identify the differences between applications and their dependencies, suspend the execution, transform the state using manually written transformation functions, and resume the execution. By contrast, the IEC 61499 standard offers the services and the possibility to create arbitrary applications with them, yet the task feels more daunting and there is generally less support. Additionally, the availability of great live debugging features, extensive unit testing support, and a general tendency to think about failures and fault tolerance in Erlang provide more confidence in the procedure. Erlang forces you to think about the "what if a process fails", whereas the IEC 61499 standard still lacks support for exceptions.

Conclusion

Despite the architectural advantages of the IEC 61499 standard as a modeling language for distributed control systems, industry acceptance is still lacking. Erlang, on the other hand, is a technology initiated and developed by the industry itself, which is improved continuously. This section outlined an IEC 61499 implementation in Erlang to pinpoint opportunities and limitations.

Erlang favors an asynchronous execution because of its process architecture, although an event-chain implementation, such as introduced by Zoitl [50], may be feasible and advantageous in the future, especially concerning real-time constraints. In the current implementation, all scheduling and load distribution is organized by the Erlang Runtime System.

In the author's opinion, the IEC 61499 ecosystem can benefit from many diverse implementations that may suit different purposes. If interoperability were guaranteed, an Erlang implementation may offer convenient scalability, availability, and multitasking in applications where soft real-time suffices. While the real-time performance may not be sufficient for safety-critical industrial automation applications, this aspect could be improved in the future by introducing new scheduling paradigms to Erlang.

2.4

Key Findings

Current standards, such as the IEC 61131-3 or the IEC 61499 standards offer only primitive support for adaptation, which significantly inhibits agility as a precursor for self-adaptation. A reconfigurable RTE for industrial control software can offer the adaptable foundation onto which an agile architecture can be built.

This chapter investigated the role of current standards in improving the agility of ICS. Therefore, first, the state of the art in reconfigurable real-time software, in particular concerning industrial control (IEC 61499) and telecommunication (Erlang), was summarized. In detail, current IEC 61499 implementations were analyzed and compared. Further, the soft real-time RTE of Erlang was evaluated for an IEC 61499 implementation using a synthetic benchmark. Lastly, a full-fledged compiler from the IEC 61499 exchange format to Erlang was implemented that allows adaptation at runtime using the high-level paradigms of Erlang, and an evaluation regarding the scalability of the implementation was performed.

2.4.1

Reuse of existing technology

In modern software engineering, the reuse and re-purposing of existing technologies and frameworks is essential. Python's success hinges on its vast ecosystem of easily integrable packages developed by a large community. This drives innovation and stimulates new ideas that emerge at the intersection of other works.

The IEC 61499 standard, as a domain-specific language, offers a promising architecture and fresh ideas for the automation industry, however, it is rarely used compared to the IEC 61131 standard. This may be partly due to many of its features only existing in research implementations or prototypes, e.g. dynamic reconfiguration. Furthermore, a lack of adoption leads to a lack of stimulation for new ideas and developments, broadening the gap to the predominant IEC 61131 standard.

Erlang, as a programming language, has been in use and development over the last thirty years. The ecosystem can be confidently described as mature, and the runtime system serves as the infrastructure for other languages (i.e., Elixir) as well. Thus, the idea of reusing the RTE for an IEC 61499 implementation is not far off. Reusing and building upon this technology has obvious advantages, most

importantly:

- Maturity of the ecosystem
- Frequent and continuous upgrades and improvements
- Large and growing user base
- Detailed documentation
- Available commercial support
- Support for *hot-code-loading*

Building an IEC 61499 RTE on top of an existing RTE allows leveraging these advantages. The resulting RTE is straightforward to implement and benefits from the capabilities of Erlang.

The main disadvantage of using Erlang under the hoods of an IEC 61499 RTE is the lack of hard real-time semantics. As was seen in Section 2.2, the performance is fast, yet not deterministic. On the one hand, this facilitated the implementation, and the soft real-time semantics of Erlang are already sophisticated compared to the IEC 61499 standard. On the other hand, the IEC 61499 models *should* be executed in hard real-time, even though the standard does not describe *how*. This led to a bit of a conundrum because there also would not have been a straightforward way of making a hard real-time RTE given that the required metrics (deadlines, cycle times, priorities) do not exist within the IEC 61499 standard.

Nevertheless, the implementation served as an evaluation framework for dynamic adaptation and the feasible execution semantics of the IEC 61499 models. Furthermore, dynamic adaptation is just one area in which inspiration from other frameworks can be favorable. Other topics include automated deployment, real-time scheduling, testing, or monitoring, which were not touched upon in this dissertation.

Reusing existing technologies can stimulate new ideas and capabilities that, in the long term, may lead to improvements in the IEC 61499 standard or other domain-specific modeling languages. It also allows the comparison of the IEC 61499 models to the state of practice in other domains, which helps to identify shortcomings and potentials.

2.4.2

Guarantees and Execution Order

PLCs should execute and behave deterministically, both from a hardware and software perspective. The programming languages commonly used to program PLCs are simpler and cater to the needs of the automation engineer. The IEC 61499 standard extends these languages with models that allow the consideration of the full, distributed system during development before making decisions about the deployment. This advantage comes with the responsibility that engineers expect the same determinism from these distributed models as they expect from a traditional PLC. However, guaranteeing predictable behavior in a distributed system is much harder.

The discussions about the IEC 61499 execution semantics focus on the ambiguity within a single RTE [71]. This ambiguity remains within the standard and has not been resolved. Precise execution semantics between distributed runtimes are even harder to guarantee and there is little work done in this area. Furthermore, splitting the connections into data and event connections complicates the implementation and semantics even more.

This issue became evident when facing the IEC 61499 implementation in Erlang in this chapter. One goal of the implementation was to achieve functionally deterministic behavior, and Erlang provides some guarantees for achieving this determinism. For example, the message ordering between two participants on the same node/device is guaranteed by Erlang [104, 105]. The arrival of a message in a distributed setting, however, cannot be guaranteed. If the participant is not available at the time or the entire node crashes, the message will not be delivered. Having explicit guarantees or an explicit lack of guarantees promotes transparency towards the developer. In the case of Erlang, this means that if guaranteed delivery is expected, you need to implement a handshake that ensures that the message has been sent and received successfully. This kind of handshake may in return prevent predictable message ordering, yet this can also be overcome by a robust architecture.

The IEC 61499 standard, on the other hand, gives very few behavior guarantees and lacks the ability for expressive error handling [55, 106]. Implementing handshakes or sophisticated, robust failure handling mechanisms within an ECC is cumbersome and inefficient, and usually leads to an explosion of states and transitions. Thus, the application developer must rely on hazy guarantees, which can and may hold for a single node, yet are impossible to guarantee in a distributed

setting. There are two paths out of this predicament: Solve the issue in the RTE by sacrificing speed and efficiency for robustness and determinism, or communicate the issue transparently and improve the developer's ability to implement the required protocols efficiently, e.g. by introducing exception handling within FBs. Either way, transparency is critical to facilitate development and interoperability.

2.4.3 Real-time Capabilities

ICS require predictable real-time behavior to guarantee the safety of the physical process. Research has provided real-time models for an IEC 61499 implementation [50]. These models are, however, not commonly applied. Erlang, on the other hand, only offers soft real-time, which cannot guarantee that deadlines will not be missed.

In this chapter, the real-time performance of Erlang was investigated, and the system was able to show fast soft real-time performance. However, an Erlang implementation cannot offer any hard real-time guarantees. The non-deterministic garbage collection can lead to unpredictable delays, and there are no real-time metrics, such as deadlines or cycle times.

Two topics should be mentioned here. First, there have been efforts in the past to achieve hard real-time scheduling in Erlang. In [107], a hard real-time scheduler for Erlang was introduced, yet it was never fully adopted. They used deadline monotonic (DM) scheduling, and an earliest deadline first (EDF) scheduler was proposed. The problem of garbage collection was still open, yet this impact is relatively small in Erlang since it is performed per process. This would solve some of the issues and make an IEC 61499 implementation in Erlang suitable for hard real-time applications. Second, most current IEC 61499 runtimes avoid the topic of real-time performance. Some runtimes use the concepts of the IEC 61131 to achieve real-time performance, others do not specifically address the topic at all. In general, the standard does not contain the necessary metrics to perform a scheduling analysis, e.g. deadlines, rates, or worst-case execution times.

Erlang does not offer the hard real-time performance necessary for safety-critical ICS applications, yet this does not mean that an IEC 61499 implementation in Erlang is pointless. It can serve as an example of an adaptable RTE with soft real-time performance, which may suffice for many applications. In the light of interoperability, it may even work together with hard real-time RTEs in a larger application.

Chapter 3

Consistent Adaptation of Industrial Control Systems

Contents

The ability to adapt is meaningless if unused. The previous chapter showed how the fundamental ability can be achieved and improved, yet adaptation remains a manual process. Closing the loop towards self-adaptation requires the ability to implement the adaptation logic automatically and autonomously, i.e., without user intervention. Instead of using manual labor to understand the requirements and implement a suitable adaptation, this procedure must be automatic. This serves two

purposes: Automatic processes are faster and, if set up correctly, can prevent user errors and bugs. These reasons are particularly important for (self-) adaptation. A fast adaptation improves the agility of the system, and buggy or incorrect adaptation renders the process ineffective. To achieve an architecture such as the WATERBEAR architecture, adaptation must be swift and safe by design.

The main problem with transitioning a manual process into an automatic one is the handling of implicit knowledge. When a developer is tasked with implementing a dynamic adaptation, the developer must understand the explicit and implicit requirements and find a suitable solution. As seen in Section 1.1.2, there are numerous requirements to be considered before, during, and after the adaptation. The most important high-level requirement is correctness or consistency. An incorrect adaptation is worse than no adaptation at all. Thus, a mechanism is needed that can automatically guarantee the correctness or consistency of the adaptation.

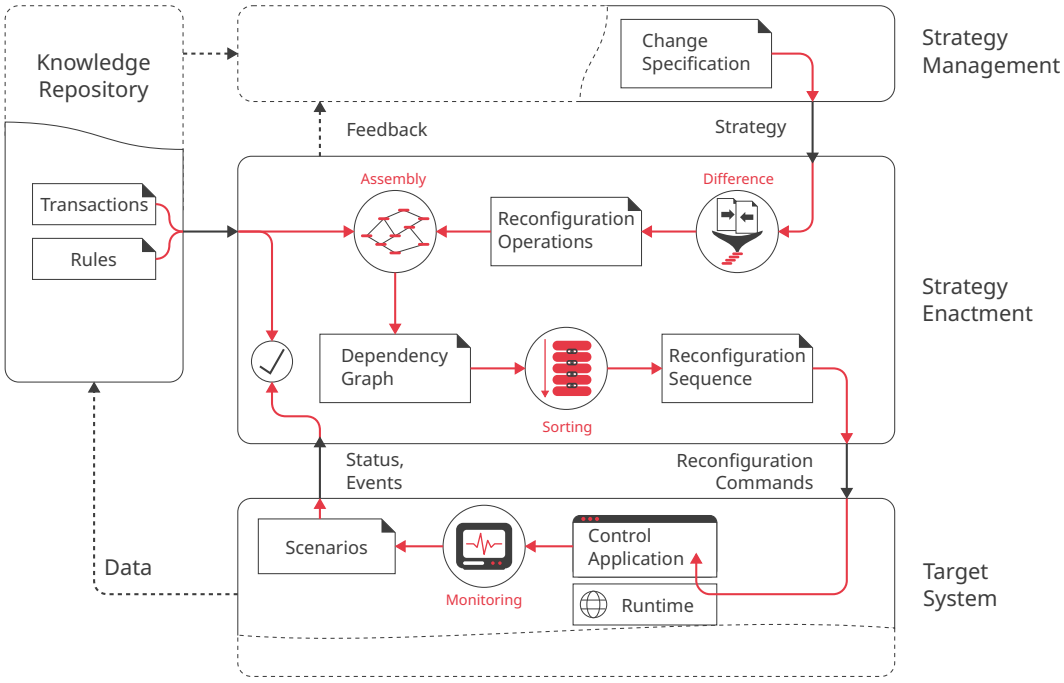


Figure 3.1: The strategy enactment layer takes the selected change specification (strategy) and devises a suitable reconfiguration sequence (change implementation) while keeping the behavior of the system in mind.

The mechanism developed in this chapter is summarized in Figure 3.1. Such a mechanism is only as good as the information it is provided. Thus, the implicit developer knowledge must be made explicit. Within the scope of the WATERBEAR architecture, all the relevant information must be stored in the knowledge reposi-

tory. Within the strategy enactment layer, the change specification provided by the strategy management must be implemented using the available information. In this mechanism, the required reconfiguration operations must be identified and sorted in an order that preserves the correctness of the running applications. The scenarios that result from these reconfiguration sequences can be validated against the correctness rules.

This chapter first introduces the state of the art in consistency management in dynamic adaptation and the issues that arise within (Section 3.1). Section 3.2 discusses the requirements of the IEC 61499 models regarding consistency, and proposes a method for achieving consistency within the scope of the IEC 61499 standard and general ICS. The algorithm is evaluated on multiple reconfiguration scenarios in Section 3.3, and difficulties and limitations are highlighted. Finally, Section 3.4 concludes the chapter with the key findings.

3.1 Consistency in Dynamic Reconfiguration

The most straightforward idea in achieving consistency during an adaptation is to wait for the perfect moment. In traditional modification processes, this perfect moment is forced: The system is brought to a safe state, turned off, and restarted with the new version. This moment, i.e., the point in time and the corresponding system state, is guaranteed to achieve consistency. Yet, for some systems, such an offline adaptation may be infeasible, and, as seen in Chapter 1, there are good reasons for dynamic adaptation, e.g., the transition towards self-adaptation.

To find a suitable safe update point, there must be transition conditions. If there are formalized transition conditions, these conditions can be used to synthesize a safe-by-design controller that navigates the system through the state space of these conditions [108]. If there are no formalized transition conditions, it is also possible to manually specify potential update points and use a verification mechanism, e.g., model checking, to verify the safety of these points [109]. Existing formal models of the application behavior, e.g., through interface automata, can also serve to synthesize safe locations [110].

For simple applications, e.g., IT server where most of the state is initialized at startup and does not change significantly during operation, *checkpoint restart* may be a solution. In this case, the old version is frozen in a quiescent checkpoint state, a new

version is initialized, and the new system is brought to the equivalent checkpoint state [111].

The problem is worsened if the adaptation changes the specification as well [112]. The problem of formalizing the requirements during an adaptation is known in research [113–115].

Generally, the validity of an online change from version 1 to version 2 with a particular state mapping in a specific state is undecidable [116]. This is due to the halting problem, which argues about the termination of an arbitrary program given an input. It was shown that there cannot be a general algorithm that can solve the halting problem for all combinations of programs and inputs. Deciding the validity of an online change can be reduced to a halting problem, since validity can usually be considered a type of reachability, i.e., the change is valid if eventually a particular state is reached. This does not indicate that halting or reachability cannot be shown for any program; for many systems, this analysis can be trivial. Yet, a general algorithm for all programs cannot be implemented. Thus, any algorithm must have three outcomes: valid, invalid, or unknown.

For future systems, it is assumed that the desirable or undesirable behavior is known. Arguments about consistency of unspecified behavior are not particularly productive, thus at least from the observed behavior, it should be possible to determine if the observations fall within the realm of expected or unexpected/undesirable outcomes. For this argument, input-output automata are used as a simple abstraction, although more complex models may be used or necessary. For the sake of visualization, this simple representation suffices.

In the following, two aspects are discussed in detail: General consistency conditions that can be used to achieve consistency during dynamic adaptation procedures, and the problems and solutions of state transformation during an adaptation. rate

3.1.1 Consistency Conditions

When updating a system on the fly, safety is of utmost importance. In addition to the validation and verification necessary for the new system version, the transition from one version to another must be validated and verified as well. This problem has been addressed in many research papers. Apart from satisfying real-time constraints, the functional consistency of the change is crucial to prevent failures. For component-

based systems, three major consistency conditions emerged: Quiescence, Tranquility, and Version Consistency [117–119].

Applying these concepts in practice is not always straightforward. In this section, the dynamic update of distributed control systems is considered, which results in new requirements and constraints. The aforementioned concepts are introduced in more detail in the following paragraphs.

Quiescence

Quiescence provides consistency by demanding that a component is "not within a transaction and will neither receive nor initiate any new transactions" during the adaptation [117]. Components can be *passivated* to achieve this by stopping the initiation of new transactions. A passivated component is unable to initiate new transactions, yet can service old transactions. By passivating the right components, eventually, the lack of new transactions will leave some components in a quiescent state.

In this context, two types of transactions are mentioned: *Independent* and *dependent* two-party transactions. For independent transactions, only the components directly adjacent to a change must be passivated to prevent these transactions from being initiated. For dependent transactions, on the other hand, all dependent transactions that eventually lead to this component must be considered. Thus, the passivated set of components and thus the caused disturbance is directly related to the interdependence of the system [117].

A downside of the original approach was its focus on components, which can lead to long passivation times of components. Wermelinger [120] instead proposes the use of passivation on the connection level, which only blocks the connections that will eventually be removed.

The strict requirements of quiescence could be relaxed if communication is synchronous and reliable and messages cannot be lost [121].

A benefit of the quiescence approach is that it can be modeled very efficiently as a state machine on the component level [122]. The component starts in the *active* state, is passivated to reach the *passive* state, and eventually reaches the *quiescent* state once all transactions involving this component have ended. Yet, these states can be hard to detect, since this requires tracking the transactions of other components as well.

Quiescence is a simple consistency condition suitable for many scenarios. How-

ever, depending on the transactions, it may take a long time for the system to reach quiescence. As a consequence, several conditions were developed that are explained in the following sections.

Tranquility

Tranquility was introduced as a condition for consistency to reduce the potential disruption caused by quiescence [118]. It takes two additional assumptions into account: Firstly, a component can be changed if there is an active transaction it has already participated in but will not again, and if there is an active transaction it will participate in, but has not yet. Secondly, when assuming a black box design, a component can only depend on directly adjacent components. Thus, a component is in a tranquil state if it is passivated and "none of its adjacent nodes [components] are engaged in a transaction in which it has both already participated and might still participate in the future" [118].

Unlike quiescence, tranquility is not forced by passivating surrounding components. As a consequence, it can occur naturally and with less disturbance, but there is also no guarantee that it ever occurs [118]. Quiescence, on the other hand, will be reached eventually, as long as transactions eventually finish. On the other hand, the black box principle is in practice often not applicable [119].

Thus, tranquility provides an extension to quiescence that relaxes the assumptions. However, it also requires additional monitoring with higher complexity. It can be beneficial in the best or average case, yet does not improve the worst-case delays over quiescence.

Version Consistency

The aforementioned consistency conditions work on the level of components. Version consistency, on the other hand, ensures global consistency with distributed transactions [123]. It requires that for every transaction, all sub-transactions are "entirely executed in the old or the new configuration" and the component is idle at the time of update [119].

If a component is in a valid state to be changed can be determined through the property of *Freeness*. Baresi et al. [119] determine *Freeness* by following dynamic dependencies. These dynamic dependencies are maintained on a configuration model of the system by adding or removing *future* and *past* dependencies on each

component. These dependencies indicate if a source node can initiate or has already initiated a transaction on the target node. If a component does not have both *past* and *future* dependencies regarding one transaction, the component is said to be *free*. In essence, this indicates that regarding one transaction, this component either will be visited, or was already visited, but not both. The dynamic dependencies must be either tracked globally or for each component.

Once it is possible to track the *Freeness* status of each component, the update must be triggered at the right moment. Baresi et al. [119] mentions three mechanisms to achieve version consistency:

- Waiting for freeness, i.e., holding the update back until the component is free,
- Concurrent versions, i.e., hosting an *old* and a *new* version of a component and choosing which version must serve a request, and
- Blocking for freeness, i.e., blocking some transactions from processing in a node to prevent *past* dependencies.

Version consistency takes a different approach than quiescence or tranquility, and offers multiple mechanisms to achieve it. The common piece to all three conditions is, on the one hand, a condition to determine a safe update point, and on the other hand, a mechanism for how to reach this point. The next section explains how to handle the state transformation problem.

3.1.2 State Transformation

The previous sections discussed possible consistency conditions that concern the timing of the update. Since most applications are stateful, and the meaning of the state may change during an adaptation, the state of the application must also be transformed during the adaptation.

The state transformation can be achieved in many ways. Seifzadeh, Abolhassani, and Moshkenani [24], for instance, mention the following types:

Copy/Shadow The data structure can either be copied to a new location, or modified by adding pointers to new fields.

Specified/Re-Execution There must either be a (manually) specified transformer function, or the state can be retrieved by re-execution from a previous checkpoint.

Global/Active Either the global state is transformed, or only actively updated states are transformed.

Eager/Lazy In an eager transformation, the entire state of the application is transformed at once. Lazy transformation updates the state of a module/process/-component when it is used for the first time.

Value/Type The transformation can be limited to values of the state, or also allow type changes.

Direct/Indirect The transformation is either performed directly by the affected module or exported to a standard format and transformed indirectly by another module.

Generally, the main problem with state transformation is the creation of the needed transformer functions. For some applications, it is possible to automatically generate these transformers by analyzing the source code [124]. In other cases, the transformers may depend on the context and origin of the adaptation. A closed-loop self-adaptation should be able to also generate the transformers, whereas an open-loop manual adaptation may require manually created transformer functions.

3.2 Automated Dependency Resolution for IEC 61499[13]

After having seen the challenges in preserving consistency and some solutions on how consistency during a dynamic adaptation can be achieved, this section applies this methodology to ICS. The existing framework for dynamic reconfiguration of the IEC 61499 standard is introduced briefly. It currently does not offer any guarantees of consistency but only offers the low-level operations or services necessary for a reconfiguration. This section, then, analyzes the system and change specifications to automatically generate a safe-by-design sequence of operations.

This section introduces a methodology to automatically generate reconfiguration sequences that allow the safe reconfiguration of distributed, component-based systems based on the IEC 61499 standard (Figure 3.2). A mechanism is developed to extract the required reconfiguration operations from the difference between two applications and required supplemental information is identified. The resulting oper-

[13]Major parts from this section are published in L. Prenzel and S. Steinhorst. "Automated Dependency Resolution in Dynamic Reconfiguration for IEC 61499". In: *Proceedings of the International Conference on Emerging Technologies and Factory Automation (ETFA)*. Västerås, Sweden: IEEE, 2021.

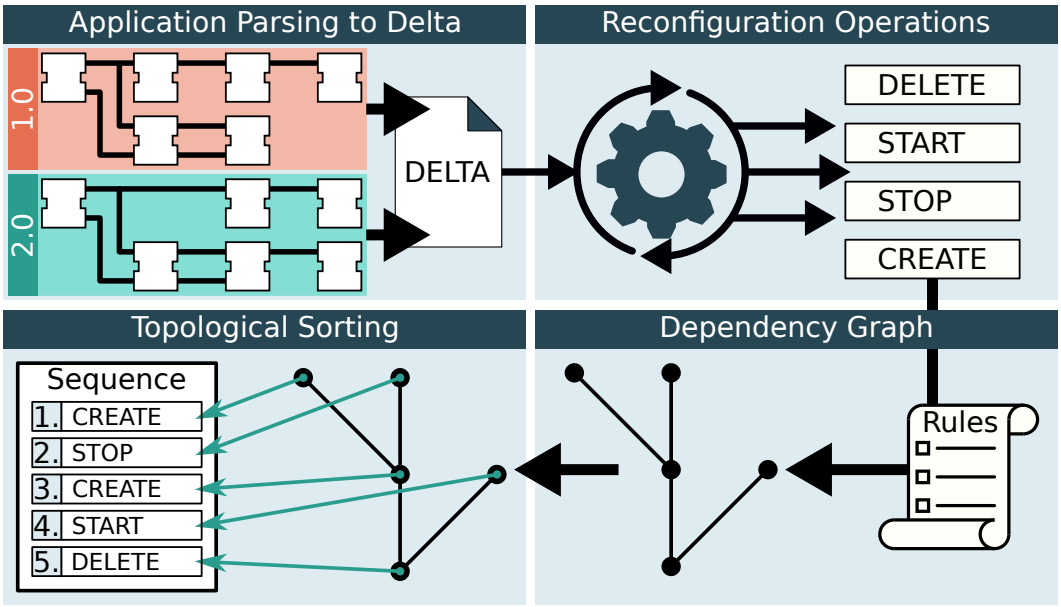


Figure 3.2: From the IEC 61499 application, the event traces can be extracted. The difference between the two applications yields the required reconfiguration operations for the change, which can be assembled into a reconfiguration sequence given the event traces.

ations may be organized into a dependency graph using a rule set, and a linearized reconfiguration sequence can be extracted via topological sorting. The methodology is demonstrated on four reconfiguration scenarios, and an extension is proposed to identify and handle feedback loops.

3.2.1 Reconfiguration Operations

The necessary operations must be identified to be able to change a component-based system. A reconfiguration operation o can be defined as a tuple (Equation 3.1), where i is the instruction or service, t is the target, e.g., a component or a connection, and D is a set of operations that this operation depends on.

$$o = (i, t, D) \tag{3.1}$$

IEC 61499 Reconfiguration Services

The IEC 61499 standard defines a set of reconfiguration services and a state machine for the operational states of an FB. These services are specific to the runtime environment and may be provided by a set of corresponding management FBs. An exhaustive list of services is given by Zoitl [50]. These management FBs can be assembled into a reconfiguration control application (RCA) or *Evolution Control Application* [50, 88]. This process was described in Section 2.1.3.

In this section, the IEC 61499 services are treated as the instructions in a reconfiguration operation. The operation to create a new FB would thus contain the CREATE FB service as the instruction i, the target t as the FB to be created, and D would indicate a set of dependencies that must be performed before the FB can be created.

Selected Reconfiguration Operations

The services as described by the IEC 61499 standard are intended to be exhaustive, i.e., they should allow any type of reconfiguration. In practice, most reconfigurations are going to be simplistic, such as tweaking a parameter. Even complicated reconfigurations do not typically need the full set of services, at least during normal operation. The KILL and RESET services, which can be used to interrupt a running FB and may lead to a corrupted state or event loss, are hardly compatible with most execution semantics as long as continuity is desired. Yet, there may be specific edge cases in which their use is required.

In this section, thus, only a selection of these services is used to demonstrate the methodology (Table 3.1). Most noteworthy, these are the services to create and delete FBs and connections. Further, the flow of events must be controlled, which can be achieved by setting the operational state of a FB to stopped or started. It is assumed that, when a FB is stopped, the events for this particular block are going to be buffered. This is crucial to achieve consistency because otherwise, unrecoverable information in the form of events is going to be lost.[14] Finally, the internal state of FBs must be accessed by reading and writing. Some services are omitted from the methodology since they do not add any meaningful value at this point. These are

[14]To the author's knowledge, this behavior is not clearly defined within the IEC 61499 standard. Some RTEs, such as 4diac FORTE do not buffer events in the stopped state. Buffering can lead to an overflow of the event queue, however, losing events is a dangerous interference with the execution semantics. Message and event loss can be tolerated in fault-tolerant languages like Erlang, but it requires special care, attention, and an underlying framework built for fault-tolerance.

the services to create and delete types and resources, because they can always be performed during the RINIT or RDINIT phases and their ordering does not matter, except that resources are added in the very beginning and deleted at the very end. The methodology can be easily extended to incorporate other or new services.

Service	Description
CREATE FB	Create a new function block.
CREATE CON	Create a new data/event connection.
DELETE FB	Delete a function block.
DELETE CON	Delete a data/event connection.
START	Set the operational state of the FB to Started.
STOP	Set the operational state of the FB to Stopped.
READ	Read the data inputs, outputs, and internal variables of a FB.
WRITE	Write the data inputs, outputs, and internal variables of an FB.

Table 3.1: Description of the reconfiguration services as defined in [50].

3.2.2

Reconfiguration Scenarios

Reconfiguration can take place in many scenarios that require slightly different handling. This section distinguishes four scenarios concerning the necessary state transformations.

State in Dynamic Reconfiguration

A major hurdle in the dynamic reconfiguration of a component-based system is the handling of state [125]. In [126], the continuity property is introduced concerning the state transformation. To achieve continuity, a service must be continued and partially executed services must be completed. In a traditional change scenario, in which the system is shut down and restarted, the state can be discarded and the new system can be restarted from a known, initial state. In the dynamic case, this continuity and integrity is a critical component. Given two arbitrary systems, the prospect of being able to elegantly transition from one to the other is rather bleak. Transforming a flight controller into an HVAC controller, however unrealistic, is difficult not because different components are used, but because of the difference in state and the difficulty to achieve continuity.

State Transformations

A state transformation offers the necessary information to transform the state, either by mapping state X to state Y or by identifying suitable safe update points, e.g., wait until state X and switch to state Y. For a small class of problems, the needed state transformation can be identified automatically. Panzica La Manna [126] model the system behavior with interface automata, which are then used to identify a correct state transformation. This requires the availability of corresponding behavior models and a framework to identify the state transformation. Other automatic frameworks require the availability of formal specifications and manually created mapping functions [108]. In this section, reconfiguration scenarios are characterized based on the state transformation needs from an architectural perspective, irrespective of their origin.

Reconfiguration Scenarios in IEC 61499

The system view of an IEC 61499 application does not comprise any information on how another system may be transformed into this one or how it could be transformed into another one. Current behavior models are not suitable to automatically identify appropriate state transformations, although that may change in the future [127]. Currently, four classes of reconfiguration scenarios can be identified with different requirements regarding the handling of state.

(I) Stateless In a stateless reconfiguration, the internal state does not matter and can be discarded, or there is no internal state. This is the case for robust processes, which can quickly recover the state, or for specific event FBs or simple FBs that contain little or no state.

(II) State mapping In most mapping reconfigurations, the application behavior is not changed but only the allocation of resources. In this scenario, the state can be mapped from one FB to another without transformation.

(III) State transformation (SISO) In the simple case of state transformation, the state of one FB is transformed into the state of one other FB, for example, if the FB version is updated and the implementation changes slightly.

(IV) State transformation (MIMO) For complex changes, the state of multiple FBs can be transformed into the state of multiple others. This can be the case when two FBs are replaced by a single new one, or a subapplication is exchanged for

another subapplication.

IEC 61131-3 Online Change

Current PLC software suites based on the IEC 61131-3 standard may encompass an online change feature [89]. Given the IEC 61131-3 execution model, this usually means that the current application program is exchanged for a new version, and the state is mapped, but not changed (Scenario II: State mapping). This allows only for very small changes and requires the care of the developer to not cause catastrophic failures. The fragmented state of the IEC 61499 models allows for much more fine-grained changes with less impact on the execution and more control over the state transformation.

3.2.3 Automatic Generation of Reconfiguration Operations

As identified by Hummer et al. [89], most of the needed reconfiguration operations can be extracted from the difference between two applications, i.e., the delta. In this section, the operations are generated automatically from the delta: After parsing the applications, their contents can be compared, and the operations needed to patch this difference are generated. Yet, the delta fails to capture the intention of the developer and thus complicates the generation of the state transformation. Panzica La Manna [126], for instance, use the behavior models to automatically generate feasible state transformations. Given the lack of fitting behavior models within the IEC 61499 standard, it is assumed that any state transformation is supplied by the system developer until suitable models are available. It may also in principle be feasible to infer state transformations from the ECC, yet most commonly, while the syntax of the change is obvious, the semantics are usually not.

Section 3.2.2 identified four scenarios with different needs for state handling. Following this, the generation of these operations is discussed.

Stateless Reconfiguration

For Scenario I, the stateless reconfiguration, the difference contains all needed information: The added and removed function blocks and connections. As a result, the

needed CREATE, START, STOP, and DEL operations can be added. Every created FB must be started, and every deleted FB must be stopped.

State Mapping Reconfiguration

For Scenario II, not all information is given in the difference. CREATE, DEL, START, and STOP operations can be generated the same as in Scenario I. The mapping of one FB to another must be performed manually and supplied to the generation process. The necessary state mapping m_{map} is defined in Equation 3.2, where s is the source FB and t is the target FB. As a result, the necessary READ and WRITE operations can be added, where the state of s is read, and the state of t is written.

$$m_{\text{map}} = (s, t) \tag{3.2}$$

SISO State Transformation Reconfiguration

Scenario III can be treated similarly to Scenario II, except for an additional state transformation operation added between the READ and WRITE operations. This state transformation operation must be provided by the application developer and may be implemented as an FB. A definition of a state transformation m_{SISO} is given in Equation 3.3, where s is the source FB, t is the target FB, and f is the transformation function.

$$m_{\text{SISO}} = \left(s, t, f(\text{state}_s) \rightarrow \text{state}_t\right) \tag{3.3}$$

MIMO State Transformation Reconfiguration

The operations for Scenario IV are identical to Scenario III. In addition to the state transformation operation between two FBs, this operation must synchronize more than one input and/or output FB. A definition for the MIMO state transformation m_{MIMO} is given in Equation 3.4, where S is a set of source FBs, T is a set of target FBs, and f is the transformation function.

$$m_{\text{MIMO}} = \left(S, T, f(\text{state}_S) \rightarrow \text{state}_T\right) \tag{3.4}$$

3.2.4

Reconfiguration Sequences

After generating the reconfiguration operations from the delta, the operations must be assembled in the right order to preserve the continuity of the application. This preservation can be achieved by updating the system from the event source to the event sink. Traditionally, the IEC 61499 reconfiguration services were implemented in management FBs, which are assembled into reconfiguration applications. In this section, the operations are mapped into a dependency graph, and a linearized sequence is extracted. In particular, the dependencies are added according to a set of rules, and the linearized sequence is found through topological sorting. The reconfiguration sequence could be implemented in a RCA or simply sent to a runtime service that executes the operations. In this section, a linearized sequence was chosen deliberately to prevent concurrency issues and increase determinism, yet the dependency graph can be easily used to create a concurrent reconfiguration plan or configuration manager as described by Wermelinger [120].

Reconfiguration Sequence Definition

A reconfiguration sequence s is an ordered sequence of reconfiguration operations o_i that must be performed to reconfigure an application. It must preserve the order of the operations to prevent undesirable side effects. For example, a new function block must be stopped before it can be connected, and it should be fully connected before the old function block is deleted. Thus, the ordering of the sequence s must ensure that the dependencies of an operation o_i are satisfied by the preceding operations o_0 to o_{i-1}.

$$s = \left[o_0, \cdots, o_n | o_i.D \subseteq \left\{ o_0, \cdots o_{i-1} \right\} \right] \tag{3.5}$$

Sequence Dependencies

Dependencies between evolution regions of interest (EROIs) were introduced by Hummer et al. [89]. In this section, it is proposed to introduce dependencies between the reconfiguration operations themselves. These dependencies lead to the identification of the EROI, since by definition, an EROI is a region that can be reconfigured independently. If a set of operations can be executed independently, this set can be

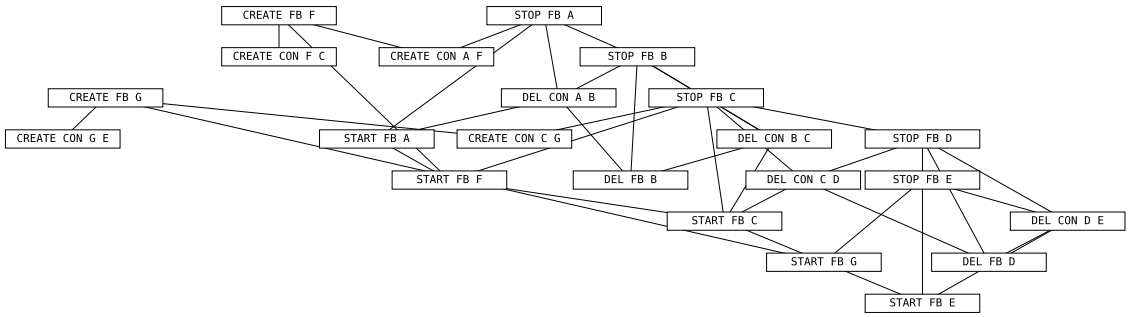


Figure 3.3: A dependency tree for the reconfiguration operations of Scenario 1 (Figure 3.4). A node represents an operation, an edge indicates that the lower operation depends on the higher one.

treated as an EROI.

The dependencies can be injected through rules. The rules are applied to the set of reconfiguration operations and yield the dependencies of each particular operation. Once all dependencies have been added, a dependency graph can be used for visualization, and a suitable reconfiguration sequence can be extracted via topological sorting.

A list of dependency rules is given in Table 3.2. These rules are applied to any operation in the reconfiguration sequence, i.e., if an operation with a CREATE FB instruction is detected for a target FB, it must occur before a corresponding START operation for this particular FB. In this method, any operation may only appear once in a reconfiguration sequence, i.e., an FB cannot be started and stopped two consecutive times since the rules don't differentiate between the operations.

The START and STOP ordering rules ensure that all FBs are stopped and started from the event source to the sink. In this manner, the change can propagate through the application while old events are processed in the old version, and new events are processed in the new version. These rules add dependencies for the event and data connections in the application. In the case of feedback loops, a circular dependency is created that cannot be resolved. A solution to this problem is discussed in Section 3.4.

An example of a resulting dependency tree is given in Figure 3.3. As can be seen, the dependencies leave several implementation choices for the order of execution. The dependencies also allow transparent and explainable reasoning as To why a specific operation is or is not currently allowed. This facilitates a conversation between the automatic tooling and the system developer, who can reproduce why a reconfiguration is or is not possible.

Rule	Description
STOP before START	Existing FBs must be stopped before they can be started.
CREATE FB before START	New FBs must be created before they can be started.
STOP before DEL	Any FB must be stopped before it can be deleted.
START ordering	All FBs must be started in the order of the FB connections.
STOP ordering	All FBs must be stopped in the order of the FB connections.
STOP before CREATE CON	Any FB must be stopped before its connections can be created.
STOP before DEL CON	Any FB must be stopped before its connections can be deleted.
CREATE FB before CREATE CON	Any FB must be created before its connections can be created.
DEL CON before DEL FB	Any connection must be deleted before the FB can be deleted.
DEL CON before START	Any connection must be deleted before the FB can be started.
STOP before QUERY	Any FB must be stopped before its state can be queried.
STOP before WRITE	Any FB must be stopped before its state can be written.
WRITE before START	Any state must be written before the FB can be started.

Table 3.2: Dependency rules for the injection of dependencies into the operations. The resulting dependencies allow the sorting of the operations.

Sequence Ordering

Once the reconfiguration operations are created and the dependencies are injected, the operations can be sorted by a simple topological sorting algorithm based on Kahn's algorithm [128], in which the operations are sorted by the dependency first, and a static priority second. The priorities are defined for every instruction i, e.g., START has a higher priority than STOP. The implementation of the algorithm in Python is given in Listing 10.

The sequence ordering according to the dependencies guarantees that any dependency is fulfilled, i.e., that any execution will be performed either by the old version, or the new version. No events are lost or have to be discarded. The priority ensures that given two choices, the more urgent operation is performed. In general, it is preferable to START as soon as possible and STOP as late as possible to minimize the disturbance. This algorithm works well for many scenarios but can suffer from priority inversion. In particular, a higher priority operation (START) could be blocked by a lower priority operation (DEL CON). The general problem could be extended to an optimization problem if an appropriate cost function was introduced, e.g., min-

```
def topological_ordering(depgraph: DepGraph) -> Sequence:
    """Find a topological ordering of the dependency graph.

    Uses a priority function to determine the next operation from
    the feasible set. Dependency loops cannot be resolved and
    will raise an exception.
    """

    L: list[Operation] = []     # Sorted list of operations
    S = set()                   # Set of feasible operations
    N = set(depgraph.items())   # Set of unfinished operations

    while S := depgraph.feasible_items(set(L)):
        # Choose an op from S and add it to L
        chosen_op = max(S, key=lambda x: priority(x))
        L.append(chosen_op)
        N -= {chosen_op}

    if len(N):
        not_perf_string = "\n".join([str(n) for n in N])
        perf_string = "\n".join([str(n) for n in L])
        print(f"Not performed operations: {not_perf_string}")
        print(f"Performed operations: {perf_string}")
        raise NotImplementedError("Not all operations were
        ↪  performed.")

    return Sequence(L)
```

Listing 10: Iterative algorithm based on Kahn's algorithm [128] to select a topological ordering from the dependency graph.

imizing the disturbance from STOP to START. This would require the quantified disturbance per operation and could also incorporate communication overhead. This optimization problem is not considered in this work.

3.3

Evaluation & Case Studies[15]

After proposing a mechanism for the automatic resolution of dependencies between components of the IEC 61499 models to provide consistency, the behavior can be exemplified and evaluated in multiple scenarios. This section first introduces four hypothetical reconfiguration scenarios, that are then solved using the proposed mechanism.

The four scenarios, as identified in Section 3.2.2, exhibit the feasibility of using the dependency graph to generate a topological sorting of the reconfiguration operations. The necessary operations and the potential impact on the execution are presented for each scenario. For the sake of brevity, only the relevant FBs of an application are displayed. All other function blocks in the application can continue without interruption unless they have connections through the affected FBs.

The different phases according to Sünder et al. [129] are indicated in each scenario. The startup sequence contains the creation of new FBs and connections. The reconfiguration sequence is when the operation is disrupted. It is marked by the first STOP operation and ends when all FBs are started again. The closing sequence deletes the remaining connections and FBs. Some services, such as CREATE TYPE or CREATE RES are omitted here. These operations would always occur during the RINIT or RDINIT phases.

3.3.1

Scenario I: Stateless Reconfiguration

Figure 3.4 shows a reconfiguration in which two FBs are replaced by two other FBs. In this particular scenario, the exchange is stateless, i.e., the states of FBs B and D are discarded, and FBs F and G start from their initial states.

[15] Major parts of this section were published in L. Prenzel and S. Steinhorst. "Automated Dependency Resolution in Dynamic Reconfiguration for IEC 61499". In: *Proceedings of the International Conference on Emerging Technologies and Factory Automation (ETFA)*. Västerås, Sweden: IEEE, 2021.

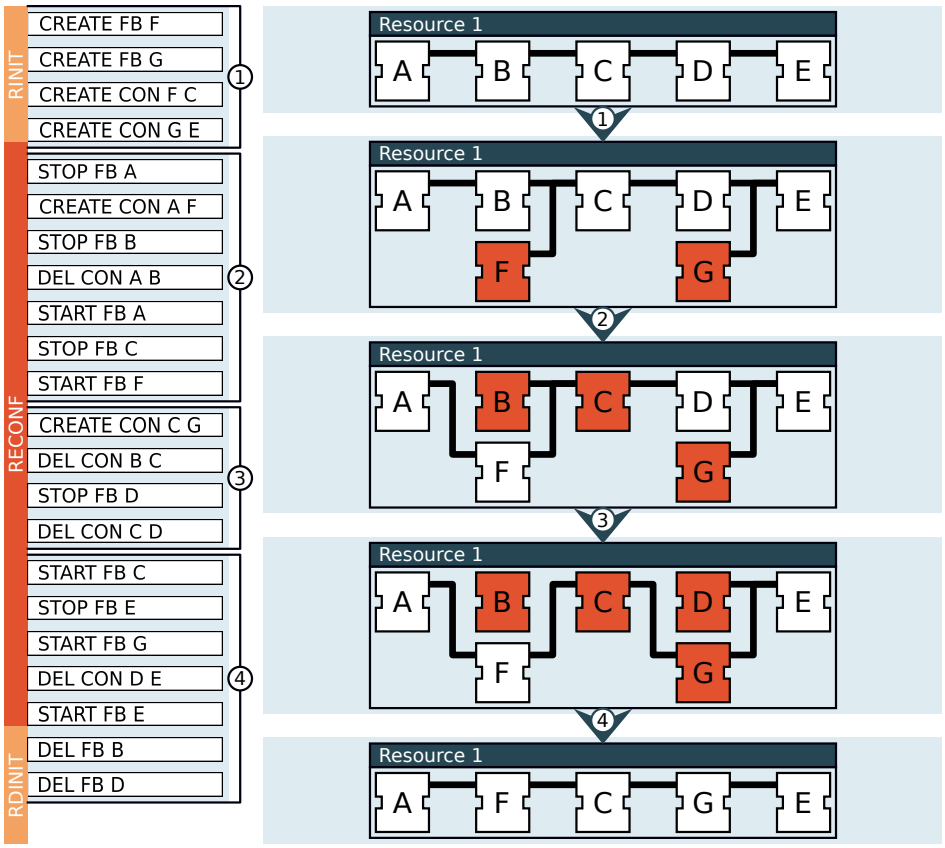


Figure 3.4: Scenario I: Function blocks B and D are replaced by function blocks F and G, respectively. A resulting reconfiguration sequence with the RINIT, RECONF, and RDINIT-phases is displayed on the left. The evolution of the application is shown on the right. The internal state is not carried over in this scenario.

Operations

The reconfiguration requires operations to create, start, stop, and delete the affected FBs in the correct order. According to the flow of events and data, the reconfiguration must occur from FB A to FB E, i.e., the FBs must be stopped from left to right and started from left to right. In this manner, the integrity and continuity of the event flow can be guaranteed. Figure 3.3 shows the dependency tree for this scenario.

Impact

After adding the new FBs, the two FBs are exchanged one after another. This allows the reconfiguration of FB B to start, while FB D can still process old events. Similarly, FB F can already continue, while FB G is still under reconfiguration.

3.3.2

Scenario II: FB Mapping Reconfiguration

The second scenario investigates the reconfiguration sequences for the redistribution of FBs from one resource to another (Figure 3.5). Two function blocks from Resource 1 (C, D) are shifted to Resource 2, which requires the addition of two communication FBs (X and Y).

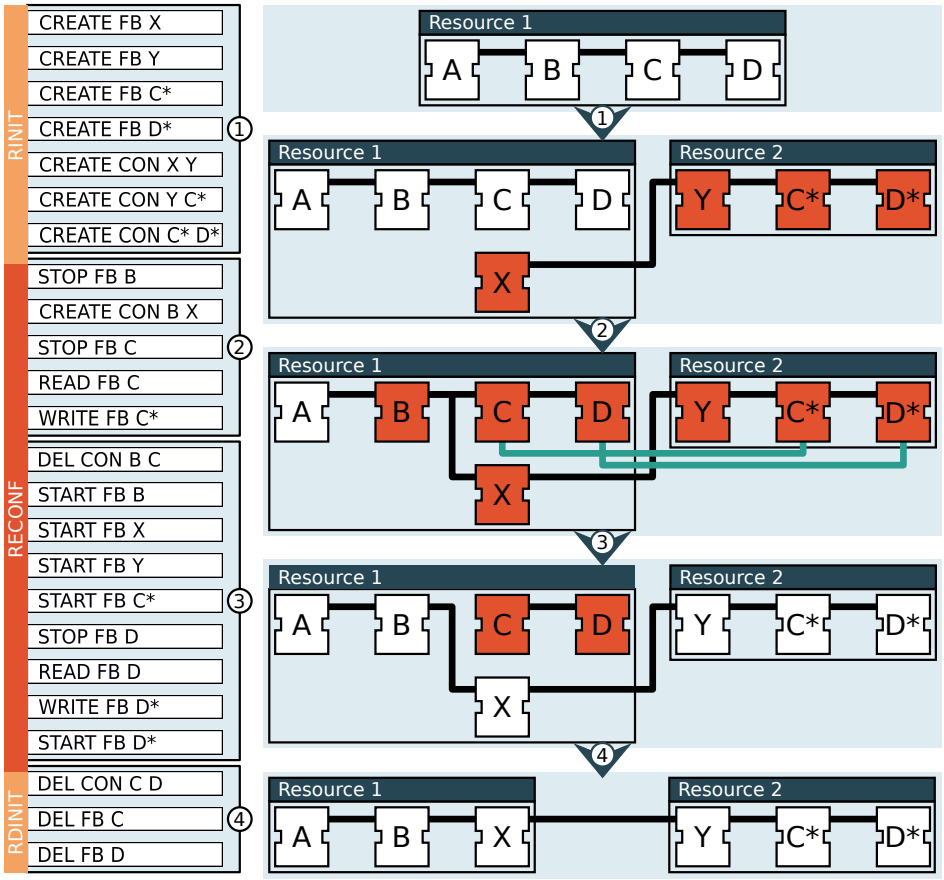


Figure 3.5: Scenario II: The mapping of FBs to resources is reconfigured. Throughout the reconfiguration, the continuity must be preserved. Thus, the internal state of FB C must be read and written before C* can be started.

Operations

Similar to Scenario I, the operations to create, start, stop, and delete are added as needed. In addition, two operations to read and write are added as well, to map the state between the resources.

Impact

Initially, the new function blocks and connections are added. During the critical phase, FB B is stopped to prevent further execution. FB C is stopped first, and the state is read and written to FB C*. This allows FBs X, Y, and C* to be started while the state from D is read and written to D*. If concurrent or parallel execution were possible, some of these operations could be performed in parallel, thus further reducing the overhead. It is noteworthy that the distinction between the RECONF and RDINIT phase starts to blur, as the execution can be reinstated continuously.

3.3.3 Scenario III: SISO State Transformation

In Scenario III (Figure 3.6), an FB is replaced by another FB with an explicit state transformation. This new FB may be a new version of the same type, or another FB altogether. The key difference to Scenario II is the explicit transformation of the state between the reading and the writing of the state.

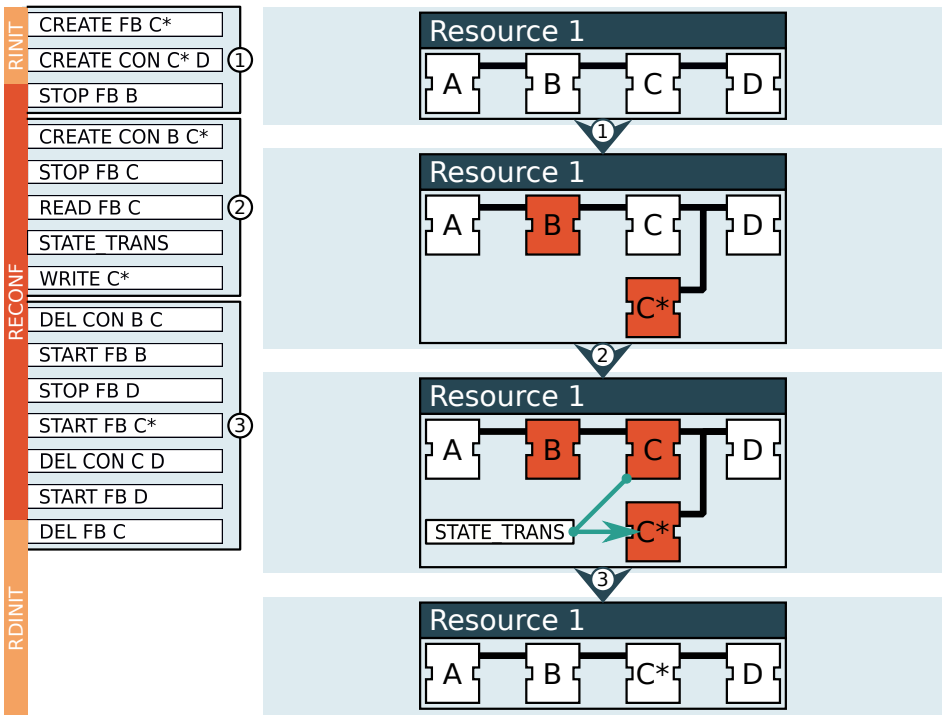


Figure 3.6: Scenario III: The version of a function block is changed, thus requiring an explicit state transformation between the stopping of FB C and the start of FB C*.

Operations

A state transformation operation is added to explicitly transform the state between the READ and WRITE operations. Other operations remain similar to scenarios I and II.

Impact

This change has a particularly small impact since only the FB before the one to be exchanged must be suspended momentarily. The biggest difficulty is the identification of the state transformation, which can be performed offline. Given the fragmented state of the IEC 61499 application, the state of an FB is small and consists of the ECC state and the set of variables (input, output, internal).

3.3.4 Scenario IV: MIMO State Transformation

In the MIMO state transformation case, the state of multiple FBs is needed to reconfigure one or multiple FBs. In this particular scenario, the state of three FBs must be read and transformed to reinitialize a new FB. FB B* replaces FBs B and E, while also requiring the state of C.

Operations

In addition to the operations discussed before, the state transformation in this scenario requires the synchronization with three READ operations. Thus, the state transformation can only start once the states of FBs B, C, and E are read.

Impact

Stopping FB A suspends the arrival of new events, while previous events can still be processed. Once the connections are rewired to FB B*, FB A can be restarted, and FB B* is started as soon as the state is transformed. The required inputs of the state transformation lead to a further synchronization of the sequence, thus also increasing the overall impact.

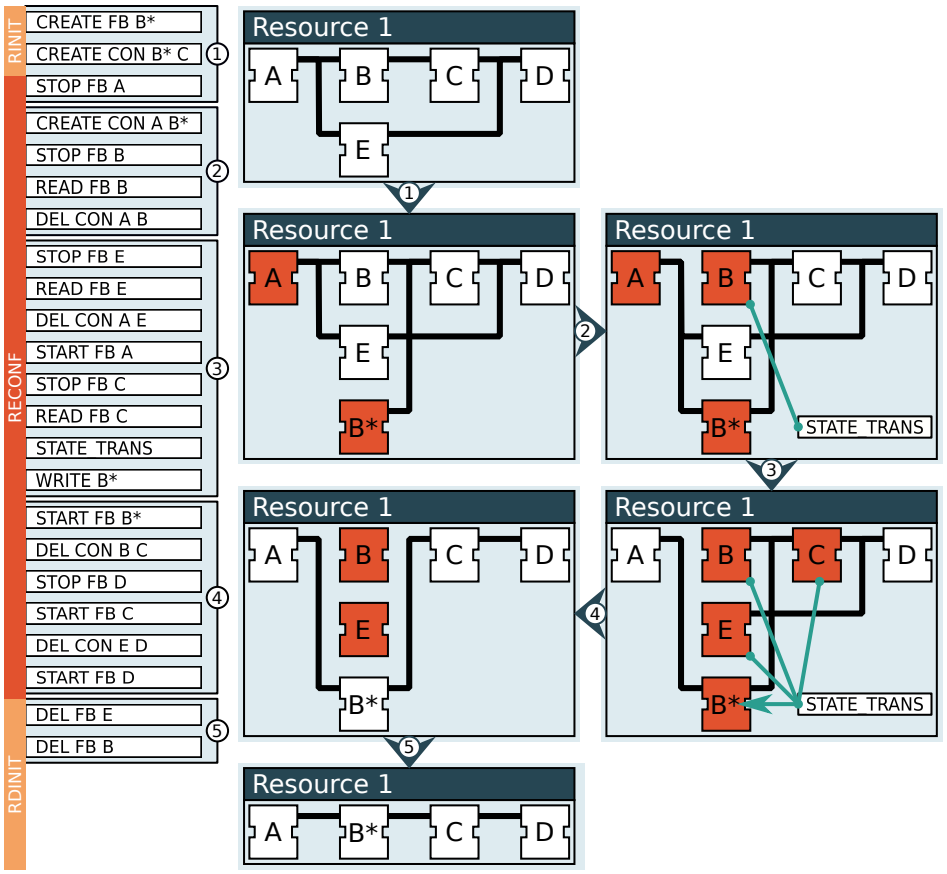


Figure 3.7: Scenario IV: The version of a function block is changed, thus requiring a state transformation.

3.3.5

Discussion

The reconfiguration scenarios show how the methodology enables the automatic generation of reconfiguration sequences that preserve the continuity of the application throughout the reconfiguration. The changes in the applications sweep through the FB networks from source to sink. They also indicate the elements that currently cannot be automated: The state mapping and the state transformation. This information is not presently included in the IEC 61499 application but is represented by the intent of the system developer. Nevertheless, this added layer of abstraction simplifies and facilitates the utilization of dynamic reconfiguration for a range of reconfiguration scenarios.

3.3.6

Conclusion

The transition from traditional manufacturing systems to more adaptable or even (partly) autonomous systems is feasible. A major step in this direction is taking the human out of the loop and automating the reconfiguration procedure as much as possible. Dynamic reconfiguration as a topic has been addressed in the scope of component-based systems, the IEC 61499 standard, and even in industrial applications with the IEC 61131-3 standard.

This section proposed a methodology to automatically generate reconfiguration sequences for component-based systems, as they are defined by the IEC 61499 standard. These sequences may be implemented in different forms, for instance, reconfiguration applications. By automatically generating the required operations and ordering them according to their dependencies, the continuity of the provided services can be guaranteed. This automatic adaptivity provides the foundation for an architecture such as the WATERBEAR architecture.

3.4

Key Findings

Adaptability does not imply *safe* adaptability. Countless things can go wrong during an adaptation. Windows users are far too familiar with things breaking after an update, and these updates usually aren't even applied on the fly. For ICS, *safe* dynamic adaptation must involve the physical process and the behavior of the control system. Functionally, the adaptation may not lead to a violation of the specification, which lays out the admissible interactions between the control system and the physical process. For instance, if a switch needs to be activated once every 24 hours, the dynamic adaptation needs to remember if the switch was already activated, and when it has to be activated again.

This chapter showed the implementation of a correct-by-design algorithm to automatically generate safe adaptation or reconfiguration sequences. The implementation is based on consistency requirements and conditions in research. An existing consistency condition (i.e., quiescence) is applied to the domain-specific modeling language of the IEC 61499 standard. Using this condition, an algorithm is imple-

mented that automatically selects a safe reconfiguration sequence of operations that modifies a system while preserving the intended application behavior. An evaluation demonstrates the algorithm on a selection of relevant adaptation scenarios and shows that the consistency of the functional behavior is guaranteed.

3.4.1 Choice of Consistency Conditions

Research has come up with multiple consistency conditions, e.g., *quiescence*, *serenity*, and *version consistency* [117–119]. Each condition extends the previous one in one way or another. Ultimately, they all make different assumptions about the system, the transactions, and the available information that describes the system. In practice, not all information may be available and this limits the applicability of these conditions.

This chapter uses a variation of *quiescence* to guarantee the consistency of the transactions. This condition is particularly suitable for the problem at hand because of the one-directional interactions between function blocks (FBs) with strictly defined interfaces. In this way, the reconfiguration sequence updates the system from the event source to the event sink while pushing old events out of the system as new events arrive. Components are passivated during the adaptation. Consequently, the consistency of the execution is guaranteed because one by one, components will be in a state where all previous transactions have finished and no *new* transactions (of the old behavior) can be launched. As components are activated again from the event sink, they start execution in the new behavior, thus achieving consistency.

While quiescence is simple to implement and, together with the IEC 61499 execution semantics leads to satisfactory results, other conditions may be just as or more suitable. Version consistency, for example, is a very high-level condition that ensures global consistency with distributed transactions [123]. This detailed transaction model does not currently exist within the IEC 61499 standard, and therefore, a simpler consistency condition was chosen in this work.

Given the current state of the IEC 61499 standard and the given information, quiescence is straightforward to implement and can handle a variety of scenarios. When better formal models of the execution behavior are available, other consistency conditions should also be evaluated. One shortcoming of quiescence is the handling of loops, which is discussed in the next section.

3.4.2

Consistency in Feedback Loops

Feedback loops lead to circular dependencies between components. These loops cannot be easily handled by quiescence. This topic is also addressed in the works on quiescence, where a possible solution is a partial passivation of the component to let the feedback loop finish [117].

Unfortunately, this partial passivation requires the knowledge of how to treat each input/output. Since the IEC 61499 standard defines an executable model, most of the information is available inside the component, which would allow the unraveling or flattening of the feedback loop. This, however, means that the behavior of the component in detail must be considered, which quickly complicates the implementation of quiescence. In principle, it is also possible to implement components that do not behave deterministically, e.g., by introducing random behavior or by relying on external communication interfaces. In this case, unraveling a feedback loop could prove impossible. Either way, proper behavior modeling of the component could solve this issue and could circumvent the need for exhaustive simulation of the models.

While in the current algorithm, feedback loops are prohibited and cannot be resolved, this extension is rather straightforward once it is known how the feedback loop should be treated. Partial passivation is a possible solution, yet neither partial nor any passivation (apart from stopping, which leads to event loss) is part of the IEC 61499 models.

3.4.3

Execution Semantics and Ambiguities

Consistency is required on the behavior level. The system behavior is defined by the executed model and the underlying execution semantics. The IEC 61499 standard partially defines the execution semantics and the language to create the application model. Issues with the application model (e.g., feedback loops) were discussed before, where the behavior models of the application are insufficient to resolve circular dependencies between components.

Furthermore, ambiguous execution semantics make it impossible to derive a clear consistency condition, since the behavior can be inconsistent even without

an adaptation taking place. For instance, the separation between data and event connections and the different scheduling of both obstruct most efforts in achieving consistent behavior. Furthermore, important mechanisms such as suspension (or even partial suspension) are not currently part of the IEC 61499 standard. Both the comparison with Erlang, as well as the analysis of the theory behind consistency conditions, lead to the necessity of passivation. While this can be fixed within an implementation by modifying the event scheduler, this further complicates the issue with data connections. Passivating event dispatches but not corresponding data changes means that preserving consistency on that level is infeasible. In Erlang, for instance, all messages can be queued and when a component is passivated, it simply stops retrieving messages. In most IEC 61499 implementations, events are queued, but data updates are not. Thus, when passivated, important data updates may be lost and all consistency conditions disappear.

Consistency in dynamic adaptation of component-based systems has been researched for decades. In practice, consistency requires clearly defined models and unambiguous execution semantics tailored for consistency. With the current state of the IEC 61499 models, achieving consistency is difficult to impossible without addressing the ambiguity in the application model and execution semantics. This chapter tried to bridge this gap and assumed smaller changes to the semantics to allow consistency to be achieved. Strong consistency guarantees, especially in a distributed setting or between different runtimes, would require much stricter semantics and guarantees than the IEC 61499 standard currently offers.

Chapter 4

Real-time Adaptation of Industrial Control Systems

Contents

The defining characteristic of dynamic adaptation is that it is performed *dynamically*, i.e., during the execution. If a real-time system is adapted dynamically, then the real-time requirements of the system must also apply during the adaptation. The previous chapter introduced the automatic generation of consistent reconfiguration sequences. However, consistency alone does not factor in the real-time constraints. While the state of the application may remain consistent, the execution of the system may be disrupted for too long, which can lead to deadline misses, failures, or catastrophic consequences. This is unacceptable for an ICS whose purpose is to be dependable and to prevent failures.

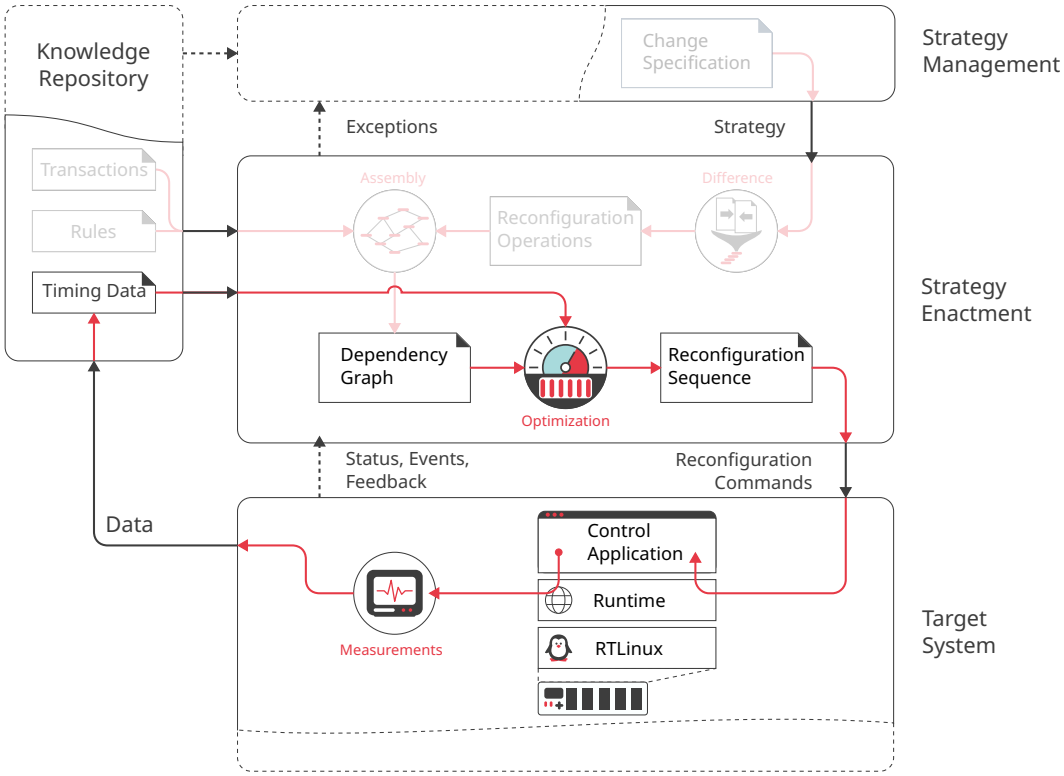


Figure 4.1: In this chapter, the previous sorting algorithm is replaced by an optimization mechanism that uses timing information. This timing information can be gathered from the system itself.

As seen in Chapter 2, the IEC 61499 execution semantics are ambiguous and there have been different interpretations in practice. Most implementations gravitate towards an event-triggered execution which also facilitates dynamic adaptation. An event-triggered execution, however, complicates the real-time analysis compared to a cyclic execution. The most sophisticated real-time model for an IEC 61499 implementation was developed by Zoitl et al. [130]. This model, unfortunately, does not consider the real-time execution during the adaptation. Additionally, the models of the IEC 61499 standard do not integrate the suspension functionality that was needed in the previous chapter to achieve consistency. As a result, guaranteeing real-time performance was up to now not feasible, yet it is urgently needed for an architecture like the WATERBEAR architecture to succeed.

This chapter bridges the gap between the existing real-time models and dynamic adaptation by extending the real-time models to cover the dynamic adaptation phase. This requires the consideration of shared access and resource contention between the real-time tasks in the system that have not been previously addressed

for the IEC 61499 models. The extended model allows the application of common schedulability conditions, which then allows the optimization of the reconfiguration sequence to not just consider consistency, but also minimize the real-time disturbance. The scope of this extension is visualized in Figure 4.1. The timing data needed for the optimization can be gathered from measurements of an actual implementation. Using this procedure, both the consistency and the timeliness of the dynamic adaptation procedure can be guaranteed. Finally, the agility of this adaptation procedure is evaluated to gather a glimpse of how quickly an architecture like the WATERBEAR architecture could adapt with currently feasible technologies.

The chapter first introduces the fundamentals of real-time scheduling for ICS in Section 4.1, particularly concerning the event-triggered execution of the IEC 61499 models. Following this, the existing real-time models of the IEC 61499 standard are extended to cover the dynamic adaptation phase in Section 4.2. Finally, the agility of an existing IEC 61499 runtime environment is evaluated in measurements (Section 4.3). The chapter concludes with the key findings in Section 4.4.

4.1
Real-time Scheduling and Reconfiguration of ICS

The fundamentals of industrial control software are detailed in Section 2.1. In this section, the existing work on real-time scheduling and reconfiguration, specifically with a focus on the IEC 61499 standard, is summarized.

4.1.1
Real-time Execution of IEC 61499

As seen in Section 2.1.2, the IEC 61499 standard can be implemented using different execution semantics. This is due to some ambiguity on the side of the standard. Nevertheless, over the years, the topic of real-time execution of the IEC 61499 models has been approached, since this is crucial for safety-critical hard real-time systems.

Generally, there are different ways to achieve real-time performance with the IEC 61499 models. One approach is the use of a scan-based execution cycle as used in regular IEC 61131 languages. By reserving execution time for each FB in every cycle, real-time performance can be simple to guarantee, yet there is a large overhead [131] and it may lead to event-loss [70]. This approach is also not particularly suitable for

reconfiguration, since the same reconfiguration problems of the IEC 61131 standard emerge (e.g., global state, monolithic execution). Thus, this section only considers event-driven execution.

The underlying event-driven execution model is proposed by Zoitl et al. [130], where the event-chain concept is introduced. The most comprehensive real-time model for the IEC 61499 standard exists in Zoitl [50]. This chapter builds upon the theory of this work, which is briefly summarized in the following. Afterward, the dynamic adaptation procedure and its impact on the real-time execution is recapped.

4.1.2 Real-time Execution Model

The most elaborate real-time execution model for the IEC 61499 standard to date was done by Zoitl [50]. This model assigns a real-time task to every event chain, i.e., every chain of FB executions starting at an event source and ending at an event sink. Zoitl [50] proposes the use of either fixed priority or dynamic priority scheduling for the IEC 61499 models.

Fixed Priority

The main scheduling paradigm for these types of systems is preemptive fixed priority scheduling, e.g., rate monotonic (RM) or DM scheduling. Zoitl [50] discusses both of these. Given that RM scheduling is only optimal for periodic tasks where the deadline is equal to the rate, DM is chosen as the most suitable paradigm for the IEC 61499 models. As was seen in Chapter 2, the main issue in implementing a real-time runtime environment is the lack of scheduling information. A solution is the use of special real-time FBs that specify the necessary scheduling information on their interfaces. A remaining issue of DM (and similar, but less significant for RM) scheduling is the relatively low schedulability bound of 58.6 %, i.e., the system must reserve considerable resources.

Dynamic Priority

This potential waste of resources can be diminished by dynamic priority scheduling. In this case, the priorities of tasks are assigned at runtime. This represents additional

load and a source of uncertainty, yet scheduling paradigms such as EDF scheduling can be optimal up to 100 % utilization.

For the IEC 61499 models, the choice of fixed or dynamic priority scheduling must be implemented in the event dispatcher that schedules the execution of FBs. This is equivalent to the assignment of events to run queues in Erlang (see Chapter 2). Unlike general fixed priority scheduling mechanisms, Erlang allows only for three priority levels. While this does limit its use for real-time systems, the frequent preemption and fair allocation within each priority level maximizes the responsiveness of the system. This can effectively reduce the impact of blocking behavior, which must be considered for the IEC 61499 execution models.

Blocking

There are different types of blocking behavior. Zoitl [50] identifies two main blocking cases: Shared access of multiple tasks to the same FB and data connections between multiple tasks. Yet, there is at least a third type of blocking, as will be seen in Section 4.2.2, which is the suspension of FBs during an adaptation.

To date, while there are many different runtime environments for the IEC 61499 standard, not many support the necessary real-time semantics. The open source implementation 4diac IDE ([81]) and the corresponding runtime environment 4diac FORTE offer an implementation of the event chain execution model and support for real-time FBs, yet this is rarely used in practice.

4.1.3 Dynamic Adaptation Procedure[16]

There is a framework for the Dynamic Reconfiguration of the IEC 61499 standard. This framework uses the IEC 61499 models as the programming language in which the reconfiguration is instructed. The rudimentary framework is integrated into the standard itself with numerous extensions [50, 63, 90, 129].

The general idea is to launch a reconfiguration control application (RCA) in parallel with the regular control application, which uses the management interface of the IEC 61499 devices to change the control application [129]. This approach has

[16] Major parts of this section were published in L. Prenzel and S. Steinhorst. "Towards Resilience by Self-Adaptation of Industrial Control Systems". In: *International Conference on Emerging Technologies and Factory Automation (ETFA) 2022*. Stuttgart, Germany: IEEE, 2022

many upsides, e.g., a low latency, access to the full semantics of the IEC 61499 models for failure handling and reconfiguration logic, and the benefits that the IEC 61499 models introduce for modeling distributed control applications. In contrast to other approaches, e.g., the *hot code loading* capabilities of Erlang (see Section 2.1.4), the major downside is that this can lead to fairly large RCAs.

Running the RCAs concurrently with the control application simplifies the satisfaction of real-time constraints because the same scheduler of the control application can be used to interleave the RCAs while keeping all deadlines.

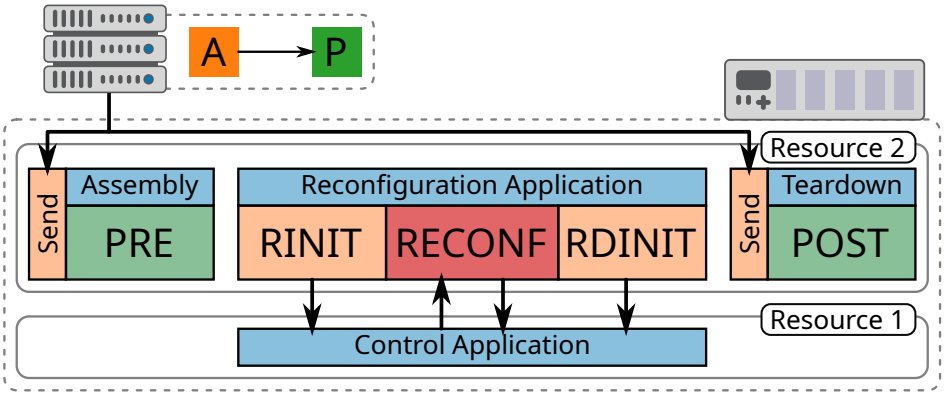


Figure 4.2: Once a reaction is decided, the RCA must be assembled on a separate resource. During its execution, it interacts with the control application. Eventually, it can be disassembled in the tear-down phase.

The general method of adaptation is displayed in Figure 4.2. The reconfiguration process is triggered from the outside, and an RCA is sent to a second resource on the same device. The individual phases are explained in the following section.

Adaptation Phases

The adaptation of ICS using the IEC 61499 standard can be split into five phases [90] that deal with the assembly, execution, and tear-down of a RCA (Figure 4.2). The RCA is executed in parallel with the control application and modifies it on the fly. The five adaptation phases are:

PRE The RCA is transmitted to the control device and assembled in a separate resource using the same operations used for the reconfiguration. This phase is non-critical and can be performed over a long time.

RINIT After the RCA is started in parallel with the control application, non-critical operations, such as the addition of FBs, can be performed. At this stage, the

RCA is executed concurrently with the control application, but with a lower priority.

RECONF In this critical phase, parts of the control application are suspended to prevent unpredictable state changes, and the real-time behavior is disturbed. In a well-designed RCA, the disturbance must not cause the violation of a deadline [7].

RDINIT Once the execution of the control application is resumed, the final non-critical operations remove left-over elements on the application resources, such as removed FBs. This can be performed concurrently with a lower priority.

POST Eventually, the RCA has to be removed to free the resource for further reconfigurations. In this phase, the RCA can either be disassembled, or the entire resource could be deleted.

Splitting the adaptation into these five phases illustrates that not everything during the adaptation is critical to the real-time performance and that there is an inherent overhead (e.g., transmission of the changes) to every adaptation. Further, the concurrent nature of the adaptation indicates a clear benefit of multi-threading, even if the critical *RECONF* phase is only single-threaded. In the following section, the timing impact of the adaptation process is further illuminated.

Timing Impact of Dynamic Adaptation

Performing a dynamic adaptation can affect the real-time performance of the control application. Ideally, this impact must be minimized and real-time constraints, such as deadlines, must be satisfied at any point during the adaptation. The previous section introduced the five phases of the adaptation process. The only critical phase is the central *RECONF* phase, in which the control application is actively modified [50, 90].

Without going into the details of each phase, the overall duration of the adaptation process can be calculated as the sum of the individual durations of each phase:

$$d_E = d_E^{\text{PRE}} + d_E^{\text{RINIT}} + d_E^{\text{RECONF}} + d_E^{\text{RDINIT}} + d_E^{\text{POST}}. \tag{4.1}$$

Each duration d depends on the execution time C, which is a result of the complexity of the adaptation and is directly affected by the utilization U of the resource. Generally, the exact duration is affected by the scheduling, yet a good estimate can be achieved by dividing the execution time C by the utilization U:

$$d_E^X = \frac{C}{1-U}. \tag{4.2}$$

Given that the FBs that make up the execution of all phases are short-running and must terminate quickly, and RCAs are commonly made up of many FBs, this estimate is accurate since small scheduling differences will average out. Equation 4.1 indicates that the overall duration during which the application is affected is longer than the critical *RECONF* phase, which also means that the response time to the triggering of an adaptation is substantially longer than the *RECONF* phase. On the other hand, only a small fraction of the overall adaptation is real-time critical, which means that most of the attention should be directed onto this phase. Further, Equation 4.2 demonstrates that while a high utilization is generally attractive, this will prolong the response time of the adaptation process.

4.2 Schedulability of Dynamic Adaptation[17]

Having seen the existing works on dynamic adaptation of the IEC 61499 models, this section addresses the schedulability of dynamic adaptation since this was not addressed in previous works such as [50]. First, the execution of the IEC 61499 standard is modeled using preemptive rate monotonic (RM) scheduling with shared resources. This model allows the incorporation of the delay introduced by a sequence of reconfiguration operations into the schedulability condition. From this model, a schedulability condition can be extracted that allows the optimization of the ordering of the reconfiguration sequence, which extends the works proposed in Chapter 3. Two examples demonstrate that an optimization algorithm using the timing information outperforms the heuristic algorithm by up to 85 %.

The system model and resulting scheduling problem are defined in Section 4.2.1. Section 4.2.2 describes the blocking duration and its calculation. Two examples demonstrate the approach in Section 4.2.3.

[17] Major parts of this section were published in L. Prenzel, S. Hofmann, and S. Steinhorst. "Real-time Dynamic Reconfiguration for IEC 61499". In: *International Conference on Industrial Cyber-Physical Systems (ICPS)*. Coventry, United Kingdom: IEEE, 2022.

4.2.1

System and Problem Definition

This section first defines the system model, emphasizes the resulting scheduling problem that occurs during the reconfiguration, and argues about the timeliness of the reconfiguration. The original scheduling problem that did not consider the reconfiguration or access to shared resources was defined by Zoitl [50].

System Model

The tasks of the system model are the execution traces or event chains within the component-based architecture. This is in line with previous models [50]. These tasks or traces can be modeled as directed acyclic graphs (DAGs). The task-set

$$\tau = (\tau_1, \ldots, \tau_n) \tag{4.3}$$

consists of n tasks that may occur concurrently. Each task τ_i is characterized by (V_i, E_i, D_i), where the vertices $V_i = (e_{i,0}, \ldots, e_{i,j})$ are distinct executions of components, in this case IEC 61499 FBs, the edges E_i are the precedence constraints between executions, and D_i is a deadline of the task that must be kept. Executions $e_{i,j} = (f_{i,j}, c^e_{i,j})$ are defined by the FB $f_{i,j}$ and a WCET $c^e_{i,j}$. FBs are shared resources that can only be used in one execution e at a time. The WCET $c^e_{i,j}$ depends on the algorithms, execution control, state, and the execution platform. A fixed, known WCET for every execution is assumed.

Next, the reconfiguration sequence is modeled that modifies the application with a set of reconfiguration operations. The ordered sequence

$$S = \langle o_0, \ldots, o_n \rangle \tag{4.4}$$

consists of reconfiguration operations $o_i = (a_i, F_i, c^o_i)$, where a_i is an action, F_i is a set of affected FBs, and c^o_i is the WCET of the operation. The actions can interfere with the execution of a particular FB, e.g., a *stop* action will suspend all affected FBs until they receive a corresponding *start* action. A suspended FB is blocked from being executed and events must be preserved. This suspension is an important mechanism to guarantee the consistency of the reconfiguration [117]. It enables reconfiguration of an application from source to sink, which causes all events to be processed according

to either the old version, or the new version, but not a mixture [6].

Scheduling Problem

The resulting scheduling problem is to determine whether all real-time tasks satisfy their deadlines when the reconfiguration sequence is applied. By itself, this problem is not well-defined, thus, the analysis is based on the following assumptions:

1. There is only one single-threaded resource that executes both the tasks and the reconfiguration sequence using rate monotonic (RM) scheduling.
2. The resource can be preempted, but access to a FB is blocking, i.e., a FB cannot be executed by a second task before the first task has released it.
3. Each real-time task τ_i has a unique rate T_i identical to its deadline D_i. It cannot be triggered more frequently than its rate.
4. The reconfiguration sequence S is not a real-time task and has no deadline. Thus, it has the lowest priority and may be preempted by real-time tasks. It is only executed once.

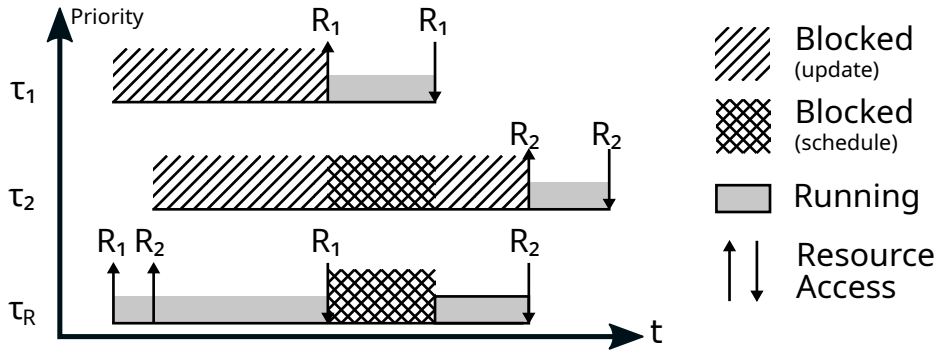


Figure 4.3: A reconfiguration (τ_R) introduces additional blocking behavior within the real-time tasks (τ_1, τ_2). The blocking duration depends on the performed reconfiguration and must be assessed before the reconfiguration is applied.

Given that the reconfiguration is not a real-time task, its impact can nevertheless not be neglected, since it may introduce blocking behavior that causes a priority inversion when the reconfiguration suspends particular FBs. To mitigate the priority inversion, the priority ceiling protocol (PCP) is utilized, which elevates the priority of a task if it blocks a resource with a higher priority ceiling. This behavior is depicted in Figure 4.3, where the sequence S blocks tasks τ_1 and τ_2, because it requested their resources before the tasks were triggered. Sha, Rajkumar, and Lehoczky [132]

define a schedulability condition for n periodic tasks using PCP and preemptive RM scheduling:

$$\forall i, 1 \leq i \leq n, \quad \underbrace{\frac{C_i}{T_i}}_{\text{Execution}} + \underbrace{\frac{B_i}{T_i}}_{\text{Blocking}} + \underbrace{\sum_{j=0}^{i-1} \frac{C_j}{T_j}}_{\text{Preemption}} \leq \underbrace{i(2^{1/i} - 1)}_{\text{Max. Utilization}} \tag{4.5}$$

This condition assumes a task-set $\tau = \{\tau_1, \ldots, \tau_n\}$, where τ_n has the lowest priority and τ_1 has the highest priority. This is compatible with the system model, which can be sorted in ascending order by their rate $T_i = D_i$ to fit the requirement. The WCET of a task τ_i with m executions can then be calculated as

$$C_i = \sum_{j=1}^{m} c_{i,j}. \tag{4.6}$$

The condition in Equation 4.5 consists of three components. The first component is the contribution of the tasks WCET to the overall utilization. The second is the contribution of blocking by other, lower-priority tasks to the utilization. Third is the cumulative preemption of tasks with higher priority. The sum of all components must be lower than the maximum utilization for a task-set of i tasks using preemptive RM scheduling. In the model, the blocking time

$$B_i = B(\tau_i) = B^{\text{FB}}(\tau_i) + B^R(\tau_i) \tag{4.7}$$

has two contributors: $B^{\text{FB}}(\tau_i)$ is caused by shared access to the FBs, and $B^R(\tau_i)$ is a result of the concurrent execution of the reconfiguration sequence S. Together, they determine how long a task is blocked by a lower-priority task due to the PCP.

Thus, the problem remains of determining the blocking durations $B^{\text{FB}}(\tau_i)$ and $B^R(\tau_i)$ for every task given a reconfiguration sequence. Once these values are known, the schedulability condition will decide if the system is schedulable and can thus meet its deadlines. The next section shows how the blocking durations can be determined. Afterward, an optimization problem is proposed to minimize the blocking time of each task by optimizing the order of reconfiguration operations.

4.2.2

Blocking Duration

The reconfiguration sequence will temporarily suspend FBs that may be used by other real-time tasks. This is a priority inversion, which can be solved using the priority ceiling protocol (PCP). To satisfy the schedulability test in Equation 4.5, the blocking time $B(\tau)$ has to be determined. The contributors of $B(\tau)$ can be seen in Equation 4.7.

Function Block Blocking Time $B^{\mathsf{FB}}(\tau_i)$

When two tasks share a FB, the higher priority task may have to wait until the lower priority task has released the FB. This execution cannot be preempted, since this would lead to an inconsistent state. With $F(\tau_i)$ defined as the set of FBs used by task τ_i, this blocking time can be calculated as

$$B^{\mathrm{FB}}(\tau_i) = \sum_{j=i+1}^{n} \left(\sum_{f_k \in F(\tau_j) \cap F(\tau_i)} c_{j,k} \right), \tag{4.8}$$

that is the sum of the WCET of all executions in tasks with lower priorities that use an FB that is also used in τ_i. More intuitively, it is the maximum duration a task may be blocked by another task with a lower priority because they require access to the same FBs.

Reconfiguration Blocking Time $B^R(\tau_i)$

The blocking time $B^R(\tau_i)$ depends on the reconfiguration sequence S. An example is given in Figure 4.4. Every reconfiguration sequence can be split into three phases: An initialization phase RINIT, a critical phase RECONF, and a de-initialization phase RDINIT [90]. The critical RECONF phase begins when the first shared resource, in this case a FB, is stopped. It ends when the last FB is started, and the execution of the real-time tasks can continue. During the RDINIT phase, leftover components are cleaned up. Using preemptive scheduling, RINIT and RDINIT do not disturb the real-time execution, since they can be prolonged indefinitely and are executed when no other real-time task is running.

Intuitively, $B^R(\tau_i)$ is the duration of the sequence S during which there is a

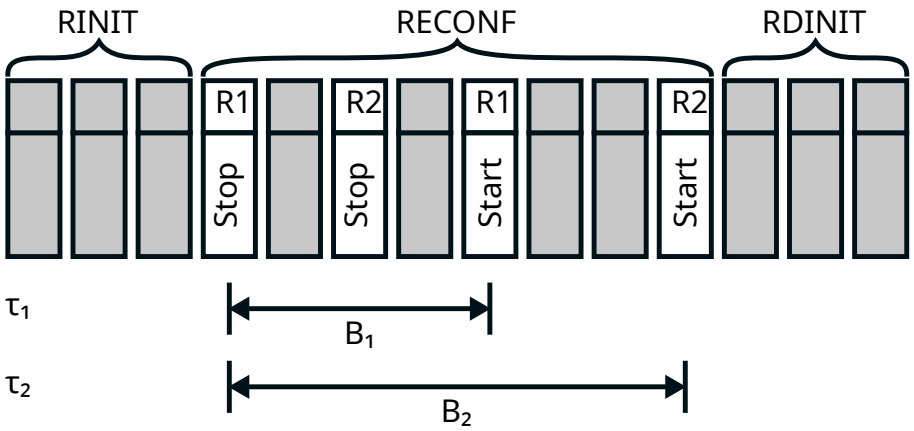


Figure 4.4: The reconfiguration sequence S can be split into three phases: RINIT, RECONF, and RDINIT [90]. The blocking time of a task lasts while a resource with a higher priority ceiling is suspended.

suspended FB with a higher priority ceiling than the priority of task τ_i. Formally, the priority function

$$\pi^{\text{FB}}(f) = \max_{\tau_i | f \in F(\tau_i)} \pi^{\tau}(\tau_i), \tag{4.9}$$

can be defined, which returns the priority ceiling of a FB f based on the priority $\pi^{\tau}(\tau_i)$, which can be assigned to each task according to its rate/deadline.

The notation $S_{p,q} = \langle o_p, \ldots, o_q \rangle$ represents the sub-sequence of $S = \langle o_0, \ldots, o_n \rangle$, where $0 \leq p \leq q \leq n$. The existence of a suspension function $\text{sus}(S_{0,q})$ is assumed, which returns the set of FBs suspended after the occurrences of the sequence $S_{0,q}$. This function must check if a FB was stopped or added during the sequence but not yet started. Then, the function

$$\pi^{R}(o_i) = \max_{f \in \text{sus}(S_{0,i})} \pi^{\text{FB}}(f) \tag{4.10}$$

calculates the priority ceiling of a reconfiguration operation as the maximum priority ceiling of any FB suspended during the occurrence of operation o_i. $\pi^{R}(o_i)$ decides whether operation o_i can preempt another real-time task. Now the blocking sequence

$$S_{\text{block}}(\tau_i) = \langle o_i \in S | \pi^{R}(o_i) \geq \pi^{\tau}(\tau_i) \rangle, \tag{4.11}$$

can be defined, which represents the sequence of operations that will block task τ_i. From the point of view of τ_i, this is the disturbance of the reconfiguration S. Finally,

the blocking time of τ_i,

$$B^R(\tau_i) = \sum_{o_j \in S_{\text{block}}(\tau_i)} c_j^o, \tag{4.12}$$

is calculated as the sum of the WCET of all blocking operations from the point of view of task τ_i. For a task-set $\tau = (\tau_1, \ldots, \tau_n)$, where $D_1 < D_i < D_n$ and, thus, τ_1 has the highest priority and τ_n the lowest, the blocking times will follow the same order

$$B^R(\tau_1) \leq B^R(\tau_i) \leq B^R(\tau_n). \tag{4.13}$$

This is because any sub-sequence of S that blocks τ_i must also block τ_{i+1}, since $\pi^\tau(\tau_i) > \pi^\tau(\tau_{i+1})$, i.e., the priority of task τ_i is larger than the priority of task τ_{i+1}.

Reconfiguration Feasibility

The original schedulability condition as proposed by Sha, Rajkumar, and Lehoczky [132] was given in Equation 4.5. By adjusting this condition, the laxity of a task τ_i can be defined as

$$L(\tau_i) = \underbrace{T_i i(2^{1/i} - 1)}_{\text{Max. Utilization}} - \underbrace{T_i \sum_{j=0}^{i-1} \frac{C_j}{T_j}}_{\text{Preemption}} - \underbrace{C_i}_{\text{Execution}} - \underbrace{B(\tau_i)}_{\text{Blocking}}. \tag{4.14}$$

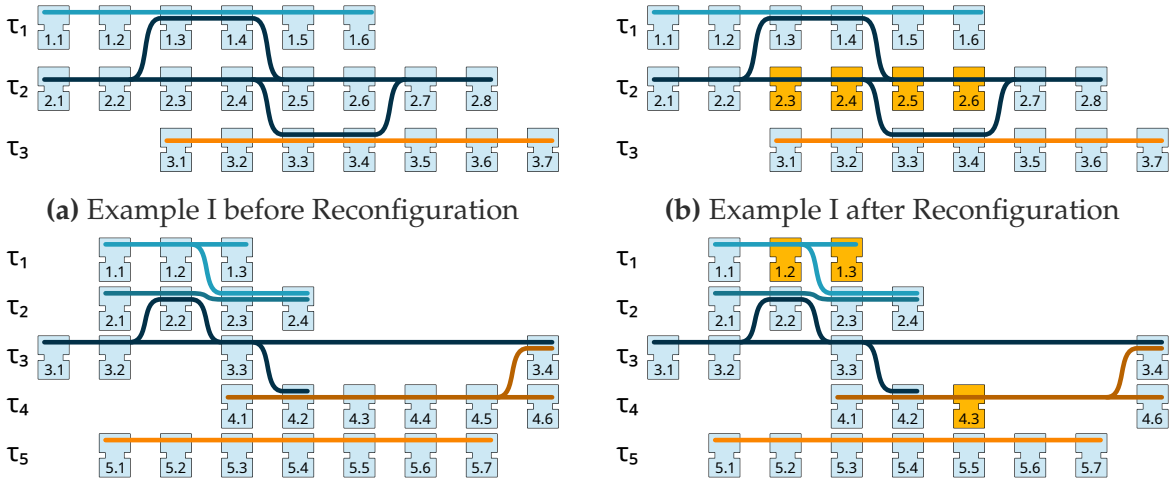
This laxity is the time that τ_i could execute longer while still keeping its deadline. For the system to be schedulable, the laxity must be positive for all tasks $\tau_i \in \tau$:

$$L(\tau_i) \overset{!}{\geq} 0. \tag{4.15}$$

This section illustrated how to compute the blocking time $B(\tau_i)$ and laxity $L(\tau_i)$ given a reconfiguration sequence S. This allows the application of the schedulability condition in Equation 4.5, which indicates whether the system can be scheduled with preemptive, single-threaded RM scheduling.

Optimization

The laxity metric can be used to define an optimization problem to identify the ordering of operations that causes minimal disruption to the real-time execution. The full optimization algorithm is presented in [7] and an extension to also consider



(a) Example I before Reconfiguration

(b) Example I after Reconfiguration

(c) Example II before Reconfiguration

(d) Example II after Reconfiguration

Figure 4.5: The connection graph shows that the reconfiguration of Example I requires a change in components/FBs 2.3, 2.4, 2.5, and 2.6. The reconfiguration from Example II-1 to II-2 modifies components 1.2 and 1.3, while combining 4.3, 4.4 and 4.5 into a new 4.3. Tasks are depicted as colored lines. In Example II, there is an additional task τ_5, which is not reconfigured.

rollback behavior in the sequence is presented in [9]. For the definition of the optimization problem, please refer to the aforementioned paper. The results of the optimization are presented in the following section.

4.2.3 Evaluation

The schedulability condition and reconfiguration optimization are demonstrated on two example systems (Figure 4.5). Example I consists of 21 FBs and three tasks τ_1, τ_2, and τ_3. Example II consists of 24 FB and five tasks, τ_1–τ_5. A fixed WCET of every FB of 50 µs is assumed. The WCETs, blocking times $B^{\text{FB}}(\tau_i)$, and deadlines are summarized in Table 4.1. The systems are inspired by real applications. To demonstrate the feasibility of the schedulability condition and optimization, naming and timing values are simplified without loss of generality.

A reconfiguration is implemented for both examples. In Example I, four FBs are replaced by new FBs, which requires one state transformation each, and all tasks are affected. In Example II, two separate locations are changed. In task τ_1, two FBs are replaced, and in task τ_4, three FBs are replaced by a single new FB. The generated

	Tasks			Undisrupted		Heuristic		Optimized		
τ	$D = T$	$C^\tau(\tau_i)$	$B^{\mathsf{FB}}(\tau_i)$	$B^R(\tau)$	$L(\tau) = B^{R,\max}$	$B^R(\tau)$	$L(\tau)$	$B^R(\tau)$	$L(\tau)$	%
Example I										
τ_1	1000	300	100	0	600.00	1280	−680.00	190	410.00	85.16%
τ_2	4000	600	100	0	1413.71	1470	−56.29	1240	173.71	15.65%
τ_3	5500	350	0	0	1463.70	1470	−6.30	1320	143.70	10.20%
Example II										
τ_1	1000	250	100	0	650.00	1280	−630.00	480	170.00	62.50%
τ_2	2500	200	50	0	1196.07	1340	−143.93	660	536.07	50.75%
τ_3	4000	300	50	0	1449.05	1600	−150.95	1300	149.05	18.75%
τ_4	5000	350	0	0	1409.14	1660	−250.86	1370	39.14	17.47%
τ_5	7000	350	0	0	1529.44	1660	−130.56	1510	19.44	9.04%

Table 4.1: Satisfaction of real-time constraints can be guaranteed using schedulability conditions, where a laxity $L(\tau)$ must be positive. A reconfiguration may lead to a negative laxity. While a heuristic ordering of reconfiguration operations cannot satisfy real-time constraints, the proposed optimized algorithm can.

dependency graph limits the feasible orderings of the necessary reconfiguration operations while guaranteeing consistency as described in Chapter 3. The reconfiguration is performed from the event source to the event sink, pushing out events following the old execution. While the reconfiguration flushes out old events, new events will follow the new system specification.

The undisrupted systems satisfy the schedulability condition (Eq. 4.5, see Table 4.1). The system is not overloaded with work, which would make reconfiguration infeasible, and it is not idle either. The laxity in the undisturbed case is equal to the maximal blocking time of each task that may be caused by the reconfiguration.

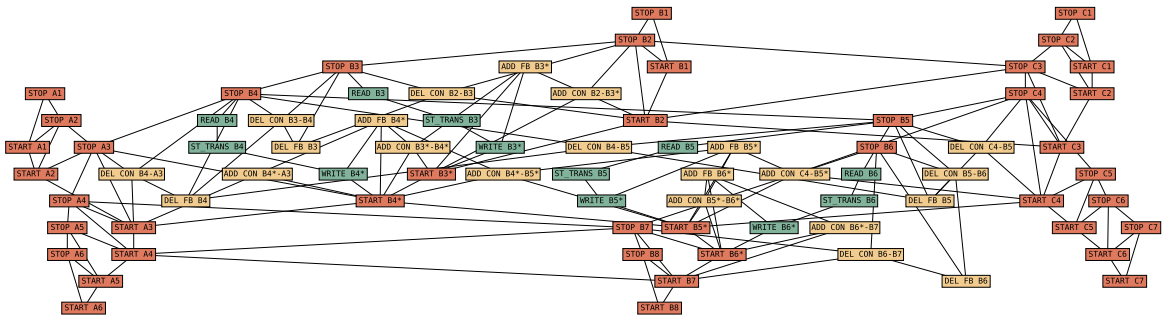


Figure 4.6: The dependency graph of Example I limits the feasible orderings that satisfy the consistency requirements. Higher nodes restrict lower nodes. The colors indicate the priority of each operation according to the heuristic in [6].

Results

As can be seen in Table 4.1, the heuristic ordering of reconfiguration operations can in both cases not satisfy the schedulability condition, since the laxity is negative. In Example I, the blocking time of the heuristic solution is significantly larger than the maximum blocking time $B^{R,\max}$. This indicates that the reconfiguration may lead to one or multiple missed deadlines. The optimized sequence that was ordered as presented in the previous section can meet the deadline in both examples.

Discussion

The heuristic algorithm does not take into account the individual deadlines of each task. Thus, it doesn't favor any task in particular but tries to start components as soon as possible and stop components as late as possible. This algorithm can find a decent solution very quickly but fails to find the optimal solution. It generates a sequence, even if a deadline is missed. Using the schedulability metric, a global minimum can be found that satisfies all constraints. In the examples, the timing-optimized solutions disrupt the time-critical tasks less. The priority inversions caused by the heuristic algorithm are resolved.

4.2.4 Conclusion

The need for adaptability was motivated in Chapter 1. Dynamic adaptability facilitates self-adaptation since it shortens the adaptation time and makes the system generally more agile.

In this section, the execution of the IEC 61499 standard was modeled as a scheduling problem that can be solved using preemptive RM scheduling and a PCP for the access to shared components/FBs. It was shown that the maximum blocking time during a reconfiguration can be calculated, and the blocking time can be used to find an optimal reconfiguration ordering. In two examples it was demonstrated that using real-time information is crucial in selecting a reconfiguration sequence. The heuristic algorithm, as proposed in Chapter 3, failed to find schedulable sequences. Using real-time information, a schedulable sequence can be selected instead.

The schedulability condition is only applicable to single-threaded, preemptive RM scheduling. Extensions to distributed control systems are necessary and valida-

tion on real applications is needed, although current IEC 61499 runtime environments do not support preemptive RM scheduling.

4.3

Agility of Dynamic Adaptation with IEC 61499[18]

The previous section investigated the real-time schedulability of reconfiguration sequences as part of an adaptation process. As seen in Section 4.1, this disturbance is only part of the entire adaptation duration. Further, it does not determine the agility of the system, which depends on how quickly an adaptation can be applied once it has been triggered.

In this section, the required adaptation time is estimated. Therefore, the execution times of the elementary reconfiguration services are measured in a simple case study. Then, the expected duration of larger adaptation is extrapolated from these measurements.

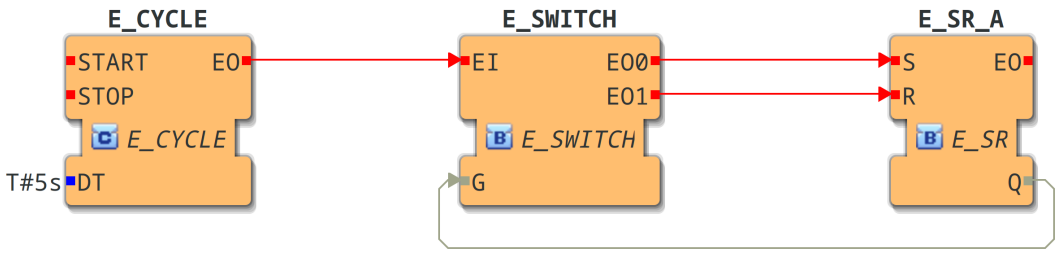
4.3.1

Measured Execution Time

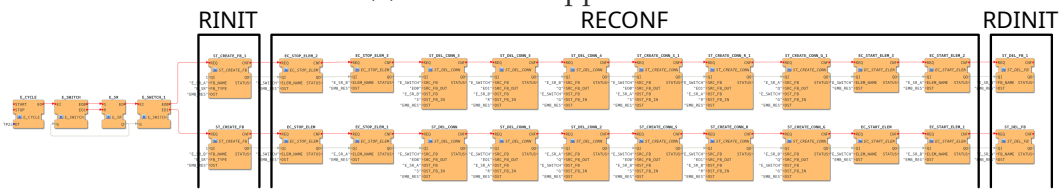
To measure the execution time of the individual reconfiguration services, a small IEC 61499 application was implemented, and a minor adaptation was performed. The application behavior is simplistic and irrelevant to the measurements (Figure 4.7a). The adaptation is the exchange of a single FB and the corresponding removing and adding of connections. After the adaptation is done, it automatically reverts the changes to the initial configuration. The reconfiguration control application (RCA) consists of two sequences of each 12 operations and is displayed with the distinct phases in Figure 4.7b. Within each sequence, first, the new FBs are created, before the old and connected FBs are stopped to preserve consistency [6]. Then, the old connections are deleted, new connections are added, the stopped FBs are resumed, and the old FB is deleted. Every two seconds, the FB is replaced and the execution times of each FB is measured within the RCA.

The measurement is performed on a Raspberry Pi 4B+ with Raspberry Pi OS

[18]Major parts of this section were published in L. Prenzel and S. Steinhorst. "Towards Resilience by Self-Adaptation of Industrial Control Systems". In: *International Conference on Emerging Technologies and Factory Automation (ETFA) 2022*. Stuttgart, Germany: IEEE, 2022



(a) IEC 61499 Application



(b) Reconfiguration Control Application

Figure 4.7: The test reconfiguration control application (4.7b) switches the FB E_SR_A in the application (4.7a) every two seconds using 12 reconfiguration operations.

with the PREEMPT_RT patch applied. The IEC 61499 application is modeled in the 4diac IDE and executed in 4diac FORTE [81] for one hour. The source code is modified to log relevant scheduling information, leading to 1800 data samples. All superfluous OS services and throttling are disabled, and the RT priority of 4diac FORTE is maximized. The results of this measurement are summarized in Table 4.2, and the distributions are visualized in Figure 4.8. Two groups can be identified: Most services are performed in around or under 4 µs, while the creation of a new FB requires on average 11.34 µs. The distributions show a long tail for longer execution times, which may be due to the execution platform, the non-real-time operating system, or the runtime environment. For the analysis, the mean value is used as the estimated execution time for each service. This data should not be used for the sake of inferring execution times for safety-critical applications, since they depend critically on the hardware and software configuration, yet they suffice to support the analysis.

4.3.2 Estimated Adaptation Times

With the measurements of the execution times for each service, it is possible to estimate the necessary adaptation time for different scenarios. Four adaptation scenarios are assumed: A minor change, as seen in the previous section, where only

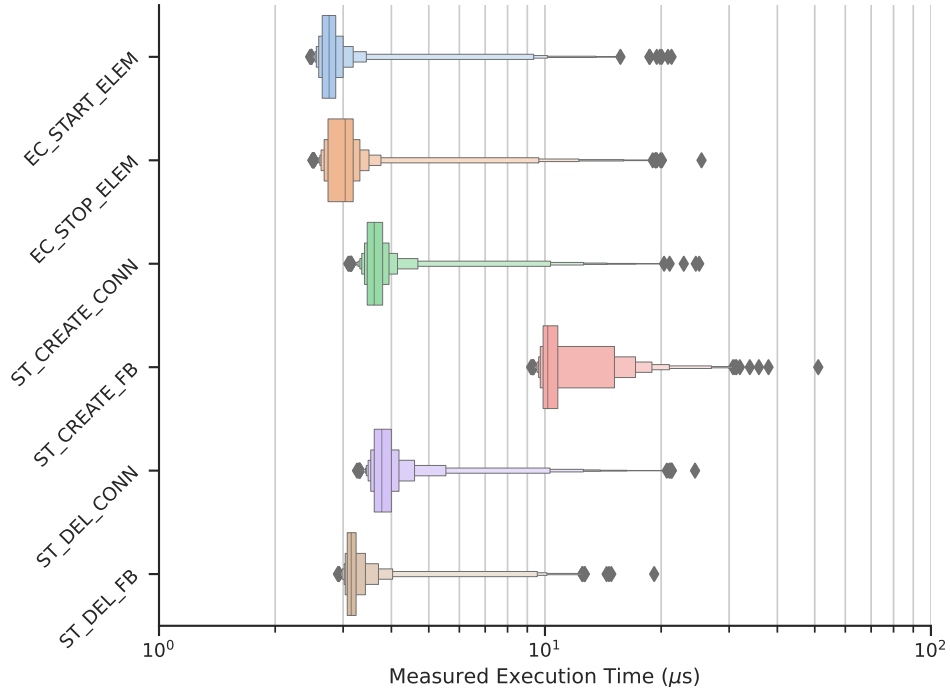


Figure 4.8: The measured execution times of the reconfiguration FBs indicate a mean execution time of under 5 µs, except for ST_CREATE_FB, which takes around 11 µs. The outliers in the measurement over one hour are rare and have an insignificant influence on overall execution times.

minimal changes are performed, a moderate change of multiple components, and a major adaptation, which affects large portions of an application. Additionally, a composite reconfiguration is considered to represent a distributed scenario in which multiple devices are reconfigured. For the sake of simplicity, additional communication overhead is ignored in this scenario because it is highly application-specific. For the moderate and major reconfigurations, 10 and 100 added/removed FBs are assumed, respectively, and an average of three connections per FB. In practice, these numbers are easily achievable if hierarchical subapplications are modified.

The adaptation is structured into five phases as described in Section 4.1.3. The resulting number of operations per phase and the corresponding estimated execution times are summarized in Table 4.3. In the example scenarios, the number of operations scales linearly with the number of changed FBs. The estimated execution time behaves identically. The PRE and POST phases represent a significant overhead, yet they do not influence the real-time execution. While a minor adaptation only requires execution time in the order of µs, a moderate to composite scenario may require

Function Block	Mean	Std	Min	Max
EC_START_ELEM	2.98 µs	1.38 µs	2.46 µs	21.28 µs
EC_STOP_ELEM	3.22 µs	1.54 µs	2.50 µs	25.45 µs
ST_CREATE_CONN	3.87 µs	1.46 µs	3.09 µs	25.09 µs
ST_CREATE_FB	11.34 µs	3.27 µs	9.20 µs	51.06 µs
ST_DEL_CONN	4.05 µs	1.38 µs	3.26 µs	24.48 µs
ST_DEL_FB	3.36 µs	1.15 µs	2.91 µs	19.19 µs

Table 4.2: Statistics of the measured execution times of the reconfiguration FBs, as performed by the test application.

	PRE	RINIT	RECONF	RDINIT	POST	$\sum$
Minor	$n = 36$	1	10	1	36	
	0.23 ms	0.01 ms	0.04 ms	0.00 ms	0.14 ms	0.42 ms
Moderate	360	10	100	10	360	
	2.29 ms	0.11 ms	0.36 ms	0.03 ms	1.38 ms	4.17 ms
Major	3600	100	1000	100	3600	
	22.90 ms	1.13 ms	3.62 ms	0.34 ms	13.75 ms	41.73 ms
Composite	36000	1000	10000	1000	36000	
	228.96 ms	11.34 ms	36.16 ms	3.36 ms	137.52 ms	417.34 ms

Table 4.3: Operations for each phase for four adaptation scenarios, and the corresponding estimated execution time (ms).

milliseconds. Ignoring the impact on the real-time execution (which is only affected by the RECONF duration), this adaptation is fast, but not instantaneous.

These results represent only the expected execution times. To reach the expected adaptation time, the utilization of the device must be taken into account since the system will be busy with other tasks. Most of the operations will be performed with a low priority to prevent any disturbance of the real-time tasks. Consequently, for a device with 80 % utilization, the adaptation time will be five times as long. This behavior is further analyzed in Figure 4.9, where the estimated adaptation times are plotted over the system utilization. At a utilization of 1.0, the adaptation is infeasible, since the additional load would result in missed deadlines. The lower utilization bound for RM scheduling (0.6931) indicates a typical load that can be expected. This would allow a major adaptation to take place in about 100 ms, and a composite adaptation in one second. Since the utilization bound only considers real-time tasks,

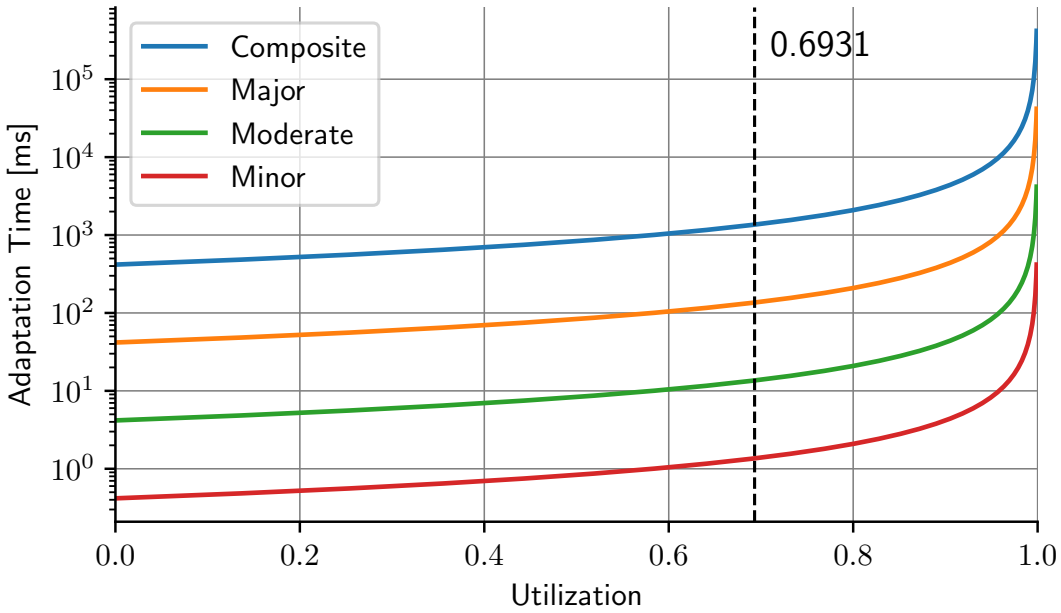


Figure 4.9: The adaptation time depends on the system utilization. For realistic utilization levels, the adaptation will terminate within seconds.

the actual utilization may be higher, and thus the adaptation time could be longer. This adaptation time represents the entire adaptation and not merely the real-time critical RECONF phase. Further communication overhead could lead to adaptation times in the order of seconds.

An important distinction is that the utilization of the system is decided at design time. While a high utilization is generally desirable to use the available resources efficiently, this may make dynamic adaptation infeasible or slow. More efficient scheduling paradigms, such as earliest deadline first scheduling, may allow even higher utilization for real-time tasks, thus making dynamic adaptation even more difficult.

4.4 Key Findings

The consistency conditions from the previous chapter enabled the generation of functionally safe and consistent reconfiguration sequences. Yet, executing these sequences may lead to intolerable delays of the real-time behavior that can lead to catastrophic failures of the ICS.

This chapter extends the necessary schedulability theory to reconfiguration

sequences, thus allowing a decisive assessment of the satisfiability of real-time constraints during an adaptation. First, the state of the art in real-time scheduling in programmable logic controllers (PLCs) is summarized, particularly focusing on dynamic reconfiguration. Next, a scheduling problem for dynamic adaptation is defined and the impact of an adaptation on the real-time behavior is investigated. A decision criterion is specified that determines if a reconfiguration sequence is schedulable for a given system. Finally, the execution times of reconfiguration services in a real system are measured, and estimated adaptation times are extrapolated that indicate the agility of the architecture.

Three key insights of this analysis are further detailed in this section.

4.4.1 Blocking Behavior

ICS are real-time systems. During an adaptation, the real-time constraints must be preserved. This is the key promise of dynamic adaptation: If the system fails and must restart because of an adaptation, it is no longer dynamic. Unlike an additional workload, the adaptation can actively modify and interfere with the execution of other real-time tasks. Thus, an analysis of the disruption caused by the adaptation is critical.

Apart from the additional workload introduced by the adaptation, the most important factor is the interference of the adaptation with the real-time tasks. Generally, this interference is caused by the starting and stopping of components, which will delay or disrupt other system behaviors. If another task needs to access a component during its reconfiguration, this will lead to a delay that may cause a deadline miss.

This chapter showed that the blocking time of the real-time tasks is the critical aspect of the adaptation. This blocking time is caused by the shared access to components that may be temporarily disrupted to achieve consistency of the process (as explained in Chapter 3). To preserve the consistency of the behavior, some components are temporarily suspended and will not react to events, which may lead to critical real-time tasks missing their deadline. Thus, it is possible to determine the blocking duration of each real-time task from the suspension duration of the used components. This blocking duration is the deciding factor when it has to be determined if a reconfiguration can be performed in real-time or not.

In the current model, the priority ceiling protocol (PCP) is used to determine the

blocking time of the real-time tasks. This only holds for rate monotonic scheduling in a single-threaded, non-distributed scenario. Especially communication delays can play a significant role in the adaptation of distributed systems. Multi-threading is a very realistic option due to its pervasiveness in modern industrial computers and may promise compelling computational gains. However, these extensions would further complicate the schedulability analysis and were consequently excluded from this analysis.

Nevertheless, this chapter applied the calculation of the blocking time from scheduling theory to the field of dynamic adaptation, which yields satisfactory results and can be used to identify suitable reconfiguration sequences in practice.

4.4.2 Schedulability

The schedulability of the dynamic reconfiguration is determined by the laxity of the undisrupted real-time behavior and the introduced blocking time of the reconfiguration. The blocking time depends on the suspension of the shared components during the reconfiguration, which is influenced by the order in which the reconfiguration operations are executed. The consistency requirements in Chapter 3 are insufficient to determine the best order of operations from a laxity perspective. The heuristic algorithm proposed in Chapter 3 can guarantee consistency, yet it does not take into account the real-time requirements. This chapter showed the optimization by incorporating timing requirements such as deadlines. This leads to a reconfiguration sequence with minimal disruption to real-time behavior.

This chapter assumed RM scheduling and the PCP, yet other algorithms are also feasible and would provide distinct advantages and disadvantages. For instance, EDF scheduling may allow for higher utilization, yet may lead to a less deterministic failure behavior. Either way, laxity is required to perform a reconfiguration. If an additional load immediately leads to a deadline miss, reconfiguration becomes impossible. Practical applications usually contain some laxity because achieving full utilization is not desirable. This chapter assumes cyclic tasks, which simplifies the schedulability analysis. Given the event-triggered nature of most IEC 61499 execution semantics, there may also be a temporal component to the utilization: The utilization may fluctuate depending on the behavior of the system.

The application of scheduling theory to dynamic adaptation permits conclusive

statements regarding the hard real-time schedulability of the system. This allows the adaptation of non-stop, safety-critical systems.

4.4.3 Domain-specific Models

Timing requirements are non-functional requirements. These can be harder to grasp or more easily overlooked. In particular, gathering these requirements in the necessary detail can be cumbersome. This chapter relies on information that is not currently part of the IEC 61499 standard and is also not necessary for many applications. Hard real-time execution, for instance, is not always mandatory. Without strict timing requirements, dynamic reconfiguration can be performed without a thorough real-time analysis. This type of execution and adaptation is also possible in programming languages such as Erlang.

If, on the other hand, hard real-time execution is required, then the timing requirements must be known explicitly. This chapter demonstrated the schedulability of dynamic reconfiguration which requires further explicit knowledge that is also not currently contained within the IEC 61499 models. The necessary information includes, among others, execution traces and the corresponding timing constraints. This is consistent with previous works on this topic [50], yet there has been little progress in this direction. Further work is necessary to show how to model this information and include it within the IEC 61499 standard.

The event-triggered nature of most IEC 61499 execution semantics allows for more flexibility in the scheduling. This chapter treated tasks within the execution as cyclic occurrences for the sake of schedulability. In practice, however, behaviors may be inherently non-cyclic or may have much larger cycles than the deadlines. As a result, it may be possible to schedule a dynamic reconfiguration at a time when the system is particularly idle. This touches on the topics of identifying safe update points from Chapter 3. In the end, this ability relies on the availability of the relevant information.

A main contribution of this dissertation is the automation of previously manual processes. While manual processes rely on implicitly known information, e.g., an automation engineer knows when to best schedule a reconfiguration, automation requires explicit knowledge. This knowledge may be included in domain-specific models, e.g., the IEC 61499 models, and needs to be included to enable automation.

Chapter 5

Agile & Resilient Industrial Control Systems

Contents

While intuitively, an architecture such as the WATERBEAR architecture extends current ICS and industrial automation systems (IAS) through self-adaptation, this benefit can be hard to quantify. Weyns [29] identifies this challenge as a need for empirical evidence for the value of self-adaptation. Arguably, in practice, complex architectures that do not offer immediately evident benefits are not approachable. The previous chapters have proposed improvements to the adaptability of runtime environments (RTEs) and mechanisms to automatically implement adaptation sequences. However, these methods are not free and require extensions to models and further implementation efforts. Thus, from the point of view of a practitioner, the effort may appear greater than the potential benefit.

The IEC 61499 standard has integrated reconfiguration services since its inception that are rarely used in practice. This may also apply to any extensions or further developments unless their benefits are quantifiable. Consequently, first of all, possible scenarios should be specified in which dynamic (self-) adaptation could prove beneficial. While the previous chapters have developed aspects of the WATERBEAR

architecture, the full potential of self-adaptability can only become evident during its application.

Resilience provides a mechanism to quantify the losses in system performance during fault or failure events. By analyzing potential scenarios that may emerge in dynamic adaptation, a comparison can be made to systems that cannot (self-) adapt. Figure 5.1 shows the meaning of these scenarios within the WATERBEAR architecture. Measurement data is used to create scenarios on the strategy management layer to facilitate the decision-making by the configuration manager. This manager uses this data to select a suitable strategy and handle potential failure scenarios.

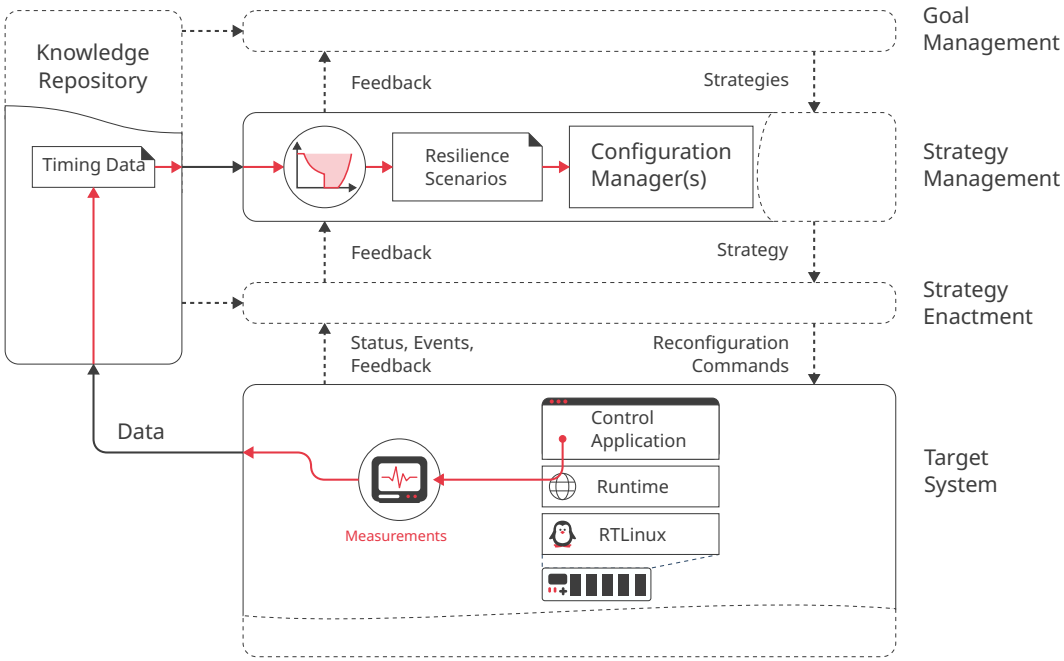


Figure 5.1: This chapter shows how the configuration manager can use measurement data to select the appropriate strategy that was handed down from the goal management layer to be implemented in the strategy enactment layer.

Furthermore, this chapter presents a decentralized implementation of the WATERBEAR architecture, underscoring its capacity for autonomous and distributed functions. Specifically, the separation of concerns between the different layers of the WATERBEAR architecture allows for an efficient distribution. The decentralization, however, necessitates synchronization via consensus algorithms that enable a decentralized decision-making process.

This chapter introduces the concept of resilience in Section 5.1, which allows the quantification of the impact of dynamic adaptation on the system performance.

Next, the initial WATERBEAR architecture is sketched out in Section 5.2 for a resilient and fully autonomous operation of an industrial automation system. Finally, the key findings are summarized in Section 5.3.

5.1 Resilience[19]

Measures of fault tolerance commonly rely on qualitative differences, i.e., whether the system is fault-tolerant of a specific fault or not. Thus, they struggle to quantify degradation [133]. By contrast, resilience is commonly used in a quantitative context, i.e., a person/system is more resilient than another. In this section, an established resilience metric is used to quantitatively assess the impact of dynamic adaptation in response to a fault.

Several definitions of resilience from assorted domains are outlined by Hosseini, Barker, and Ramirez-Marquez [134]. A common definition goes as follows: "Resilience is the ability to prevent something bad from happening, or the ability to prevent something bad from becoming worse, or the ability to recover from something bad once it has happened." [135]. There are three components in this definition: Prevention, survival, and recovery.

Resilience can be quantified as a function of time and visualized in a resilience graph [136]. An example is displayed in Figure 1.3, where the phases of the MAPE loop are inserted. Henry and Emmanuel Ramirez-Marquez [136] identify three distinct system states: The *original* state, a *disrupted* or *degraded* state, and a *recovered* state. In our specific example, the *original* state ends with the failure, and the *degraded* state ends when an adaptation is executed. In practical applications, the graph may be arbitrarily complex.

A failure generally leads to a loss in some metric. If a failure does not create any loss, then no action is required because it has no effect. Computation of a loss requires the definition of a *figure of merit* $F(\bullet)$ [136]. For the domain of industrial control, possible metrics could be *productivity*, *process quality*, or *process/communication delays*. Since this dissertation focuses on the methodological aspects of resilience quantification, a *figure of merit* $F(\bullet)$ is assumed for which increasing values are

[19] Major parts of this section were published in L. Prenzel and S. Steinhorst. "Towards Resilience by Self-Adaptation of Industrial Control Systems". In: *International Conference on Emerging Technologies and Factory Automation (ETFA) 2022*. Stuttgart, Germany: IEEE, 2022

preferred, and it is assumed that the system can be sufficiently quantified using a single metric. For practical applications, multiple metrics may be necessary to fully capture the complexities.

5.1.1 Calculation

Once a figure of merit is chosen as the metric, the loss of this metric can be calculated over time. Bruneau et al. [137], for instance, introduces the calculation of a resilience loss

$$RL = \int_{t_0}^{t_1} [100 - Q(t)] \mathrm{d}t \tag{5.1}$$

by integrating the loss of quality over time, where the quality $Q(t)$ ranges from 0 to 100. A smaller resilience loss indicates higher resilience. For this dissertation, an abstract quality of service is chosen, e.g., productivity, as the *figure of merit*, which can be measured in percent. By integrating this metric over time, the lost productivity is calculated in the unit of time. For example, if a failure causes a total loss of function for 5 seconds, this would lead to an equivalent resilience loss as a partial loss of 50 % for 10 seconds.

A difficulty of resilience quantification is the choice of metric, and the comparison between metrics. Mitigating a loss in one metric by compensating it with another metric requires a conversion factor that is often hard to derive. In the case of this dissertation, a fault is assumed to lead to a failure of the ICS, which directly causes a loss of productivity. Faults and failures that immediately lead to a full shutdown to prevent catastrophic consequences, for example, an emergency stop, are explicitly excluded because there can be no meaningful quantification of losses in this case. This does not mean that dynamic adaptation cannot handle these faults and failures, if anything it may be the only reasonable solution for true non-stop systems. However, the added risk of failure during the adaptation must be carefully considered.

5.1.2

Impact of Dynamic Adaptation on Resilience[20]

The previous section outlined a decentralized, autonomous mode of operation for ICS. The results show that consensus algorithms enable fully autonomous closed-loop self-adaptation and provide the system with unprecedented flexibility.

A detailed analysis of the resilience scenarios and behaviors is performed in this section and the resilience gains and losses are quantified to allow an assessment of the benefit of dynamic adaptation as part of a self-adaptive architecture.

Resilience Scenarios

Using the estimated adaptation times from Section 4.3, the effect of dynamic adaptation on the system resilience can be quantified. First, the *survival* and *recovery* scenarios are analyzed, before the *prevention scenario* is analyzed as part of Figure 5.4

The analysis is based on a couple of assumptions. Degradation and recovery follow the resilience graph as introduced in Section 5.1. A fully-available system has a quality of service (QoS) of 100 %, while a degraded system retains 25 %. During a restart, the system is disabled. The fault takes place after 5.5 s, and the failure after 10 s. Degradation, recovery, and ramp-down/ramp-up require 2 s to transition from 100 % to 0 % and a restart takes 5 seconds. The values are inspired by realistic applications, however, tailored to provide a meaningful analysis. A fault in a real system may remain dormant for days or weeks, and a restart could require a ramp-down in the order of hours to reach a safe state.

In Figure 5.2, a survival scenario is displayed. The reaction takes place after 5.5 seconds (d_{MAP}), when the failure has happened and degradation has begun. The restart action quickly shuts down the system and performs the modification offline. Using a fast dynamic adaptation, the system can survive the failure, without reaching a degraded state or having to shut down. The major adaptation ($d_E = 0.1s$) can be performed nearly instantaneously. The resilience loss RL_A shows a loss of 0.6 s of production time, which is mostly caused by the degradation and ramp-up. The restart requires a significant loss of productivity ($RL_R = 7.0s$).

The recovery scenario in Figure 5.3 results from a delayed reaction time ($d_{\text{MAP}} =$

[20]Major parts of this section were published in L. Prenzel and S. Steinhorst. "Towards Resilience by Self-Adaptation of Industrial Control Systems". In: *International Conference on Emerging Technologies and Factory Automation (ETFA) 2022*. Stuttgart, Germany: IEEE, 2022

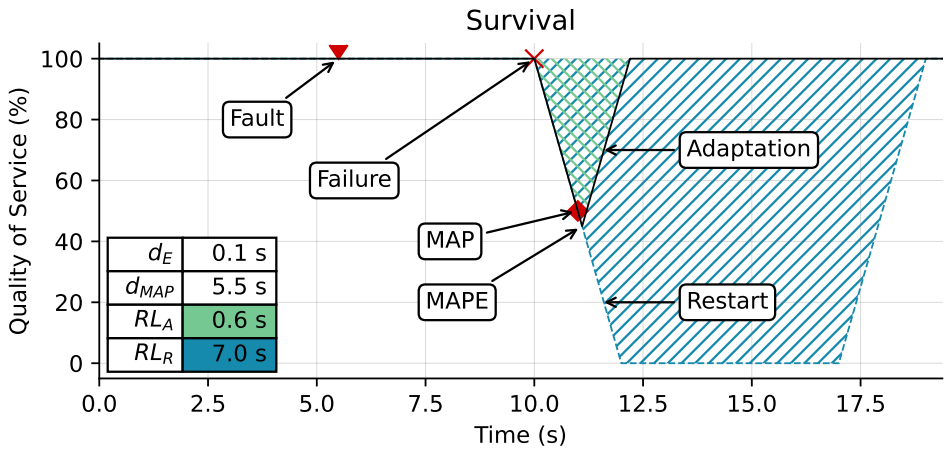


Figure 5.2: In a survival scenario, the adaptation takes place after the failure occurred, but before a degraded state is reached.

$10s$), which could be caused by a slow detection, or a complex decision-making algorithm. The system reaches its degraded state until finally a reaction is triggered. In the restart reaction, the system will quickly ramp down and perform a restart. In the adaptation scenario, the (in this case, complex) adaptation is triggered, which will cause the system to remain degraded until the adaptation is done. Finally, the recovery can begin. Similarly to the survival scenario, dynamic adaptation provides a significant resilience advantage over a restart. The loss RL_A of 4.9 seconds is still half the expected loss RL_R of a restart.

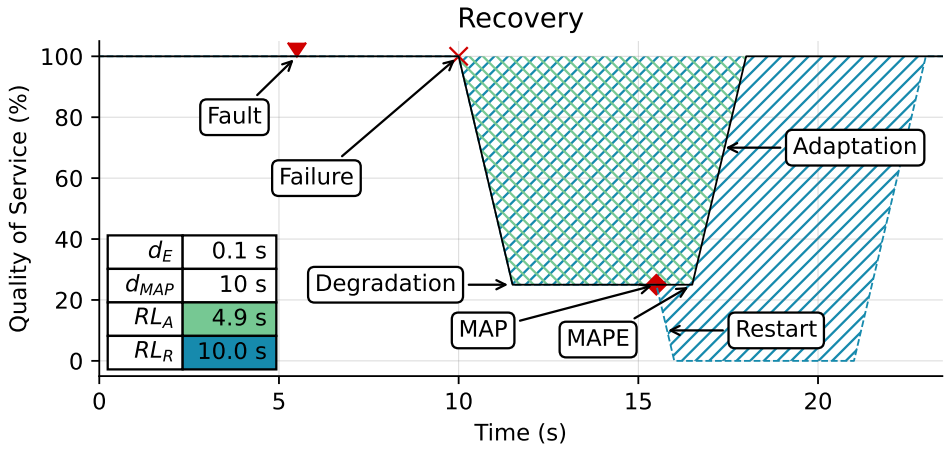


Figure 5.3: The recovery scenario is a delayed reaction, once the degraded state has been reached. The resilience loss (RL) in the survival scenario is significantly lower, yet even in recovery, the adaptation provides a clear advantage over a restart.

Further scenarios are sampled in Figure 5.4. There are four types of reaction

speeds (early, fast, late, and delayed), and three levels of adaptation complexities are considered (Minor/Moderate, Major, and Composite). An early reaction coupled with a minor to major adaptation can, in this example, prevent an impact on the system, which results in a loss of 0, i.e., the system is perfectly resilient against this kind of fault/failure. The composite adaptation will lead to the manifestation of the failure, and a brief degradation. It must be noted that here, a non-critical failure is assumed that does not require an immediate shutdown. If the failure must be prevented at any cost, and the risk of its manifestation during the adaptation period is too large, then an emergency shutdown is necessary.

For the fast, late, and delayed reactions, different resilience losses can be observed. In all cases, the resilience loss of the adaptation (RL_A) is lower than the loss of the restart scenario (RL_R). This is evident from the fact that the adaptation time is always shorter than the restart time, which is a valid assumption. What is noteworthy is that once dynamic adaptation is available, the reaction time becomes the critical factor that determines the system's resilience. Without dynamic adaptation, the reaction time is less important, since the restart will lead to a loss either way. Consequently, the currently feasible dynamic adaptation mechanisms require and facilitate a shift of focus to the monitoring, analysis, and planning phases.

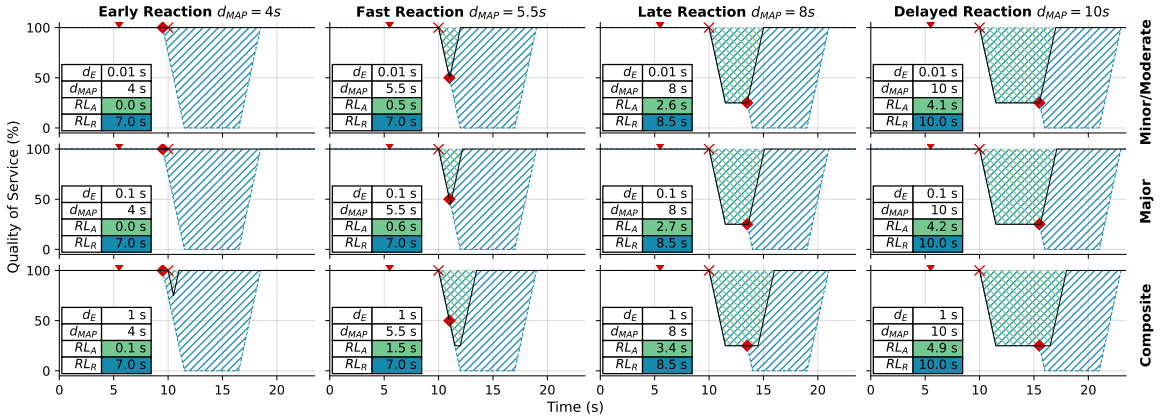


Figure 5.4: Dynamic adaptation leads to a significantly smaller resilience loss RL compared to a traditional restart. If the detection and decision-making (MAP, ⧫) after the fault (▼) takes place before the failure (×), a degradation can be prevented (RL = 0). Otherwise, dynamic adaptation avoids lengthy downtimes and can recover much faster.

Discussion

There are two key insights: First, the advantage of adaptation over a restart depends on how difficult a restart is. In the scenarios, a restart is assumed to be feasible and reasonably fast, i.e., within five seconds. For many applications, this is highly optimistic and unrealistic. Restarting a PLC can require an extensive ramp-down phase to bring the system into a safe and known state, from which the system can be safely started. If, on the other hand, a restart is feasible within a short time frame, e.g., because the system is stateless or the change does not affect the state, then the advantage of dynamic adaptation fades. Arguably, many bugs, failures, or attacks require more complicated modifications.

Second, once dynamic adaptation is feasible, available, and sufficiently fast, the main resilience gain can result from better monitoring, analysis, and planning methods. An adaptation cannot be faster than instantaneous. More importantly, dynamic adaptation facilitates further developments in monitoring, analysis, and planning, since it significantly lessens the burden of adaptation. A fast reaction that is implemented too late is similarly ineffective as a quickly implemented reaction that is detected too late. Yet the ability to quickly adapt together with fast detection results in exceptional flexibility and agility.

Dynamic adaptation allows the implementation of any imaginable adaptation of the system. This allows the reaction to events that cannot be anticipated, e.g., attacks or bugs. Many faults and failures, by contrast, can be anticipated, and a reaction can be arranged beforehand, or even directly implemented within the control application. This allows for a faster reaction without requiring a lengthy adaptation or a restart, yet additional resources must be reserved. The preparation of a reaction may also be coupled with dynamic adaptation to allow a fast implementation of complex reactions.

5.2

Resilient Autonomous Operation for IAS[21]

The previous chapters have shown specific contributions towards agility and adaptability of ICS. The big picture of how this adaptability can be integrated into a self-adaptive architecture for ICS has not yet been addressed. This especially affects the upper layers of the WATERBEAR architecture where strategies are generated and decisions are made. For large-scale systems, these layers are also not singular devices, and thus centralized techniques and mechanisms can reach a limit.

Commonly, self-adaptive systems are implemented as multi-agent systems (MAS) [138]. This approach is particularly suited to decentralized systems, where techniques such as *swarm intelligence* or *auction-based voting* can be employed. Yet, decentralizing the optimization algorithm for the adaptation of safety-critical systems can be troublesome. As identified by [139], certifying Reconfigurable Manufacturing Systems (RMS) is an unsolved problem and may require changes to current standards and best practices. V&V of MAS can be difficult due to their flexible behavior, which further complicates their use in safety-critical systems [140].

An alternative to the use of MAS is to not decentralize the optimization, but to decentralize only the agreement. The *planning* phase of the MAPE loop (see Section 1.2.1) can thus be further divided into finding a solution and agreeing to a solution. This is already integrated into the WATERBEAR architecture, where generated strategies are fed to configuration managers after a verification procedure. This allows the configuration manager to disagree with the proposed strategies. Similarly, the control device in the target system can verify the reconfiguration commands before their execution and reject the changes. Consequently, there is no need for a decentralized optimization, yet there must be a synchronization mechanism. In this section, the use of consensus algorithms is proposed for this purpose.

In this section, a decentralized architecture based on the WATERBEAR architecture is developed, together with a two-stage consensus mechanism. The architecture is demonstrated in a case study in Section 5.2.2.

[21] Major parts of this section were published in L. Prenzel and S. Steinhorst. "Decentralized Autonomous Architecture for Resilient Cyber-Physical Production Systems". In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. Grenoble, France: IEEE, 2021.

5.2.1

Decentralized Architecture

As identified by Koren, Gu, and Guo [141], the real-time decision-making mechanisms needed for production planning in manufacturing systems are difficult to achieve. Nevertheless, real-time adaptability, especially when facing failures or disruptions, is critical for any autonomous manufacturing system [142, 143]. As a result, most current approaches either focus on the optimization problem, using algorithms such as Genetic Algorithms [144] or focus on dynamic real-time production planning using multiagent systems [145]. While centralized approaches can solve arbitrarily complex optimization problems, the necessary orchestrators represent single points of failure. Furthermore, with cloud computing and manufacturing, a production plan may be generated by potentially untrusted devices. On the other hand, fully decentralized approaches, such as multiagent systems, struggle to find the optimal production plan. This section focuses on the problem of how centralized optimization algorithms can be used in a resilient and decentralized architecture for the autonomous generation and application of production plans.

Cloud Manufacturing Cloud Manufacturing promises the service-oriented pairing of manufacturing demand and supply through a cloud architecture [146]. As such, the abstraction level is much higher and the production plan does not consider safety- or timing-related properties. Within Cloud Manufacturing, the issue of trust and security has been touched and may be solved by blockchain technology, but since the abstraction level is higher, safety validation is not commonly applied [147]. In contrast to Cloud Manufacturing, the changes in this architecture manifest on a lower abstraction layer in which safety- and timing-related properties must be guaranteed.

Therefore, a novel decentralized system architecture for CPPS is proposed, combining an optimization algorithm with a decentralized validation and consensus framework. The phases of the life cycle are depicted in Figure 5.5. The production plan, which details the mapping of devices and tasks, is generated automatically. Functional validation is applied to the plan and a two-stage consensus algorithm provides autonomy, resilience, and real-time decision-making capabilities to the decentralized architecture. The proposed system architecture has three distinct phases:

Automatic Production Plan Generation The new production plan is generated based on a system description and formalized specifications. This is identical to the *configuration generation* task in the WATERBEAR architecture.

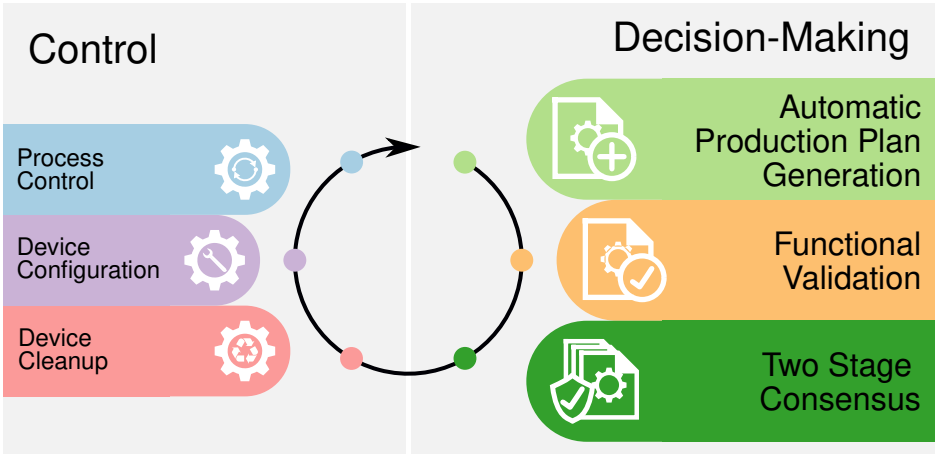


Figure 5.5: Life-cycle of the autonomous system architecture, separated by decision-making tasks and control tasks.

Production Plan Validation The timing, safety, and functional correctness of the generated production plan are validated. This task can be found in the *verification services* in the WATERBEAR architecture.

Decentralized Two-Stage Consensus The decision-making process is built on a two-stage consensus algorithm, featuring a majority agreement on the safety and optimality and a unanimous agreement of all executing devices on the feasibility and authenticity of the plan. In the WATERBEAR architecture, the first stage takes place on the strategy management layer, whereas the second stage can be found on the target system layer.

The hardware design of the architecture is visualized in Figure 5.6. A mesh network of heterogeneous devices with varying functionalities and computational capabilities is considered. The generation of the production plans is outsourced to cloud devices, whereas the functional validation can be performed on the computationally stronger devices in the mesh network as well.

Automatic Production Plan Generation

The automatic production plan generation is performed by an optimization algorithm. The inputs to the optimization are a formalized system description and one or multiple product specifications. The goal of the optimization is to find a production plan that uses the devices in the system description to fulfill (ideally) all product specifications. This problem can be formalized as a version of the Job Shop Scheduling

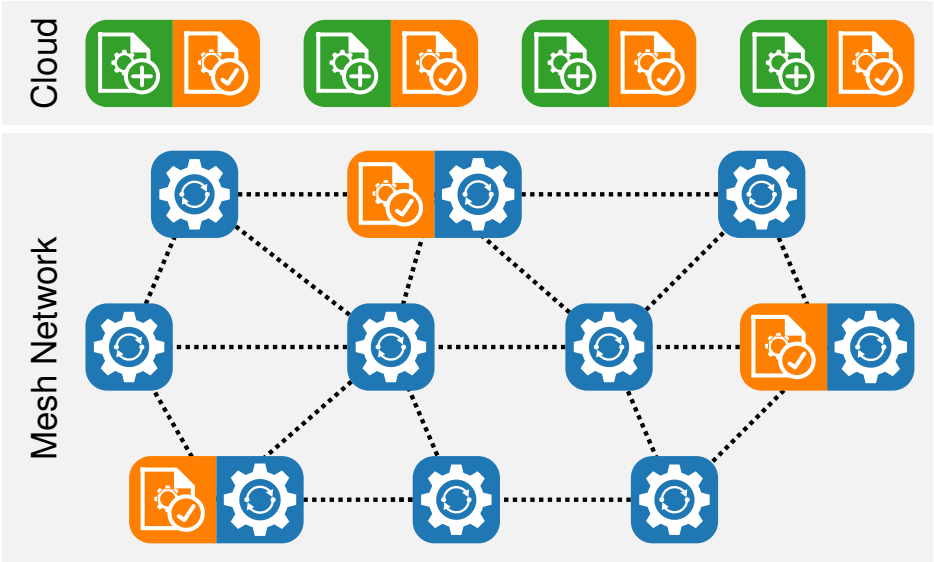


Figure 5.6: Hardware architecture featuring three types of heterogeneous devices. Devices with additional computational resources can participate in the validation, whereas the production plan generation is outsourced to the cloud.

problem, to which a great number of solutions already exist [148]. In this architecture, a greedy algorithm is used to find a suitable plan, although different algorithms can be plugged in. The reconfiguration of the architecture is enabled by adaptable control devices on the lowest layer, as seen in Chapter 2.

Production Plan Validation

The new proposed production plan must be validated. Given that it may be generated remotely, this plan is not immediately trustworthy. Thus, the validation can be used to generate this trust [149].

The validation may encompass the verification of different timing- and safety-related properties. Since the inputs of the generation (system description, specifications) are openly available, they can be used in the validation procedure as well, leading to a more meaningful validation. Additional complexity is introduced by considering a stateful reconfiguration, in which the system state is (partially) preserved during a modification of the production plan. Depending on the expressiveness of the system description and specifications, a formal verification may be feasible, e.g., to verify safety-critical time constraints. The validation can be performed in a distributed manner, with devices validating a hierarchical sub-component of the

overall production plan.

Decentralized Two-Stage Consensus

The two-stage consensus starts when a new production plan is proposed. The first stage synchronizes the results of the functional validation between the involved devices. During this stage, unsafe or functionally incorrect plans are filtered out, and a new plan may be selected based on its optimality. The second stage empowers all devices affected by the new plan to veto the change.

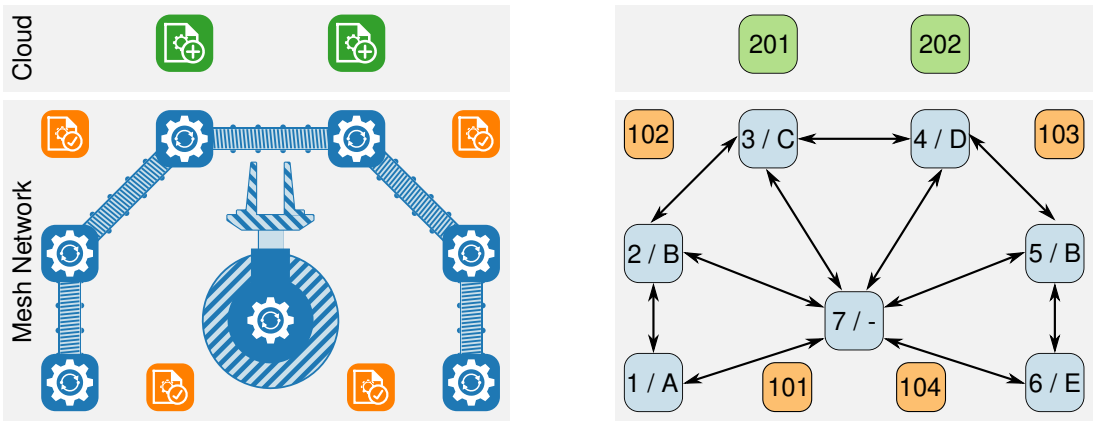
Majority Consensus The first consensus uses majority voting to select the most suitable new plan after performing an in-depth functional validation, as detailed in the previous subsection. Given the computational complexity of this validation, this consensus is formed between the computationally stronger devices. A majority vote is chosen over a unanimous vote since the functional validation leads to the same result on all devices.

Unanimous Consensus The second consensus requires all affected devices to unanimously agree to this new plan. Every device must perform a quick feasibility analysis and verify its authenticity. Only if every device can fulfill this new production plan, it can be applied to all devices.

In case a consensus cannot be formed in the second stage, the production plan may be changed incrementally to exclude problematic devices. Thus, a malicious device in the second stage would not be able to block the consensus indefinitely.

Simulation Framework

The architecture is implemented in a custom discrete event simulation using the Python library simpy [150]. It can simulate arbitrary mesh networks, which are defined in a YAML file and can be visualized in a graphical user interface. The optimization is currently performed using a greedy algorithm on a graph representation of the system. The simulation may be executed in real-time or simulation time, and can automatically output the timestamped interactions between all system components (see for example Figure 5.8).



(a) Example System **(b)** Abstract Depiction

Figure 5.7: Example system of a flexible manufacturing system with conveyor belts and a backup robot. A workpiece must follow the specification A–B–C–D–B–E. Functionality B is redundant in stations 2 and 5, every other function is only available once.

5.2.2

Case Study

To highlight the features of the simulation and to demonstrate the feasibility of the architecture, a case study is implemented. Figure 5.7a depicts a conveyor belt system with an additional gripper robot that may serve as a backup to reroute workpieces in a factory. An abstract system view is visualized in Figure 5.7b, detailing the specific functions of the stations. Each station (1–6) has a distinct function, where station 2 and 5 are identical. Station 7 is the robot, which has no specific function apart from transportation. The specification defines that every workpiece must pass all functions in the order A-B-C-D-B-E. Every station features a device, and there are four additional gateway devices (101–104) with additional computational power and no functional capabilities, and two cloud devices (201–202). This case study illustrates three contributions:

System Configuration The system can autonomously generate and apply a production plan.

System Reconfiguration When facing a failure, the system can reconfigure itself to circumvent the failed device and continue operation.

Decentralized Consensus Using the two-stage consensus algorithm, the devices can decentrally agree on the course of action.

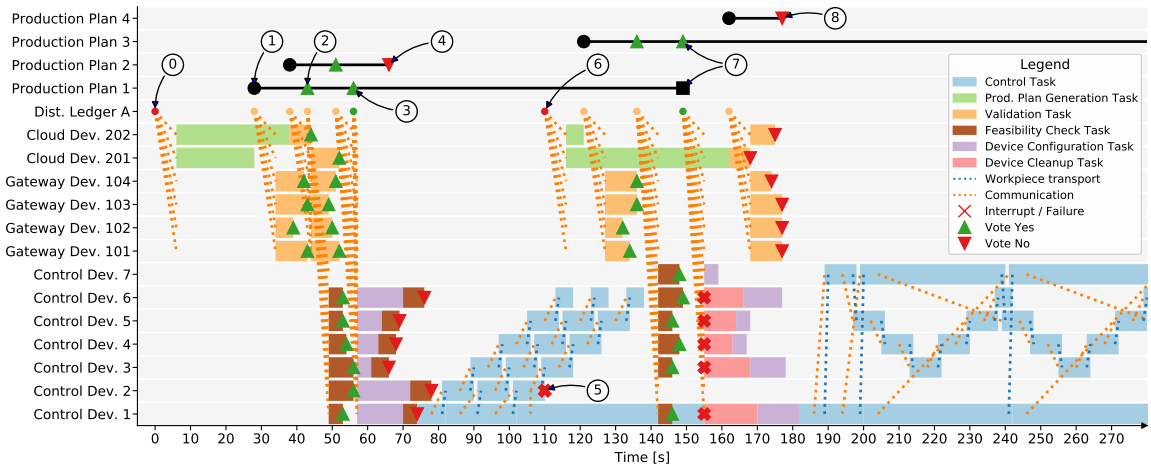


Figure 5.8: Time diagram representing the result from the discrete event simulation. After system initialization (0), a production plan is generated (1). From the point when all devices agree (3), the new plan runs until a device failure appears (5), at which point new production plans are generated. Eventually, a new plan is found and applied (7). Timing is exaggerated for better visualization.

The simulation architecture uses a distributed ledger to share the production plans and synchronize the system state. The system starts with all devices available, configures itself, and starts producing. At a predefined point in time, Device 2 fails, causing a reconfiguration. Because Device 5 is still available, the system remains functional and continues production at a slower rate by using Device 5 twice and involving the robot arm. To better visualize the timing behavior, the durations of computational tasks and communication are exaggerated.

Results

The timing behavior of the simulated case study is displayed in Figure 5.8. Particularly interesting events are marked with numbers. At Event 0, the system description and specifications are initialized, which leads to the generation of new production plans. At Event 1, the first plan is created, and the first stage of the consensus begins, in which the gateway and cloud devices validate the plan. After a majority of devices agrees (Event 2), the second stage of the consensus takes place, which ends at Event 3. As a result, the production plan is set to valid and the devices are configured. Consequently, the second production plan, which is now outdated, is dismissed in the second phase of the consensus (Event 4). When Device 2 fails (Event 5), this is recognized as a system change (Event 6) and leads to the generation of new plans.

Eventually, a new plan passes both stages of the consensus and replaces the old plan (Event 7). Since the last plan is already outdated when it is ready, it is discarded in the first stage of the consensus (Event 8).

Discussion

The case study highlights the features of the simulation and demonstrates the feasibility of the architecture and the two-stage consensus algorithm. The system can automatically generate a production plan that enables the control devices to produce. The greedy graph-search algorithm can find a valid production plan and may in the future be exchanged for a more elaborate algorithm that can consider a multitude of factors. The control devices configure themselves when the production plan is selected and begin the process control autonomously.

After the failure of Device 2, new production plans are generated automatically. Following a cleanup phase, in which the state of the previous production is discarded, the involved devices configure themselves again and begin producing according to the new plan. In the future, this cleanup phase may be replaced by a stateful reconfiguration, where leftover workpieces are considered. An approach similar to the calculation of migration routes proposed by Pourmohseni et al. [151] may be utilized.

The feasibility of the two-stage consensus algorithm is demonstrated. The first stage, featuring a functional validation, can take place without interrupting the safety- and timing-critical process control. Only the second stage requires the synchronization between all involved devices. As a result, the devices decentrally configure and reconfigure themselves, showing resilience against failures. The custom simulation framework performs a meaningful quantitative timing analysis and provides an extensible skeleton for future implementations.

5.2.3 Conclusion

This section presented an autonomous mode of operation for cyber-physical production systems (CPPS) based on the WATERBEAR architecture. This autonomous system operation mode closes the self-adaptation loop through an automatic production plan generation, a functional validation, and a two-stage consensus mechanism.

A simulation demonstrated the feasibility and benefits of such an architecture: By closing the self-adaptation loop and integrating consensus mechanisms, the system autonomously reacts to unforeseen events and recovers its functionality independently in a decentralized manner. This indicates an increased resilience over current, traditional architecture.

This section also highlighted the need for future work to close the loop. Specifically, multi-objective optimization to automatically generate configurations, fast real-time capable consensus mechanisms, and sophisticated validation mechanisms to check proposed adaptations in a decentralized manner are needed.

5.3 Key Findings

As identified by Weyns [29], the value of self-adaptation needs to be shown with empirical evidence. Quantifiable benefits are required to offset the high initial cost of agility and self-adaptability of ICS. The IEC 61499 standard included dynamic reconfiguration support since the beginning, yet it is not used in practice. Until now, the cost has seemed to overshadow the potential gains, which were elusive and hard to quantify. Consequently, this chapter shines some light on this topic.

This chapter addressed the quantification of resilience gains resulting from dynamic adaptation. The state of the art in resilience and resilience quantification of technical systems was analyzed to identify a suitable metric. It was found that single metrics are inherently flawed in the quantification, however generally, productivity and availability can be used until better metrics are available. The resilience gains of an adaptable architecture in contrast to a non-adaptable architecture were evaluated. The results show that some architectures benefit from adaptability more than others. Most importantly, fast detection and decision-making mechanisms can increase the resilience gain, and applications that are costly to restart benefit the most. Finally, a closed-loop optimization mechanism for decentralized, autonomous, and resilient operation of IAS was established. The multilayered architecture allows an efficient separation of concerns with facilitates decentralization. However, decentralization requires synchronization mechanisms, such as consensus algorithms.

5.3.1

Prevention, Survival, Recovery

Resilience, as a word, conveys an abstract idea that is broadly understood. How this idea materializes in a technical system is usually harder to grasp. Scientific literature has spent a lot of effort on the definition and quantification of resilience which aids in understanding and implementation.

Dynamic adaptation, on the other hand, is generally understood as a kind of evolution, or for technical (software) systems, updating of a system. While the concept is very comprehensible, it's often underestimated and seen as "just updating", especially when considering that the process could be extensively automated. The link between adaptability and resilience is often missed.

In this chapter, the effects of dynamic adaptation were explicitly demonstrated to lead to resilient behaviors, namely *prevention*, *survival*, and *recovery*. These terms are easily understandable and illustrate the significance of dynamic adaptation in future system architectures. Moreover, they allow a comparison between different behaviors. The benefit of (fast) dynamic adaptation becomes obvious when it enables a system to *prevent* a failure instead of just *recovering* from it. This particularly applies to unanticipated faults and failures, which cannot be accounted for at design time.

The three resilient behaviors shown in this chapter are placeholders for all kinds of resilient reactions to a fault or failure. The focus of this dissertation was on realizing the ability to adapt, not developing or evaluating ingenious adaptations. The fundamental adaptability enables all types of complex, resilient reactions and scenarios to be implemented. For instance, a mixed-criticality approach can be used to prioritize specific functions over others, leading to a simultaneous degradation of one function to prevent a loss in another function. The solution space is limited only by the creativity of the developer, or, in the future, by the generating algorithm.

5.3.2

Monitoring, Analysis, Planning

Closed loop adaptation requires more than just adaptability. To close the loop, a decision has to be made to trigger an adaptation and the adaptation has to be planned. In scientific literature, the MAPE feedback loop ([19, 44]) offers a self-adaptation model with four phases: Monitoring, Analysis, Planning, and Execution.

The MAPE model applies to all types of adaptive systems and is not specific to ICS. Thus, an integration of the MAPE model is necessary and ICS-specific requirements must be considered. Or, as stated by Weyns [29], domain-specific modeling languages need to be integrated into self-adaptive engineering.

This chapter proposed a decentralized, autonomous mode of operation for ICS enabling resilient behavior. A two-stage consensus algorithm permits decentralized decision-making between the participants while considering resource constraints of the control devices (or PLCs). The evaluation in Section 5.1.2 further indicates the relationship between the different phases when it comes to resilient behavior. The main goal of this dissertation was to realize adaptability and agility in ICS. In terms of the MAPE loop, this is the fundamental requirement to achieve self-adaptation, since an architecture that cannot be *adapted* can also not self-adapt. Once this precursor is available, the other phases start to matter more: Elaborate data collection, fast decision-making, and sophisticated planning algorithms are the keys to closing the loop from adaptability to self-adaptability.

Consequently, once a sufficient level of adaptability and agility is achieved, the focus must shift towards these three phases. As was seen in Section 5.1.2, the feasible speed of adaptation, i.e., up to one second for complex changes, means that the other phases need to reach similar ranges as well, or faster. Monitoring and analysis must be fast enough to detect a deviation before it can lead to dangerous consequences. The adaptation must be planned in time, yet this can also be partly achieved at design time by conceiving fallback configurations that can be used when needed. Regarding the integration of domain-specific models into self-adaptation, this dissertation has shown some progress by linking the IEC 61499 standard. Yet, there are further needs. For instance, the IEC 61499 standard does not have satisfactory behavior models that could be monitored [127].

5.3.3 Choice of Metric

To make a compelling case for the adoption of (self-) adaptation, a quantitative analysis is invaluable. On the qualitative level, the term fault tolerance describes the ability to tolerate faults and failures, however, it is hard to quantify how beneficial a fault tolerance mechanism is. Finding a suitable, quantifiable metric, thus, permits a comparison between different techniques and scenarios.

A resilience metric and the corresponding resilience graphs allow a simple (and graphical) analysis of the quantitative benefit of a resilience action. Nevertheless, the resilience graphs require a metric over which a loss of performance can be measured.

This chapter showed how the quantitative impact of dynamic adaptation can be determined on a simple productivity or availability metric. The resilience loss caused by a resilience action was measured in lost production time, which is easily accessible. This metric is on a high enough abstraction level to be easily understood, yet is straightforward to calculate or simulate.

Often a single metric is incapable of capturing the entire system behavior during a failure scenario [136]. In these cases, multiple metrics should be combined to paint a better picture of the multifaceted losses and gains. For instance, lowered productivity could be overcome by lowering production quality. Then, a compromise must be made about the acceptable product quality and the required production rate. In a closed-loop scenario, the system itself must be able to decide in this multi-objective optimization problem, thus a conversion factor between metrics may be necessary.

5.3.4 Decentralized Decision-Making

As was seen in Chapter 1, multilayered self-adaptive architectures provide separation of concern which facilitates their implementation. It also enables the decentralization of tasks, e.g., the generation of configurations. Generally, decentralization allows for better scalability, since the number of participants can be increased. For instance, the configuration generation can be outsourced to a powerful cloud server to relieve the resource-constrained edge and control devices from this task. Furthermore, decentralization eliminates single points of failure and allows for scalable introduction of redundancies. However, decentralization requires some form of synchronization and often leads to communication overhead.

This chapter demonstrated that a two-stage consensus algorithm is suitable for the synchronization of the decision-making process between decentralized participants in the WATERBEAR architecture. This enables a highly scalable, distributed generation and validation process. Furthermore, security can be increased by enabling the control devices to object to the adaptation and thus preventing compromised devices from triggering malicious adaptations.

The behavior of the architecture was shown in a simulation, yet real implementa-

tions are necessary. For instance, the consensus mechanism relies on functional validation, which can be difficult to implement for complex systems. In practice, the issue of certification and validation of reconfigurations and adaptation is unsolved [139] and requires further work as well. Nevertheless, decentralized self-adaptation for ICS was shown to be feasible.

Chapter 6

Conclusion

Contents

This dissertation has advanced the state of research in dynamic adaptation and self-adaptation within industrial control systems (ICS). Within Chapter 1, three principal challenges were identified: the development of an adaptable runtime environment, the consideration of consistency during the adaptation, and the examination of timeliness during the adaptation. The development of an adaptable runtime environment is crucial for displaying practical applications and demonstrating technical feasibility, while consistency and timeliness during the adaptation are critical to ensure the reliability and dependability of the methods.

This dissertation has provided contributions to all three challenges. Most importantly, it demonstrated for the first time, how functional consistency and timeliness can be guaranteed during the dynamic adaptation of an ICS. This is an important step in bringing dynamic adaptation and self-adaptation into practice, since ICS must fulfill strict safety and real-time requirements. Previous works in this domain were either not specific to the domain of ICS, or did not fully consider the consequences to the functional or real-time behavior.

The previous chapters have shown the contributions of this dissertation to these three key challenges. This chapter not only summarizes the contributions of this dissertation but also sets the stage for discussing its limitations and future directions,

underscoring the impact and potential of the findings within the ICS context.

6.1 Key Findings

This section summarizes the key findings and contributions of the previous chapters and relates them to each other to underline their significance. The contributions are ordered by the chapters they appeared in, which also follow the research challenges from Chapter 1. This analysis lays the groundwork for the limitations and future steps presented in the next section.

6.1.1 Reconfigurable Runtime Environment

Adaptability and reconfigurability are key enablers to increase the flexibility, maintainability, and autonomy of industrial control software. Current standards, such as the IEC 61131-3 and the IEC 61499 standards offer only primitive support for dynamic adaptation, a precursor for self-adaptation. Chapter 2 investigates the role of the runtime environment in the dynamic adaptation of ICS. Therefore, first, the state of the art in reconfigurable real-time software, in particular for industrial control (IEC 61499) and telecommunication (Erlang), was summarized. Specifically, current IEC 61499 implementations were analyzed and compared concerning their execution semantics. Further, the soft real-time runtime environment of Erlang was evaluated for an implementation of the IEC 61499 standard. Lastly, a full-fledged compiler from IEC 61499 models to Erlang was implemented that allows adaptation at runtime using the high-level paradigms of Erlang, and an evaluation of the scalability of the implementation was performed.

As described in Chapter 1, a reconfigurable runtime layer is crucial for self-adaptation, because it can provide effectors that can execute an adaptation designed by another entity. This allows the middleware to implement the change through services, while higher levels consider the use of these services to guarantee consistency or timeliness (as discussed in Chapters 3 and 4). Three findings of Chapter 1 should be highlighted specifically:

Reuse of existing technology The reuse of existing technologies for an IEC 61499

implementation provides stimulus for new developments and allows the comparison with the state of practice of other domains. Even though industrial automation is considered a rather conservative domain, observing other domains and learning from their innovations and mistakes can initiate unique developments.

Guarantees and Execution Order Guarantees given by the runtime environment must be communicated transparently. Better guarantees reduce the effort for the adaptation designer, yet may exponentially increase the complexity of the runtime. However, the adaptation designer must be aware of the guarantees that exist, or more importantly, do not exist. For the IEC 61499 execution model, this, for instance, concerns the possibility of event loss or changes in the event order. Clear execution semantics are needed, especially if interoperability is desired.

Real-time Capabilities Sophisticated real-time models require information. The IEC 61499 model does not provide all of the information necessary for a sophisticated event-triggered real-time execution, for instance, deadlines or priorities. Erlang, on the other hand, cannot guarantee hard real-time performance but offers very efficient and sophisticated soft real-time mechanisms that work with less information. There needs to be a conscious decision about the level of real-time guarantees and how the necessary information is going to be provided.

The IEC 61499 standard offers a promising framework for implementing an adaptable runtime environment. However, it is also lacking in clarity about the execution semantics and the real-time behavior. These issues are magnified during an adaptation procedure, and when questions of *correct behavior* arise, as was seen in the following chapters.

This dissertation compared the support of existing ICS architectures for dynamic adaptation to a runtime environment used for high-availability telecommunication systems. While the boundary conditions differ, similar lessons can be learned. Given that Erlang and its *hot-code loading* feature have been used in production systems for decades, these lessons could prove invaluable for the future of ICS.

6.1.2

Consistency During Reconfiguration

Dynamic adaptation itself does not imply *safety* during the adaptation. As was seen in Chapter 1, there are as many types of adaptation as there are reasons to adapt. For ICS, *safe* adaptation must consider the physical process and the behavior of the control system. Functionally, the adaptation may not lead to a violation of the specification, which lays out the admissible interactions between the control system and the physical process.

This dissertation implemented a correct-by-design algorithm to automatically generate safe adaptation or reconfiguration sequences. First, consistency requirements and conditions in research were reviewed. Then, an existing consistency condition (i.e., quiescence) was applied to the domain-specific modeling language defined by the IEC 61499 standard. This allowed the implementation of an algorithm that automatically selects a safe reconfiguration sequence of operations to modify a system while preserving the intended application behavior. An evaluation demonstrated the algorithm on a selection of relevant adaptation scenarios.

From the implementation and evaluation, three key findings are presented here.

Choice of Condition In research, multiple consistency conditions have emerged over the last decades, e.g., *quiescence*, *tranquility*, and *version consistency* [117–119]. For application in practice, one condition must be chosen and implemented. In Chapter 3, a variation of quiescence was used to guarantee consistency during a reconfiguration of the IEC 61499 models. This condition is particularly suitable due to the one-directional interactions between components and their well-defined interfaces. Updating the system from event source to event sink allows the update to push old events out of the system as new events arrive. While quiescence is simple to implement, other conditions may be just as or more suitable. Version consistency, for example, is a high-level condition, yet it requires a detailed transaction model which does not currently exist for the IEC 61499 models. Given the current state of these models and the provided information, quiescence is straightforward to implement and can handle a variety of scenarios.

Feedback Loops Feedback loops lead to circular dependencies between components. This cannot be handled easily by quiescence unless more information about the execution behavior is available. Since the IEC 61499 standard defines

an executable model, most of the information is available inside the component but must be inefficiently extracted. Non-deterministic behaviors may also be infeasible to resolve. A solution could be to integrate better behavior models into the IEC 61499 standard which enables the decomposition of feedback loops.

Execution Semantics Consistency is required on the system behavior level which is defined by the executed model and the underlying execution semantics. Ambiguous execution semantics make it impossible to derive a clear consistency condition because the behavior can be inconsistent even without an adaptation taking place. Two issues in the IEC 61499 models are the separation between data and event connections, and the lack of suspension mechanisms. While these issues can be and have been solved, they would need to be formalized within the IEC 61499 standard.

Consistency is crucial for the adoption of dynamic adaptation and eventual self-adaptation. Yet, consistency during adaptation depends upon a consistent execution model to build on or transparency about the inconsistencies. For ICS, the existing consistency conditions suffice, yet they have to be properly integrated into the domain-specific languages. For the IEC 61499 standard, this would require the community to settle on one semantic and accept the necessary changes to the standard. On the other hand, settling on a compromise, e.g., regarding the choice between event-triggered and cyclic execution, or the type of event-triggered real-time execution, may water down the consistency guarantees.

6.1.3 Timeliness during Reconfiguration

The consistency conditions developed in this dissertation allow the generation of functionally safe reconfiguration sequences. Yet, the execution may lead to intolerable delays of the real-time behavior that can lead to catastrophic failures of the ICS. Chapter 4 extended the necessary schedulability theory to reconfiguration sequences, thus allowing a decisive assessment of the satisfiability of real-time constraints during an adaptation. First, the state of the art in real-time scheduling in programmable logic controllers (PLCs) was summarized, particularly focusing on dynamic reconfiguration. Next, a scheduling problem for dynamic adaptation was defined and the impact of an adaptation on the real-time behavior was investigated. A decision criterion was specified that determines if a reconfiguration sequence is schedulable

for a given system. Finally, the execution times of reconfiguration services in a real system were measured, and estimated adaptation times were extrapolated to indicate the agility of the architecture.

This dissertation advanced the state of research on schedulability analysis regarding the IEC 61499 standard and dynamic adaptation. While real-time performance was always expected and partially shown (e.g., in [50]), this did not cover the adaptation phase and most importantly not while simultaneously guaranteeing consistency of the running application. From the achievements of Chapter 4, three notable contributions are:

Quantifiable Blocking Behavior The timing disruption of a dynamic adaptation task is the blocking time that results from the suspension of other components. The disruption can be quantified using the priority ceiling protocol (PCP). The blocking time depends on the order of operations within the reconfiguration sequence. Smaller, incremental reconfigurations would thus be less disruptive to the real-time behavior.

Extended Schedulability Blocking caused by the adaptation affects the schedulability. In this dissertation, a schedulability condition for rate monotonic scheduling was shown. This scheduling condition can be used to find an optimal reconfiguration sequence with a minimal disturbance of the real-time performance, yet not every adaptation can be achieved in real-time. Consequently, developing a reconfiguration sequence should be done iteratively, while keeping the real-time laxity in mind. In addition, this laxity needs to be considered during the initial setup of the system. If the system is at capacity without the extra load of a reconfiguration, the adaptation may be infeasible.

Domain-specific Models Current domain-specific models lack the necessary information to perform the schedulability analysis necessary to guarantee hard real-time behavior. While it was shown that dynamic adaptation can be reasonably fast, the event-triggered execution of the IEC 61499 standard requires more advanced scheduling models but does not currently hold the information to implement them.

This dissertation investigated the real-time feasibility of dynamic adaptation and the reachable agility of such an architecture. It was shown that hard real-time schedulability with strict guarantees is feasible if the necessary models are available. For the IEC 61499 models, this is complicated by the event-triggered execution. In

practice, dynamic adaptation can apply complicated changes within seconds without disrupting real-time behavior, which promises great agility compared to the state of practice.

6.1.4

Agility and Resilience

Dynamic reconfiguration has been a less-marketed feature of the IEC 61499 standard. Similarly, in Erlang, *hot-code loading* is known but rarely applied in production. This can be blamed on incomplete or poorly documented implementations, yet another reason is that the benefit may be hard to quantify. For instance, the *nine-nines availability*[22] of Erlang was only impressive after it was achieved and had been measured.

The agility and self-adaptability of ICS are irrelevant if the benefits cannot be quantified. Although the IEC 61499 standard has included dynamic reconfiguration in principle since its inception, it is rarely used in practice. In past applications, the cost overshadowed the potential gains, which were elusive and hard to quantify. Chapter 5 addressed the quantification of the resilience gain that results from the use of dynamic adaptation. The state of the art in resilience and resilience quantification for technical systems was examined to identify a suitable metric. Next, a closed-loop optimization mechanism for decentralized, autonomous, and resilient operation of IAS was established to extend the concepts of self-adaptation. Finally, the resilience and resilience gains of an adaptable architecture in contrast to a non-adaptable architecture were evaluated in several scenarios.

Resilience is not a prerequisite for most technical systems. Even when a type of *recovery* is needed, in most applications, this can be done via manual intervention, e.g., manually resetting the system and fixing the issue. However, for modern software architectures and distributed, non-stop systems, resilience can be highly desirable. A central argument of this dissertation is the value of dynamic adaptation, as it enables a transition towards self-adaptation, which offers its own set of benefits:

Prevention, Survival, Recovery Adaptability and agility of the ICS enables resilient behavior in response to unanticipated events. Depending on the fault and reaction speed, and the agility of the system, a degradation can either be prevented, survived, or recovered. The questions of which of these scenarios

[22] Erlang is said to have achieved nine-nines of uptime, i.e., 99.9999999 %.

takes place should be considered during system design.

Monitoring, Analysis, Planning In addition to adaptability, closed-loop adaptation requires monitoring, analysis, and planning. Once the system demonstrates sufficient agility, the monitoring, analysis, and planning phases become significantly more important. Thus, adaptability serves as a multiplier for the relevance of the other phases. Considering the importance of big data, it is surprising how much the planning and execution phases have been neglected.

Choice of Metric Comparing resilience actions requires a metric over which they can be compared. Productivity is an obvious choice, but can be difficult to measure. Generally, relating low-level actions to high-level effects (i.e., costs) promises the highest benefit.

In Chapter 5, the step from agility to resilience was shown to indicate the significance of dynamic adaptation for ICS. It was shown that dynamic adaptation is a precursor to self-adaptive behavior, and given the current feasibility of dynamic adaptation, the focus can shift towards the earlier phases of the MAPE self-adaptation loop.

6.2 Research Limitations and Future Directions

While this dissertation provided distinct contributions to the area of dynamic adaptation and self-adaptation, it is not without limitations. The contributions of this dissertation build upon previous scientific contributions, and themselves provide a foundation for future scientific work. Applying the results to practice would be its own, fascinating journey, but unquestionably out of scope for this work. Thus, this section elaborates on the limitations of this work and highlights a selection of the next steps that seem appealing and important.

6.2.1 Research Limitations

The limitations are split into two categories: methodological limitations and generalizability limitations. The methodological limitations address the choices made during the dissertation. The generalizability limitations, on the other hand, describe the reservations when introducing this research into practical application.

Methodological Limitations

The main parts of this dissertation, namely chapters 2 to 5, have shown different aspects of the design, implementation, and validation of adaptable and self-adaptive systems. This required several choices to be made, that need to be critically assessed.

Design Choices Two noteworthy design choices of this dissertation are using the IEC 61499 models as an input, and applying the MAPE and MORPH self-adaptation models. The MAPE model is commonly used and operates on a high abstraction level. The MORPH architecture is less known, however the usage in this dissertation is exemplary and not prescriptive. In this sense, these models could be replaced as new models appear. The IEC 61499 standard, on the other hand, can appear as a questionable choice given the lack of adoption in practice. The selection of the IEC 61499 models in this dissertation is a strategic choice, grounded in their relevance to the research challenges, rather than an arbitrary decision. Shortcomings of the current state of the IEC 61499 standard are made clear throughout this work. The standard does, however, provide the best starting point for the path taken in this dissertation. The obvious alternative, the IEC 61131 standard, lacks some of the required characteristics, namely the reconfiguration framework and the fragmentation of the global state. Nevertheless, given some modifications, the contributions of this dissertation could be coined for any component-based architecture for ICS.

Behavior assumptions The generation of reconfiguration sequences in Chapter 3 relies on the knowledge of the execution traces between components. This knowledge is not defined within the IEC 61499 standard. The presence of this information is necessary to guarantee the functional consistency of the behavior. Without it, the algorithms presented in this dissertation may fail to uphold consistency conditions, potentially resulting in unpredictable behavior. In its absence, it is possible to approximate the execution traces from the component connections. It is also possible to observe the execution traces in a real system or a simulation. Better information, in this case, leads to a smaller disruption to the system, because fewer components need to be stopped. A lack of information, in contrast, would require larger areas of the application to be stopped.

Real-time assumptions Similarly, the existence of real-time scheduling information was taken for granted in Chapter 4, although it is not part of the IEC 61499 standard and is not available in current IDEs. Without this information, schedulability analysis is impossible and the timeliness of the adaptation cannot be verified. Also, generally,

real-time execution is not possible without knowing the constraints you need to satisfy. This usually includes deadlines, cycle times, and execution times.

Validation Chapter 4 and Chapter 5 both rely on measurements within a controlled environment and assumptions about how real-world examples would scale. In practical applications, this may be different. For instance, a novel runtime could be much faster or slower in executing individual reconfiguration instructions. Alternatively, a practical application may be larger or smaller than the scenarios identified within this dissertation, or their composition may differ. For instance, the critical RECONF phase of an application could be longer in comparison with the other phases. For this dissertation, shorter reconfiguration times or smaller reconfigurations would not undermine the overall conclusions. Significantly longer execution times of individual reconfiguration services could represent an issue that would need to be compensated through system design. However, as defined in the standard, these services should be executed quickly. Different sizes of applications were considered in the adaptation scenarios with generous margins.

These methodological limitations show the decisions and assumptions necessary to achieve the contributions. Individual choices can be seen as questionable, and some assumptions could have been weakened. Nevertheless, choices, such as using information that is not part of the IEC 61499 standard, were necessary to challenge the state of the art.

Generalizability Limitations

Apart from the methodological choices, the contributions of this dissertation could be seen as too narrow or only applicable to a small subset of systems. These generalizability limitations are discussed here.

Use case specificity As was previously seen, dynamic adaptation and particularly self-adaptation are not strict requirements for most technical systems, and they introduce overhead. Thus, the benefit of dynamic adaptability could be much smaller for some systems. For instance, there is a common argument that dynamic adaptation is less important for cars because they spend half the day in a stationary or safe position and don't need to be adapted while they are driving. Similarly, manufacturing systems that only produce during the day can be easily modified during the night shift. Consequently, the use case for dynamic adaptation may be smaller than expected. Nevertheless, it should not be forgotten that dynamic adaptation can be a stepping

stone towards self-adaptation, and that today's systems are not necessarily predictive of how future systems will operate. In a drift towards more software-defined systems, self-adaptation can displace hardware redundancies and enable fine-grained process optimizations within seconds. In connection with artificial intelligence, dynamic adaptation provides the agile *execution* step of the MAPE cycle, whereas AI could fit into the *analysis* and *planning* phases.

Ongoing developments The technology world is changing continuously. While the OT world is changing much slower than the IT world, it is changing nevertheless. This can lead to developments that make the contributions of this dissertation obsolete, for instance, a true "IT-OT convergence" in the sense of abandoning traditional PLC languages or introducing serverless architectures. However, the gap between IT and OT systems is not arbitrary. The requirements are different. While Javascript has taken a dominant position in web applications and is more and more leaking into the backend, it was simply not made to satisfy real-time requirements. And while the IT world is transitioning towards cloud architectures where serverless functions make sense, embedded systems are always linked to a physical system. The contributions of this dissertation build upon basic architectures for ICS and component-based systems, which are unlikely to become obsolete in the future.

Anticipating the future is inherently uncertain. Whether or not self-adaptation will eventually persist needs to be seen. The currently ongoing developments are exciting, yet do not pose threats to the applicability of dynamic adaptation as shown here. If anything, the softening of real-time constraints could make this research more likely to become practical.

6.2.2 Future Directions

Application A key component of extending this research should go towards applicability. This specifically means going back to practitioners, understanding their needs, problems, and capabilities, and identifying the correct solution that pushes the envelope of what is currently possible. On the other hand, one should not get stuck in the weeds of trying to implement small, incremental changes. This particularly concerns the application of real-time theory in practice. The current status quo in industrial automation seems to rely on cyclic execution because this is what is known to work. However, this theory fails for distributed applications. The IEC 61499 stan-

dard thrives on event-driven execution, however offers very little help to achieve this in real-time. Research has shown how event-driven execution can be achieved for the IEC 61499 standard, and this dissertation has shown that even dynamic adaptation can be performed under full real-time constraints. However, the real-time models rely on deadlines of event chains which most IEC 61499 users have no understanding of and wouldn't know how to specify. This gap needs to be closed, or other real-time models need to be found, that end-users would be comfortable with.

Implementation The Erlang implementation from Chapter 2 is prototypical and not necessarily suitable for hard real-time execution. The existing IEC 61499 implementations are lacking parts of the required features, e.g. full dynamic adaptation support and event-driven real-time execution. Furthermore, the higher levels of the WATERBEAR architecture, for instance, configuration generators and managers are non-existent in practice. Consequently, these hold potential for future research. Three areas are of particular interest:

1. A **fully-adaptable runtime environment** with a strong focus on clear real-time execution semantics,

2. **Configuration managers and validators** which can verify an application or configuration given a behavior specification or high-level constraints, and

3. **Configuration generators** which can autonomously create applications or configurations according to high-level descriptions or change requests.

The runtime environment serves as the base layer for experiments and practical use cases. The configuration managers and validators support the developer, emphasize the benefits of unambiguous execution semantics, and provide the ability to differentiate between "good" or "valid" and "bad" applications. Finally, the generators close the feedback loop to fully demonstrate the benefits of an adaptable runtime in practice.

Validation Closing the gaps in the theories and implementing a prototype for use in production systems are the necessary steps to finally validate the closed-loop behavior on a real-world system. Theoretical results suffice for research purposes, but to achieve adoption in a conservative domain such as industrial automation, the capabilities must be demonstrated in a real use case. This cannot be the first step, but it also certainly can't be the last—because enabling adaptability and agility in automation systems is not the end goal, it's the starting point for new research.

After agility The works in this dissertation should not be self-serving. The contributions are building blocks intended to make the idea of dynamic adaptation and self-adaptation digestible and palatable. Once they have arrived, the real work can begin to properly control and operate such systems optimally. How can we tell a self-adaptive control system what we want it to achieve? How can it explain its behavior and decision-making process back to the operator? How should these systems work together in global supply chains? What are the consequences on the hardware and human layers, when the software is fully adaptable? These questions will need to be answered by the following research.

6.3 Concluding Remarks

Current industrial control architectures must provide increased agility and adaptability to cope with the uncertainties and complexities of the modern, connected world. Yet, while the ideas of adaptability and agility have been discussed since the 1990s, modern closed-loop self-adaptation has not been achieved in industrial control systems (ICS).

In this dissertation, the gap between IT and OT concerning adaptability and agility was narrowed, and a roadmap towards resilience was brought to light. Explicitly, four contributions should be highlighted.

- The adaptability of current ICS middlewares was inspected and compared to IT environments. An adaptable middleware was implemented that allows the reuse of a modern IT ecosystem for ICS.
- The automatic generation of reconfiguration sequences was investigated and implemented using a correct-by-design algorithm that resolves the dependencies and respects the functional consistency requirements.
- Existing real-time models for ICS were extended to cover the dynamic adaptation phase. A metric was defined that allows the optimization of the reconfiguration sequences to respect both functional consistency and non-functional timing requirements.
- The effect of dynamic adaptation on resilience as part of a self-adaptive architecture was demonstrated. It was shown that all applications can benefit from dynamic adaptation, yet hard-to-restart systems benefit the most, and dynamic

adaptation capitalizes on fast detection and decision-making mechanisms.

Self-adaptation is not one discrete task but requires many distinct pieces to work together. Within this dissertation, some pieces to the puzzle were put into their correct location. Other pieces were sketched out, and new pieces were found. In the author's opinion, the task of self-adaptation is so vast, that there will always be areas of improvement. The best way to handle this prospect is to divide and conquer and build extensible interfaces between the components of these exciting, self-adaptive architectures.

Bibliography

[1] L. Prenzel and J. Provost. “Dynamic Software Updating of IEC 61499 Implementation Using Erlang Runtime System”. In: *World Congress of the International Federation of Automatic Control*. Vol. 50. Elsevier, 2017.

[2] L. Prenzel and J. Provost. “Implementation and Evaluation of IEC 61499 Basic Function Blocks in Erlang”. In: *International Conference on Emerging Technologies and Factory Automation*. Torino, Italy: IEEE, 2018.

[3] L. Prenzel, A. Zoitl, and J. Provost. “IEC 61499 Runtime Environments: A State of the Art Comparison”. In: *International Conference on Computer Aided Systems Theory*. Springer, 2019.

[4] L. Prenzel and J. Provost. “FBBeam: An Erlang-based IEC 61499 Implementation”. In: *International Conference on Industrial Informatics*. IEEE, 2019.

[5] L. Prenzel and S. Steinhorst. “Decentralized Autonomous Architecture for Resilient Cyber-Physical Production Systems”. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. Grenoble, France: IEEE, 2021.

[6] L. Prenzel and S. Steinhorst. “Automated Dependency Resolution in Dynamic Reconfiguration for IEC 61499”. In: *Proceedings of the International Conference on Emerging Technologies and Factory Automation (ETFA)*. Västerås, Sweden: IEEE, 2021.

[7] L. Prenzel, S. Hofmann, and S. Steinhorst. “Real-time Dynamic Reconfiguration for IEC 61499”. In: *International Conference on Industrial Cyber-Physical Systems (ICPS)*. Coventry, United Kingdom: IEEE, 2022.

[8] L. Prenzel and S. Steinhorst. “Towards Resilience by Self-Adaptation of Industrial Control Systems”. In: *International Conference on Emerging Technologies and Factory Automation (ETFA) 2022*. Stuttgart, Germany: IEEE, 2022.

[9] L. Prenzel, S. Hofmann, and S. Steinhorst. “Rollback Sequences for Dynamic Reconfiguration of IEC 61499”. In: *International Conference on Industrial Informatics (INDIN) 2022*. Perth, Australia: IEEE, 2022.

[10] A. Madhok. “Globalization, de-globalization, and re-globalization: Some historical context and the impact of the COVID pandemic”. In: *BRQ Business Research Quarterly* 24.3 (2021).

[11] V. Masson-Delmotte et al. “Climate Change 2021: The Physical Science Basis”. In: *Contribution of working group I to the sixth assessment report of the intergovernmental panel on climate change* (2021).

[12] M. S. Golan, L. H. Jernegan, and I. Linkov. “Trends and applications of resilience analytics in supply chain modeling: systematic literature review in the context of the COVID-19 pandemic”. en. In: *Environment systems & decisions* 40.2 (2020).

[13] H. Siagian, Z. J. H. Tarigan, and F. Jie. “Supply Chain Integration Enables Resilience, Flexibility, and Innovation to Improve Business Performance in COVID-19 Era”. en. In: *Sustainability: Science Practice and Policy* 13.9 (2021).

[14] C. S. Gascon. “Labor Constraints Remain Greatest Challenge for Resurgent Manufacturing Sector”. In: *The Regional Economist* (2022).

[15] R. N. Nagel and R. Dove. *21st Century Manufacturing Enterprise Strategy: An Industry-Led View*. en. Iacocca Institute, Lehigh University, 1991.

[16] M. G. Mehrabi, A. G. Ulsoy, and Y. Koren. “Reconfigurable manufacturing systems: Key to future manufacturing”. en. In: *Journal of intelligent manufacturing* 11.4 (2000).

[17] I. Baldini et al. “Serverless Computing: Current Trends and Open Problems”. In: *Research Advances in Cloud Computing*. Ed. by S. Chaudhary, G. Somani, and R. Buyya. Singapore: Springer Singapore, 2017.

[18] A. Jindal and M. Gerndt. “From DevOps to NoOps: Is it worth it?” In: *Communications in Computer and Information Science*. Communications in computer and information science. Cham: Springer International Publishing, 2021.

[19] J. O. Kephart and D. M. Chess. “The vision of autonomic computing”. In: *Computer* 36.1 (2003).

[20] IBM. “An architectural blueprint for autonomic computing”. In: *IBM White Paper* (2006).

[21] T. Goldschmidt, S. Hauck-Stattelmann, S. Malakuti, et al. “Container-based architecture for flexible industrial control applications”. In: *Journal of Systems* (2018).

[22] P. Gould. “What is agility?” en. In: *Manufacturing Engineering and Materials Processing* 76.1 (1997).

[23] H. Muccini, M. Sharaf, and D. Weyns. “Self-adaptation for cyber-physical systems: a systematic literature review”. In: *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS ’16. Austin, Texas: Association for Computing Machinery, Inc, 2016.

[24] H. Seifzadeh, H. Abolhassani, and M. S. Moshkenani. “A survey of dynamic software updating”. In: *Journal of Software: Evolution and Process* 25.5 (2013).

[25] D. Gupta and P. Jalote. “On-line software version change using state transfer between processes”. In: *Software: practice & experience* 23.9 (1993).

[26] M. A. Wermelinger. “Specification of software architecture reconfiguration”. en. PhD thesis. 1999.

[27] E. Miedes and F. D. Muñoz-Escoi. “A survey about dynamic software updating”. In: *Instituto Universitario Mixto Tecnologico de Infromatica, Universitat Politecnica de Valencia, Campus de Vera s* 46022 (2012).

[28] I. Mugarza, J. Parra, and E. Jacob. “Analysis of existing dynamic software updating techniques for safe and secure industrial control systems”. In: *International Journal of Safety and Security Engineering* 8.1 (2018).

[29] D. Weyns. “Software Engineering of Self-adaptive Systems”. In: *Handbook of Software Engineering*. Ed. by S. Cha, R. N. Taylor, and K. Kang. Cham: Springer International Publishing, 2019.

[30] B. H. C. Cheng et al. “Software engineering for self-adaptive systems: A research road map”. In: *Dagstuhl Seminar Proceedings*. 2008.

[31] K. Gama, W. Rudametkin, and D. Donsez. “Resilience in Dynamic Component-Based Applications”. In: *2012 26th Brazilian Symposium on Software Engineering*. IEEE, 2012.

[32] V. Ilvonen, P. Ihantola, and T. Mikkonen. “Dynamic Software Updating Techniques in Practice and Educator’s Guides: A Review”. In: *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*. IEEE, 2016.

[33] C. Ghezzi. “Of software and change”. In: *Journal of Software: Evolution and Process* 29.9 (2017).

[34] ISO/IEC/IEEE. *ISO/IEC/IEEE 14764:2022Software engineering — Software life cycle processes — Maintenance*. Tech. rep. 14764:2022. 2022.

[35] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan. “Types of software evolution and software maintenance”. In: *Journal of Software Maintenance and Evolution: Research and Practice*. NBS Special P 13.1 (2001).

[36] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. “Towards a taxonomy of software change”. In: *Journal of Software Maintenance and Evolution: Research and Practice*. Java Series 17.5 (2005).

[37] M. Salehie and L. Tahvildari. “Self-adaptive software: Landscape and research challenges”. In: *ACM Trans. Auton. Adapt. Syst.* 4.2 (2009).

[38] G. S. Blair, L. Blair, V. Issarny, P. Tuma, and A. Zarras. “The Role of Software Architecture in Constraining Adaptation in Component-Based Middleware Platforms”. In: *Middleware 2000*. Springer Berlin Heidelberg, 2000.

[39] D. Mlinarić. “Challenges in dynamic software updating”. In: *TEM Journal* 9.1 (2020).

[40] R. Lounas, M. Mezghiche, and J.-L. Lanet. “Formal methods in dynamic software updating: A survey”. In: *International Journal of Critical Computer-Based Systems* 9.1-2 (2019).

[41] N. Feng, G. Ao, T. White, and B. Pagurek. “Dynamic evolution of network management software by software hot-swapping”. In: *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings*. IEEE, 2001.

[42] Q. Wang, J. Shen, X. Wang, and H. Mei. “A component-based approach to online software evolution”. en. In: *Journal of Software Maintenance and Evolution: Research and Practice* 18.3 (2006).

[43] R. de Lemos, H. Giese, H. A. Müller, and M. Shaw. *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[44] Y. Brun et al. “Engineering Self-Adaptive Systems through Feedback Loops”. In: *Software Engineering for Self-Adaptive Systems*. Ed. by B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[45] P. Arcaini, E. Riccobene, and P. Scandurra. “Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation”. In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2015.

[46] D. Weyns et al. “On Patterns for Decentralized Control in Self-Adaptive Systems”. In: *Software Engineering for Self-Adaptive Systems II*. Ed. by R. de Lemos, H. Giese, H. A. Müller, and M. Shaw. Vol. 7475 LNCS. Springer, 2013.

[47] V. Braberman, N. D’Ippolito, J. Kramer, D. Sykes, and S. Uchitel. “MORPH: a reference architecture for configuration and behaviour self-adaptation”. In: *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*. CTSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015.

[48] J. Iber, T. Rauter, M. Krisper, and C. Kreiner. “The potential of self-adaptive software systems in industrial control systems”. In: *European Conference on Systems, Software and Services Process Improvement, EuroSPI 2017*. Vol. 748. Springer, Cham, 2017.

[49] D. Sloan, R. Alves Batista, and A. Loeb. “The Resilience of Life to Astrophysical Events”. en. In: *Scientific reports* 7.1 (2017).

[50] A. Zoitl. *Real-time Execution for IEC 61499*. en. Instrumentation, Systems, and Automation Society, 2009.

[51] J. Kramer and J. Magee. “Dynamic Configuration for Distributed Systems”. In: *IEEE Transactions on Software Engineering* SE-11.4 (1985).

[52] J. Kramer and J. Magee. “A model for change management”. In: *Proceedings of the Workshop on the Future Trends of Distributed Computing Systems in the 1990s*. IEEE, 1988.

[53] D. Weyns. “Software engineering of self-adaptive systems: an organised tour and future challenges”. In: *Chapter in Handbook of Software Engineering* (2017).

[54] P. Oreizy and R. N. Taylor. “On the role of software architectures in runtime system reconfiguration”. In: *IEE Proceedings - Software* 145.5 (1998).

[55] V. Vyatkin. “IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review”. In: *IEEE Transactions on Industrial Informatics* 7.4 (2011).

[56] Schneider Electric. *Universal Automation - A call for change*. Tech. rep. 998-21066447. 2020.

[57] G. Lyu and R. W. Brennan. “Towards IEC 61499-Based Distributed Intelligent Automation: A Literature Review”. In: *IEEE Transactions on Industrial Informatics* 17.4 (2021).

[58] V. R. Segovia and A. Theorin. “History of Control History of PLC and DCS”. In: *University of Lund* (2012).

[59] M. Tiegelkamp and K. H. John. *IEC 61131-3: Programming Industrial Automation Systems — SpringerLink*. `https://link.springer.com/content/pdf/10.1007/978-3-662-07847-1.pdf`. Accessed: 2018-8-10. 1995.

[60] B. Vogel-Heuser and F. Ocker. “Maintainability and evolvability of control software in machine and plant manufacturing—An industrial survey”. In: *Control engineering practice* (2018).

[61] H. Kopetz. “Event-triggered versus time-triggered real-time systems”. In: *Operating Systems of the 90s and Beyond*. Springer Berlin Heidelberg, 1991.

[62] B. Werner. “Object-oriented extensions for iec 61131-3”. In: *IEEE Industrial Electronics Magazine* 3.4 (2009).

[63] M. N. Rooker, C. Sünder, T. Strasser, A. Zoitl, O. Hummer, and G. Ebenhofer. “Zero Downtime Reconfiguration of Distributed Automation Systems: The ϵCEDAC Approach”. en. In: *Holonic and Multi-Agent Systems for Manufacturing*. Springer, 2007.

[64] V. Vyatkin and H. M. Hanisch. “Formal modeling and verification in the software engineering framework of IEC 61499: a way to self-verifying systems”. In: *International Conference on Emerging Technologies and Factory Automation*. Vol. 2. IEEE, 2001.

[65] C. Schnakenbourg, J. M. Faure, and J. J. Lesage. “Towards IEC 61499 function blocks diagrams verification”. In: *International Conference on Systems, Man and Cybernetics*. Vol. 3. IEEE, 2002.

[66] T. Strasser, A. Zoitl, J. H. Christensen, and C. Sünder. “Design and Execution Issues in IEC 61499 Distributed Automation and Control Systems”. In: *IEEE Transactions on Systems, Man and Cybernetics* 41.1 (2011).

[67] A. Zoitl and R. Lewis. *Modelling control systems using IEC 61499: Applying function blocks to distributed systems*. Vol. 95. IET, 2014.

[68] L. Ferrarini and C. Veber. “Implementation approaches for the execution model of IEC 61499 applications”. In: *International Conference on Industrial Informatics*. IEEE, 2004.

[69] V. Vyatkin. “The IEC 61499 standard and its semantics”. In: *IEEE Industrial Electronics Magazine* 3.4 (2009).

[70] G. Cengic, O. Ljungkrantz, and K. Akesson. “Formal Modeling of Function Block Applications Running in IEC 61499 Execution Runtime”. In: *Conference on Emerging Technologies and Factory Automation*. IEEE, 2006.

[71] G. Cengic and K. Akesson. “On Formal Analysis of IEC 61499 Applications, Part B: Execution Semantics”. In: *IEEE Transactions on Industrial Informatics* (2010).

[72] C. Sünder et al. “Usability and Interoperability of IEC 61499 based distributed automation systems”. In: *International Conference on Industrial Informatics*. IEEE, 2006.

[73] A. Zoitl, G. Grabmair, F. Auinger, and C. Sünder. “Executing real-time constrained control applications modelled in IEC 61499 with respect to dynamic reconfiguration”. In: *International Conference on Industrial Informatics*. IEEE, 2005.

[74] G. Cengic and K. Akesson. “Definition of the execution model used in the Fuber IEC 61499 runtime environment”. In: *International Conference on Industrial Informatics*. IEEE, 2008.

[75] K. Thramboulidis and A. Zoupas. “Real-time Java in control and automation: a model driven development approach”. In: *Conference on Emerging Technologies and Factory Automation*. Vol. 1. IEEE, 2005.

[76] G. S. Doukas and K. C. Thramboulidis. “A real-time Linux execution environment for function-block based distributed control applications”. In: *International Conference on Industrial Informatics*. IEEE, 2005.

[77] K. Thramboulidis and N. Papakonstantinou. “An IEC 61499 Execution Environment for an aJile-based Field Device”. In: *Conference on Emerging Technologies and Factory Automation*. IEEE, 2006.

[78] holobloc. *FBDK 8.0 - The Function Block Development Kit*. `https://www.holobloc.com/fbdk8/index.htm`. Accessed: 2019-5-24.

[79] V. Vyatkin and J. Chouinard. “On Comparisons of the ISaGRAF implementation of IEC 61499 with FBDK and other implementations”. In: *International Conference on Industrial Informatics*. IEEE, 2008.

[80] 4diac. *4diac FORTE - The 4diac runtime environment*. `https://www.eclipse.org/4diac/en_rte.php`. Accessed: 2019-5-24. 2019.

[81] A. Zoitl, T. Strasser, and A. Valentini. “Open source initiatives as basis for the establishment of new technologies in industrial automation: 4DIAC a case study”. In: *International Symposium on Industrial Electronics*. IEEE, 2010.

[82] L. I. Pinto, C. D. Vasconcellos, R. S. U. Rosso, and G. H. Negri. “ICARU-FB: An IEC 61499 Compliant Multiplatform Software Infrastructure”. In: *IEEE Transactions on Industrial Informatics* 12.3 (2016).

[83] J. H. Christensen et al. “The IEC 61499 function block standard: Software tools and runtime platforms”. In: *ISA Automation Week* (2012).

[84] nxtcontrol. *nxtControl - nxtIECRT*. `https://www.nxtcontrol.com/en/control/`. Accessed: 2019-5-24. 2019.

[85] P. Lindgren, M. Lindner, A. Lindner, D. Pereira, and L. M. Pinho. “RTFM-core: Language and implementation”. In: *Conference on Industrial Electronics and Applications*. IEEE, 2015.

[86] P. Lindgren, M. Lindner, A. Lindner, V. Vyatkin, D. Pereira, and L. M. Pinho. “A real-time semantics for the IEC 61499 standard”. In: *Conference on Emerging Technologies Factory Automation*. IEEE, 2015.

[87] International Electrotechnical Commission. *IEC 61499*. Tech. rep. International Electrotechnical Commission, 2012.

[88] O. Hummer, C. Sünder, A. Zoitl, T. Strasser, M. N. Rooker, and G. Ebenhofer. “Towards Zero-downtime Evolution of Distributed Control Applications via Evolution Control based on IEC 61499”. In: *IEEE Conference on Emerging Technologies and Factory Automation*. IEEE, 2006.

[89] O. Hummer, C. Sünder, T. Strasser, M. N. Rooker, and G. Kerbleder. “Downtimeless System Evolution: Current State and Future Trends”. In: *IEEE International Conference on Industrial Informatics*. Vol. 2. IEEE, 2007.

[90] C. Sünder, V. Vyatkin, and A. Zoitl. “Formal Verification of Downtimeless System Evolution in Embedded Automation Controllers”. In: *ACM Transactions on Embedded Computing Systems* 12.1 (2013).

[91] M. Cronqvist. *The Nine Nines and How To Get There*. Erlang Factory SF Bay 2010. San Francisco, 2010.

[92] F. Cesarini and S. Thompson. *Erlang Programming: A Concurrent Approach to Software Development*. en. "O'Reilly Media, Inc.", 2009.

[93] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. en. Pragmatic Bookshelf, 2007.

[94] F. Cesarini and S. Vinoski. *Designing for Scalability with Erlang/OTP: Implement Robust, Fault-Tolerant Systems*. en. "O'Reilly Media, Inc.", 2016.

[95] S. St. Laurent. *Introducing Erlang*. en. "O'Reilly Media, Inc.", 2013.

[96] F. Hebert. *Learn You Some Erlang for Great Good!: A Beginner's Guide*. en. No Starch Press, 2013.

[97] E. Stenman. *The BEAM Book*. `https://github.com/happi/theBeamBook`. Accessed: 2019-10-22. 2019.

[98] H. Soleimani. *Erlang Garbage Collection Details and Why It Matters*. `https://hamidreza-s.github.io/erlang%20garbage%20collection%20memory%20layout%20soft%20realtime/2015/08/24/erlang-garbage-collection-details-and-why-it-matters.html`. Accessed: 2017-6-9. 2015.

[99] A. B. Ericsson. *System Architecture Support Libraries (SASL) Reference Manual Version 4.2*. `https://www.erlang.org/`. Accessed: 2022-8-22. 2022.

[100] K. Kruger and A. Basson. "Erlang-based control implementation for a holonic manufacturing cell". In: *International Journal of Computer Integrated Manufacturing* 30.6 (2017).

[101] T. Lindahl and K. Sagonas. "Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story". In: *Programming Languages and Systems*. Springer Berlin Heidelberg, 2004.

[102] R. Wilhelm et al. "The worst-case execution-time problem—overview of methods and survey of tools". In: *ACM Transactions on Embedded Computing Systems* 7.3 (2008).

[103] A. A. Santos, A. F. Silva, M. de Sousa, and P. Magalhães. "An IEC 61499 Replication for Distributed Control Applications". In: *International Conference on Industrial Informatics*. IEEE, 2018.

[104] G. Chilingarov. *Message order and delivery guarantees in Elixir/Erlang*. `https://medium.com/learn-elixir/message-order-and-delivery-guarantees-in-elixir-erlang-9350a3ea7541`. Accessed: 2022-5-20. 2017.

[105] J. Högberg. *A few notes on message passing*. `https://www.erlang.org/blog/message-passing/`. Accessed: 2022-5-20. 2021.

[106] A. Zoitl and H. Prähofer. "Guidelines and Patterns for Building Hierarchical Automation Solutions in the IEC 61499 Modeling Language". In: *IEEE Transactions on Industrial Informatics* 9.4 (2013).

[107] V. Nicosia. "Towards hard real-time erlang". In: *Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*. ERLANG '07. Freiburg, Germany: Association for Computing Machinery, 2007.

[108] L. Nahabedian et al. "Dynamic Update of Discrete Event Controllers". In: *IEEE Transactions on Software Engineering* (2018).

[109] M. Zhang, K. Ogata, and K. Futatsugi. "Towards a Formal Approach to Modeling and Verifying the Design of Dynamic Software Updates". In: *2015 Asia-Pacific Software Engineering Conference (APSEC)*. Vol. 2016-May. IEEE Computer Society, 2015.

[110] V. Panzica La Manna. "Local Dynamic Update for Component-based Distributed Systems". In: *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*. CBSE '12. Bertinoro, Italy: ACM, 2012.

[111] C. Giuffrida, C. Iorgulescu, and A. S. Tanenbaum. “Mutable checkpoint-restart: automating live update for generic server programs”. In: *Proceedings of the 15th International Middleware Conference*. Middleware ’14. Bordeaux, France: Association for Computing Machinery, 2014.

[112] C. Ghezzi, J. Greenyer, and V. P. L. Manna. “Synthesizing dynamically updating controllers from changes in scenario-based specifications”. In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2012.

[113] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster. “Specifying and Verifying the Correctness of Dynamic Software Updates”. In: *Verified Software: Theories, Tools, Experiments*. Vol. 7152. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[114] J. Zhang and B. H. C. Cheng. “Using temporal logic to specify adaptive program semantics”. In: *The Journal of systems and software* 79.10 (2006).

[115] J. Zhang and B. H. C. Cheng. “Specifying adaptation semantics”. In: *Proceedings of the 2005 workshop on Architecting dependable systems*. WADS ’05. St. Louis, Missouri: Association for Computing Machinery, 2005.

[116] D. Gupta, P. Jalote, and G. Barua. “A formal framework for on-line software version change”. In: *IEEE Transactions on Software Engineering* 22.2 (1996).

[117] J. Kramer and J. Magee. “The evolving philosophers problem: dynamic change management”. In: *IEEE Transactions on Software Engineering* 16.11 (1990).

[118] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D’Hondt. “Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates”. In: *IEEE Transactions on Software Engineering* 33.12 (2007).

[119] L. Baresi, C. Ghezzi, X. Ma, and V. P. L. Manna. “Efficient Dynamic Updates of Distributed Components Through Version Consistency”. In: *IEEE Transactions on Software Engineering* 43.4 (2017).

[120] M. Wermelinger. “A hierarchic architecture model for dynamic reconfiguration”. In: *International Workshop on Software Engineering for Parallel and Distributed Systems*. IEEE, 1997.

[121] J. Kramer and J. Magee. “Analysing dynamic change in software architectures: a case study”. In: *Proceedings. Fourth International Conference on Configurable Distributed Systems (Cat. No.98EX159)*. IEEE, 1998.

[122] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, and D. A. Menascé. “Software adaptation patterns for service-oriented architectures”. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC ’10. Sierre, Switzerland: Association for Computing Machinery, 2010.

[123] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu. “Version-consistent dynamic reconfiguration of component-based distributed systems”. In: *ACM SIGSOFT symposium and the European conference on Foundations of software engineering - SIGSOFT/FSE ’11*. ACM Press, 2011.

[124] Z. Zhao, Y. Jiang, C. Xu, T. Gu, and X. Ma. “Synthesizing Object State Transformers for Dynamic Software Updates”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021.

[125] A. Rasche and A. Polze. “Dynamic reconfiguration of component-based real-time software”. en. In: *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. IEEE, 2005.

[126] V. Panzica La Manna. “Dynamic Software Update for Component-based Distributed Systems”. In: *International Workshop on Component-oriented Programming*. WCOP ’11. New York, NY, USA: ACM, 2011.

[127] B. Wiesmayr and A. Zoitl. “Requirements for a dynamic interface model of IEC 61499 Function Blocks”. In: *International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. IEEE, 2020.

[128] A. B. Kahn. “Topological sorting of large networks”. In: *Communications of the ACM* (1962).

[129] C. Sünder, A. Zoitl, B. Favre-Bulle, T. Strasser, H. Steininger, and S. Thomas. “Towards Reconfiguration Applications as basis for Control System Evolution in Zero-downtime Automation Systems”. In: *Intelligent Production Machines and Systems*. Oxford: Elsevier Science Ltd, 2006.

[130] A. Zoitl, R. Smodic, C. Sünder, and G. Grabmair. “Enhanced real-time execution of modular control software based on IEC 61499”. In: *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.* IEEE, 2006.

[131] A. Zoitl, T. Strasser, K. Hall, R. Staron, C. Sünder, and B. Favre-Bulle. “The Past, Present, and Future of IEC 61499”. en. In: *Holonic and Multi-Agent Systems for Manufacturing*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, Berlin, Heidelberg, 2007.

[132] L. Sha, R. Rajkumar, and J. P. Lehoczky. “Priority inheritance protocols: an approach to real-time synchronization”. In: *IEEE Transactions on Computers* 39.9 (1990).

[133] I. Koren and C. Mani Krishna. *Fault-Tolerant Systems*. en. Morgan Kaufmann, 2020.

[134] S. Hosseini, K. Barker, and J. E. Ramirez-Marquez. “A review of definitions and measures of system resilience”. In: *Reliability Engineering & System Safety* 145 (2016).

[135] E. Hollnagel, D. D. Woods, and N. Leveson. *Resilience Engineering: Concepts and Precepts*. Ashgate Publishing, 2006.

[136] D. Henry and J. Emmanuel Ramirez-Marquez. “Generic metrics and quantitative approaches for system resilience as a function of time”. In: *Reliability Engineering & System Safety* 99 (2012).

[137] M. Bruneau et al. “A Framework to Quantitatively Assess and Enhance the Seismic Resilience of Communities”. In: *Earthquake Spectra* 19.4 (2003).

[138] D. Ye, M. Zhang, and A. V. Vasilakos. “A Survey of Self-Organization Mechanisms in Multiagent Systems”. In: *IEEE transactions on systems, man, and cybernetics* 47.3 (2017).

[139] D. Etz, P. Denzler, T. Frühwirth, and W. Kastner. “Functional Safety Use Cases in the Context of Reconfigurable Manufacturing Systems”. In: *International Conference on Emerging Technologies and Factory Automation (ETFA)*. Stuttgart, Germany: IEEE, 2022.

[140] R. H. Bordini, M. Dastani, and M. Winikoff. “Current Issues in Multi-Agent Systems Development”. In: *Engineering Societies in the Agents World VII*. Springer Berlin Heidelberg, 2007.

[141] Y. Koren, X. Gu, and W. Guo. “Reconfigurable manufacturing systems: Principles, design, and future trends”. In: *Frontiers of Mechanical Engineering in China* 13.2 (2018).

[142] S. Jeschke, C. Brecher, T. Meisen, D. Özdemir, and T. Eschert. “Industrial Internet of Things and Cyber Manufacturing Systems”. In: *Industrial Internet of Things: Cybermanufacturing Systems*. Cham: Springer International Publishing, 2017.

[143] P. Skobelev and D. Trentesaux. “Disruptions Are the Norm: Cyber-Physical Multi-agent Systems for Autonomous Real-Time Resource Management”. In: *Service Orientation in Holonic and Multi-Agent Manufacturing*. Springer International Publishing, 2017.

[144] R. Ramezanian, D. Rahmani, and F. Barzinpour. “An aggregate production planning model for two phase production systems: Solving with genetic algorithm and tabu search”. In: *Expert systems with applications* 39.1 (2012).

[145] N. He, D. Z. Zhang, and Q. Li. “Agent-based hierarchical production planning and scheduling in make-to-order manufacturing system”. In: *International Journal of Production Economics* 149 (2014).

[146] X. Xu. “From cloud computing to cloud manufacturing”. In: *Robotics and computer-integrated manufacturing* 28.1 (2012).

[147] X. Zhu, J. Shi, S. Huang, and B. Zhang. “Consensus-oriented cloud manufacturing based on blockchain technology: An exploratory study”. In: *Pervasive and mobile computing* 62 (2020).

[148] J. Zhang, G. Ding, Y. Zou, S. Qin, and J. Fu. “Review of job shop scheduling research and its new perspectives under Industry 4.0”. In: *Journal of intelligent manufacturing* 30.4 (2019).

[149] B. Shala, U. Trick, A. Lehmann, B. Shala, B. Ghita, and S. Shiaeles. “Trust-Based Composition of M2M Application Services”. In: *2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN)*. IEEE, 2018.

[150] K. Muller and T. Vignaux. “Simpy: Simulating systems in python”. In: *ONLamp. com Python Devcenter* 650 (2003).

[151] B. Pourmohseni, S. Wildermann, M. Glaß, and J. Teich. “Hard real-time application mapping reconfiguration for NoC-based many-core systems”. In: *Real-Time Systems* 55.2 (2019).

Acronyms

CPPS cyber-physical production system

CPS cyber-physical system

DAG directed acyclic graph

DM deadline monotonic

ECC execution control chart

EDF earliest deadline first

EROI evolution region of interest

ERTS Erlang Runtime System

FB function block

FBD function block diagram

IAS industrial automation system

ICS industrial control system

IL instruction list

IT information technology

LD ladder diagram

MAS multi-agent system

OT operational technology

OTP Open Telecom Platform

PCP priority ceiling protocol

PID process identifier

PLC programmable logic controller

POU program organization unit

QoS quality of service

RCA reconfiguration control application

RM rate monotonic

RMS Reconfigurable Manufacturing System

RTE runtime environment

SFC sequential function chart

ST structured text

V&V validation & verification

WCET worst case execution time