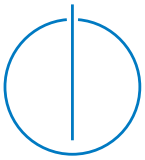


Technische Universität München
Fakultät für Informatik
Lehrstuhl III - Datenbanksysteme

A Distributed Data Processing Perspective on Industrial Real-Time Systems

Thomas Kothmayr
Master of Science with honours

Technische Universität München
Fakultät für Informatik
Lehrstuhl III - Datenbanksysteme

A Distributed Data Processing Perspective on Industrial Real-Time Systems

Thomas Kothmayr, M.Sc. with honours

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Prof. Dr. Uwe Baumgarten
Prüfer der Dissertation:	1.	Prof. Alfons Kemper, Ph. D.
	2.	Prof. Dr. Harald Kosch, Universität Passau

Die Dissertation wurde am 03.03.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 22.06.2017 angenommen.

Abstract

The current generation of manufacturing systems relies on monolithic control software which provides real-time guarantees but is hard to adapt and reuse. These qualities are becoming increasingly important for meeting the demands of a global economy. Ongoing research and industrial efforts therefore focus on service-oriented architectures (SOA) to increase the control software's flexibility while reducing development time, effort and cost. With such encapsulated functionality, system behavior can be expressed in terms of operations on data and the flow of data between operators. In this thesis we consider industrial real-time systems from the perspective of distributed data processing systems. Data processing systems often must be highly flexible, which can be achieved by a declarative specification of system behavior. In such systems, a user expresses the properties of an acceptable solution while the system determines a suitable execution plan that meets these requirements. Applied to the real-time control domain, this means that the user defines an abstract workflow model with global timing constraints from which the system derives an execution plan that takes the underlying system environment into account. The generation of a suitable execution plan often is NP-hard and many data processing systems rely on heuristic solutions to quickly generate high quality plans. We utilize heuristics for finding real-time execution plans. Our evaluation shows that heuristics were successful in finding a feasible execution plan in 99% of the examined test cases. Lastly, data processing systems are engineered for an efficient exchange of data and therefore are usually built around a direct data flow between the operators without a mediating entity in between. Applied to SOA-based automation, the same principle is realized through service choreographies with direct communication between the individual services instead of employing a service orchestrator which manages the invocation of all services participating in a workflow.

These three principles outline the main contributions of this thesis: A flexible reconfiguration of SOA-based manufacturing systems with verifiable real-time guarantees, fast heuristics based planning, and a peer-to-peer execution model for SOAs with clear semantics. We demonstrate these principles within a demonstrator that is close to a real-world industrial system.

Kurzfassung

Die heutige Generation von Fertigungssystemen nutzt monolithische Kontrollsoftware, welche zwar Echtzeitgarantien bietet, aber dafür schwer wartbar und kaum wiederverwendbar ist. Diese Eigenschaften werden jedoch zunehmend wichtiger um den Anforderungen einer globalisierten Wirtschaft gerecht zu werden. Die aktuelle Forschung und Entwicklungsbestrebungen der Industrie setzen daher auf serviceorientierte Architekturen (SOAs), um die Flexibilität der Kontrollsoftware zu erhöhen und dadurch Entwicklungsaufwand, -Zeit und -Kosten einzusparen. Durch die Kapselung der Funktionalität in SOAs kann das Systemverhalten dabei auf der Ebene von Operatoren, sowie dem Datenfluss zwischen diesen Operatoren, ausgedrückt werden. In dieser Arbeit betrachten wir daher industrielle Echtzeitsysteme vom Standpunkt der verteilten Datenverarbeitung.

Datenverarbeitungssysteme müssen oft hoch flexibel sein, was sich durch eine deklarative Spezifikation des Systemverhaltens umsetzen lässt. Nutzer beschreiben in solchen Systemen die Eigenschaften von akzeptablen Lösungen woraufhin das System selbsständig einen Ausführungsplan erstellt, welcher diese Eigenschaften erfüllt. Angewendet auf die Domäne von Echtzeitsystemen bedeutet dies, dass Nutzer einen abstrakten Ablaufplan in Form eines Graphen mit globalen zeitlichen Anforderungen defineren ausgehend von dem das System eigenständig einen Ausführungsplan erstellt, welcher die konkrete Systemumgebung berücksichtigt. Die Erzeugung solch eines Ausführungsplans ist meist NP-schwer, weswegen viele Datenverarbeitungssysteme heuristische Lösungen nutzen um in kurzer Zeit hochqualitative Lösungen zu erstellen. Wir nutzen ebenfalls Heuristiken um echtzeitfähige Ausführungspläne zu erstellen. Unsere Auswertung zeigt, dass Heuristiken in 99% der untersuchten Fälle einen echtzeitfähigen Ablaufplan erstellen konnten, falls solch ein Plan existiert. Datenverarbeitungssysteme sind zudem auf effiziente Kommunikation ausgelegt und tauschen daher Daten meist direkt zwischen den einzelnen Operatoren aus ohne über Dritte zu kommunizieren. In SOAs wird dieses Prinzip durch Servicechoreographien mit direkter Kommunikation zwischen den einzelnen Services umgesetzt, welche im Gegensatz zur Serviceorchestrierung durch einen Orchestrator stehen der die einzelenen Services aufruft und die Kommunikation zwischen ihnen verwaltet.

Die drei obenstehenden Prinzipien skizzieren bereits die wissenschaftlichen Beiträge dieser Dissertation: Die flexible Rekonfiguration von SOA-basierten Fertigungssystemen mit verifizierbaren Echtzeiteigenschaften, schnelle Generierung von Ausführungsplänen für solche Systeme mit Hilfe von Heuristiken und ein Peer-to-Peer Ausführungsmodell für SOAs mit einer klar definierten Semantik. Zusätzlich verdeutlichen wir diese Prinzipien in einem Demonstrator, der an ein Echtweltsystem angelehnt ist.

Acknowledgements

I would like to express my gratitude to Prof. Alfons Kemper for enabling me to work on this research topic at his chair for the last years and for providing me with guidance and support throughout. Dr. Jörg Heuer and Dr. Andreas Scholz from our industry partner Siemens CT have also played a large role in the successful conduction of this research endeavor. I would like to thank them for the valuable discussions and their input to both the alignment and execution of my research. I also thank Prof. Kosch from the University of Passau for reviewing this thesis.

Special thanks go to Michele Blank who setup the electromechanical components of the real-world demonstrator described in this thesis. He also implemented the event-driven control software, which served as a basis for the time triggered architecture of the demonstrator. I also thank all students who worked in the context of the rtSOA project.

I would also like to thank all current and former members of the chair for database systems at TUM for their collegiality, input and the enjoyable working atmosphere. I would like to especially mention Aleksandar Bojchevski for giving me the idea to examine the information gain of various factors influencing the effectiveness of the rtSOA heuristics. Thank you to Silke Prestel for keeping my working hours largely free of paper work.

Lastly, thank you to my family and friends for the support and encouragement, whenever necessary.

Für Hannah, meinen Grund hart -
aber nicht zu lange - zu arbeiten.

Contents

"Then there is the man who drowned crossing
a stream with an average depth of six inches."
- W.I.E. Gates

CHAPTER 1

Introduction

This thesis takes a distributed data processing perspective on industrial real-time systems. The main building block of this approach is a service-oriented architectural style that encapsulates mechatronic or computational functionality in self-describing services. We have therefore named our approach *rtSOA*. With these self-contained, reusable services the development of automation systems can be shifted to reasoning over the flow of data between service instances, thus taking a step into the direction of distributed data processing (DDP) systems. To realize a high level of flexibility, users of DDP systems often do not have to explicitly specify concrete execution plans. Users instead list the properties of acceptable solutions for which the DDP system will automatically generate a matching execution plan. This plan fulfills the specified requirements while hiding the complexities of the underlying networked system from the user. rtSOA applies this principle by only requiring the specification of global real-time constraints for a given task-graph, which constitutes the distributed real-time automation task. Timing constraints for individual service instances are derived automatically from the global constraints, the concrete placement of the services on machines and the configuration of the connecting network. The execution plan then consists of a schedule for the network and each participating device, so that the specified real-time requirements are realized.

rtSOA is focused on enabling wide-reaching reuse of existing code modules and enabling fast, iterative and incremental development for distributed real-time systems. The main use case for rtSOA lies within the area of industrial manufacturing. In recent years increasing uncertainty about the production volume of a product is accompanied by demand fluctuations over the product's life cycle [53, 103, 117]. To address these issues, the paradigm of the *reconfigurable manufacturing system* (RMS) has been postulated in research. Software reconfiguration is a key component in

enabling reconfigurable manufacturing systems, which therefore benefit immensely from a modular and reconfigurable software architecture, supported by a declarative approach to the generation of execution plans. To the best of our knowledge, rtSOA is the first approach that achieves predictable and deterministic execution plans by applying distributed data processing principles in a flexible and reconfigurable service-oriented architecture.

In the following, we present basic principles of distributed data processing systems in Section 1.1 and real-time systems in Section 1.2 before giving an overview of the goals and contributions of this thesis in Section 1.3 and detailing the outline in Section 1.4.

1.1 Distributed Data Processing

Distributed data processing may become necessary due to different reasons: Either because the amount of data cannot be handled on a single device or because the processing of data in the network is more efficient in terms of computation time, energy efficiency or other resources, including organizational resources such as money or employee working hours. In this section we will outline several different systems which are contained under the umbrella of distributed data processing and which demonstrate certain properties or principles of DDP that are applicable to the rtSOA approach.

In recent years, distributed data processing has often become synonymous with processing of big data sets that cannot be contained on a single machine. One of the most successful programming paradigms in that area is MapReduce [31], which requires the programmer to specify computation in terms of a map and a reduce function. The underlying library then takes care of issues such as parallelization, inter-machine communication and fault tolerance. Efficient scheduling of network resources becomes particularly relevant when aiming for high overall system performance. Starting from the observation that the increase of computational resources has progressed at a faster pace than the increase of network communication speeds, Rödiger et al. have analyzed the impact of network scheduling on joins in distributed databases [96] and designed a network-optimized join that is based on solving two distinct optimization problems. The first of these problems is the optimal partition assignment which specifies which parts of the input data will be processed on which machine. By choosing an optimal partition assignment, the length of the network transfer phase can be minimized. The second optimization problem lies with determining a suitable communication schedule. Cross traffic and other inefficiencies can render an otherwise efficient execution plan ineffective, meaning that much time is spent idly waiting for network packets that obstruct each other during transmission. Applied to the field of industrial real-time systems, the two optimization problems solved by Rödiger et al. map to determining a suitable assignment of (computational) tasks to machines and determining an efficient overall execution plan that takes network particularities into account.

Stream processing systems are moving away from traditional database systems, which operate on stored data, towards execution semantics over data streams which operate on in-flight data, meaning data that is analyzed as soon as it is produced and may also carry a temporal component. Data for stream processing systems is often generated by distributed systems and the stream processing system may itself be a distributed system. Stonebraker et al. [105] detail eight general rules for real-time stream processing. Among these rules are performance requirements, demanding that data should be processed "in-stream" (1) without a separate storage step to achieve suitably low latency and the general requirement that the system must be able to "keep up" (2) with the amount of data generated. Rules for consistency demand that a system must process data in a repeatable and predictable (3) manner while handling stream imperfections (4) such as delayed, missing or out of order data. The stream processing systems must also be highly available and ensure data integrity under hardware failures (5). Usability requirements for a declarative stream query language (6) with easy integration of stored data (7) and support for automatic partitioning and scaling (8) are classic principles of DDP systems.

Many of the outlined principles can also be applied to industrial real-time systems, which must process sensor data as it is generated (1) while always offering guaranteed timing properties (2). Automation systems should remain deterministic (3), even when handling communication (4) and hardware failures (5). rtSOA aims to introduce some of the usability principles described by Stonebraker et al. to the area of industrial real-time systems by allowing the user to specify high-level timing constraints without manual consideration of network particularities (c.f. rules 6 and 8). Real-time in the context of the above paragraph means "a timely manner", without timing guarantees, whereas real-time in the context of this thesis always means real-time guarantees. Section 1.2 introduces our notion of the term.

Another requirement for using a declarative approach to generating execution plans is the presence of well understood and stringent semantics for the specification language. In traditional database management systems, the relational algebra [60, Chapter 8] provides the semantics and logical basis for the generation of the query execution plan. The semantics of a relational algebra expression are contained in the semantics of the individual operators used in the expression and the data flow between them. Contrastingly, rtSOA views individual services in a SOA as black-box operators and only reasons about the data flow between them. Section 3.4 defines these semantics.

The main driver for distributed data processing in embedded systems lies not in handling otherwise intractable amounts of data. Instead, achieving a balance of flexibility and efficiency in highly complex systems of systems is often the main motivation behind applying distributed data processing methods in embedded networks. One example for such a system is TinyDB [78]. TinyDB is an acquisitional query processing system that explicitly addresses, and deeply integrates with, the particularities of wireless sensor networks. Sensor networks consists of tens or hundreds of

self-organizing battery powered nodes that operate without human intervention for months or years. Since sensor nodes are usually battery powered and severely energy constrained, optimization for energy efficiency and limiting network communication are essential for realizing a practical system. One way to realize long battery life spans is to configure sensor nodes to sleep in a low-power state for most of the time and only wake up periodically to perform data acquisition, processing and network communication. Communication often follows a multi-hop pattern to one or more gateways which act as data sinks. By performing in-network aggregation along these multi-hop communication paths, the network traffic and thus the energy usage can be greatly reduced. Acquiring sensor data and performing data aggregation and analysis is greatly simplified by using TinyDB's declarative SQL-like query interface instead of manually configuring the data acquisition and in-network aggregation behavior of each sensor node. In contrast to traditional database or stream processing systems, TinyDB can control the frequency of when sensor data is acquired instead of acting on stored data or only analyzing incoming data streams. Other issues, such as determining which sensor nodes have relevant data and in which order samples should be taken, are more closely related to traditional query optimization and stream processing.

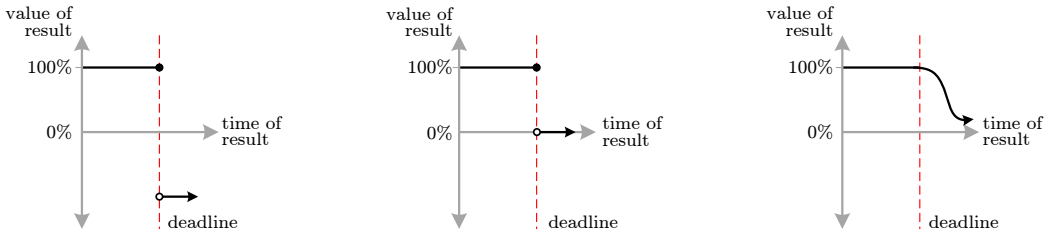
TinyDB illustrates a flexible but specialized solution to distributed data processing in embedded (sensor) networks. Scholz et al. presented a more general solution in the form of a service-oriented middleware for embedded networks named ϵSOA [98]. ϵSOA takes a data centric view on service-oriented architectures and models service interaction following a push-based communication pattern. As explained in the context of TinyDB, various data flows inside the network should be optimized in terms of the timing and frequency of the individual messages. The ϵSOA stream dispatcher therefore applies principles of data stream management systems to optimize the rate of all data streams in terms of energy efficiency and desired message interval. The ϵSOA middleware additionally tackles the service placement problem with the simulated annealing meta heuristic. Based on an abstract representation of the workflow to be executed by the devices in the embedded network, ϵSOA transparently applies its optimization techniques and thus hides the details of the underlying network structure from the application developer. This approach follows the same high-level structure as the MapReduce library where users are required to specify their application following certain development principles but are freed from explicit consideration of network issues because the middleware is addressing them. Users can thus reason about the implementation on a higher level of abstraction.

The examples in this section outline that distributed data processing systems may share some commonalities in their principles but have to be carefully adapted to their intended target area. In Section 1.2 we therefore give more background information about general real-time systems before detailing the system environment for our envisioned distributed data processing approach to industrial real-time automation in Section 2.

1.2 Background: Real-Time Systems

This short introduction to real-time systems is based on the book "Hard Real-Time Computing Systems" by Buttazzo [14]. The fundamental difference between a real-time system and other computing systems is the fact that the correctness of a computational result depends on the time at which the result is produced. The presence of a deadline thus marks a real-time system. The reason for this is that the system interacts with its environment, often meaning the physical environment, thus making real-time requirements an integral part of many embedded systems. Examples for real-time systems include automotive and avionic systems, telecommunication systems, robotics, environmental monitoring and industrial manufacturing systems, which are the main focus of this thesis.

Figure 1.1 illustrates different types of real-time systems. They are classified by the value of a computational result after its expected deadline. In hard real-time systems a single missed deadline may have catastrophic consequences for the system itself, attached equipment, the environment or humans. The value of a late result is therefore negative and deadlines must be guaranteed. Examples for these kind of systems could be chemical process plants, safety critical systems in airplanes or automobiles or in general systems that incorporate sensing, actuating and control activities. In firm real-time systems deadline misses are not catastrophic. Late results are nevertheless useless and have no value. Examples would be video or audio decoding systems where frames may be dropped occasionally. In soft real-time systems a late result still has value but the performance of the system is degraded. Examples would be systems handling user input or graphical displays.



(a) Hard real-time (b) Firm real-time (c) soft real-time

Figure 1.1: A schematic visualization of the value of a late computational result under the hard, firm and soft real-time paradigms.

An often held misconception is that real-time simply means "fast", but execution speed only relates to the average response time of a system. Hard real-time systems require guarantees for the worst case and not for the average case. These timing requirements are determined by the system's environment and the system must always react within the time frame set by the environment to be fit for its purpose. Experience with real-time systems shows that the worst case scenario can, does and will happen (c.f. the anecdotes presented by Buttazzo in the introduction of his book [14]). Buttazzo defines the following basic properties that real-time systems should possess:

- **Timeliness:** Results must not only be correct but also be available for the consuming entity within a specific time frame.
- **Predictability:** The system must be analyzable in order to determine the consequences of any scheduling decision. For safety critical applications, timing requirements should be guaranteed before the system is put into operation.
- **Efficiency:** Most real-time systems are embedded devices with restricted storage, computational and energy resources. This must be taken into account when developing the system's hardware and software.
- **Robustness:** Real-time systems must be robust under peak-load conditions and should be designed to handle all anticipated load scenarios.
- **Fault tolerance:** Single failure of hardware or software should not cause the overall system to fail. Critical components of the system should therefore be designed to be fault tolerant.
- **Maintainability:** The architecture of a real-time system should be modular, ensuring that future system modifications are easy to perform.

To achieve these properties in distributed real-time systems, all components of the system need to be chosen accordingly. For example, fault tolerance in the area of networking may require the presence of physical duplication of the communication equipment. When writing software for these systems, developers are severely restricted in the primitives available to them. Dynamic memory allocation at run time might not be possible because this could introduce unbounded delays. Similarly, unbounded recursion or unbounded loops are often not allowable in real-time programs [14].

1.3 Goals and Contributions

The overall goal of this thesis was the development of an engineering approach for distributed hard real-time systems, with a special focus on reconfigurable manufacturing systems. By following principles of distributed data processing systems the approach should reduce development time, effort and cost. The design of rtSOA was based on the following desired system properties while addressing the general properties for hard real-time systems stated at the end of Section 1.2:

Goal 1 rtSOA systems are adaptable at run time with minimal downtime.

Goal 2 rtSOA systems are hard real-time systems. They have deterministic and verifiable behavior.

Goal 3 The development of automation workflows with rtSOA is decoupled from timing and networking concerns.

Goal 4 Development with rtSOA is iterative, visual and has short feedback cycles.

Resulting from the stated set of goals, the contribution of this thesis are:

- To the best of our knowledge, rtSOA is the first approach that achieves predictable and deterministic execution plans by applying distributed data processing principles in a flexible and reconfigurable service-oriented architecture.
- We compare the use of domain specific heuristics with a state-of-the-art linear program solver for the generation of distributed real-time workflow execution plans. This analysis is performed over a benchmark data set with 1.2 million feasible problem instances.
- We propose two new domain specific heuristics that contribute a large number of unique execution plans. Unique means that no other heuristic contributed a feasible solution for the same test case.
- We propose several adaptions of existing heuristics to increase their effectiveness for the generation of rtSOA execution plans.
- We showcase the adaption of DDP principles for industrial real-time systems with a demonstrator that is close to a real industrial system.

1.4 Outline of the Thesis

The rest of this thesis is structured in the following way: Chapter 2 expands upon the general introduction to real-time systems given in Section 1.2 by outlining past and present developments in industrial automation from a hardware and software perspective. Chapter 3 introduces our approach, named rtSOA, for real-time service-oriented architectures from a distributed data processing perspective. It reconciles the flexibility and reconfigurability of a service-oriented architecture with the timing guarantees and determinism required to verify the correctness of hard real-time systems. rtSOA achieves these properties via cyclic execution plans for each device that participates in the execution of the automation workflow. These execution plans follow a stringent set of dataflow semantics, also defined in Chapter 3. Chapter 4 details how execution plans can be derived from an abstract representation of the workflow. The chapter includes an extensive evaluation of the alternatives. To provide confidence that a set of execution plans conforms to the required real-time properties engineers rely on verification or simulation of the overall system. Chapter 5 presents how these methods can be applied to rtSOA. In Chapter 6 we present a prototype for the overall rtSOA system that ranges from configuration of a workflow with a graphical user interface (GUI), over generation of execution plans with the methods presented in Chapter 4, to simulation, deployment, and execution on a real-world demonstration system. The thesis ends with a summary of our approach and results in Chapter 7 wherein we also point out further directions for research.

"Any customer can have a car painted any color that he wants so long as it is black."
- Henry Ford

CHAPTER 2

History and Trends in Industrial Automation

Parts of this chapter have been previously published in [64, 65, 67].

Products, markets and societies are experiencing an increasing pace of change which requires flexibility and changeability from the manufacturing industry to meet the demands of a global economy [62, 117]. Indeed, the technological and sociological conditions under which the industry has operated have constantly been changing for over 100 years. Figure 2.1 shows the changes in manufacturing paradigms in the automobile industry. In the middle and end of the 19th century the automobile (or in the beginning the carriage making) industry was following the paradigm of craft production, meaning that each product is essentially unique and hand crafted by skilled workers. In 1913 Henry Ford introduced the moving assembly line method in production of the famous Ford Model T. The assembly line ushered in the era of mass production which saw a drastic reduction in product variety while increasing the production numbers of each variant. This development in the United States' automobile industry reached its peak in 1955 [62]. From then on more product variants have been introduced to meet customers' different needs and desires. The 1980s mark the emergence of mass customization as a new paradigm in which customers are able to customize a product by choosing from different options for its individual characteristics, e.g.,different accessories in their car or different components of their laptop computers. Koren observes that the 2000s have marked another paradigm shift triggered by the increased interrelations of global value chains and a worldwide customer base [62]. The customization trend may be taken to its extreme by paradigms such as personalized production which offers fully individualized products to the customer. Another trend is regionalization of products, as manufacturers have to take the cultural and legal particularities of different countries and regions

into account. A certain make and model of a car will have different components if manufactured for the European market when compared to the Chinese market. Chryssolouris remarks:

> "It is increasingly evident that the era of mass production is being replaced by the era of market niches. The key to creating products that can meet the demands of a diversified customer base, is a short development cycle yielding low cost, high quality goods in sufficient quantity to meet demand. This makes flexibility an increasingly important attribute to manufacturing." [22]

Each of the manufacturing paradigms is best realized by a different manufacturing system. Craft production of one of a kind items is best realized with the most adaptable of all production systems: A skilled human worker with general purpose machine tools.

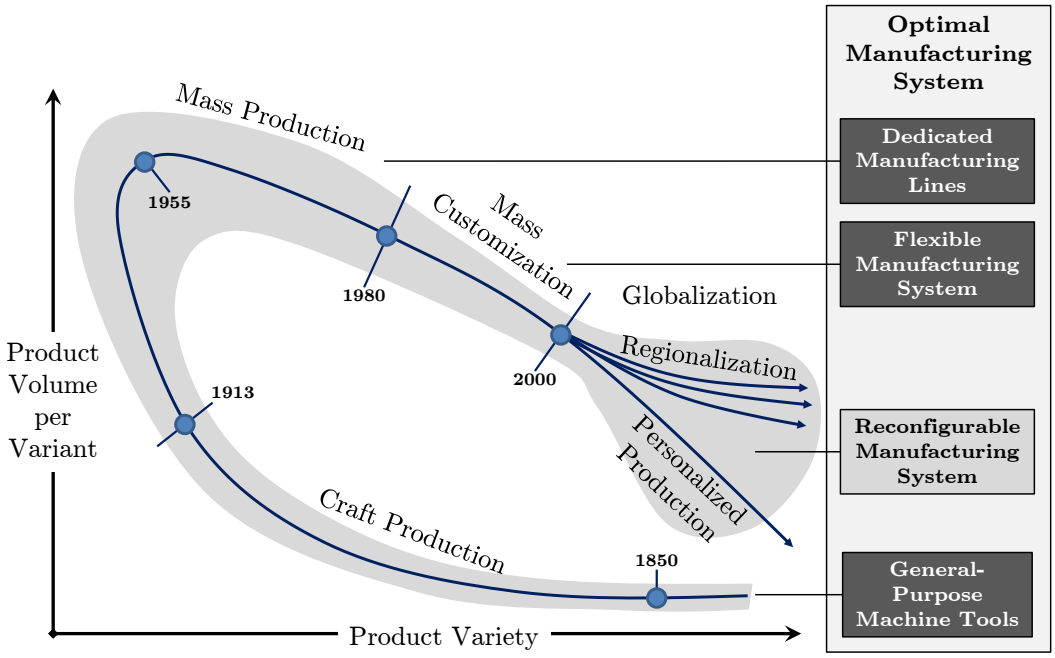


Figure 2.1: Changing manufacturing paradigms over time in the context of the automobile industry. [62]

Since the 1980s software controlled manufacturing systems have increasingly come to dominate the automation landscape, making software another key factor in the success or failure of manufacturing solutions. *Dedicated manufacturing systems* (DMS) excel at mass production of goods at high volume and quality while keeping product cost low through economies of scale. These centralized, monolithic and scan-based control systems are optimized for a given physical and network configuration allowing them to fully operate with high production rates after a long and costly setup period.

However, these dedicated lines can only be used to manufacture a narrow range of products and cannot be adapted in terms of production volume. The per-unit cost of goods manufactured on a DMS rises whenever it is not run at full capacity. There is also no quick way to increase the capacity of a DMS if demand exceeds supply. From a software perspective, the modularization and reuse of source code running on these systems is hindered by a tight coupling with their environment.

Flexible manufacturing systems (FMS) emerged as a way to handle changing customer demands within a given family of products or parts through rapid reconfiguration of their hardware and software [117]. As such, they are an enabling technology for the mass customization manufacturing paradigm [62]. Computerized numerical control (CNC) machines are an example for flexible production systems. These machines can be programmed to mill and bore work pieces from a solid block of material with high precision, granting them an unprecedented flexibility in the variety of products that they can manufacture, albeit at the cost of reduced throughput. FMSs are designed for a specific range of work pieces and often feature overprovisioning of tools or transport capabilities because retrofitting of new capabilities is expensive [117]. Section 2.1 outlines the state of the art in current industrial manufacturing from a software and networking perspective, thus showing the challenges to reconfigurability inherent in traditional architectures.

In recent years increasing uncertainty about the production volume of a product is accompanied by demand fluctuations over the product's life cycle [53, 103, 117]. To address these issues, the paradigm of the *reconfigurable manufacturing system* (RMS) has been postulated in research [36, 62, 117]. Koren is giving the following definition:

> "A reconfigurable manufacturing system [...] is designed for rapid adjustment of production capacity and functionality [...] by rearrangement or change of its components (hardware and software)." [62]

A RMS is modular and designed to evolve both in terms of capacity and capability over its lifetime. Apart from reconfigurable hardware, software reconfiguration has been identified as a key technology for RMS [36]. This need for reconfigurability and flexibility of the control software has led to research endeavors which aim at fulfilling these requirements through implementation of service-oriented architectures (SOAs) [106] [23] [46], which encapsulate hardware behavior and software capabilities in services which can be combined in a modular way. Software capabilities also play a large role in the digitalization of the manufacturing industry. The German initiative *Industrie 4.0* envisions a landscape of smart factories which are fully integrated with the manufacturer's other plants, its suppliers and its customers [53]. These smart factories also offer a detailed view of their current state to enterprise systems concerned with production planning and logistics. SOAs are likely to play a central role in realizing the capabilities envisioned under the Industrie 4.0 term [97]. In Section 2.2 we therefore give an overview of the role of service-oriented architectures in future manufacturing systems with a special focus on the Industrie 4.0 vision.

2.1 State of the Art in Industrial Manufacturing

Today's manufacturing plants are usually designed following the hierarchical architecture described in the IEC 62264 standard [5] and visualized in Figure 2.2. Therein, sensors and actuators on level 1 are controlled by a control system on level 2. Communication between field devices and controllers is strictly hierarchical and follows a polling model or a cyclic publish-subscribe relation. Communication between controllers on level 2, or higher levels, may be peer-to-peer. Levels 3 and 4 are the domains of operations control and enterprise planning systems, respectively. Sensors and actuators are usually controlled by *programmable logic controllers* (PLCs), also following a cyclic model. At the beginning of each *scan cycle* an input scan is performed which obtains readings from all connected sensors. Based on these updated values, the PLC performs its logic computations, updates all outgoing communication values and sends commands to the connected actuators.

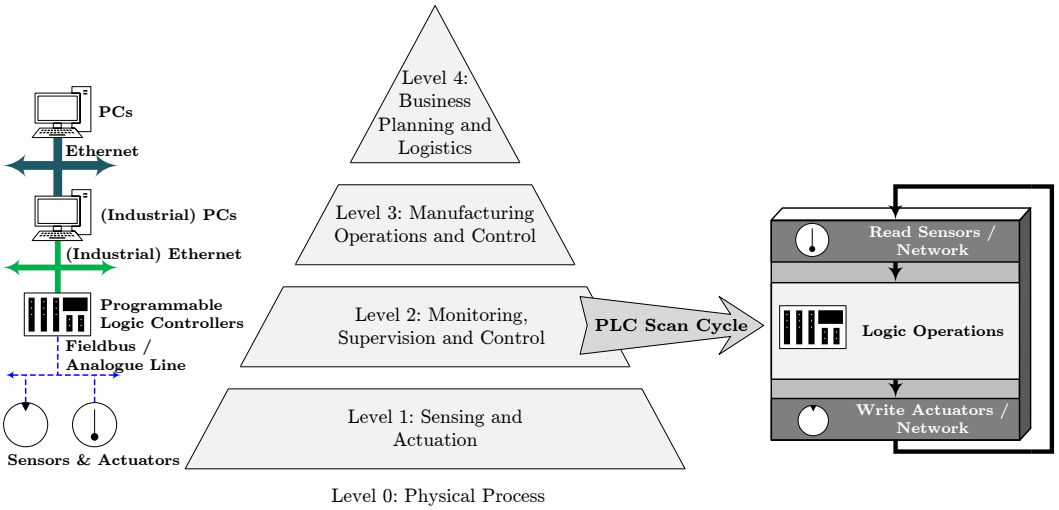


Figure 2.2: Traditional Automation Pyramid with passively queried sensors and actuators. Control programs are realized through programmable logic controllers (PLCs) following a processing architecture called a scan cycle.

IEC 61131-3 is the prevalent standard for PLC programming. This standard specifies the function block diagram (FBD) and structured text (ST) languages which are higher level and, from a software engineering perspective, should have higher productivity than the ladder diagram (LD) and instruction list (IL) languages included in the same standard. Structured text is based on, and similar to, the programming language Pascal. It can be used to define function blocks that can then be used in function block diagrams. FBDs are a graphical representation of programs consisting of (elementary) function blocks such as add, compare, logic functions, and FBs defined with structured text. Instruction list programs resemble code written in assembly. Lastly, ladder diagrams are a program representation form that resembles a notation form for electrical circuits. It is thus focused on logic relations. Although recent industry efforts target increased reusability of code blocks, the control soft-

ware is often rewritten from scratch when integrating new devices [123]. Software architectures of higher level systems mostly follow object-oriented principles with local method calls or direct data access. Services may be available, but not necessarily accessible for remote applications [5].

Networking and data acquisition between level 1 and level 2 devices are strictly hierarchical. Devices may be connected to PLCs either with analogue connections or, increasingly, over field buses or even Ethernet based solutions [5]. In most cases the communication is polling-based although cyclic publish-subscribe alternatives exist [5]. Because all input / output operations of a PLC are performed during each scan cycle, the tightest timing requirement determines the available runtime for the whole scan cycle. Additionally, the scan cycle is tightly coupled with the network cycle. Controllers on level 2 of the automation pyramid are usually able to communicate in a peer-to-peer fashion but messages are also often transmitted in a cyclic fashion. Alarms are special high-priority events which are always transmitted in an event-based fashion [5], other data may also be transmitted in an event-based manner instead of a cyclic manner. Networking between controllers also requires time determinism which is guaranteed by the utilized network protocols.

These traditional control applications offer a high degree of determinism and contribute to the high throughput of today's manufacturing systems. However, their setup and installation cost contribute up to one third of their total lifecycle costs [47]. Manufacturing strategies based on reconfigurable manufacturing systems are therefore hard to implement with the traditional automation architecture. More flexible hardware and software architectures which offer extensive interoperability and quick reconfiguration are thus needed to react to today's changing market demands. The following section outlines modern approaches to achieve these requirements.

2.2 Emerging Technologies

The key differences between traditional and future manufacturing systems can be summarized by two factors from a hardware perspective: Increased networking, based on a more unified protocol and hardware landscape, and ubiquitous computation resources leading to smart devices and even smart products or work pieces. These advances in hardware to support manufacturing are already evident. Industrial Ethernet is gaining traction [28] while smarter embedded devices are performing an increasing number of orthogonal tasks, for example wireless sensor networks (WSNs) monitoring machine health. Recent developments include the roll out of smart field devices, like Ethernet-equipped sensors and actuators. In the automotive and avionic industries a different process with similar results can be observed. Subsystems, that were previously separated, are consolidated through a single real-time communication network to reduce weight and costs. In both domains the result is a hardware architecture of multiple networked processing units.

These developments lead to a more interconnected and heterogeneous automation landscape as depicted in Figure 2.3 [7, 57, 112]. *Smart devices*, consisting of sensors and actuators with direct networking and computation capabilities, are blurring the line between data acquisition and processing devices. Traditional hardware and software architectures, together with traditional approaches to software development for these systems, are not capable of addressing the challenges brought on by the heterogeneous automation landscape shown in Figure 2.3. Effectively leveraging the new capabilities offered by these interconnected systems is the topic of numerous ongoing and completed research projects. Research concerning service-oriented architectures in the industrial domain is addressing the need for interoperability between devices near the physical process and enterprise applications, which are placed several layers apart in the classical automation pyramid, thus realizing a vertical integration. Industry and research have reacted by developing the IEC 61499 standard for distributed automation systems which we introduce briefly in Section 2.2.1. SOAs are also poised to simplify the composition of multiple devices so that they may cooperate to realize an automation workflow, implementing a horizontal integration of devices. Section 2.2.2 is giving an overview of research endeavors covering the topic of SOAs for industrial automation.

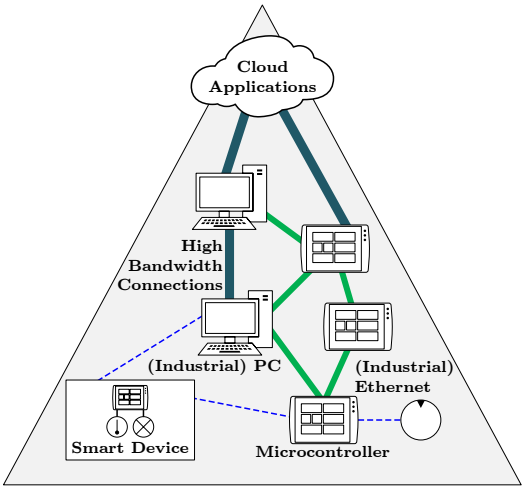


Figure 2.3: The emerging automation landscape is characterized by an increasing heterogeneity and interconnectedness between devices.

The term Industrie 4.0 encompasses horizontal and vertical integration, not of individual devices in a factory but of the factory, then termed a smart factory [7], with its surroundings. Horizontal integration in the Industrie 4.0 world describes the integration of the smart factory in the value chain of the enterprise, meaning an integration with systems used in different stages of the manufacturing and business planning process, whether in-house or with external business partners. Vertical integration refers to the integration of systems on different levels of the automation process with higher level systems [53], similar to the usage of the term when applied to SOAs. Section 2.2.3 will provide further introduction to the concept of Industrie 4.0.

2.2.1 Distributed Systems with the IEC 61499 Standard

The IEC 61499 standard has been proposed as a way to address distributed automation in industrial systems. Its main concept is the hierarchic encapsulation of functionality in *function blocks* (FBs) which offer a high level of abstraction and promise greater code reuse. An abstract example of a FB is shown in Figure 2.4. This way, "the IEC 61499 architecture exploits the familiarity among control engineers accustomed to a block-diagram way of thinking" [114]. Internally, the execution of an FB is driven by an execution control chart (ECC) state machine [122]. Multiple instances of the same function block type may exist in the same block diagram. Input and output ports are separated into event ports and (typed) data ports as shown in Figure 2.4. Signals on event ports are either present or absent, they carry no additional data but may be associated with data ports. New data is loaded and emitted from data ports whenever an associated event is present [122]. Abstract, more high-level function blocks often interact with the underlying hardware through service interface function blocks (SIFBs), which are conceptually similar to device drivers or hardware abstraction layers in traditional embedded systems development [122]. SIFBs provide services such as sending or receiving data over the network or access to IO-pins. Algorithmic FBs may thus easily interface with different hardware.

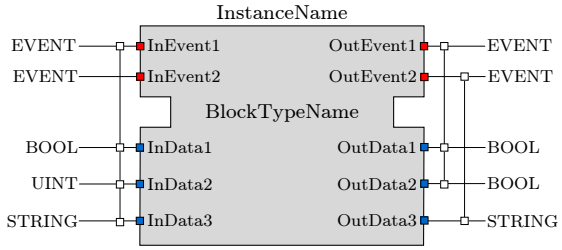


Figure 2.4: A function block instance in the IEC 61499 standard

Function blocks can be nested arbitrarily and allow the combination of different diagram types, among them IEC 61131-3 languages other than FBDs. IEC 61499 models also include specifications of the automation workflow's infrastructure, i.e.,the associated devices and the network structure. This results in an executable specification of the distributed automation system [114]. The execution semantics of IEC 61499 blocks are event-driven, they are activated whenever an event reaches one of their event inputs. It is assumed that the FB cannot be reactivated before it has finished its execution, however the standard does not specify how this should be enforced and which measures should be taken if events arrive faster than the FB is executed [114]. Encapsulation of data into FBs is another principle from software engineering that has been applied in IEC 61499. Data encapsulation was often not present in traditional PLC programming and communication between different modules often happened via shared variables [114]. The principles of encapsulation of data and functionality, together with a distributed execution of function blocks, is conceptually consistent with service-oriented architectures [27] which we discuss in detail in the next section.

2.2.2 The Role of Service-Oriented Architectures

There are many definitions of the term service-oriented architecture. The author prefers the following definitions from the OASIS "Reference Model for Service Oriented Architecture":

> "[Service-oriented architectures are] a means of organizing solutions that promotes reuse, growth and interoperability." [86]

and from the SIRENA EU project on SOAs in industrial manufacturing:

> "A service-oriented architecture [...] is a set of architectural tenets for building autonomous yet interoperable systems." [47]

A service-oriented architecture in itself is therefore not a concrete implementation or a set of technologies, but rather an architectural style for implementing systems that achieve the conflicting goals of autonomy and interoperability. In many cases SOAs have been realized with web service standards, as is also the case in industry driven initiatives (Section 2.2.2.1) and collaborative research projects sponsored by the European Union (Section 2.2.2.2). However, this introduction will not focus on technology details of web services but instead outline the general characteristics of SOAs in the industrial context. The context in which a SOA is deployed has an influence on its characteristics. SOA as an architectural style has originally emerged as a way to organize business processes on an enterprise IT level. Owing to the particularities of the manufacturing domain, some of the original tenets of a service-oriented architecture need to be adapted for use in industrial automation scenarios [88].

The basic building block of a SOA is a *service*. A service encapsulates functionality, which may be either software functionality or mechatronic functionality. To outside entities, like service orchestrators or other services, the service is fully described via its service interface. Services with the same interface should be seamlessly exchangeable in classical enterprise-IT SOAs. This may not be the case in the automation domain, as the location of a service plays a role in the semantics of its execution [88]. A conveyor belt may offer the same service interface as the neighboring belt, but the correct execution of the automation task will depend on choosing the component with the right physical location for the given context. The objective of a SOA in industrial automation is therefore not distributed computation but instead the execution of a technical process [88]. A service may therefore have certain requirements for the hardware environment where it is executed, meaning that it cannot be placed freely on any device. This is already implicit in the fact that a service can encapsulate mechatronic behavior. A service that offers temperature measurements will need access to a temperature sensor. However, this does not mean that services should be defined technology centric. Instead each service should be rather coarse grained and oriented around business functions [47], meaning it should encapsulate functionality that is already useful on its own without the need to be composited

with other services. When services are composited in a sequence we are referring to this composition as a *workflow*. The execution of a workflow may be controlled centrally by a service orchestrator invoking the individual services, receiving their responses and then using these responses in invocations of succeeding services, thus implementing a *service orchestration*. Workflows may also be executed decentrally with each service being aware of its successor in the workflow. This mode of workflow execution is called a *service choreography*.

Service-oriented architectures are designed to address the weaknesses, such as inflexibility and high setup costs, of traditional automation architectures. The strengths of SOA lie in its flexibility and adaptability. By encapsulating basic functionality in interchangeable services the programming of automation systems can be lifted to a higher level [47, 88]. Instead of dealing with individual IO-signals, new control flows can be created by rearranging pre-existing services and rerouting the data flow between them. Our prototype implementation in Chapter 6 demonstrates this principle with a real-world system. Ideally, the interface descriptions of services are standardized and physically compatible systems would offer services that are transparently interchangeable with each other, thus allowing fully vendor independent planning and programming phases [48]. To leverage the full strength of SOAs, the hardware should therefore also be designed in a modular fashion and allow quick reconfiguration on the physical side. Reconfigurable manufacturing systems are thus an ideal fit for service-oriented architectures, and vice versa [97, 112]. In the transition phase legacy automation systems can be integrated in higher level SOAs via service facades, allowing a gentle transformation [47].

In SOAs messages between services and devices are sent in a push-based manner making more efficient use of the limited communication resources than the traditional scan or pull-based communication patterns [47]. Extensive research has been undertaken to push the performance envelope of SOA (or more specifically, web-service) technology on constrained embedded devices [123]. These endeavors have seen SOA solutions developed for lower and lower layers of the automation pyramid [46]. It is, however, difficult to verify the emergent properties of distributed event-based systems. Implementations with centralized control are therefore likely to be favored in the near future [58]. Centralized control is realized through a service orchestration which is executed by an orchestration engine that invokes each individual service in a workflow [48]. While such an orchestration represents an improvement over the traditional model, the full impact of the SOA paradigm could be realized through service choreographies providing global cooperation without central coordination [104]. In Chapter 3 we show how decentralized execution plans can be used to realize service choreographies for cyclic control tasks with real-time constraints.

The clear benefits offered by service-oriented architectures have been recognized by the industry and research communities. The remainder of this section covers the current state of SOAs in industry (Section 2.2.2.1) before presenting the research results realized in several framework programs of the European Union (Section 2.2.2.2).

2.2.2.1 Industry Efforts

The automation community has long identified the need for self-describing devices and easy discovery and configuration of automation devices. For this purpose numerous standards and vendor specific implementations have been developed. We have chosen relevant examples based on their importance in industry and research as well as their relation to service-oriented principles. The two most prominent specifications concerning service orientation for industrial devices are the Devices Profile for Web Services (DPWS) [87] and the Object Linking and Embedding for Process Control Unified Architecture (OPC UA) [89]. Both are conceptually similar: They implement a SOA through web services and rely on built-in base services for discovery and service reservation. In both cases, SOAP over HTTP is the standard message binding and messages are either encoded with XML or in a binary format for increased performance. In the following we will look closer at their similarities and differences.

OPC UA

OPC UA is a service-oriented version of the original OPC architecture and its main mission is still to connect industrial devices to control and supervision applications [16]. It is therefore not directly aimed at the communication between the devices themselves and follows a client-server architecture. Connecting devices to monitoring and human-machine interaction (HMI) interfaces may entail a large amount of data to be transferred, as one of the roles a OPC UA client may occupy is that of a data logger. For this reason OPC UA also specifies a binding to optimized native binary protocols which may reduce network traffic and processing requirements. Typically lower-level devices take the role of OPC UA servers, as they make their information available to higher level devices [16]. OPC UA servers implement a predefined set of (web)services, which include discovery services, services for reading and writing of attributes or remote invocation of methods, as well as services for subscription to data or negotiation of secure communication channels between client and server [75]. Semantics of data exchanged via OPC UA are defined via the OPC UA meta model.

The client-server architecture as well as the predefined service sets of OPC UA make it well suited for its intended task of data export from lower layers to higher layers of the automation pyramid, especially when taking the existing model for data semantics into account. However, OPC UA is not suited for implementation of true peer-to-peer general purpose SOAs. The client-server architecture limits the communication patterns of OPC UA devices and the predefined service sets do not allow for definition of additional services that can encapsulate device behavior [16].

DPWS

DPWS is a web service middleware and profile that aims to constrain the WS-* set of standards to make them suitable for embedded use. It is aimed directly at the devices performing the automation task and therefore supports a peer-to-peer as well as a client-server architecture [123]. DPWS defines two basic elements: the

individual devices and the services hosted upon the devices. This corresponds to our earlier observation that descriptions of services in the industrial context need to be considered together with additional information about the device which provides the service. Similar to OPC UA, DPWS also specifies a set of basic services for device discovery, publish / subscribe services for events and metadata exchange services. Additionally, DPWS allows the definition of user specified hosted services which encapsulate device functionality [16]. DPWS does not specify a meta model like OPC UA, or a similar framework for providing semantics. Semantic descriptions for DPWS services are an area of active research [42]. Efficient binary representation of messages is not part of DPWS, but the profile is open enough to allow for usage of data reduction techniques such as the Efficient XML Interchange (EXI) standard [115]. Käbisch et al. [52] showed how EXI can be applied in embedded networks.

DPWS closely fits our description of characteristics for service-oriented architectures in industrial automation. DPWS is directly aimed at constrained devices with only a few kB of RAM and ROM [123], thus allowing an integration of individual devices in the overall SOA. User defined services enable the encapsulation of the device's mechatronic capabilities. The discovery and metadata services provided by DPWS, along with the defined description format for the device itself as well as the hosted services, are a necessary component in an interoperable software architecture. The focus on peer-to-peer messaging between devices allows using DPWS services as part of control algorithms. Thus, we consider DPWS to be an enabling technology for service oriented architectures in the industrial context. The development of DPWS, as well as its use in service-oriented architectures, has been the subject of several EU research projects which we will consider next.

2.2.2.2 Research Projects on Industrial SOAs

The concept of service-oriented architectures emerged during the early 2000s as a way to realize interoperable and cooperating systems in the e-business domain [13]. Research into service-oriented architectures for industrial automation started in 2003 with the "Service Infrastructure for Real-time Embedded Networked Applications" (SIRENA) project which was financed with grants from the European Union. Major progress into service-oriented manufacturing architectures has since been made in several additional EU-projects. Because much of the academic research is conducted in this context this section will focus on work performed as part of the initiatives outlined in Figure 2.5. The projects depicted therein are not following the traditional hierarchical approach, the automation pyramid is only used in this figure to convey an intuition about the relative target area of the differing projects.

The need for modular, interoperable and reconfigurable manufacturing systems was already recognized before the start of the SIRENA project or the advent of service-oriented architectures. Earlier research was focused on agent-based architectures [100], which share some similarities with SOAs. Agents encapsulate functionality and may interact with each other. In contrast to services, an agent is not invoked

but self activating. Agents are therefore often used to implement the concept of *holonic manufacturing systems* which are composed of self controlled modules, called *holons*. A holon is "an autonomous and cooperative building block of a manufacturing system" [100]. The concept of the holon can also be applied to modules in a reconfigurable or modular manufacturing system which offer appropriate software services. The remaining conceptual difference between service-oriented and agent-oriented systems lies in their activation semantics: while agents are self-activating and self-regulating, services are often activated by a central orchestrator. Another possibility for service invocation are distributed service choreographies in which services are themselves invoking their successors in a workflow. With choreographies the line between the agent-oriented and service-oriented architectural styles is blurring. For the remainder of the thesis we will continue to use the service-oriented terminology.

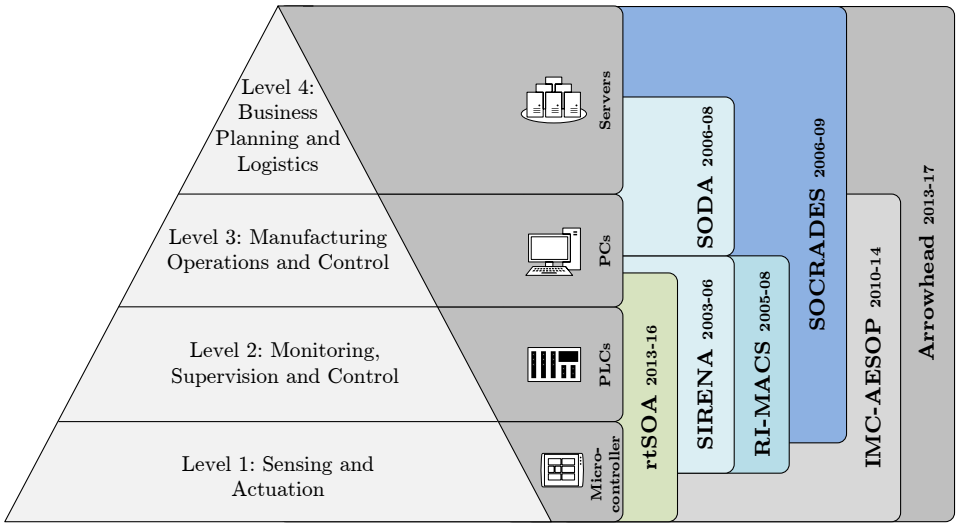


Figure 2.5: An overview of the rtSOA research project described in this thesis and several research projects into SOAs for industrial manufacturing financed by grants from the European Union

SIRENA 2003-2006

The SIRENA project was started in 2003 to examine the use of SOAs for interconnecting embedded devices in the industrial, automotive, telecommunication and home automation domain [11]. Initial development efforts still used the Universal Plug and Play (UPnP) technology, as DPWS was still being standardized. After the completion of the standardization, DPWS became the intended target technology for the project [11]. One of the major outcomes of the SIRENA effort was the first embedded DPWS-stack, which had a memory footprint of under 200kB [47]. Several tools to simplify the development of DPWS based systems, such as a generator for marshalling and demarshalling of SOAP-XML messages, have also been developed during the project. Based on this implementation an industrial demonstrator [48] was developed which consisted of a dose maker filling granules into a container.

SODA 2006-2008

The SIRENA follow-up project "Service Oriented Device and Delivery Architecture" (SODA) extended the SIRENA framework by providing a toolkit for manageability, orchestration and security [79]. Apart from furthering the integration of DPWS services with high-level business processes, SODA also focused on improving the performance and serviceability of the DPWS infrastructure [79]. Together, these two projects proved the feasibility of DPWS-based architectures on embedded devices.

RI-MACS 2005-2008

The "Radically Innovative Mechatronics and Advanced Control Systems" (RI-MACS) project is the first effort that takes real-time requirements into account, which could not be fulfilled by DPWS or other web service technologies at the time [19, 26]. DPWS is used for interaction with higher level systems which have no, or only relaxed, timing requirements. More time critical tasks are implemented through a separate communication stack which is build directly upon TCP/IP or UDP/IP instead of the HTTP / SOAP / DPWS protocols used by general services. A custom protocol stack may also be used to interact with legacy or hard real-time systems.

SOCRADES 2006-2009

The "Service Oriented Cross-Layer Infrastructure for Distributed Smart Embedded Devices" (SOCRADES) project built on the SIRENA and SODA results to further the vertical cross-layer integration between shop floor and enterprise systems [102]. The project also focused on the questions of how legacy devices could be integrated into a service-oriented architecture and how the evolution of existing manufacturing systems and plants towards a SOA can be accomplished [55]. When executing a step-wise transformation to a SOA, it may not always be feasible to replace legacy devices with SOA-capable devices. In these cases gateways or service mediators can be used. Gateways function as a service-facade for an individual device whereas service mediators offer a higher level view focused on functionality instead of representing an individual device. Therefore, service mediators may have internal data aggregation or composition functionality [55].

SOCRADES achieved an initial integration of constrained devices on the lower levels of the automation pyramid, but identified orchestration, determinism of the SOA run-time behavior, decentralization and effective reconfiguration as open research issues [106]. The project also focused on the engineering aspects of networks of smart devices, leading to new tools and technologies for modeling, design, implementation and operation of these new kinds of distributed embedded systems [17]. One of the major take-away messages from the project is that the effort required to add web service capabilities increases for devices further down the automation hierarchy. However, the benefits of having a service enabled device are also increasing when moving further down in the traditional automation pyramid [55].

AESOP 2010-2014

The SOCRADES follow-up project "Architecture for Service Oriented Process - Monitoring and -Control" (AESOP) [23] investigated the feasibility and limits of using a SOA-based approach inside control loops [46]. By implementing several prototypes, the project closely investigated the performance implications of using web services for the concurrent control of several thousand devices, thus addressing some of the open issues raised by SOCRADES. One real-world showcase included the migration of a legacy plant lubrication system to a service-oriented architecture [84], addressing issues such as SOA on low-level devices and in closed-loop control [24]. AESOP achieved cross-layer collaboration of services and devices mainly through use of an orchestration engine, which constitutes an event based model with central control through an orchestrator. Other research within the project highlighted the benefits of decentralized execution plans for service choreographies over using centralized service orchestrations for automation tasks. Starke et al. implemented an event-based choreography engine for service-oriented automation and observed a higher degree of performance and reactivity in an choreography based approach when compared to an approach based on orchestration [104].

Arrowhead 2013-2017

The Arrowhead project consolidates previous research into a comprehensive framework that supports the development, deployment and operation of cooperative systems in a SOA [110]. Its main contributions are design guidelines and patterns for faster engineering; documentation guidelines and templates on the levels of individual services, systems, and systems-of-systems; and a software framework build around core services which support the interaction of application specific services across different base technologies. Among the provided core set are services for discovery, authorization, orchestration and status monitoring. They address the main technical questions examined in the Arrowhead project: How can systems advertise their services and discover remote services? How can systems ensure that only authorized entities may consume its services? How can systems-of-systems be orchestrated?

Comparison with our approach

SIRENA, SODA and SOCRADES achieved the vertical integration of industrial control devices with higher-level enterprise systems. Arrowhead mostly focused on engineering related questions and orchestration of systems-of-systems. Since our work on rtSOA is focused on providing hard real-time guarantees on the underlying device level, RI-MACS, and especially AESOP, can be considered conceptually similar projects. AESOP showed the feasibility of integrating embedded devices in a control loop through a SOA. In our opinion, message exchange in a control loop is possible by leveraging one of the protocol stacks investigated by AESOP [46]. Additionally, work performed in the context of the AESOP project has already pointed out that a distributed choreography approach to SOA is preferable to the classical orchestration-based approach [104]. Our work focuses on the planning required to achieve performant, hard real-time execution plans for devices in a tight control loop.

Our proposal for a real-time service-oriented architecture (rtSOA) reconciles the SOA paradigm with predictable execution semantics while enabling decentralized coordination of devices. rtSOA implements decentralized execution plans for service choreographies and focuses on the temporal aspects required to offer deterministic, predictable and verifiable real-time properties, thus addressing the research questions raised by SOCRADES. The AESOP design can be united with the rtSOA approach to achieve both event-based flexibility and cyclic determinism where needed: A cyclic sub-system, scheduled with rtSOA, can periodically trigger events which are then processed by an event-based architecture on higher layers.

2.2.3 Industrie 4.0

Industrie 4.0 is heralded as the fourth industrial revolution, with the three previous revolutions being the mechanization of industry beginning in the 1780s, the electrification starting in the 1870s and finally the introduction of computer control in the 1970s. The unique aspect of Industrie 4.0 is seen in the advent of *Cyber-Physical Systems* (CPS) [7,53,112]. Cyber-Physical Systems are computer systems which register properties from the physical world via sensors and are often also able to affect these properties via actuators. So far, this is identical to the definition of an embedded system. The defining properties of cyber-physical systems are the aspects of networking, interconnectedness and advanced decision making and reasoning [7, 53, 73].

In the United States a different classification and terminology is being used: The *Industrial Internet* is seen as the third wave of innovation and productivity increase after the industrial revolution and the internet revolution. The Industrial Internet is characterized as the convergence of industrial systems with sensing and computing power, as well as networking capabilities [37]. These technical enablers are combined with an analytical and predictive evaluation of the gathered data. This description is similar to the definition of a CPS in the Industrie 4.0 context.

The Industrie 4.0 vision also calls for a horizontal and vertical integration of smart factories within a company and even with other companies along its value chain [53]. The definition of vertical integration is identical to the meaning of the term in the SOA context as it implies an interconnection between processes on the lowest layer of the production floor with enterprise planning modules. Leveraging this data effectively through analytical and predictive evaluation is the area of big-data and complex event processing systems [7, 97]. Horizontal integration encompasses more than interconnection of devices within a factory or within a single enterprise. In the Industrie 4.0 model, there should be a connection between the manufacturing systems of a company, its logistics process, its suppliers, and customers. The horizontal and vertical integration is envisioned through a large scale application of service-oriented architectures, called the *Internet of Services* [7, 53]. Service-orientation of individual machines and devices is an important part of this architecture [97]. Finally, end-to-end engineering in the context of Industrie 4.0 refers to the use of model-based techniques in an integrated engineering environment for the development and optimization of products and the associated manufacturing system [53].

To conclude, Industrie 4.0 and the Industrial Internet describe the concept of leveraging Cyber-Physical Systems for increased flexibility, awareness and data-driven decision support in several industries with the manufacturing industry at the forefront. Service-oriented architectures and model-based design, simulation and visualization are integral parts of this vision [7, 37, 53]. Efforts in developing a reference architecture for Industrie 4.0 name service-oriented architectures as a key component and enabler [111]. The SOA should extend down to the manufacturing process where suitable orchestration techniques are an important building block [53]. Our work inside the rtSOA project focuses at providing engineers with an easy and intuitive way to specify service compositions in a declarative way influenced by distributed data processing principles. rtSOA users can quickly evaluate the implications of their design decisions through simulation, verification and visualization of the generated execution plans. These decentralized plans can be deployed automatically to devices in a service-oriented architecture. The next chapter will introduce the rtSOA architecture and its execution semantics before we detail and evaluate the heuristics-based rtSOA planning process in Chapter 4.2. Additional challenges arise from the vision of hosting some parts of the automation logic in the cloud [56], especially when considering the lack of fault tolerance in current publish / subscribe middleware facing complex failures scenarios such as byzantine failures or selfish node behavior [80].

"I love deadlines. I love the whooshing noise they make as they go by."
- Douglas Adams

CHAPTER 3

The rtSOA Approach

Parts of this chapter have been previously published in [64–67].

This chapter introduces our approach, named rtSOA, for service-oriented architectures with real-time execution plans. It reconciles the flexibility and reconfigurability of a SOA with the timing guarantees and determinism required for hard real-time systems. The development of systems with rtSOA follows principles of distributed data processing (DDP) systems and hides much of the networking and timing issues from the engineer. The design goals and principles of the rtSOA architecture are explained in Section 3.1. Based on these principles, and the assumptions detailed in Section 3.2, Section 3.3 introduces the basic rtSOA architecture. Coherent semantics are required to generate execution plans for any distributed data processing system. Section 3.4 is therefore giving an in-depth analysis of the dataflow-driven execution semantics of rtSOA. We conclude the chapter by discussing alternative design decisions for rtSOA and placing it in context with related work in Section 3.5.

3.1 Design Goals and Principles

The target domain of rtSOA are distributed hard real-time systems for industrial automation. The changing realities of the manufacturing world necessitate a focus on flexibility, reconfiguration and reuse (c.f. Chapter 2). rtSOA is thus focused on applying DDP principles to modular and reconfigurable manufacturing systems in the smart factory ecosystem envisioned with Industrie 4.0. This section presents the design goals for rtSOA and the principles behind the rtSOA design philosophy. The term rtSOA refers to a service-oriented architecture for embedded real-time systems based on DDP principles. We distinguish between the rtSOA architecture and the following pieces of software:

- **rtSOA planner:** The planner is the algorithmic core that generates rtSOA execution plans from abstract workflow descriptions. Its inputs are one or multiple workflows modeled as a task graph and a representation of the target system's infrastructure, i.e.,the available machines and the network configuration. The planner's output is a static schedule for each of the machines. These schedules constitute the rtSOA execution plan.
- **rtSOA runtime:** The rtSOA runtime is the software running on the individual devices partaking in the execution plan. The runtime is tasked with executing the schedule generated by the rtSOA planner by invoking the individual service instances and routing the messages between them.
- **rtSOA user interface (GUI):** The rtSOA GUI is an engineering tool used to composite individual services into workflows by arranging the services in a graph through specification of the messages passed from one service instance to another. The feasibility of a workflow on a given infrastructure can be determined by the rtSOA planner, which can be triggered from the GUI. The GUI can also be used to deploy a new execution plan to the machines.

The following chapters will focus on the rtSOA planner, as it is the main contribution of this thesis. The GUI and runtime have been implemented prototypicaly and are presented in Chapter 6. The design of rtSOA was based on the following desired properties:

Goal 1 rtSOA systems are adaptable at run time with minimal downtime.

Goal 2 rtSOA systems are hard real-time systems. They have deterministic and verifiable behavior.

Goal 3 The development of automation workflows with rtSOA is decoupled from timing and networking concerns.

Goal 4 Development with rtSOA is iterative, visual and has short feedback cycles.

These four design goals serve as the basis and rationale for the high-level design decisions which guide the development of rtSOA. These design principles are:

Distributed Data Processing Principles

Manually dealing with the complex interrelations of large distributed systems is a time-consuming and error prone activity. By applying principles of distributed data processing systems, such as the automatic generation of execution plans that handle the timing issues arising from network communication and data dependencies, rtSOA aims to reduce development effort and time. It thereby enables a fast change of system behavior (Goal 1) while providing verifiable real-time guarantees (Goal 2). Decoupling system development from the concrete network situation, as stated in Goal 3, is a common principle of DDP systems.

Service-Oriented Architecture

Section 2.2.2 has already outlined the role of service-oriented architectures in future manufacturing systems. The concept of a service encapsulates mechatronic functions of the individual devices which may then be composited into larger workflows, thus enabling easy reuse of existing functionality. Another important property of services is their loose coupling at run-time which enables modification of the system's behavior by changing the service composition. The SOA architectural style is thus an excellent conceptual match for the flexibility demanded in Goal 1.

Dataflow Driven System Behavior

In a SOA, the services are more permanent than the ephemeral connections between them. New services offering additional software functionality may be installed on the devices, but their basic mechatronic services change rarely, if ever, because this would require the installation of new hardware. Changing a system's behavior, as mandated by Goal 1, is thus realized by changing the flow of data between services. The dataflow driven system behavior applies DDP principles and accomplishes the decoupling of functionality from low-level timing concerns (Goal 3) by raising the level of abstraction for the implementation of automation workflows to reasoning over dataflow between mostly invariable services. The semantics described in Section 3.4 offer the deterministic behavior and verifiable real-time properties required by Goal 2.

Decentralized Execution

Traditional approaches to service orchestration or service choreographies pose a challenge in the context of real-time environments. Decentralized service choreographies have been shown to possess superior runtime performance [104], which is a necessity for fulfilling the performance requirements raised in Goal 2. Decentralized execution with peer-to-peer messaging can also reduce the amount of messages sent.

Schedule-Based Execution Plans

Hard real-time systems often require formal verification or otherwise need to give certain performance guarantees. This is stated in the second part of Goal 2. The rtSOA approach therefore bases its execution plans upon static, cyclic, non-preemptive schedules for each device, providing determinism and verifiability through these schedules. Validation of rtSOA schedules is discussed in detail in Chapter 5. Schedule based execution plans also fit well with the previous principle of decentralized execution and allow for lightweight runtime systems, which are the next design principle of rtSOA.

Lightweight Runtime

To realize the general reconfiguration of automation systems, as stated in Goal 1, rtSOA systems need to reach down to the lowest layers of the automation pyramid which encompasses sensors and actuators with limited computational resources. The possibility for a lightweight runtime environment suitable for embedded systems is a prerequisite of the real-time requirements stated in Goal 2 in combination with the flexibility mandated by Goal 1.

Model-Based, Interactive Engineering Environment

Model-Driven Engineering (MDE) describes an approach wherein an abstract model of a system is created and then systematically transformed to a concrete implementation [38]. This methodology promises the decoupling of functionality from timing and network issues, as stated in Goal 3, and raises the abstraction to a higher level, thus fulfilling Goal 4. It also fits well with the declarative approach for specifying system behavior that is often present in DDP systems. The rtSOA GUI is a prototypical implementation of a model-based engineering environment. Although rtSOA does not currently offer full end-to-end model driven engineering of all aspects of the automation system, the visual service composition offered by the GUI together with an interactive simulation of rtSOA execution plans already implements important aspects of a model-driven engineering approach. Currently missing from rtSOA are other MDE aspects such as models at different abstraction levels, e.g.,models of the behavior inside individual services, and support for automatic transformation between models, which may cumulate in the generation of runnable code. rtSOA is therefore only model-based in the context of service composition and choreographies.

Fast and Effective Heuristics

Short feedback cycles enabled, among other factors, by quick compile and build times are known to increase the efficiency of software development as well as the quality of the outcome [44]. Following the same reasoning, explorative and iterative development within the rtSOA framework would be severely hindered by answer times of multiples minutes or even hours. Similarly, the rtSOA planner and GUI can be viewed as an information system supporting the user in the decision whether a specified automation workflow can be executed on a given infrastructure. Fast feedback times are paramount in this scenario as well. This requirement is contained in Goal 4 and, to a lesser degree, in Goal 1. Our approach employs domain specific heuristics which constitute a proof-by-construction for the existence of a feasible execution plan. Exhaustive state space search methods may have unsuitably long response times (c.f. Chapter 4).

3.2 System Model and Assumptions

This section describes the system model and the assumptions on which rtSOA is based. The basic assumption behind rtSOA is the presence of services on each device participating in an rtSOA workflow. A service encapsulates (mechatronic) functionality and has well-defined input and output ports, described by its service definition. A *port* in this context can be seen as an interface which describes the type and size of data consumed by the service. A service can have multiple input and output ports. An output port may be connected to an arbitrary amount of input ports. However, there may not be any cyclic dependencies between service instances, i.e.,the graph formed by connections between instances' ports must be a directed acyclic graph (DAG). In addition to ports, service instances can have attributes which do not change over the instance lifetime. A *service instance* is the

concrete instantiation of an abstract service definition, similar to how an object is the concrete instantiation of a class in object-oriented programming. There may be multiple instances of the same service definition. The lifetime of a service instance is the time from the creation and scheduling of the instance to its destruction. Cyclic, static, non-preemptive schedules implement a service choreography which constitutes the rtSOA execution plan. Service instances are created on individual machines and must have concrete values for their worst case execution time (WCET) and their schedule time. Each machine follows a schedule. Multicore CPUs are not explicitly modeled but could be represented by pinning a separate schedule to each core. Timing side effects from parallel execution are outside of the scope of this thesis. Workflows are a composition of multiple service instances and a service choreography may comprise multiple workflows. Service instances may only communicate via their input and output ports. The invocation of a service is side-effect free, meaning that a service may not free or allocate memory, write outside of its private memory or otherwise influence the execution of other services apart from explicit messages to their input ports or via effects on the physical environment. Figure 3.1 illustrates the domain model of rtSOA in UML.

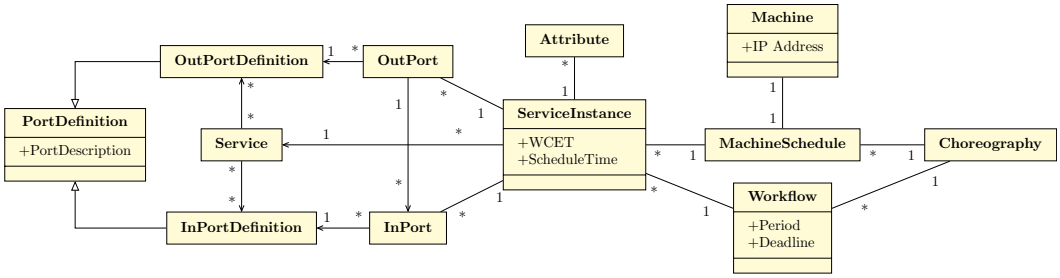


Figure 3.1: UML domain model for rtSOA

Because we need to provide real-time guarantees for the control loop in a distributed system, the network also needs to offer these guarantees. We therefore assume all devices in the control loop are connected by a real-time capable network with bounded message delays. The network also provides reliable message transfer in bounded time, the application layer therefore does not have to include acknowledgment mechanisms. The predominant message exchange mode in industrial control applications is cyclic; thus, we also assume a cyclic communication model. In this model, each network cycle is divided into a number of time slots that are assigned to a device, i.e.,TDMA. We do not assume a master-slave relationship on the system or network level. Each device can potentially send data to any other device in the network. The tight time-synchronization required for distributed real-time execution plans are also needed by the TDMA-network so there is no additional overhead. Communication between service instances only happens over pre-determined TDMA-slots. These slots may not be used otherwise, neither by other rtSOA service instances or by other tasks running on the same machine or in the same network. We assume broadcast messaging semantics, meaning that all devices in the same TDMA-network

receive the data sent inside the slot. Another assumption is that the devices in an rtSOA execution plan start the current iteration of the plan at the same point in time and that this instant coincides with the beginning of a TDMA-cycle. Figure 3.2 illustrates the resulting execution model as described in this section.

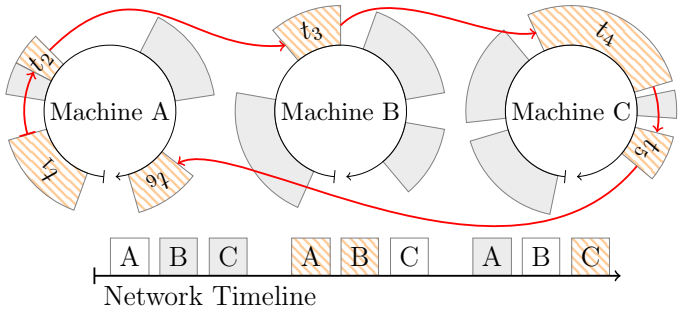


Figure 3.2: Devices cooperate in a distributed execution plan by locally executing cyclic schedules and communicating over a deterministic real-time TDMA network. Gaps in the machine time-line and white slots on the network time line represent unused network and computational resources. The hatched areas indicate resources used by an example workflow, arrows represent data dependencies.

3.3 The rtSOA Architecture

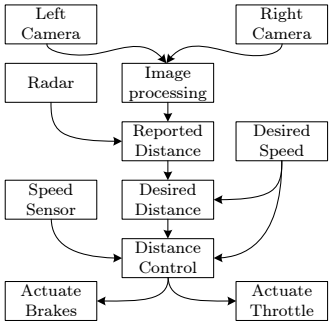
Our proposal for a real-time service-oriented architecture (rtSOA) applies principles of distributed data processing to reconcile the flexibility and reconfigurability of DDP systems and service-oriented architectures with the timing guarantees required by hard real-time systems. The approach enables global coordination of field devices through deterministic execution plans that comprise communication and computation schedules with verifiable real-time properties. The execution plan is derived from a model-based representation of the control loop. During design and development, the control loop is modeled as a directed acyclic graph (DAG) of dependent tasks, similar to an automation workflow incorporating individual services in a SOA approach. We therefore also refer to the task-DAG as workflow. The workflow carries global timing information, such as its global deadline and period. Timing restrictions on a per-job level are derived from global constraints when binding a set of workflows to devices connected through a real-time network. This binding is performed by a skilled engineer who is supported by the rtSOA GUI. The rtSOA planner generates a static, cyclic, non-preemptive schedule for each device that includes all relevant tasks from a real-time workflow. The network communication is implicitly included in the generated schedules as each task is scheduled to finish before a certain communication deadline and the receiving tasks on different devices are only scheduled to start after the delivery of the relevant data from their predecessors.

This approach allows for a separation of concerns: When specifying workflows, an engineer is freed from timing constraints imposed by the hardware and network. New devices can quickly be integrated into the existing infrastructure by deploying those sub-tasks of a workflow that are specific to the device and subsequently generating new schedules that include the device. While we focus on applications in the context of manufacturing, rtSOA could also be applied in other areas, e.g.,the automotive or avionic industries.

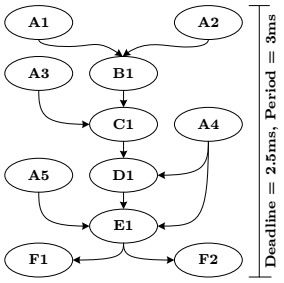
As explained in Section 2.2.2, there are two separate domains that need to be specified when deploying a SOA in the industrial context: the automation workflow and the infrastructure on which it will be executed. The infrastructure comprises the devices which partake in the automation process, the physical capabilities of each device (e.g.,provision of sensor data, high computational capability, etc.) and the characteristics of the network which connects the devices. Previous work has demonstrated that web service technology can be leveraged in the embedded context and efforts to standardize DPWS are well under way. The capabilities of a device can therefore be described, advertised and discovered through industry standards (c.f. Section 2.2.2.1). Service discovery at run-time is not required for planning and deploying the tasks of the control loop but offers possibilities for seamless integration into less time critical applications. Similarly, the rtSOA planner does not require run-time discovery of network particularities. Our assumption is that the network configuration, including addressing, message delay and TDMA slot assignment, is made available to the planner together with matching device descriptions.

The goal of the rtSOA planner is to provide a distributed execution plan wherein each device fulfills its part to cooperatively realize the control loop. The target platforms for rtSOA span from large control systems to very small embedded devices, such as smart sensors or actuators. We use the term smart device to describe a sensor or actuator attached to a system on a chip with several kilobytes of memory, a CPU clock rate of a few Megahertz and integrated networking capability. We do not assume that any advanced real-time operating system (RTOS) is available. The output of the rtSOA planning stage is a static, non-preemptive, cyclic schedule for each device. The job timing in each schedule is adjusted in such a way that the devices cooperate in a distributed service choreography without a centralized point of control. The exact execution semantics for rtSOA execution plans are given in Section 3.4. Advanced RTOS features are not required but can be leveraged to provide additional quality of service (QoS) levels beneath the critical real-time task.

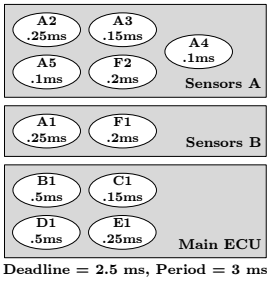
Figure 3.3 depicts an overview of the planning steps necessary in our architecture. We chose an adaptive cruise control system as an example. In this example, a 3D-vision system is used together with a radar system to measure the distance to vehicles in front of the object vehicle and regulate vehicle acceleration and deceleration accordingly. The resulting workflow is shown in Figure 3.3a. This only covers the functional dependencies and modular decomposition of the system so far. The global deadline and period of the workflow are derived from physical requirements



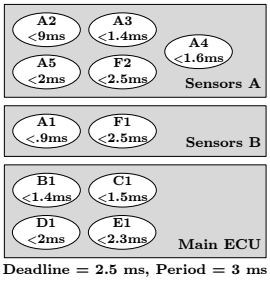
(a) Task in the embedded application



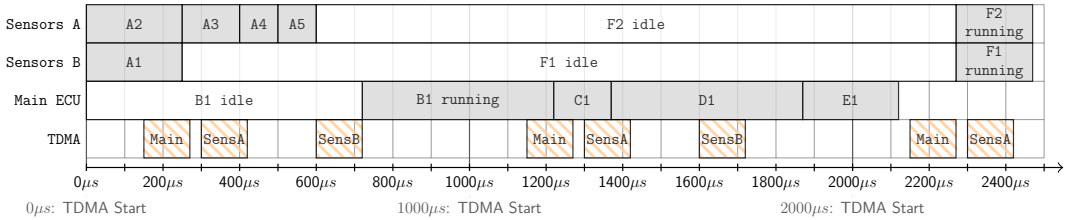
(b) Workflow layout and global timing constraints



(c) Task assignment and task WCET on assigned machines



(d) Local timing constraints as input for scheduling algorithm



(e) Simulation of a three machine execution plan for the workflow in Figures (a) - (d). Grey blocks denote task execution, white blocks show time spent waiting for messages from preceding tasks. Hatched blocks show the TDMA-slots.

Figure 3.3: A complex workflow in an adaptive cruise control scenario

and / or control theory. Once the automation task has been modeled according to the modular decomposition (Figure 3.3a), these global deadlines are attached to the workflow (Figure 3.3b). Because embedded systems often require platform specific implementations for each functional module and the modules themselves make use of sensors and actuators, the assignment of jobs to machines can be viewed as a design time decision performed by a skilled engineer. For any given assignment, the worst case execution time (WCET) of each task in the workflow can be measured or estimated. This estimation leads to the situation shown in Figure 3.3c where the global deadline and period of the workflow are known and the machine placement and WCET of each workflow task have been determined. Afterwards, the heuristics take over: a deadline assignment algorithm can be used to generate local constraints (Figure 3.3d), which are then used to generate a matching execution

plan that consists of a feasible task ordering. Alternatively, a suitable distributed scheduling algorithm can directly determine the task ordering. Methods for solving the scheduling problem, with an emphasis on suitable domain specific heuristics, are presented in Chapter 4. In the final step, the system is validated through discrete event simulation (c.f. Section 5.2). Output from this simulation step for our example is shown in Figure 3.3e. The fundamental difference between the planning and the execution phase of the control loop is important: whereas the execution plan is generated in a centralized, offline fashion, the execution of this plan is distributed without a central point of control.

In communicating systems with tight timing requirements, the network configuration plays an essential role in finding valid execution plans. We cannot simply place an upper limit on the communication delay and add it to the WCET of each task as this action would prevent us from finding a feasible schedule in Figure 3.3e and many other situations. Instead, timing information about each individual TDMA slot has to be considered when generating the execution plan. As shown in our example, the slots may be distributed irregularly. Real life examples would be an application sharing the same communication medium with a legacy application or communication protocols, such as Flexray, which set aside a portion of each cycle for lower priority traffic. We therefore consider the available TDMA slots as an input to our schedule synthesis instead of searching for a suitable slot assignment for a given schedule.

3.4 Execution Semantics

The semantics of our execution plans are strongly related to principles of dataflow programming which were presented by Dennis in the 1970s [32]. The original usecase for dataflow was as a programming model for massively parallel computing architectures which were seen as an alternative to classical von Neumann machines [2, 32]. In the abstract dataflow model, programs are modeled as graphs with data values, called tokens, flowing on their edges and operators being located in the nodes of the graphs. The activation of nodes follows a data-driven approach where a node becomes fireable when a token is present at each of its input edges. The node then fires some undetermined time after it has become fireable [2, 50]. When firing, the node removes a token from each of its input edges and produces a new token on each output edge. An example dataflow graph is shown in Figure 3.4. The example illustrates two key properties of the dataflow model. Nodes without any direct connection can potentially be evaluated in parallel. The other property is determinancy, meaning that the result does not depend on the order in which potentially parallel nodes are executed [2]. Acyclic dataflow graphs with side-effect free nodes are always *well-behaved*, meaning that a single wave of input tokens produces exactly one wave of output tokens [2, 50].

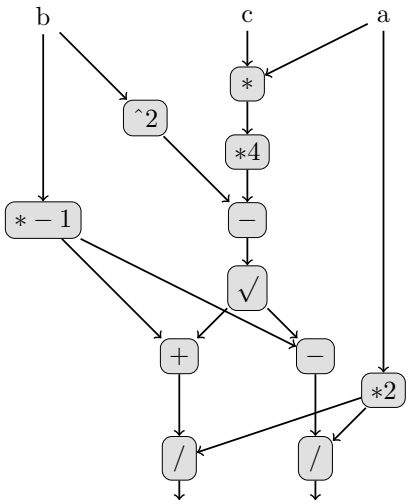


Figure 3.4: An acyclic dataflow graph for calculating the quadratic formula $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$. Operators are evaluated in infix notation with the leftmost incoming edge being placed before the operator and the rightmost edge after the operator. The edges labeled a, b and c are called input edges.

For conditional and loop programs control operators are required. The switch operator has one input and two outputs, labeled true and false, and an additional second input, called the control input, that takes a Boolean value. The Boolean value received by the control input determines the output edge on which the input token is forwarded. The counterpart of the switch operator is the merge operator. It also has a special control input. In the case of the merge operator, the control input determines whether the token from the "true" or "false" input edge is forwarded on the node's single output. With these two control operators conditional and loop programs can be represented.

The details of handling tokens is an important distinction between different dataflow models. In the abstract model, edges in the graph are assumed to be unbounded first-in first-out (FIFO) queues storing the tokens. Implementations of the dataflow architecture diverge from this idealized view. The *static dataflow* approach allows only one token to exist on an edge at any time whereas the *tagged-token dataflow* approach allows an unbounded number of tokens but enforces no ordering. A later development is *synchronous dataflow* (SDF) wherein the number of consumed and produced tokens on each edge is known at compile time [74]. This places restrictions on the type of programs than can be implemented in SDF, for example the maximum number of iterations for loops must be specified beforehand [50]. The benefit of this approach is that SDF programs can be statically scheduled and do not need the dynamic scheduling inherent in the other activation semantics [74]. This makes SDF an attractive execution semantic for embedded and real-time systems. The popular real-time programming language LUSTRE [40] is based on SDF. Synchronous dataflow is built around *large grain dataflow* [74], which means that the nodes in the dataflow

graph do not represent individual operators but rather groups of von Neumann operators, i.e.,larger code modules or functions. Large grain dataflow has become the dominant model since the 1990s as it has been noted that pure, fine grain dataflow networks do not offer the expected performance benefits [50]. The mathematical properties of dataflow do not change with the granularity of its nodes [50].

Our approach is mapping dataflow semantics to execution plans in service-oriented architectures. The natural fit for this approach is a large grain dataflow semantic. The individual service instances form the nodes in the dataflow graph and the communication between the instances maps to the edges in the dataflow graph. Visual programming with dataflow semantics is a paradigm that many engineers are already used to through commercial products such as LabVIEW[1] or Simulink[2]. Surveys have found that the visual aspect of LabVIEW is rated more positively than its textual aspects, indicating an inherent benefit of visual programming with dataflow semantics [116]. We therefore believe that the same programming paradigm can be quickly applied by domain experts for service composition in industrial automation.

rtSOA is targeted at hard real-time systems and small embedded devices. This application domain prohibits the use of tagged dataflow semantics. Tagged dataflow requires scheduling decisions and buffer management at run time. Both of these requirements lead to indeterministic timing behavior. Memory allocation at run time may not even be possible on embedded devices. We have therefore chosen to model the execution semantics of rtSOA choreographies after the static dataflow model. This model requires that at most one token be present on an edge in the dataflow graph at any time. This allows constant time checking of a node's ability to fire. As described in Section 3.2, each service instance has several input ports and output ports. These ports are the targets of connections between different instances and represent memory areas that can be allocated before the execution of the choreography. Each of the input ports has a flag which represents that data is present. Since the number of input ports is constant, assessing the fireability of a service instance can also be performed in constant time.

Classical dataflow literature mentions severe problems with enforcing the "one token per edge" restriction [2, 50]. The problem lies in the requirement that a node may only be fired if there is no token present on any of its output edges. This is usually enforced through adding acknowledgment edges in the opposite direction of each edge or by following a demand driven approach where a node is only activated after receiving a request via its output edge [50]. These semantics would both add additional network communication and reduce the performance of the system, similar to pull-based communication patterns. Another drawback mentioned in literature is the limit that this architecture places on parallelism as loops in the graph may not be dynamically unrolled. This means that a second loop iteration may not begin until the first iteration has finished [2].

[1] http://www.ni.com/labview
[2] http://www.mathworks.com/products/simulink

In the context of hard real-time systems, the restriction on loop parallelism is not a serious problem. In fact, real-time determinism requires the specification of the maximum number of iterations inside a loop (c.f. Section 1.2) which allows unrolling of loops at compile or specification time [2]. The dataflow graphs rtSOA has to handle are therefore always directed acyclic graphs (DAGs). The increase in network traffic caused by enforcing the "one token per edge" property, however, is a concern for rtSOA. For well-behaved graphs, the acknowledgment edges can be dropped if we ensure that each node is activated exactly once before the next tokens are placed on the input edges of the graph. New input may thus only be placed on the input edges after all nodes in the graph have been activated once. This is enforced through predetermined static schedules. The real-time domain requires reliable message transfer, so it can be assumed that tokens sent will always reach their destination, eliminating the need for acknowledgment edges for this purpose. When enforcing the predetermined schedule, data consistency and determinism of the dataflow DAG can be established by viewing the problem in the light of transaction processing in databases [60, Chapter 11]. The external actions of a node activation can be modeled as a transaction as shown in Equation 3.1.

$$T_1 = r_1(In_1) \rightarrow \ldots \rightarrow r_1(In_n) \rightarrow w_1(In_1) \rightarrow \ldots \rightarrow w_1(In_n) \rightarrow w_1(Out_1) \rightarrow \ldots \rightarrow w_1(Out_n) \rightarrow c_1 \quad (3.1)$$

A node will first read from all its input ports, then delete all tokens from its input ports before writing to its output ports and committing the transaction. Read / write conflicts could potentially arise from data dependencies between services. Viewing the two transactions T_1 and T_2 in Equation 3.2 without rules for ordering or scheduling of input tasks, T_2 could read from B before T_1 has written B, leading to an inconsistent state. However, the dataflow activation rules demand that T_2 may only ever be started after T_1 has finished its execution because there is a direct data dependency between them expressed through the token T_1 places in the input buffer of T_2. The read / write conflict between T_1 and T_2 is thus eliminated.

$$\begin{aligned} T_1 &= r_1(A) \rightarrow w_1(A) \rightarrow w_1(B) \rightarrow c_1 \\ T_2 &= r_2(B) \rightarrow w_2(B) \rightarrow w_1(C) \rightarrow c_2 \end{aligned} \quad (3.2)$$

The entry c_1 in transaction T_1 stands for a commit in the database world. Applied to the rtSOA architecture, committing means delivery of the tokens written by T_1 to the input ports of T_2. If both T_1 and T_2 are executed on the same machine, this means writing data to the memory location representing the input port. This is either performed by the service itself or by a data routing sub layer (c.f. Chapter 6) and may be added to the WCET of the service instance performing T_1. If the two services are located on different nodes data must be delivered over the network. If we assume bounded message delay, we can include an upper bound for the network delivery time to the execution time of T_1, thus ensuring that the "commit" is completed before the dependent service, represented by transaction T_2, is started. rtSOA uses a more fine-grained model that schedules message transfer in individual slots of a TDMA-cycle.

Under these circumstances, T_2 may only start after the tokens from its predecessor T_1 have been transmitted over the network.

$$\begin{aligned} T_1 = r_1(X) &\rightarrow w_1(X) \rightarrow w_1(A) &\rightarrow c_1 \\ T_2 = r_2(Y) &\rightarrow w_2(Y) \rightarrow w_2(A) &\rightarrow c_2 \\ T_3 = r_3(A) &\rightarrow w_3(A) \rightarrow w_3(B) &\rightarrow c_3 \end{aligned} \tag{3.3}$$

Given no further restriction, there is indeterminism inherent in two edges connecting to the same input port, as is apparent in Equation 3.3. This violates the determinism property of dataflow architectures. Graphs with two connections to an input port are therefore forbidden and the engineer must specify an additional arbitration node which will determine the canonical value written to the original target port. The addition of the arbitration service would transform the set of transactions to the following form, which is free of write / write conflicts and free of read / write conflicts when scheduled according to rtSOA rules, which is shown in Equation 3.4

$$\begin{aligned} T_1 = r_1(X) \rightarrow w_1(X) \rightarrow w_1(A) \rightarrow c_1 \\ T_2 = r_2(Y) \rightarrow w_2(Y) \rightarrow w_2(A) \rightarrow c_2 \\ T_{ar} = r_{ar}(A_1) \rightarrow r_{ar}(A_2) \rightarrow w_{ar}(A_1) \rightarrow w_{ar}(A_2) \rightarrow w_{ar}(A) \rightarrow c_{ar} \\ T_3 = r_3(A) \rightarrow w_3(A) \rightarrow w_3(B) \rightarrow c_3 \end{aligned} \tag{3.4}$$

rtSOA activates individual service instances at a predetermined offset from a global time instant. Conceptually, this can be modeled as an additional input edge for each node in the dataflow graph which writes a token to the node at the schedule time of the node, thereby making the node fireable. We call this token the *trigger token*. We distinguish between two types of schedules generated by rtSOA: *non-blocking* and *blocking* schedules. In non-blocking schedules, the arrival of the trigger token makes the node fireable and the node is executed immediately. In blocking schedules the trigger token arrives before all data tokens have arrived from other nodes. The node becomes fireable and is executed upon arrival of a data token. This indicates an imperfect schedule which may still be a feasible schedule given the global workflow deadline. Our simulation and verification modules (Chapter 5) therefore consider blocking schedules under the assumption that the blocking schedule can be transformed to a non-blocking schedule by delaying the arrival of the trigger token. Our real-world demonstrator (Chapter 6) only implements non-blocking schedules, assuming that feasible blocking schedules have been transformed to non-blocking schedules before starting the execution of the service choreography.

As pointed out by Lee and Messerschmitt, runtime overhead from dataflow architectures exists in the forms of buffering overhead and scheduling overhead where the system dynamically determines which nodes should be activated [74]. Our approach eliminates both of these sources of overhead since only a single token must be

stored per edge and all scheduling decisions are performed a priori. Lee and Messerschmitt also introduced static scheduling for synchronous dataflow (SDF) systems which form a sub set of dataflow systems. In contrast to traditional dataflow, the amount of data produced and consumed by a node is specified a priori. For example, a node may consume two tokens of input for each token of output it generates. These kind of different sampling rates are common in DSP programming. To achieve real-time properties, SDF also restricts the form of a dataflow graphs that can be modeled. Restrictions are placed on conditionals where Lee and Messerschmitt distinguish between data dependent and state dependent conditional control flow [74]. An example for state depended control flow would be bounded loops that are not dependent on input data. These can be handled in SDF by unrolling the loop through transformation of the dataflow graph. A data dependent example are nodes that generate tokens on one output edge if their input is below a certain threshold and generate tokens on another output edge when input is over the threshold value. Data dependent control flow is not explicitly handled in the synchronous dataflow model.

The same restrictions apply in the dataflow model applied by rtSOA. We additionally restrict our model to well-behaved graphs where only one single wave of tokens is active at any given time, making the rtSOA dataflow semantics a subset of the SDF model. rtSOA thus does not currently support nodes with different sampling rates in the dataflow graph nor does it support different cycle lengths for the schedules performed by each device collaborating in a given execution plan. Although these additional restrictions further limit the range of systems that can be modeled through rtSOA, when compared with general SDF, they allow efficient runtime implementations on constrained embedded devices. Although it is possible to allocate a bounded amount of buffer space for a given schedule for a SDF graph, the algorithm given by Lee and Messerschmitt for scheduling of these graphs is not optimal in terms of required buffering space [74]. rtSOA eliminates the requirement for buffer management for input tokens while allowing run time reconfiguration of the system without recompilation. Limiting the execution model to well-behaved graphs also reduces the memory requirements of the rtSOA runtime system because at most one token must be kept in memory for each edge in the dataflow graph. Future work could assess the implementation of synchronous dataflow semantics for distributed execution plans.

3.5 Discussion

This section reviews the design decisions and implications of rtSOA. First, Section 3.5.1 presents alternatives to the high-level decisions made for the design of rtSOA and states the benefits and drawbacks of each. Related systems in the field of industrial automation and real-time service-oriented architectures are examined in Section 3.5.2.

3.5.1 Design Alternatives

Time-triggered Architecture vs. Event-triggered Architecture

The question whether distributed real-time systems should be based around time-triggered or event-triggered designs has been debated since the 90's [61]. rtSOA follows a time-triggered architecture. Event-based systems offer benefits in terms of resource utilization, because they do not follow a static schedule that is provisioned to meet the worst case requirements. In high-load situations the additional overhead introduced by task-switching and decision making in event-triggered systems may make time-triggered systems more competitive as they have a lower runtime overhead. Another argument against time-triggered systems is that adding additional tasks often requires a total replanning of the whole schedule while event-triggered systems appear to be easier to extend since all scheduling decisions are taken locally. However, adding another task to a machine in an event-triggered system may also change the temporal dynamic of the system [61], which is compounded by the fact that event-triggered systems are hard to verify and predict. The quick replanning offered by rtSOA mitigates this classic drawback of time-triggered systems, allowing engineers to extend the system with confidence that the change will not introduce unpredictable temporal behavior.

A disadvantage of rtSOA, when compared with purely event-based approaches, is that the upper bound for the runtime of the execution plan also constitutes the lower bound. rtSOA users need to specify the WCET of each task and, due to the time triggered execution semantics, an early completion of a task will not lead to early completion of the workflow or offer additional processing time to succeeding tasks. rtSOA is thus well suited to critical cyclic control tasks but may overprovision resources in a more unpredictable environment, for example one where a human is interacting frequently with the system. A more event-driven approach could offer faster response times at the cost of predictability.

Run Time Reconfiguration vs. Code Generation

rtSOA provides flexibility and adaptability of manufacturing services through reconfiguration at run time without the need for providing new binaries or source code that implements the changed functionality. New system behavior is achieved strictly by reordering the invocation sequence of precompiled services and rerouting the dataflow between them. As such, reconfiguration fits well within the DDP and SOA paradigms as it keeps coupling between services low and offers a fast change of functionality.

An alternative approach would have been the generation of optimized binaries for each device that participates in the execution plan. An example for such an approach is realized in the AESOP project through the real-time programming language Timber [91] where an optimizing compiler allows for efficient implementations of real-time choreographies, thus reducing run time overhead.

In our opinion, the biggest issue with approaches based on code generation is the reprogramming of embedded devices with the new binaries. It is common for embedded systems to execute a single binary which may or may not include a real-time operating system kernel. These RTOSs often do not support dynamic linking and loading, meaning that the usual way to update code on embedded systems consists of bringing the system to a stop, reprogramming the system by replacing the executed binary and then restarting the device [43]. The reprogramming operation often requires physical access and cannot be performed over the network. This has been recognized in recent research endeavors aiming to offer lightweight solutions to the problem of run time code updates on embedded systems [43]. The rtSOA approach works with the service binaries residing on a device, requiring no re-programming, re-linking or re-loading of binaries. This means than engineers changing the choreography of a system do not need access to the services' code or binaries, they may simply change the system's configuration. rtSOA still benefits from run time code updates for loading new services onto a device or updating existing services to newer versions. However, as it stands, the rtSOA architecture can be implemented with very lightweight runtime environments that require no run time code update features.

Service Choreographies vs. Service Orchestrations

The SOCRADES project has studied the differences between service orchestrations and choreographies [15]. Among the benefits of service orchestrations identified by SOCRADES are the isolation of the workflow logic to a single device, the exposition of the orchestrated workflow itself as a single service and the simplicity of the approach. The drawbacks of service orchestration were listed as the absence of horizontal interaction due to orchestration being a strictly hierarchical approach that pushes the decision logic out of the individual devices onto the orchestrator. Additionally, the request / response messaging pattern of service orchestrations means an increase in network traffic [104].

The identified benefits of service choreographies mentioned by Candido et al. [15] include the truly distributed control with peer-to-peer communication and decision making at the individual devices. Starke et al. elaborated on this by explaining that service orchestrations lead to longer reaction times because the orchestration engine is an intermediate device which is one additional step (and network hop) removed from sensing and actuation [104]. Candido et al. stated the drawbacks of choreographies as the distribution of workflow logic to each participating device and the scaling to large and complex systems.

rtSOA is aimed at hard real-time sub systems in automation settings. The choreographies executed by these systems are limited to a smaller number of devices which must interact with each other within tight timing bounds. Higher level interactions between multiple sub systems may follow different interaction models and could again be implemented through service orchestrations. Integration of multiple systems performing a mixture of service orchestrations and choreographies is outside of the scope of this thesis. With rtSOA the localized view on a service choreography is realized in

terms of the schedules executed by each device together with the information about where and when messages generated by the services should be sent. This realizes the benefits of a truly distributed execution while addressing some of the open questions raised by SOCRADES about the semantics and implementation of hard real-time service choreographies. The choreography model is also a good fit for the dataflow semantics described in Section 3.4.

Dataflow Semantics vs. Petri Net Semantics

Synchronous dataflow is a special case of Petri nets [74] which can model asynchronous systems, as compared to the synchronous systems modeled by SDF, and have been used as execution semantics in the SOCRADES and AESOP projects [81]. With the rtSOA dataflow semantics being a subset of SDF, and SDF being a special case of Petri nets, one could also model the rtSOA semantics based on Petri nets. Related work using Petri nets has remarked that run time reconfiguration of small embedded devices may not be feasible when using interpreted Petri nets as the underlying semantics [81]. In contrast, we have realized a very lightweight yet still run-time-reconfigurable and verifiable implementation with the simple but restricted execution semantics of rtSOA. The richer Petri net formalism should therefore only be used when the system behavior cannot be expressed through simpler semantics. Since rtSOA systems are synchronous systems by construction, much of the complexity needed to model event-based asynchronous systems is not required. The richer semantics of Petri nets also lead to a more complicated graphical notation. The dataflow notation is very similar to existing standards for programming industrial control systems, for example the IEC 61131-3 function block diagrams (FBDs) which are a standardized language for programming PLCs.

Heuristic Schedules vs. Optimal Solutions

Applying heuristics for the generation of execution plans is the usual approach in interactive data processing systems which focus on providing an overall low response time to user queries. We view the rtSOA planner as a component of such an interactive systems as it may be used early and often in the engineering process of the automation system. The question remains if these heuristic schedules are suitable for continued execution in the automation system. The hard real-time domain has a binary value function for computation results. Anytime before the deadline the computation result has a relative value of 100%, anytime after the deadline its value drops to 0% or even a negative value. Therefore, an algorithm solving the optimization problem inherent in rtSOA's schedule based execution plans must find a schedule that is verifiably below the specified workflow deadline. Assuming that there is some slack time in the workflow, this schedule need not necessarily be the shortest possible schedule. Although heuristics may fail to find such a schedule, even if it exists, they are a feasible alternative because of their speed and effectiveness. Our work has proven that a combination of heuristics will find feasible schedules in the overwhelming majority of cases. Details are given in Section 4.2. However, heuristics cannot prove the absence of a feasible schedule. We therefore also give a

fast formulation of the rtSOA scheduling problem as a mixed integer linear program (MILP) in Section 4.1.2. However, the formulation is optimized for finding any feasible schedule, not the shortest schedule. Even with this restriction, the exhaustive search performed by the MILP solver may not lead to acceptable response times when included in an interactive design environment as described in Section 3.1.

3.5.2 Related Work

Commercial State of the Art

In today's manufacturing plants sensors and actuators are usually controlled by programmable logic controllers (PLCs), following a cyclic model. At the beginning of each scan cycle an input scan is performed which obtains readings from all connected sensors. Based on these updated values, the PLC performs its logic computations, updates all outgoing communication values and sends commands. Although recent industry efforts target increased reusability of code blocks, the control software is often rewritten from scratch when integrating new devices [77]. The tightest timing requirement determines the available runtime for the whole scan cycle, which is tightly coupled with the network cycle. The resulting hierarchical communication with tight cycle times can quickly exhaust existing network resources and lead to difficulties during network scheduling.

The cyclic rtSOA execution plans seem conceptually similar to the PLC scan cycle. However in a PLC scan cycle, code blocks are executed in sequence and without specified offsets from the start of the cycle time. Additionally, the majority part of a PLC program is executed in each cycle. In the rtSOA runtime, at most one service is triggered per iteration of the main loop. Only sensors required by the currently scheduled service instance are queried over analog / digital IO or pushed to the service instance over the network. Users do not have to specify the task order explicitly, it is instead determined by the dataflow and may easily be reconfigured. Compared to traditional, hierarchical control paradigms based on PLCs, rtSOA offers easier reconfiguration and more efficient network communication. Reconfiguration is enabled by an encapsulation of device capabilities in reusable services, and the separation of functionality from the timing and network scheduling aspects which are intermingled when writing PLC programs, thus limiting the reuse of existing code. The true peer-to-peer relationship of rtSOA devices and services reduces network communication overhead when compared with traditional, polling based approaches. The rtSOA planner explicitly generates a network schedule, so only required TDMA-slots are used. The assignment of TDMA-slots to devices is seen as an input to the planner, which means an rtSOA workflow can coexist with other workflows or legacy applications on the same network. The cyclic execution model of rtSOA has little communication jitter, thus allowing easy composition of rtSOA-controlled cells with other modular systems in a hierarchic manner.

IEC 61499

The IEC 61499 architecture has a number of parallels with the rtSOA architecture, for example the encapsulation of functionality in function blocks which can be freely placed among devices in a distributed system. Vyatkin names a number of concerns among researchers and practitioners about the IEC 61499 architecture [114]. Chief among them is the issue of determinism, arising from the purely event driven execution model of the standard. As events may potentially arrive at any time, the execution of function blocks implies that those events need to be stored in queues. Events may be explicitly or implicitly lost if the storage capacity of those queues is exceeded. This may lead to non-deterministic behavior given the same input conditions but slightly different message timing. Research has tried to address these issues through cyclic or synchronous activation semantics. However, the cyclic and synchronous IEC 61499 event processing models do not address situations in which multiple inputs arrive simultaneously at the same FB. Correct handling of event flows has also been shown to add substantial performance overhead [114].

rtSOA follows well-defined, conflict free data flow semantics (c.f. Section 3.4) which can be implemented on resource constrained devices without large overheads or message buffering. Also, by design, IEC 61499 is a programming and development model for distributed automation which aims to improve the development of such systems. Therefore, its focus is more on efficient code generation than on changing system behavior through reconfiguration at run-time, which is offered by the rtSOA architecture. Real-time execution of IEC 61499 systems is usually based on preemption [125], requiring more advanced RTOS features than rtSOA.

RI-MACS

Cucinotta et al. presented a real-time enhanced SOA for industrial automation within the RI-MACS project [26]. RI-MACS supports services with different criticality (hard, soft and non real-time) with temporal isolation of the more critical levels from the less critical ones. The presented SOA uses a modification of the WS-Agreement protocol to negotiate real-time and Quality of Service (QoS) properties for the individual services. Timing properties are enforced by a modified Linux Kernel that supports real-time scheduling through the earliest deadline first (EDF) algorithm. EDF is an optimal scheduling algorithm for preemptive uniprocesors [124].

RI-MACS is the first project to specifically target real-time SOAs in industrial automation. Compared to rtSOA it offers more QoS levels than just the hard real-time task. However, the rtSOA runtime is more lightweight, requiring no complex operating system, such as the real-time Linux used by RI-MACS. Additionally, rtSOA offers end-to-end timing guarantees for complex workflows instead of isolated guarantees for single service instances.

SOCRADES

The approach to workflow composition and execution within the SOCRADES project is based upon high level Petri nets [82]. The Petri nets serve as formal specification, provide semantics for composition and support conflict resolution in the SOCRADES architecture. Automation processes can be modeled on a high layer of abstraction because the chosen Petri net formalism distinguishes between immediate transitions for tasks such as message transmission and timed transitions for longer running tasks such as movements of a robot arm [81]. During composition these timed transitions are replaced with their concrete Petri net representations.

Petri nets also provide the run time semantics and control in the SOCDRADES project. An orchestration engine interprets a Petri net, which has been provided to the engine at run time, and therefore executes the workflow composition represented by the Petri net. Multiple devices may cooperate in the execution of larger Petri nets where synchronization and communication between the models running on each devices is performed at run time. Precompiled versions of Petri nets may be employed for more resource constrained devices that are unable to perform an interpretation of a Petri net model at run time.

The SOCRADES approach based on the Petri net formalism offers a clear semantic for complex composed models at run time and allows thorough formal verification and analysis of its behavioral properties during design. However, the flexibility of the presented approach is not complemented by reliable timing guarantees because the Petri net orchestration engine does not consider timing issues. Petri net models may also quickly become very large and complex [81], leading to high overhead in interpretation. Both of these issues indicate that the presented approach is not well suited to the rtSOA target domain of hard real-time systems with tight control cycles.

AESOP

The AESOP project investigated the possibilities of bringing SOAs into industrial control loops [46]. From a technology perspective, the Efficient XML Interchange (EXI) binary XML format has been identified as a crucial complement to the XML based DPWS messages, if cycle times of a few milliseconds are targeted [71]. AESOP achieved cross-layer collaboration of services and devices mainly through use of an orchestration engine, which constitutes an event based model with central control through an orchestrator. Other research within the project highlighted the benefits of service choreographies over service orchestrations for automation tasks [104]. Starke et al. implemented an event-based choreography engine for service-oriented automation and observed a higher degree of performance and reactivity compared to a traditional approach based on orchestration. Choreographies are composed of individual roles which specify the behavior of devices participating in the choreography which is realized through invocation of the device's services. Based on its role description, a device will establish connections to other devices where it subscribes to the relevant events published by the remote device. Starke et al. do not mention end-to-end performance guarantees for these event based choreographies [104].

rtSOA also offers decentralized service choreographies, but additionally focuses on the temporal aspects and offers deterministic, predictable and verifiable real-time properties. Users of the choreography engine described by Starke et al. first have to specify the individual roles before those roles can be combined into the choreography and deployed to the devices. In contrast to this approach, rtSOA choreographies are composited from available service descriptions which are bound to devices at a later stage. The push-based communication paths of rtSOA are automatically derived from this binding. Our approach is therefore more top-down than the AESOP choreographies.

The SOCRADES or AESOP design can be united with the rtSOA approach to achieve both event-based flexibility and cyclic determinism where needed: A cyclic sub-system, scheduled with rtSOA, can periodically trigger events which are then processed by an event-based architecture on higher layers. The DPWS protocol together with the binary EXI format is a possible candidate for more full featured service discovery and message exchange than the minimal functionality implemented for our real-world demonstrator (Chapter 6).

RT-Llama

RT-Llama is a real-time middleware for service-oriented architectures [85, 90] with support for end-to-end deadlines. It achieves predictable execution times for complex workflows forming DAGs through pre-reservation of resources for the whole workflow. Similar to our model, services in a workflow must be assigned a WCET. Based on the workflow structure and the service WCETs, intermediate deadlines are derived and used for scheduling. We discuss a similar approach in Section 4.2.1. RT-Llama is targeted at dynamic SOAs, which are defined as systems in which workflows may join or leave the system at arbitrary times [85]. Resource reservation is performed for periodic workflows which can have differing periods from one another. The remaining resources are available for sporadic workflows without predefined periods if enough capacity is available for the tasks. An efficient admission test is another contribution of the RT-Llama system [85]. RT-Llama supports a wider range of scheduling situations by being designed for multiple periodic and aperiodic workflows. However, the industrial automation domain is not as dynamic as the dynamic SOAs described by RT-Llama. RT-Llama may deny admission of a workflow at run time which could be catastrophic in hard real-time systems. A system for industrial automation therefore will be pre-planned with all necessary tasks before it is deployed to ensure that critical tasks will always be executed on time. Scheduling with rtSOA provides these kinds of guarantees. Compared to rtSOA the RT-Llama system is resource intensive because its prototype is implemented using a real-time Java Virtual Machine (JVM) [90]. The scheduling algorithms of rtSOA also work on a more fine-grained level. Our work schedules real-time tasks with enough precision to meet individual slots in a TDMA-cycle whereas RT-Llama assumes only a bounded message delay. The overall target systems of RT-Llama thus have more resources and less stringent timing requirements than the industrial automation domain targeted by rtSOA.

iLAND

iLAND is a middleware for service-oriented soft real-time systems [39] that supports time-bounded reconfiguration. After an initial offline phase in which the workflows and their properties are analyzed the iLAND middleware supports time bounded operation with soft real-time semantics. Time-bounded reconfiguration means that the workflows can be modified to include new services or capabilities and the generated communication and computation schedules are adjusted accordingly. iLAND can perform automatic service composition, meaning that it can derive a new workflow which conforms to given functionality and properties. Apart from the focus on soft real-time, the main difference to rtSOA is the higher resource usage of iLAND as it is implemented on top of the JVM and requires the presence of a RTOS which supports task preemption. Time-bounded reconfiguration may be relevant in the area of industrial automation insofar as switching between different modes of operation of the whole system is concerned. These modes of operation are distinct preplanned execution states, similar to how an rtSOA execution plan is a single preplanned state. If mode switching were implemented, reconfiguration of an rtSOA system could be performed by configuring a new execution plan during idle times of the nodes and switching to the new mode in a coordinated manner. However, reconfiguration in industrial automation often also involves modification of the physical environment of the system, for example repositioning modules in a modular production system or removing leftover work pieces from the manufacturing line. The quick redeployment of execution plans with rtSOA, as demonstrated in Chapter 6, is not the bottleneck in such a scenario, limiting the requirement for run-time reconfiguration as implemented by iLAND in the rtSOA target domain.

AutoFOCUS

Voss and Schätz [113] present a model based approach for development, deployment and scheduling of embedded applications. The application is modeled as a network of automata. The hardware resources, to which the application should be deployed, can consist of multiple micro controllers connected to shared memory via a bus system. The authors employ Satisfiability Modulo Theories (SMT) solvers to find both a suitable task placement and schedules for each device. From these generated schedules individual binaries for each target system can be derived through automatic code generation. Voss and Schätz target systems with a bus-based communication channel that has a fixed bound on latency, whereas our model adopts a fine-grained communication model on the level of the individual TDMA-slots. Section 4.1.3 shows that the employed SMT-solvers are an order of magnitude slower than mixed integer linear programming (MILP) solvers for similar feasibility problems [66]. Chapter 4 presents our MILP formulation along with several fast and efficient heuristics which are again orders of magnitude faster than the MILP solver approach. Together with the fact that the output of the AutoFOCUS development tool is a binary for each module, which carries the drawbacks of reconfiguration through reprogramming that we have laid out earlier (c.f. Section 3.5.1), the AutoFOCUS approach is not suitable

for rtSOA's target systems. However, the system does prove the value of model based development and automatic synthesis of schedules for distributed systems. As such, AutoFOCUS was one of the design inspirations for rtSOA.

MGSyn

Another example for a non-SOA based approach to flexible and reconfigurable manufacturing systems is MGSyn [20], which synthesizes controller programs with principles adapted from game theory. The game is modeled with two players, the system and environment, which alternate taking turns. The system wins the game if it can always reach winning conditions, which are specified in a subset of linear temporal logic (LTL). The environment may play any allowed move at any time and is not required to cooperate with the system. Goals for this synthesis are specified in a modified version of the Planning Domain Definition Language (PDDL). The generated control program is essentially a state-transition diagram that fulfills the specification. It also offers the possibility to generate programs that fall beneath a specified upper bound for the overall execution time or other metrics, such as power consumption. MGSyn is also based on code generation and replacement of binaries on the controllers which is not in accordance with a SOA-based approach. The main questions raised by the approach based a formal specification language, when compared with a SOA-based approach, are reusability of individual components, modularity and approachability. Studies have shown the benefits of using visual programming together with large grain dataflow architectures for productivity and correctness [6, 116]. This programming paradigm is combined with high-level reuse of individual services in rtSOA.

Conclusion

This chapter described the rtSOA approach for realizing distributed real-time systems with SOAs and distributed data processing principles. The planning process for generating real-time capable execution plans is a key component to this architecture. The next chapter (Chapter 4) therefore presents several possible methods for deriving the execution plan and performs a thorough evaluation of the alternatives. Our real-world demonstrator for the rtSOA GUI and runtime is presented in Chapter 6.

"Choose well. Your choice is brief, and yet endless."
- Johann Wolfgang von Goethe, translated by Thomas Carlyle

CHAPTER 4

From Specification to Solutions

Parts of this chapter have been previously published in [65, 66].

This chapter covers the different approaches to generating a feasible execution plan for a workflow following the principles expressed in Chapter 3. rtSOA execution plans comprise of static, cyclic schedules for each device that participates in the plan.

Workflows are modeled as directed, acyclic graphs of tasks. The edges in a graph represent data flow between individual tasks. Each task is annotated with a worst case execution time (WCET). The range of possible start and completion times of the task can be constrained by setting a task release time and deadline. Each workflow is annotated with a global deadline and the period after which the workflow is repeated.

The overall goal is to find a suitable task ordering for a given assignment of tasks to machines. Tasks on a single machine cannot overlap or be preempted. This task ordering must fulfill all local deadlines (on the task level), local release times and global deadlines (on the workflow level). A task cannot be started before all transitively preceding tasks have been completed. For describing the different algorithmic approaches, we use the following formal notation:

The basis for deadline assignment is a workflow $\mathfrak{W} = \langle \mathcal{T}, \mathcal{G}, D_{\mathfrak{W}}, P_{\mathfrak{W}} \rangle$ with a task set $\mathcal{T}$, the precedence graph $\mathcal{G}$ formed by the tasks, a global deadline $D_{\mathfrak{W}}$, and a period $P_{\mathfrak{W}} \geq D_{\mathfrak{W}}$ after which the execution of the task set is repeated. The task set $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$, contains the individual tasks t_i. Each task has a (worst case) execution time written as $|t_i|$, a start time $t_i.S$ and a completion time $t_i.C = t_i.S + |t_i|$. The release time $t_i.R$ and deadline $t_i.D$ constitute constraints on the earliest start

time and latest completion time, respectively. The precedence graph $\mathcal{G} = \langle \mathcal{T}, \prec \rangle$ places constraints on the task ordering. A precedence relation between two tasks t_i and t_j is written as $t_i \prec t_j$ if t_i is a direct predecessor of t_j and as $t_i \prec\!\!\prec t_j$ if t_i is a transitive predecessor of t_j, e.g.,$t_i \prec \ldots \prec t_j$. $\mathcal{G}$ forms a directed, acyclic graph. Each task is assigned to a machine $m_i \in \mathcal{M}$ by the function $\mu_t(t_i) : \mathcal{T} \rightarrow \mathcal{M}$. μ_t is defined prior to the scheduling problem described in this section.

Some of the scheduling methods described in this chapter also take the underlying network configuration into account. As mentioned in the previous chapter, we assume real-time communication is provided by a network protocol which only allows communication at predetermined time instances, i.e.,a TDMA protocol. The assignment of TDMA-slots to machines and the timing properties of the slots are used as input parameters to the scheduling process. Our model of a TDMA-configuration consists of a number of TDMA-Slots with a fixed slot start and end time. Each slot is assigned to exactly one machine and is repeated after one TDMA cycle period. A TDMA slot may carry messages from multiple tasks on one machine to receivers on all other machines, i.e.,we assume a broadcast semantic. Tasks on the same machine communicate without time delay.

The TDMA-slot configuration is represented as $\mathfrak{S} = \langle \mathcal{S}, \mu_s, P_{\mathfrak{S}} \rangle$. Therein the set of TDMA slots is represented by $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ with each slot s_i having a slot-length $|s_i|$, a slot start time $s_i.S$ and a slot completion time $s_i.C = s_i.S + |s_i|$. Each slot is assigned to a single machine by the function $\mu_s(s_i) : \mathcal{S} \rightarrow \mathcal{M}$. The TDMA cycle is repeated after period $P_{\mathfrak{S}}$. Tasks in $\mathfrak{W}$ either communicate locally if they are on the same machine, or they send a message in a TDMA-slot that is assigned to the same machine. $\tau(t_i)$ is the slot assignment function for tasks, its output is either a single TDMA slot s_i, which is used by the task t_i to broadcast its result, or the empty set $\emptyset$ if task t_i only has successors on the same machine.

In classical scheduling theory scheduling problems are classified according to the $\alpha|\beta|\gamma$ scheme where α describes the machine environment, β describes the processing details and γ denotes the objective function that should be minimized. The general complexity of scheduling problems can vary immensely, depending on these parameters. For example, the problem $1|prec|L_{max}$, denoting a single machine problem with general precedence constraints on the tasks where the objective is minimizing the maximum lateness, can be solved in polynomial time [72]. The slightly different problem $1|r_t|L_{max}$, where r_t denotes that tasks have a minimum release-time after which they may be activated, is strongly NP-hard [76].

How would the scheduling problem for rtSOA be classified? One could classify the problem as $P_m|prec|L_{max}$, where P_m denotes m parallel machines. This problem is strongly NP-hard, as it generalizes $P2|chains|C_{max}$, which has been proven as strongly NP-hard [34]. This assertion can be made based on the complexity hi-

The complexity results mentioned in this section were found using the search functionality of "The Scheduling Zoo", located at http://www-desir.lip6.fr/~durrc/query/

erarchy of scheduling problems [92]. P_m is a more general form of $P2$ [92](meaning two parallel machines), *prec* is more general than *chains* [34], meaning chain-structured graphs, and the makespan C_{max} can be reduced to the maximum lateness L_{max} [92]. However, the classification as $P_m|prec|L_{max}$ is too pessimistic. For the rtSOA scheduling problem, we assume that the engineer has fixed the allocation of tasks to machines before the scheduling process, thus relieving the scheduling algorithm from the duty of finding a suitable task allocation. Therefore, we could conceptually view the problem as a single-machine scheduling problem, although this constitutes an oversimplification. Additionally, the maximum lateness is an unsuitable objective function. rtSOA targets hard real-time systems where the workflow must be completed at a given deadline. If we minimize the total makespan C_{max}, we minimize the total completion time of the workflow. If the makespan is then smaller than the global deadline $D_{\mathfrak{W}}$, the scheduling process was successful.

Can we then classify the problem as $1|prec|C_{max}$, meaning that it can be solved in polynomial time? [1] In general, the answer is no due to network communication. $1|prec|C_{max}$ would only be a correct classification of the scheduling problem if we added an upper-bound for transmission delay to the processing time of each task that has successors placed on different machines. This approach would unnecessarily lengthen the sum of processing times in the workflow, leading to a lower success rate when scheduling the workflow with a makespan smaller than $D_{\mathfrak{W}}$. Can we model the individual TDMA-slots as release-times for each task, classifying the problem as $1|prec; r_t|C_{max}$? Lawler has proven this problem to be in complexity class P as well [72]. This approach is better and we indeed model the problem in this way for some of our heuristics (c.f. Section 4.2). However, the release time r_j of a task t_j with predecessor t_i can depend on the completion time of the predecessor t_i: Because t_j must be ready to run at time r_j, t_i must have completed at that time. Additionally, data from t_i must have been received by t_j via the network. This means transformation of the scheduling problem in this way is either pessimistic, by using an upper limit for r_j, or must be repeated after each scheduling step, thus invalidating the assumptions necessary to prove membership in the complexity class P.

We derive that the rtSOA scheduling problem is strongly NP-hard from the following argument: When two tasks are placed on different machines, a non-zero communication delay occurs. In the general scheduling literature this is modeled as task-dependent time lags l_{ij} between tasks t_i and t_j, meaning that t_j can only start after at least l_{ij} time units have passed between the two tasks. Assuming that an oracle could provide us with valid values for all l_{ij}, we could classify the rtSOA problem as $1|prec; l_{ij}|C_{max}$. Wikum et al. proved that $1|chains; l|C_{max}$ is strongly NP-hard [118], meaning that the idealized rtSOA problem is strongly NP-hard by extension. Here, l stands for task-independent time lags which are generalized by task-dependent time lags. The generalization of *chains* by general precedence constraints *prec* remains [34].

[1] $1|prec|L_{max}$ is in complexity class P and generalizes $1|prec|C_{max}$

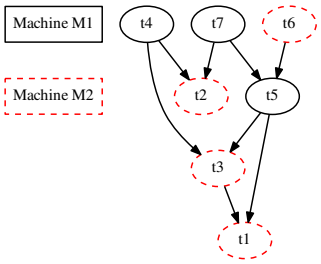
The target area of rtSOA are hard real-time systems, meaning that our objective function C_{max} could be regarded as too fine-grained. Hard real-time systems follow a step wise binary function for evaluating their performance. A result either fulfills the deadline constraints or it violates them. If the deadline is violated, the value of the result does not degrade but immediately reaches zero or becomes negative (c.f. Section 1.2). Conversely, it does not matter how far a solution is beneath the deadline limit, it is always valued the same. We are thus more accurately searching for a feasible solution to the scheduling constraints instead of trying to minimize an objective function. However, this does not result in a reduction of the complexity class for the scheduling problem, because the deadline could be chosen so that it is equal to the smallest possible makespan C_{max}.

However, this argument hints at the general structure of the problem: Finding the optimum, in terms of the smallest makespan, may generally not be the goal of the scheduling process. Instead we are interested in an effective method for finding feasible solutions to the scheduling problem described above. The remainder of this chapter explores different approaches based on either state space exploration (Section 4.1) or heuristic approaches (Section 4.2).

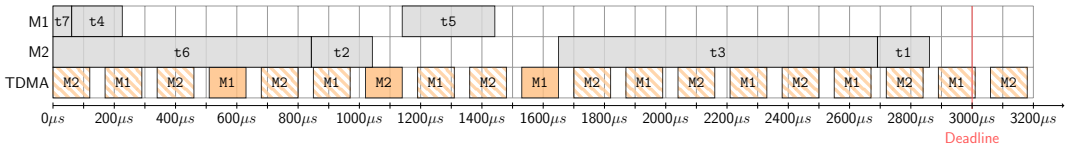
4.1 State Space Exploration

A naïve way of performing state space exploration for the rtSOA scheduling problem would be the enumeration of all topological orderings of the workflow with subsequent validation of the resulting schedules' feasibility. A small example for such an approach is shown in Figure 4.1. Generating a single topological order is easy and has a time complexity of $\mathcal{O}(|V| + |E|)$, where $|V|$ is the number of vertices, or tasks, in the workflow and $|E|$ is the number of edges, or precedence relations, between the workflow tasks. Enumerating all topological orders and checking them for feasibility is #P-complete [12]. Valiant defines the #P complexity class and shows that problems in #P are at least as hard as NP-complete problems [109]. While NP-hard problems conceptually often deal with questions of the sort "Is there a satisfactory solution that fulfills these constraints", #P-hard problems cover the task of enumerating all solutions to such problems [1]. From this high-level description it intuitively follows that the two classes are related, and that #P-hard problems are at least as hard as NP-hard problems.

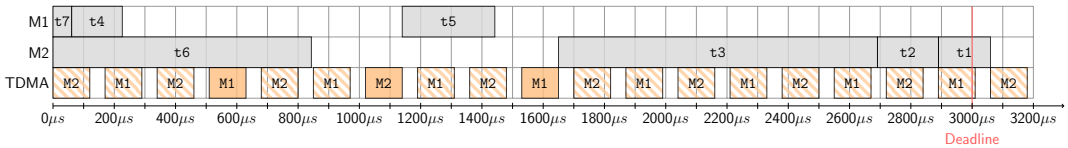
The remainder of this chapter covers more efficient methods for finding feasible solutions through state space exploration. Section 4.1.1 describes an approach based on a Satisfiability Modulo Theories (SMT) solver, which combines a Boolean satisfiability (SAT) solver with other theories, such as linear arithmetic, bit-vectors or arrays [30]. The next section Section 4.1.2 shows how the problem can be formulated as a Mixed Integer Linear Program (MILP), that can solved with many of today's highly tuned solvers. Section 4.1.3 covers related work in this area.



(a) An example workflow with 7 tasks, allocated to two different machines



(b) A feasible task ordering for the workflow shown in subfigure a)



(c) An infeasible task ordering, resulting from the switch of tasks t_3 and t_2

Figure 4.1: All topological orderings are potential valid task orderings. The example workflow in subfigure a) has 32 different topological orderings. In practice, switching two tasks can result in large differences in schedule length, as shown by subfigures b) and c). The schedule shown in subfigure b) is feasible, the schedule in subfigure c), differing only by the order of tasks t_2 and t_3 from the former, is infeasible.

4.1.1 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) solvers are efficient tools for finding feasible solutions to general first order logic problems that can be extended with a number of other theories, such as arithmetic, bit-vectors, arrays and uninterpreted functions [30]. As such, they offer an expressive and natural way of formulating the rtSOA scheduling problem in mathematical terms. In general a SMT solver is not an optimizer, because its output is a model, i.e.,a variable valuation, that renders all specified constraint formulas `true`. Other techniques, such as a binary search over possible values for the deadline constraints [113] with subsequent reevaluation of the SMT problem, have to be employed when optimization of an objective value is desired. Because our approach is only aimed at finding a feasible solution, a single evaluation of the SMT problem described below is sufficient.

The SMT solver receives the workflow $\mathfrak{W}$ and TDMA configuration $\mathfrak{S}$ as input and generates start times $t_i.S$ for each task $t \in \mathfrak{W}.\mathcal{T}$ as its output, so that the global deadline $D_{\mathfrak{W}}$ is adhered to. If the tasks are distributed over several machines, the solver also finds a valid TDMA-slot assignment $\tau(t_i)$ for each task t_i with successors on different machines.

Equation 4.1 is an input constraint for the SMT solver and describes the bounds for the task start times and completions times. It requires that all task start times have to be non negative, completion times have to be smaller than the global deadline and the completion time must be the start time plus the task execution time.

$$\forall t_i \in \mathcal{T} : t_i.S \geq 0 \wedge t_i.C \leq D_{\mathfrak{W}} \wedge t_i.C = t_i.S + |t_i| \tag{4.1}$$

Equation 4.2 requires that all tasks on the same machine do not overlap. For any two tasks on the same machine the start time of one task should be larger than, or equal to, the completion time of the other task. Alternatively t_i and t_j could refer to the same task.

$$\forall t_i, t_j \in \mathcal{T} : \mu_t(t_i) = \mu_t(t_j) \implies t_i.C \geq t_j.S \vee t_j.C \geq t_i.S \vee t_i = t_j \tag{4.2}$$

Equation 4.3 places constraints on two tasks, placed on different machines, that have a predecessor relationship between them. The constraint requires the solver to choose a TDMA slot for the first task via the function $\tau()$ and enforces the completion of task t_i before the beginning of the chosen slot. Similarly, task t_j may only start after the completion of the TDMA slot, i.e.,after t_j has received the message from t_i contained within the TDMA-slot.

$$\begin{aligned} \forall t_i, t_j \in \mathcal{T} : \mu_t(t_i) \neq \mu_t(t_j) \wedge t_i \prec t_j \implies & \tau(t_i) \neq \emptyset \wedge \mu_s(\tau(t_i)) = \mu_t(t_i) \\ & \wedge t_i.C \leq \tau(t_i).S \wedge t_j.S \geq \tau(t_i).C \end{aligned} \tag{4.3}$$

The SMT formulation for the rtSOA scheduling problem is complete with these equations. In the next section we present an equivalent formulation as a mixed integer linear program. Both formulations are compared in Section 4.1.3.

4.1.2 Mixed Integer Linear Programming

An Integer Linear Program (ILP) consists of three parts: A linear objective function to be maximized, e.g.,

$$f(x_1, \ \ldots \ , x_n) = c_1 x_y + \ldots + c_n x_n$$

a number of problem constraints of the form

$$a_1^1 x_1 + \ldots + a_n^1 x_n \leq b_1$$

$$a_1^2 x_1 + \ldots + a_n^2 x_n \leq b_2$$

$$\ldots$$

$$a_1^m x_1 + \ldots + a_n^m x_n \leq b_m$$

and variables $x_1, \ldots, x_n \in \mathbb{Z}$.

For a Mixed Integer Linear Program (MILP) not all variables x_i are required to be integers. While MILPs are not as expressive, or the formulation of a problem might not be as natural, when compared with a SMT problem formulation there has been a tremendous amount of research and engineering in the last decades to enable efficient solving of MILPs. Intuitively, being formulated in a single theory instead of multiple theories, as is the case with our SMT formulation, a MILP solver should be faster than a SMT solver. The evaluation in Section 4.1.3 shows that this is indeed the case for our problem.

Keha et al. [59] examined the performance of four different MILP formulations for single machine scheduling problems. They note that "MLIP formulations for scheduling problems are often classified based on the choice of the decision variables. The different decision variables used to distinguish between the different MILP formulations are: completion time variables (F1) [4], time index variables (F2) [101], linear ordering variables (F3) [35] and assignment and positional date variables (F4) [94]." Their paper shows that the first approach (F1), based on completion time variables, generates the highest number of feasible solutions in a given time although other formulations are faster in finding the optimum solution.

Since our focus is on quickly finding feasible solutions, we have based our MILP formulation around completion time variables and extended the formulation to multiple processors communicating via TDMA. The time index variable approach(F2) can be rejected, because it uses binary variables x_{ij} if a task t_i starts at time index j, which must be chosen at a set granularity before. We would either loose precision or face formulations with a huge number of variables. The approach based on linear ordering variables (F3) uses the variables y_{ij} which are set to 1 if task t_i precedes task t_j. We have not evaluated this approach, because the results of Keha et al. show that it often fails to produce a feasible solution in an allocated amount of time. Additionally, specifying constraints representing TDMA-communication is cumbersome in F3, at best. The last formulation (F4) uses assignment variables u_{ij} which are set to 1 if task t_i is assigned to position j. The positional variables C_k specify the completion time of the task at position k. This formulation is also competitive for finding feasible solutions in a specified time budget. However, in previous work we determined that this MILP-formulation performs worse than F1 for scheduling in-tree task sets with release times and deadlines [63] on a single machine. This problem is similar to the rtSOA scheduling problem, as the communication over TDMA can be represented by attaching release times and deadlines to the task set. We have therefore only adapted the formulation based on completion time variables to the more general rtSOA scheduling problem.

All solver specific remarks apply to the software package Gurobi[2] in version 6.5 which we used in our experiments.

[2] http://www.gurobi.com/

The LP variables are thus $C_i = t_i.C$ for all tasks $t \in \mathcal{T}$ and binary variables $y_{ij} = 1$ that denote that task t_i is scheduled before task t_j. Note that $t_i \prec\!\!\!\prec t_j \implies y_{ij} = 1$ but $y_{ij} = 1 \not\implies t_i \prec\!\!\!\prec t_j$, i.e y_{ij} only denotes precedence in the schedule produced by the solver. Additionally we introduce the binary variables $u_{ik} = 1$ that indicate task t_i sending its data via TDMA-slot s_k.

Equation 4.4 and Equation 4.5 define the value ranges for the completion time variables and the precedence variables. The implication in Equation 4.5 is not part of a valid LP constraint. We use this notation as a shorthand to denote that we only added the constraints in the consequent of the implication if the antecedent of the implication was given. In the case of Equation 4.5, we fix the value for the ordering variables y_{ij} and y_{ji} so that the precedence constraints expressed by the workflow graph are adhered to.

$$\forall t_i \in \mathcal{T} : C_i \geq |t_i| \wedge C_i \leq D_{\mathfrak{W}} \tag{4.4}$$

$$\forall t_i, t_j \in \mathcal{T} : y_{ij} \in 0, 1 \wedge t_i \prec\!\!\!\prec t_j \implies y_{ij} = 1 \wedge y_{ji} = 0 \tag{4.5}$$

The constraints in Equation 4.6 prohibit two tasks from overlapping. This means that for every two tasks on a machine, one of them needs to finish before the start of the other. Two tasks on different machines may overlap at will and thus there are no restrictions placed on their corresponding y-variables. As with Equation 4.5, the consequent of the implication in Equation 4.6 is only added to the MILP constraints if the antecedent is true ,i.e.,if the tasks t_i and t_j are placed on the same machine.

$$\begin{aligned} \forall t_i, t_j \in \mathcal{T} : \; & \mu_t(t_i) = \mu_t(t_j) \implies \\ & t_i.C + |t_j| \leq t_j.C + D_{\mathfrak{W}} * y_{ji} \\ & \wedge \; t_j.C + |j_i| \leq t_i.C + D_{\mathfrak{W}} * (1 - y_{ji}) \end{aligned} \tag{4.6}$$

The last set of constraints expresses communication over the TDMA-network. If two tasks are in a direct predecessor relation, but not running on the same machine, they need to communicate over the network. This means firstly that the preceding task t_i has to finish before any TDMA slot assigned to its machine. This is implied in the first two (in-)equalities in Equation 4.7. Lastly, the receiving task t_j can only start after the TDMA slot which transports the message from t_i has ended. Once more, the implication indicates that these constraints are only added to the MILP formulation if the antecedent of the implication is true.

$$\begin{aligned} \forall t_i, t_j \in \mathcal{T} : t_i \prec t_j \wedge \mu_t(t_i) \neq \mu_t(t_j) \implies & \sum_{s_k \in \mathcal{S}} u_{ik} = 1 \\ & \wedge \, t_i.C \leq \sum_{s_k \in \mathcal{S}} u_{ik} * s_k.S \\ & \wedge \, t_j.S \geq \sum_{s_k \in \mathcal{S}} u_{ik} * s_k.C \end{aligned} \tag{4.7}$$

We have found that Gurobi is faster in finding a feasible solution if no objective function is defined, therefore we omit its specification.

Workflow Tasks	Feasibility	Heuristic	Runtime	Timeouts
16	infeasible	Gurobi	3.50ms	0
		Z3	72.88ms	0
	feasible	Gurobi	5.88ms	0
		Z3	254.79ms	0
32	infeasible	Gurobi	10.70ms	0
		Z3	753.94ms	0
	feasible	Gurobi	37.66ms	0
		Z3	7,389.46ms	0
48	infeasible	Gurobi	23.86ms	0
		Z3	3,650.26ms	3
	feasible	Gurobi	114.21ms	0
		Z3	44,905.51ms	0
64	infeasible	Gurobi	47.01ms	0
		Z3	27,916.04ms	66
	feasible	Gurobi	268.07ms	0
		Z3	77,707.18ms	0

Table 4.1: Runtime comparison of a MILP-solver (Gurobi) with an SMT-solver (Z3). The runtime is given as the geometric mean over the observed test cases, timeout was after 20 minutes.[4]

4.1.3 Discussion and Related Work

Given the SMT-formulation presented in Section 4.1.1 and the MILP-formulation in Section 4.1.2, the question arises which of these two approaches is preferable. The ease of expression is higher in the SMT-formulation which provides a very straight forward approach to constraint programming. However, this expressiveness comes at the cost of higher runtimes with the current generation of SMT-solvers. We have chosen Z3[3] in version 4.3, an efficient SMT-solver from Microsoft Research [30]. The benchmarks in Chapter B.3 show that Z3 represents the current state of the art in SMT-solvers. However, benchmarks of the rtSOA scheduling problem show that Z3 is orders of magnitude slower than Gurobi for the formulations presented in Section 4.1.1 and Section 4.1.2. Table 4.1 shows the comparison of our SMT-formulation evaluated with Z3 and our MILP-formulation solved through Gurobi. Benchmarks in Section B.1 and Section B.2 show that Gurobi is one of the fastest solvers, both for finding optimal and feasible solution to MILP-problems. Gurobi used up to 6 concurrent threads, corresponding to the number of physical cores on our machine. Z3 remained single threaded, thereby explaining a large part of the initial performance difference. However, the relative difference in runtime increases, indicating that the lack of multithreading is not the only reason why Z3 is less performant than Gurobi in our experiment.

[3] https://github.com/Z3Prover/

[4] Runtimes reported in Table 4.4 may be different because the analyzed task graphs vary.

Related Work

Voss and Schätz [113] use SMT-solvers to find both a suitable task placement and schedules for each device. They target systems with a bus-based communication channel that has a fixed bound on latency, whereas our model adopts a fine-grained communication model on the level of the individual TDMA-slots. The novelty of their approach lies in the binary search over a target parameter (i.e.,the makespan) to reach a near-optimal solution. As such, their work is representative of other research in the area of system synthesis. System synthesis covers the allocation of computational and communication resources as well as the binding of tasks and messages to those resources together with their scheduling. Reimann et al. . [95] offer an approach based on SMT-principles in the presence of nonlinear constraints which often stem from nonfunctional requirements, e.g.,power consumption or thermal characteristic of systems. Their approach combines efficient SAT-solvers with domain specific background theories that evaluating those complex nonfunctional constraints. Reimann et al. thereby implemented an indirect SMT approach which allows the integration of external analysis tools and accelerates the synthesis problem by identifying infeasible regions in the state space instead of only identifying singular infeasible states. Compared to the more focused feasibility problem inherent in rtSOA scheduling, the area of system synthesis solves a more general optimization problem which is only indirectly applicable to reconfigurable manufacturing systems. System synthesis occurs during the design time of embedded systems, which is less frequent than the reconfiguration expected to occur in modular production systems. Therefore, the solvers employed to find high quality solutions to the system synthesis problem may run for a longer time span. In contrast, the response time of the rtSOA planner should lie within the human attention span to realize an iterative and interactive development and engineering tool (c.f. Section 3.1).

Other approaches to state space exploration have been built around principles from game theory [18, 20]. In these approaches, the system and its environment are modeled as two competing parties which alternate taking turns. In the approach of Cassez et al. [18] the game is solved by UPPAAL-TiGA [8] which employs model checking techniques to the area of timed game automata. This approach is more general than the feasibility problem solved by rtSOA, but comes at the price of higher runtime complexity and the known scalability issues of model checkers. The work of Cheng et al. [20] synthesized control software from solutions to the game representation identified with a specialized solver called GAVS+ [21] which implements different solving algorithms, mostly based on reachability, depending on the type of game. The runtime of this solver may also suffer from the state space explosion problem.

Other possible state space exploration approaches are based on the A* algorithm. While faster than a simple branch-and-bound algorithm, the evaluation by Kwok and Ahmad [70] indicates that this approach may not be feasible for larger task graphs, as the solving time required for graphs with 32 tasks approaches multiple days in their 2005 paper. Although modern computer hardware provides significantly

more performance, runtimes of multiple days point to a general scalability issue. A performance comparison for multiprocessor task scheduling with communication delays by Jin et al. [49] supports our conclusion. Jin et al. evaluated approaches based on genetic algorithms, simulated annealing, tabu search, the A* algorithm and domain specific heuristics. The implementation of A* in the paper was modified so that the algorithm was applicable to scheduling problems with communication delay. This modification restricted the search space of A*, reducing its runtime complexity but also reducing the quality of the generated solutions. The results by Jin et al. show that the A* based approach was significantly slower than all other methods. In general, the quality of the generated solutions, expressed in terms of the makespan of the resulting schedules, was high enough for the domain specific heuristics so that the increase in runtime for iterative search methods is not justified.

Given the rtSOA design goals (c.f. Section 3.1), domain specific heuristics are a promising alternative to state space exploration, which is in line with the benchmark results by Jin et al. [49]. The next section presents several domain specific heuristics for the rtSOA scheduling problem. These heuristics are compared with the MILP-based state space exploration approach in Section 4.3.

4.2 Heuristic Approaches

The remainder of this section presents different heuristics for the rtSOA scheduling problem, starting with a two-phase approach based on intermediate deadline assignment with subsequent scheduling in Section 4.2.1, followed by a single-phase approach based on distributed scheduling heuristics in Section 4.2.2.

4.2.1 Deadline Assignment Heuristics

The approach based on deadline assignment heuristics solves the task ordering problem in two steps. In the first step a deadline assignment heuristic attaches local deadline and release-time constraints to each task. Afterwards, Potts' heuristic [93] generates a task ordering which is validated through simulation. The scheduling task is performed in a per-machine fashion, meaning that Potts' heuristic is used to find a schedule for the single machine case with release times and deadlines. This section describes the employed deadline assignment heuristics. Figure 4.2 shows an example workflow with a deadline of 10 ms and a total execution time of 5 ms. The deadlines generated by the heuristics described in this section are shown in Table 4.2.

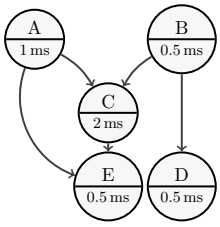


Figure 4.2: Example graph

Node	ED	EQS	EQF	PD	Slice Pure	Slice Norm	Slice Adapt-G	Slice Adapt-L
A	7.00ms	3.67ms	4.00ms	3.33ms	3.50ms	3.75ms	3.67ms	3.77ms
B	7.00ms	3.67ms	4.00ms	3.33ms	3.50ms	3.75ms	3.67ms	3.77ms
C	9.50ms	7.83ms	9.00ms	6.67ms	7.50ms	8.75ms	8.08ms	7.83ms
D	9.50ms	7.83ms	9.00ms	6.67ms	10.00ms	10.00ms	10.00ms	10.00ms
E	10.00ms	10.00ms	10.00ms	10.00ms	10.00ms	10.00ms	10.00ms	10.00ms

Table 4.2: Deadlines assigned to the tasks of the example graph shown to the left. Overall workflow deadline was 10 ms, individual task deadlines are shown below the task name in the graph.

We introduce additional notation for formal definition of the heuristics: The direct predecessor set of a task t_i, defined as $Prec(t_i) = \{t_j \in \mathcal{T} | t_j \prec t_i\}$ with the analogous successor set $Succ(t_i) = \{t_j \in \mathcal{T} | t_i \prec t_j\}$. The set of tasks with in-degree 0, i.e.,the roots of the DAG are specified as $Roots = \{t_i \in \mathcal{T} | Prec(t_i) = \emptyset\}$, with the leafs given by $Leafs = \{t_i \in \mathcal{T} | Succ(t_i) = \emptyset\}$. Additionally, let $\lceil t_i \prec\!\!\!\prec t_j \rceil$ be the length of the longest path from t_i to t_j. The length of $t_i \not\prec\!\!\!\prec t_j$ is ∞, the length of the path from a task t_i to itself is 0 and $\lceil t_i \prec t_j \rceil = 1$. We then define the top-level $t_i.L^t$ of task t_i as $min(\lceil t_r \prec\!\!\!\prec t_i \rceil)$ for $t_r \in Roots$. Analogously the bottom-level $t_i.L^b$ of task t_i is $min(\lceil t_l \prec\!\!\!\prec j_i \rceil)$ for $t_l \in Leafs$. The expression $\mathcal{T}^t_\odot = \{t_j | t_i, t_j \in \mathcal{T} \wedge t_j.L^t \odot t_i.L^t\}$ describes a set of tasks which fulfill a condition on their level L^t where $\odot$ can be any binary operator, e.g.,$\mathcal{T}^t_> = \{t_j | t_i, t_j \in \mathcal{T} \wedge t_j.L^t > t_i.L^t\}$. As a shorthand for all TDMA-slots assigned to the same machine as a given slot introduce the following notation: $Slots(t) = \{s \in \mathcal{S} | \mu(s) = \mu(t_i)\}$

4.2.1.1 Simple Heuristics

The first set of heuristics we implemented was described by Kao and Garcia-Molina [54] for soft real-time tasks with serial-parallel dependencies. We adapted these heuristics to general DAGs by grouping tasks together by their levels, meaning the distance of a task from a root or leaf of the workflow graph.

The simplest heuristic is called *Effective Deadline* (ED), which assigns a deadline to each task that is equal to the global deadline minus the execution time of all succeeding levels. The formal definition is shown in Equation 4.8. It is greedy in the sense that it assigns all available slack to the root tasks of the DAG. Slack is defined as the time span between task release time and deadline minus the task's or level's WCET. This can be seen in the example table where all of the available slack (5 ms) is assigned to tasks in the first level from the top (A and B) which means the following levels, comprising of tasks C and D in level 2 and task E in level 3, are assigned no slack at all.

$$t_i.D = D_{\mathfrak{W}} - \sum_{t \in \mathcal{T}^t_>} |t| \tag{4.8}$$

Equal Slack (EQS) tries to avoid ED's bias by assigning equal amounts of slack to the individual levels of a workflow. The formula for deadline assignment is shown in Equation 4.9, the release times are specified by Equation 4.10. In the running example there are 5 ms of total slack available which are distributed over 3 levels, leaving 1.66 ms per level. These are added to the WCET of all tasks in the level plus the deadline of the previous level.

$$t_i.D = t_i.R + \sum_{t \in \mathcal{T}^t_=} |j| + \frac{D_{\mathfrak{W}} - t_i.R - \sum_{t \in \mathcal{T}^t_\geq} |j|}{max(L^t) - L^t_i + 1} \tag{4.9}$$

$$t_i.R = max(0, t_p.D) \quad \text{for} \quad t_p \in \mathcal{T}^t_< \tag{4.10}$$

Equal Flexibility (EQF) follows a similar strategy, but it scales the amount of slack assigned to a task proportionally to its length, as shown in Equation 4.11 and Equation 4.12. Larger levels receive more slack in EQF. In the example, level 1 receives 40 % of the slack because it makes up 40 % of the total WCET in the workflow.

$$t_i.D = t_i.R + \sum_{t \in \mathcal{T}^t_{=}} |j| + \frac{\left(D_{\mathfrak{W}} - t_i.R - \sum_{t \in \mathcal{T}^t_{\geq}} |t|\right) * \sum_{t \in \mathcal{T}^t_{=}} |t|}{\sum_{t \in \mathcal{T}^t_{\geq}} |t|} \tag{4.11}$$

$$t_i.R = max(0, t_p.D) \quad \text{for} \quad t_p \in \mathcal{T}^t_{<} \tag{4.12}$$

The *Proportional Deadline* (PD) heuristic follows a different strategy. After dividing the graph into n levels it divides the global deadline into n parts of equal length which are assigned to each level. In contrast to the previous heuristics, PD uses the bottom levels of each task, as shown in Equation 4.13. In the example graph with 3 levels, the second level from the top is assigned a deadline that corresponds to $\frac{2}{3}$ of the global deadline.

$$j_i.D = \frac{1 + L^b_i}{1 + max(L^b)} * D_{\mathfrak{W}} \tag{4.13}$$

We modified these heuristics by introducing a corrective factor for communication over the TDMA network. In a first step, the mean communication delay κ is calculated for each machine in the TDMA network. This number specifies the average time a task has to wait for the next available TDMA slot on a given machine. If any task in a level has a successor on a different machine, the maximum κ of the communicating machines is added as a "hidden" level representing a virtual processing time. The *TDMA-modified Equal Slack* (EQS-TDMA) and *TDMA-modified Equal Flexibility* (EQF-TDMA) heuristics then distribute the slack between the normal, non-hidden, levels the same way as their non-modified counterparts. This method leads to an improved effectiveness of the heuristics as shown by the evaluation in Section 4.3.1.

We also propose deadline assignment based on the earliest release and latest finish time (ERT-LFT) of each task. The *earliest release time* (ERT) of each task is determined by traversing the DAG from its sources and calculating the minimum time at which all previous tasks have finished and, if the tasks are located on other machines, have sent their data via the next available TDMA-slot. Similarly, the *latest finish time* (LFT) is obtained by attaching a deadline to each task such that its successor can finish during the global workflow deadline. In the ERT-LFT heuristic, the ERT is used as the release time of a task while the LFT is used as its deadline.

Given the initial values for the latest finish time and earliest start time from Equation 4.14 and the function definition of Equation 4.15, we can specify the latest finish time as shown in Equation 4.16.

$$\begin{aligned} \forall t \in \mathit{Roots} : t.ERT = 0 \\ \forall t \in \mathit{Leafs} : t.LFT = D_{\mathfrak{W}} \end{aligned} \tag{4.14}$$

The LFT will be propagated from the leafs of the DAG to the roots, as defined in *lft_succ*. If two tasks are located on the same machine, the successor task propagates its own LFT minus its WCET. If the tasks are on two different machines we identify the last possible TDMA slot for the predecessor task to communicate with its successor. The start time of that slot is then propagated as LFT to the predecessor task form its successor.

$$\mathit{lft\_succ}(t_i, t_j) = \begin{cases} t_j.LFT - |t_j|, & \text{if } \mu_t(t_i) = \mu_t(t_j) \\ max(s.S) : s.C \leq t_j.LFT - |t_j|, & \text{else for } s \in \mathit{Slots}(t_i) \end{cases} \tag{4.15}$$

Equation 4.16 finally specifies that a task is assigned the minimum propagated LFT from all its successors.

$$\forall t_i \in \mathcal{T}, \forall t_j \in \mathit{Succ}(t_i) : t_i.LFT = min(\mathit{lft\_succ}(t_i, t_j)) \tag{4.16}$$

The earliest release time is specified analogously, but in direction from the DAG roots to the leafs, by propagating the ERT from a predecessor task to the successor task. Equation 4.17 defines this propagation on the same machine and over the network while Equation 4.18 specifies that a task's ERT is the maximum ERT propagated by its predecessors.

$$\mathit{ert\_prec}(t_i, t_j) = \begin{cases} t_j.ERT + |t_j|, & \text{if } \mu_t(t_i) = \mu_t(t_j) \\ min(s.C) : s.S \leq t_j.ERT + |t_j|, & \text{else for } s \in \mathit{Slots}(t_j) \end{cases} \tag{4.17}$$

$$\forall t_i \in \mathcal{T}, \forall t_j \in \mathit{Prec}(t_i) : t_i.ERT = max(\mathit{ert\_prec}(t_i, t_j)) \tag{4.18}$$

4.2.1.2 Slicing technique

Jonsson and Shin [51] presented a set of deadline assignment heuristics based on the slicing technique, which works by identifying the critical path through a DAG based on one of several path metrics. The global deadline is then distributed by assigning non-overlapping execution windows (slices) to the tasks on the critical path. Algorithm 1 shows the deadline distribution algorithm as defined in the original paper. In the following we will outline the different metrics used for finding the critical path in a DAG.

Algorithm 1 Slicing algorithm by Jonsson and Shin [51]

```
1:  function SLICING
2:      T_working ← T
3:      while T_working ≠ ∅ do
4:          Find critical path φ in G that minimizes metric R^slice
5:          Distribute deadline D_φ of φ to all tasks in φ
6:          for all t_i ∈ φ do
7:              for all t_p : t_p ≺ t_i do
8:                  t_p.D = t_i.R
9:              for all t_s : t_i ≺ t_s do
10:                 t_s.R = t_i.D
11:         T_working ← T_working without φ
```

The *Pure* (Slice-Pure) metric is similar to the EQS heuristic in so far as it distributes the available slack equally between all tasks on the critical path, analogously, the *Normalized* (Slice-Norm) metric is similar to the EQF heuristic and scales the assigned slack with the task length. Table 4.2 demonstrates the outcome: The slicing technique will identify the path $A \prec C \prec E$ as the critical path in the workflow. The total slack available on this path is 6 ms, which results in 2 ms of slack being allotted to tasks A, C, E. The next critical path in the graph is $B \prec D$ where 9 ms of slack would be available. However, B must still finish before C, resulting in the same deadline (3.5 ms) being assigned to B and all of the leftover slack being added to D. Slice-Norm works after the same principle but the assigned slack is scaled by the task length. In Equations 4.19 to 4.22 we give the definition of the different metrics R^{slice} used for finding ϕ in Algorithm 1. $|\phi|$ is the number of tasks in the path ϕ.

Slice-Pure metric:

$$R^{slice}_{Pure} = \frac{\left(D_\phi - \sum_{t\in\phi} |t|\right)}{|\phi|} \tag{4.19}$$

$$t_i.D = t_p.D + |t_i| + R_{Pure} \quad \text{for} \quad t_p \in \phi : t_p \prec t_i \tag{4.20}$$

Slice-Norm metric:

$$R^{slice}_{Norm} = \frac{D_\phi - \sum_{t\in\phi} |t|}{\sum_{t\in\phi} |t|} \tag{4.21}$$

$$t_i.D = t_p.D + |t_i| * (1 + R_{Norm}) \quad \text{for} \quad t_p \in \phi : t_p \prec t_i \tag{4.22}$$

The *Globally Adaptive* (Slice Adapt-G) metric only scales the task execution time if the length of a task is over a certain threshold. As in the original paper, we use the mean task execution time [51] as threshold. In the globally adaptive metric, the length of a task is then scaled by a constant factor that depends on the number of machines on which the workflow is executed and a global metric that measures the

degree of parallelism in the workflow. In the example case, tasks A and C would be scaled by the factor 1.5, because they meet or exceed the mean execution time of 1 ms. Working with these virtual execution times, the total remaining slack to distribute along the path is $10\,\text{ms} - (1.5\,\text{ms} * 1.5) - (2\,\text{ms} * 1.5) - 0.5\,\text{ms} = 2.25\,\text{ms}$. Thus, the resulting deadline for task A is $(1.5\,\text{ms} * 1.5) + \frac{4.25\,\text{ms}}{3} \approx 3.67\,\text{ms}$. Virtual task execution times for the globally adaptive slicing are defined in Equation 4.23, the deadline assignment in Equation 4.24.

$$|t_i^{virt}| = \begin{cases} |t_i| & \text{if} |t_i| < thres \\ |t_i| * (1 + k_G * \xi / |\mathcal{M}|) & \text{if} |t_i| \geq thres \end{cases} \tag{4.23}$$

$$t_i.D = t_p.D + |t_i^{virt}| \quad \text{for} \quad t_p \in \phi : t_p \prec t_i \tag{4.24}$$

The *Locally Adaptive* (Slice Adapt-L) metric scales the task execution length based on the local level of task parallelism instead of scaling by a global metric for parallelism. For example, the degree of parallelism for task A in Figure 4.2 is 2 because there are no data dependencies with tasks B and D. As with the Adapt-G metric, tasks A and C are assigned a longer virtual execution time and the remaining slack is distributed along the critical path. Equation 4.25 defines how virtual task execution times are calculated in the locally adaptive slicing metric, Equation 4.26 defines the deadlines assigned by this heuristic.

$$|t_i^{virt}| = \begin{cases} |t_i| & \text{if} |t_i| < thres \\ |t_i| * (1 + k_L * |\Psi_i| / |\mathcal{M}|) & \text{if} |t_i| \geq thres \end{cases} \tag{4.25}$$

$$t_i.D = t_p.D + |t_i^{virt}| \quad \text{for} \quad t_p \in \phi : t_p \prec t_i \tag{4.26}$$

Jonsson and Shin [51] set the release times of each task to the deadline of the previous task in their original paper. We relax those release times to the Earliest Release Time (ERT) of each task, because our benchmarks have shown that the traditional method is too restrictive for the rtSOA scheduling problem. The presented heuristics are evaluated in Section 4.3 together with more advanced heuristics which we present in the next section.

4.2.2 Distributed Scheduling Heuristics

Scheduling algorithms for tasks with communication usually comprise two different phases [29]. At First a task selection phase, also called the prioritization phase, takes place and determines which task should be scheduled next. The second phase, a processor selection phase, determines the processor on which the task should be executed. In our scenario, the processor selection is fixed a priori, which means we focus on the task ordering mechanism of each heuristic.

The *Heterogeneous Earliest Finish Time* (HEFT) [108] and *Dominant Sequence Clustering* (DSC) [121] heuristics are examples for list-scheduling algorithms that maintain a fixed priority list of tasks which is calculated once. Both heuristics use the length of the longest path (in terms of WCET and communication time) from a task t, including the communication times, to a sink of the DAG for their task prioritization. We will use $exit(t)$ to denote this path. HEFT simply ranks tasks by increasing $exit(t)$ (c.f. Algorithm 2). The DSC heuristic uses the ERT of t plus $exit(t)$ as the priority of t as shown in Algorithm 3.

Algorithm 2 Heterogeneous Earliest Finish Time (HEFT) heuristic

function SCHEDULEAFTER($t,\ release_t,\ \mathcal{T}^{sched}$)
  $blockedTime \leftarrow 0$
  for all $t_{sched} \in \mathcal{T}^{sched}$ **do**
    if $\mu_t(t) = \mu_t(t_{sched}) \wedge t_{sched}.C > release_t$ **then**
      $blockedTime \leftarrow max(blockedTime,\ t_{sched}.C - release_t)$
  $t.S = release_t + blockedTime$
  return $\{\mathcal{T}^{sched} \cup t\}$

function HEFT($\mathcal{T}$)
  $\mathcal{T}^{sched} \leftarrow \emptyset$
  $\forall t \in \mathcal{T}$ calculate $exit(t)$ and $t.ERT$
  for all $t \in \mathcal{T}$ sorted by decreasing $exit(t)$ **do**
    $\mathcal{T}^{sched} \leftarrow$ SCHEDULEAFTER($t,\ t.ERT,\ \mathcal{T}^{sched}$)

Algorithm 3 Dominant Sequence Clustering (DSC) heuristic

function DSC($\mathcal{T}$)
  $\mathcal{T}^{sched} \leftarrow \emptyset$
  $\forall t \in \mathcal{T}:$ calculate $t.ERT$
  $\forall t \in \mathcal{T} : blevel(t) = exit(t)$

  while $\mathcal{T} \setminus \mathcal{T}^{sched} \neq \emptyset$ **do**
    $\forall t \in \mathcal{T} : tlevel(t) = t.ERT$
    $t \leftarrow \arg\max(tlevel(t) + blevel(t))$ over $t \in \mathcal{T} \setminus \mathcal{T}^{sched}$
    $\mathcal{T}^{sched} \leftarrow$ SCHEDULEAFTER($t,\ t.ERT,\ \mathcal{T}^{sched}$)
    $\forall t \in \mathcal{T} \setminus \mathcal{T}^{sched}$ calc $t.ERT$ using $t_{sched}.S$ as $t.ERT$ for all $t_{sched} \in \mathcal{T}^{sched}$

The *Mobility Directed* (MD) [120] heuristic chooses tasks based on their mobility, defined as the difference between a task's LFT and its earliest start time (EST), divided by the task's WCET. Although the EST is similar to the ERT, it is recalculated after each task selection and takes into account the scheduling time of the other workflow tasks. The EST is calculated in line 7 of Algorithm 4.

Algorithm 4 Mobility Directed (MD) heuristic

1: **function** MD($\mathcal{T}$)
2: $\quad\mathcal{T}^{sched} \leftarrow \emptyset$
3: $\quad\forall t \in \mathcal{T}:$ calculate $t.ERT$ and $t.LFT$
4: $\quad$**while** $\mathcal{T} \setminus \mathcal{T}^{sched} \neq \emptyset$ **do**
5: $\quad\quad t \leftarrow \arg\max(\frac{t.LFT - t.ERT}{|t|})$ over $t \in \mathcal{T} \setminus \mathcal{T}^{sched}$
6: $\quad\quad\mathcal{T}^{sched} \leftarrow$ ScheduleAfter($t,\ t.ERT,\ \mathcal{T}^{sched}$)
7: $\quad\quad\forall t \in \mathcal{T} \setminus \mathcal{T}^{sched}$ calc $t.ERT$ using $t_{sched}.S$ as $t.ERT$ for all $t_{sched} \in \mathcal{T}^{sched}$

Earliest Task First (ETF) [45] picks a task among the ready tasks, meaning tasks whose predecessors have already been scheduled, by choosing the task with the minimum EST. Ties are broken by the task with the smallest LFT minus WCET as shown in Algorithm 5.

Algorithm 5 Earliest Task First (ETF) heuristic

1: **function** ETF($\mathcal{T}$)
2: $\quad\mathcal{T}^{sched} \leftarrow \emptyset$
3: $\quad\forall t \in \mathcal{T}:$ calculate $t.ERT$ and $t.LFT$
4: $\quad$**while** $\mathcal{T} \setminus \mathcal{T}^{sched} \neq \emptyset$ **do**
5: $\quad\quad ReadyTasks \leftarrow \arg\min(t.ERT)$ over $t \in \mathcal{T} \setminus \mathcal{T}^{sched}$
6: $\quad\quad t \leftarrow \arg\min(t.LFT - |t|)$ over $t \in ReadyTasks$
7: $\quad\quad\mathcal{T}^{sched} \leftarrow$ ScheduleAfter($t,\ t.ERT,\ \mathcal{T}^{sched}$)
8: $\quad\quad\forall t \in \mathcal{T} \setminus \mathcal{T}^{sched}$ calc $t.ERT$ using $t_{sched}.S$ as $t.ERT$ for all $t_{sched} \in \mathcal{T}^{sched}$

We propose two additional scheduling heuristics: an adapted version of Potts' heuristic [93] that works in a distributed environment with fixed task-placement, and the *Least Delay* heuristic (LD). Our proposed LD heuristic tries to determine the implications of scheduling each task in the ready set. The heuristic schedules each of the tasks in the ready set in a "what-if" manner and determines the EST of all tasks in the DAG based on this speculative scheduling. The resulting EST of each sink task is compared to its EST before the speculative scheduling, yielding a value for the expected $delay_t$ of the sink task t. The maximum $delay_t$ yields the delay for the entire workflow. The heuristic now chooses the task from the ready set that resulted in the minimum workflow delay. This heuristic has a high run-time complexity, but our evaluation (Section 4.3.1) shows it can generate solutions in many cases where the other heuristics failed to do so. The pseudocode is given in Algorithm 6.

The adapted Potts' heuristic is set up by picking the task with the minimum LFT from the available ready set, meaning the current time on the task's machine is equal or greater than the task's EST. This initial step is also known as Schrage's heuristic [93]. In most cases, Schrage's heuristic yields valid schedules, meaning the DAG sinks do not violate the workflow deadline in the schedules. If the first pass of Schrage's heuristic was unsuccessful, Potts' heuristic then analyzes the resulting schedules and looks for a task A that violates its LFT. A is called the critical task. This means there could be another task B with a smaller EST than A but with a larger latest starting time (LST) scheduled before A because A was not ready at that moment. If such a task B, also called the interference task, exists on the same machine as A, we introduce an additional edge in the DAG from A to B to ensure that A will be scheduled before B. If no such task exists on the same machine as the critical task, we look for an interference task B' on a different machine. We then take B' as the new critical task and try to locate an interference task C on the same machine as B'. If C exists, we introduce an additional edge from C to B' and continue as previously described. The modified workflow is then rescheduled with Schrage's heuristic. Pseudocode for the modified Potts' heuristic is shown in Algorithm 7.

Algorithm 6 Least Delay heuristic (LD)

1: **function** LEASTDELAY($\mathcal{T}$, $\mathcal{G}$)
2: $\mathcal{T}^{sched} \leftarrow \emptyset$
3: $\forall t \in \mathcal{T}:$ calculate $t.ERT$ and $exit(t)$
4: **while** $\mathcal{T} \setminus \mathcal{T}^{sched} \neq \emptyset$ **do**
5: $\mathcal{T}^{tmp} \leftarrow \mathcal{T}^{sched}$
6: $minDelay \leftarrow \infty$
7: $t_{minDelay} \leftarrow \emptyset$
8: **for all** $t_{root} \in \mathcal{T} \setminus \mathcal{T}^{sched} : Prec(t_{root}) = \emptyset$ **do** ▷ Unscheduled roots
9: $\mathcal{T}^{tmp} \leftarrow$ SCHEDULEAFTER(t_{root}, $t_{root}.ERT$, $\mathcal{T}^{tmp}$)
10: $Tmp \leftarrow \forall t \in \mathcal{T} \setminus \mathcal{T}^{tmp}$ calc. $t.ERT$ using $t.S$ as $t.ERT$ for $t \in \mathcal{T}^{tmp}$
11: $delay \leftarrow max(Tmp(t_{leaf}) - t.ERT)$ for $t_{leaf} \in \mathcal{T} \setminus \mathcal{T}^{sched} : Succ(t_{leaf}) = \emptyset$
12: **if** $delay < minDelay \vee (delay = minDelay \wedge maxLen < exit(t_{root}))$ **then** ▷ Scheduling current root leads to a smaller delay in the leafs than before
13: $t_{minDelay} \leftarrow t_{root}$
14: $minDelay \leftarrow delay$
15: $maxLen \leftarrow exit(t_{root})$
16: $\mathcal{T}^{sched} \leftarrow$ SCHEDULEAFTER($t_{minDelay}$, $t_{minDelay}.ERT$, $\mathcal{T}^{sched}$)
17: $\forall t \in \mathcal{T} \setminus \mathcal{T}^{sched}$ calc $t.ERT$ using $t_{sched}.S$ as $t.ERT$ for all $t_{sched} \in \mathcal{T}^{sched}$

Algorithm 7 Modified Potts' heuristic for distributed systems

```
1: function Potts(𝒯)
2:     AbsoluteEarliest ← ∀t ∈ 𝒯 : calculate t.EST
3:     for i ≤ number of tasks in 𝒯 do
4:         𝒯^Schrage ← SchragesHeuristic(𝒯)
5:         if 𝒯^Schrage is a valid schedule then ▷ Valid if t.C of all DAG leafs ≤ D_𝔚
6:             return 𝒯^Schrage
7:         else
8:             interference ← IdentifyInterference(𝒯^Schrage)
9:             if interference ≠ ∅ then
10:                add new precedence relation crit ≺ interference to 𝒢
11:            else
12:                return Infeasible with Potts' heuristic
13:    return Infeasible with Potts' heuristic
14: function SchragesHeuristic(𝒯)
15:    𝒯^sched ← ∅
16:    ∀t ∈ 𝒯 : calculate t.LFT
17:    while 𝒯 \ 𝒯^sched ≠ ∅ do
18:        for all t_i ∈ 𝒯 \ 𝒯^sched ordered by increasing t.LFT do
19:            machineClock ← max(t_j.C) over t_j ∈ 𝒯^sched ∧ μ_t(t_i) = μ_t(t_j)
20:            if t_i.ERT ≥ machineClock then
21:                t ← t_i
22:                break for-loop
23:        𝒯^sched ← ScheduleAfter(t, t.ERT, 𝒯^sched)
24:        ∀t ∈ 𝒯 \ 𝒯^sched calc t.ERT using t_sched.S as t.ERT for all t_sched ∈ 𝒯^sched
25:    return 𝒯^sched
26: function IdentifyInterference(𝒯)
27:    interference ← ∅
28:    for all t ∈ 𝒯 do                                  ▷ Identify critical task
29:        if (t.C > t.LFT ∧ t.S > AbsoluteEarliest(t)) ∨ t.C > D_𝔚 then
30:            crit ← t          ▷ t violates its own deadline or the workflow deadline
31:            break for-loop
32:    for all t ∈ 𝒯 do          ▷ Identify interference task delaying the critical task
33:        if μ_t(t) = μ_t(crit) ∧ crit.LFT < t.LFT ∧ t ⋠ crit then
34:            interference ← t                ▷ t violates local or workflow deadline
35:            break for-loop
36:    if interference ≠ ∅ then return interference
37:    crit ← arg max(t.ERT − AbsoluteEarliest(t)) over t ∈ Prec(crit)
38:    for all t ∈ 𝒯 do
39:        if μ_t(t) = μ_t(crit) ∧ crit.LFT < t.LFT ∧ t ⋠ crit then
40:            interference ← t
41:            break for-loop
42:    return interference
```

4.3 Evaluation and Discussion

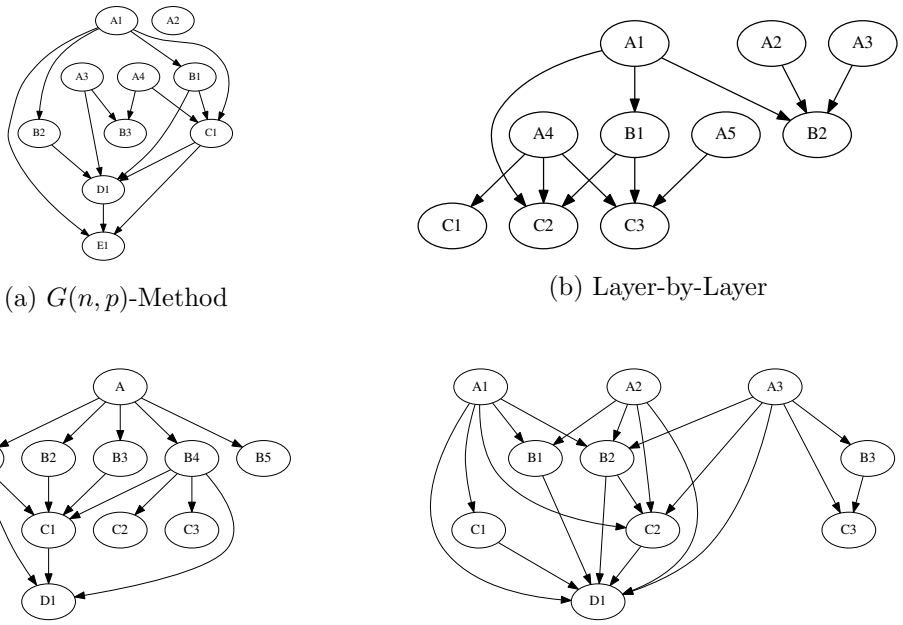
Heuristics approach the rtSOA scheduling problem from a different angle than solutions based on state space exploration. Instead of trying to find a feasible task ordering somewhere in the state space of the problem, heuristics directly generate a "reasonable" task order based on certain rules that are successful in a large number of cases. Heuristics prove the existence of a feasible solution by construction, but cannot be used to prove the absence of a feasible schedule. Lower-bound calculations can exclude the existence of a feasible solution in some cases, but an area of uncertainty remains where only approaches based on state space exploration can prove the infeasibility of a given scheduling problem. When comparing different heuristics with each other, and with a state space exploration approach, we measure the effectiveness of each approach. It is defined as:

> *The effectiveness of a heuristic over a set of feasible scheduling problems is the percentage of all problem instances in the set for which the heuristic found a feasible task ordering in regards to a given deadline and network configuration.*

The effectiveness of state space exploration techniques, given enough time, is always 100 %. Because heuristics only explore a small percentage of the solution space, their effectiveness is smaller, but they are much cheaper computationally. This thesis shows that all heuristics combined together have an effectiveness of almost 99 % when applied to our set of test cases.

Since industrial use cases span a wide range of potential layouts of the resulting task graphs, we rely on synthetic benchmarks, based on several well-known graph generation methods [25]. The *Erdős-Rènyi* $G(n, p)$ method generates an unbiased DAG out of all possibilities, therefore our benchmarks contain this form of task graph weighted with 50 %. The *Layer-by-Layer* method allows specifying the maximum depth of the graph and was developed specifically for validation of scheduling algorithms. It is contributing 20 % to the overall number of test cases. Similarly, *Task Graphs for Free* (TGFF) is another method of generating task graphs for the validation of scheduling methods. It is also weighted with 20 %. The *Random Orders* method generates a partial order (i.e.,a DAG) by intersecting several total orders, which are constructed by shuffling the nodes of the graph. This method produces graphs with all transitive edges and is used for generating the last 10 % of the test cases. Figure 4.3 shows examples of the graphs generated by these methods.

We generated a total of 1 200 000 feasible workflows, in the aforementioned ratio, with either 16, 32, 48 or 64 tasks which were randomly assigned to either 2, 4 or 8 machines connected via TDMA. Random workflows were generated by the graph generation methods shown in Figure 4.3 and scheduled with Gurobi and the MILP-formulation described in Section 4.1.2 until we had found 1.2 mio feasible test cases



(a) $G(n, p)$-Method

(b) Layer-by-Layer

(c) Task Graphs for Free

(d) Random Orders

Figure 4.3: Example output of the used graph generation methods

with the described distribution. To avoid bias in the evaluation, the amount of feasible workflows is the same for each combination of machine count and task count, i.e.,there are 100 000 workflows for each combination of number of machines and number of tasks in a workflow. The workflows all have a common deadline and period of 10 ms, which is also the length of a TDMA round. TDMA slots of 120 µs length (the time needed to transmit 1500 bytes, which is the largest allowed UDP packet size, over 100 Mbit Ethernet) were assigned in round-robin style to the machines. The workflows were then run through the heuristics pipeline, described in Section 3.3, to determine which percentage of feasible solutions a heuristic can find, i.e.,the effectiveness measure defined at the beginning of this section. The effectiveness of the state space exploration-approach naturally is 100 %, therefore it is not shown in the performance plots below. A detailed description of how the benchmark set was generated and how the individual attributes, against which we measure the heuristics' effectiveness below, are distributed in the benchmark set is given in Section A.3.

In the following we will evaluate the effectiveness of the presented heuristics in Section 4.3.1 and analyze the influence of different attributes, such as the number of machines or the size of the workflow, on the heuristics' performance in Section 4.3.2. The runtime characteristics of the presented heuristics are evaluated in Section 4.3.3 before we give our overall conclusions on the presented evaluation in Section 4.3.4.

Heuristic	Effectiveness
ED	61.44%
EQS	75.91%
EQF	75.92%
PD	75.12%
EQS-TDMA	79.50%
EQF-TDMA	79.50%
Slice Pure	74.24%
Slice Normalized	72.11%
Slice Adapt-L	74.46%
Slice Adapt-G	74.24%
ERT-LFT	90.70%
HEFT	80.31%
DSC	92.65%
MD	86.05%
ETF	94.44%
Potts	95.39%
LD	92.47%
combined	99.00%

Table 4.3: Overall effectiveness of heuristics

4.3.1 Effectiveness

Table 4.3 shows the overall effectiveness of the presented heuristics in finding feasible execution plans (schedules) together with a separate scheduling phase. The simple Effective Deadline (ED) heuristic is only able to find feasible schedules in 60 % of the examined cases. Equal Slack (EQS) and Equal Flexibility (EQF) perform nearly identical at almost 76 % effectiveness. The reason for this is that they do not lead to different task orderings after scheduling with Potts' algorithm. The release times and deadlines generated by both EQS and EQF traverse the task graphs by the top-level $t_i.L^t$ of the tasks. The minor differences between both heuristics are caused by random ordering of tasks within the same level. Analogously, the Proportional Deadline (PD) heuristic traverses task graphs by the bottom-level $t_i.L^b$ of the tasks. It has a slightly lower effectiveness at 75 %. The Slicing heuristics have mostly identical performance, with the normalized slicing heuristic performing a bit worse than the others. However, they are the worst overall deadline assignment heuristics, apart from the ED heuristic. Our TDMA-modified Equal Slack (EQS-TDMA) and TDMA-modified Equal Flexibility (EQF-TDMA) heuristics also traverse the task graph in the same way as the unmodified versions. The modification of adding the average time between TDMA-slots on each machine in the network has lead to an increase of the heuristics' effectiveness. EQS-TDMA and EQF-TDMA both have an overall effectiveness of 79.5 %. Our next new deadline assignment heuristic, Earliest Release and Latest Finish Time (ERT-LFT), has by far the highest effectiveness of

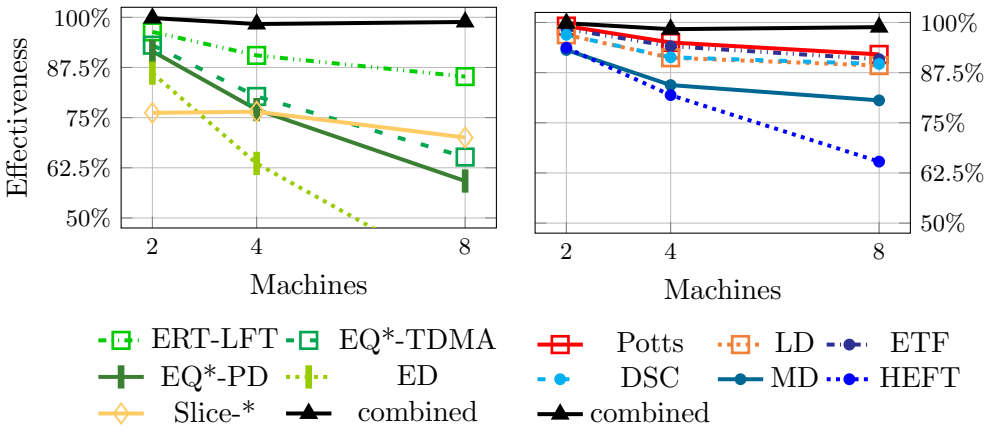
all examined deadline assignment heuristics at 90.7 %. This heuristic attaches the most relevant set of timing constraints on the individual task level when compared with the other deadline assignment heuristics.

Apart from ERT-LFT, the deadline assignment heuristics perform worse than the distributed scheduling heuristics. The performance of HEFT is similar to the effectiveness of the modified EQS-TDMA and EQF-TDMA heuristics. Heterogeneous earliest finish time (HEFT) is a very simple heuristic as it only ranks tasks by the length of the longest path from a task to an output (or exit) node of the graph. This is denoted as $exit(t)$ (c.f. Section 4.2.2). HEFT neglects to take the earliest release time (ERT) into account. The dominant sequence clustering (DSC) heuristic also uses $exit(t)$ but additionally considers the ERT of a task when prioritizing tasks. Tasks which have a long $exit(t)$ and a late ERT are prioritized because they are more critical. DSC therefore performs with 92.6 % effectiveness, which are 12 % more than offered by HEFT. The mobility directed (MD) heuristic has moderate effectiveness at 86 %. MD only considers paths to a task in its scheduling decisions, not paths from a task to an leaf of the graph. The best scheduling heuristic from literature is the earliest task first (ETF) heuristic, reaching 94.4 % effectiveness. It mostly considers tasks' release times for prioritization, but uses the latest finish time of a task as a tie-breaker.

Our modification to Potts' heuristic is the most effective heuristic at over 95 % effectiveness. The heuristic iterates over the scheduling problem multiple times and tries to eliminate scheduling conflicts if it detects an infeasible outcome. Our second new distributed scheduling heuristic, the least delay (LD) heuristic, reaches nearly 92.5 % effectiveness making it the fourth best heuristic ranked by effectiveness. Overall, the heuristics found feasible solutions for 99 % of all generated problem instances, meaning only 1 % of all instances were unsolvable through heuristics. This high overall effectiveness shows the general feasibility of using heuristics to generate schedule based execution plans.

In the following we will study the impact of different attributes of the scheduling problem on the heuristics effectiveness. We have grouped the EQS, EQF and PD heuristics together under the name EQ*-PD, as there is barely any difference between their performance, no matter the examined attribute of the scheduling problem. For the same reason, we have grouped the EQS-TDMA and EQF-TDMA heuristics together under the label EQ*-TDMA. The same behavior can be observed for the slicing heuristics which are grouped together under the label Slice-*.

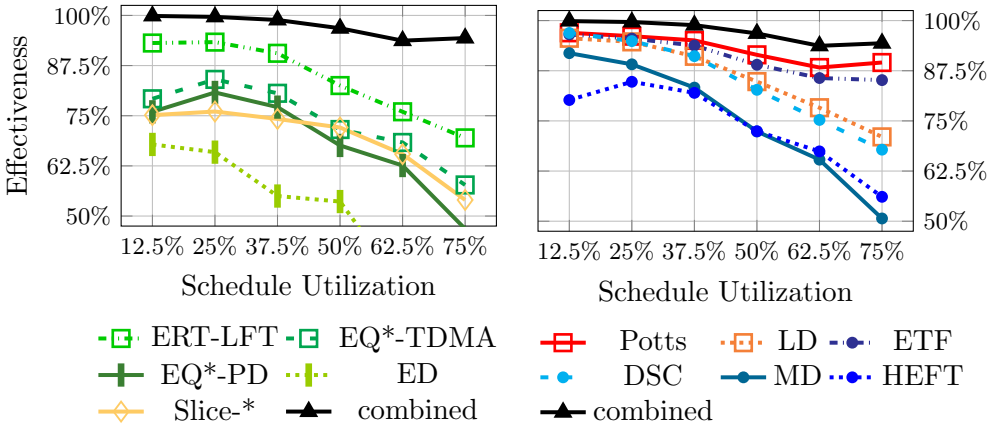
We first consider the impact of the number of devices participating in the execution plan on the effectiveness of the heuristics. This relation is plotted in Figure 4.4. For simple cases with only 2 machines, all heuristics show good performance. These cases reduce the impact of the network and of inter-machine parallelism on the outcome. The single machine case of scheduling tasks with precedence constraints is solvable in polynomial time [72]. As the number of machines in the execution plan increases, the impact of good heuristics on the outcome rises. The simplistic ED heuristic drops



(a) Deadline assignment heuristics (b) Distributed scheduling heuristics

Figure 4.4: Effectiveness of heuristics plotted against the number of machines in the network. The "combined" plot line contains all heuristics.

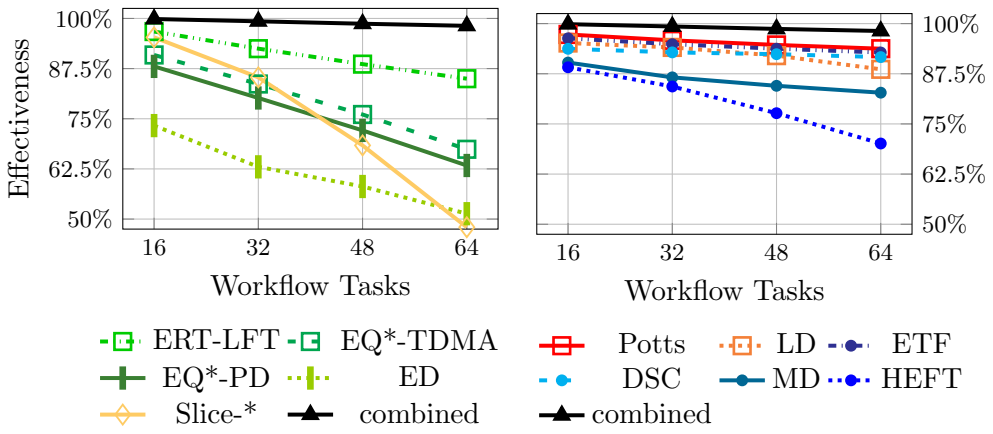
to 35 % effectiveness in the 8 machine case whereas the ERT-LFT heuristic, as the best of the deadline assignment heuristics, is able to maintain an effectiveness of over 85 %. The HEFT heuristic struggles to reach high effectiveness in situations with many machines as well. The slicing heuristics start with moderate effectiveness but their performance remains relatively stable. Another group of scheduling heuristics (DSC, ETF, LD and Potts') also shows stable performance and remains near 90 % effectiveness in the 8 machine case, outperforming the slicing heuristics. The gap between the best heuristics and the combined effectiveness widens as more machines are added to the scheduling problem. All heuristics together achieve over 98.8 % effectiveness in the 8 machine case.



(a) Deadline assignment heuristics (b) Distributed scheduling heuristics

Figure 4.5: Effectiveness of heuristics plotted against the schedule utilization

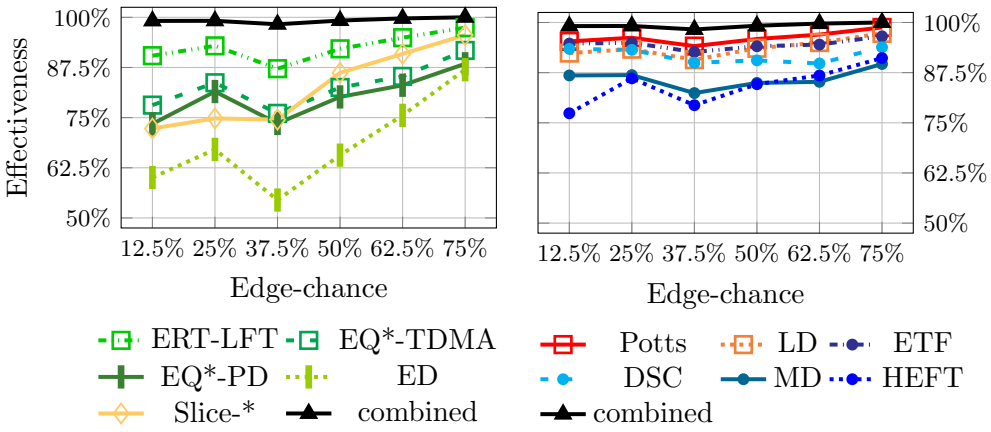
Similarly, the schedule utilization has a large impact on the effectiveness of the examined heuristics. Figure 4.5 expresses the schedule utilization as a relative percentage. 50 % schedule utilization in a problem instance with 4 machines and 10 ms deadline would represent a total of 20 ms of processing time distributed among the machines. Time spent waiting on TDMA-slots and actually transmitting messages via TDMA is not included in the schedule utilization. The utilization impacts the heuristic effectiveness so much because it reduces the resilience to inefficiency. An inefficient task ordering produced by a heuristic could still be feasible in a problem instance with low schedule utilization whereas an instance with high utilization would be less forgiving. Our modified EQS-TDMA and EQF-TDMA heuristics outperform the deadline assignment heuristics published in literature, but they do not perform satisfactorily in high-utilization situations. The Slicing heuristics perform similar to the other deadline assignment heuristics. ERT-LFT has an all round better performance than the other deadline-assignment heuristics. However, its performance still drops as the schedule utilization rises. This reduction in effectiveness is also seen with most distributed scheduling heuristics. Only ETF and our modified Potts' heuristic are able to achieve a relatively stable performance in situations with high schedule utilization. ETF and modified Potts' are the only two heuristics that consider the minimum LFT of a set of ready tasks as a metric during scheduling.



(a) Deadline assignment heuristics (b) Distributed scheduling heuristics

Figure 4.6: Effectiveness of heuristics plotted against the number of workflow tasks

Figure 4.6 shows that the effectiveness of all heuristics decreases as the number of tasks in a workflow grows larger. This behavior is expected because the state space in which the feasible solution could lie increases with the number of tasks. Each heuristic only examines a single "state" for each run, so naturally the chances of this state being a feasible solution are reduced with an increasing number of states. Apart from HEFT, all distributed scheduling heuristics are relatively stable in their effectiveness. The Slicing heuristics show the biggest drop in effectiveness, meaning that they are not well suited for larger task graphs.

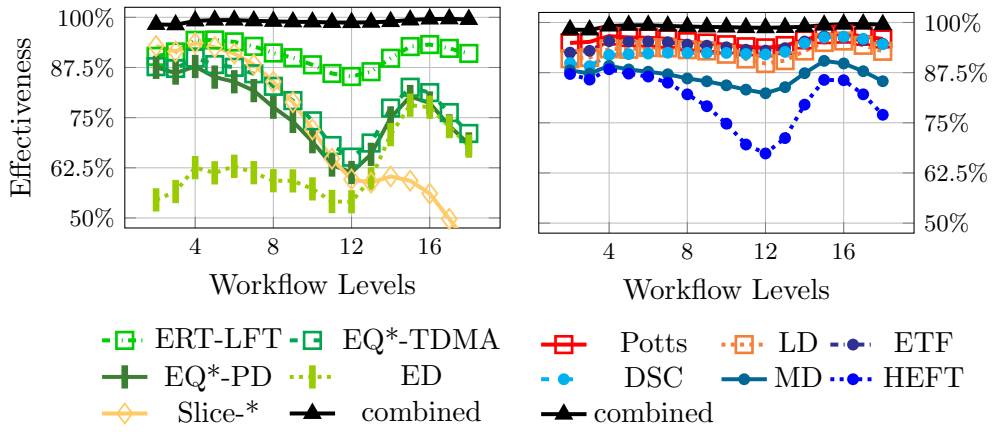


(a) Deadline assignment heuristics (b) Distributed scheduling heuristics

Figure 4.7: Effectiveness of heuristics plotted against the edge generation chance

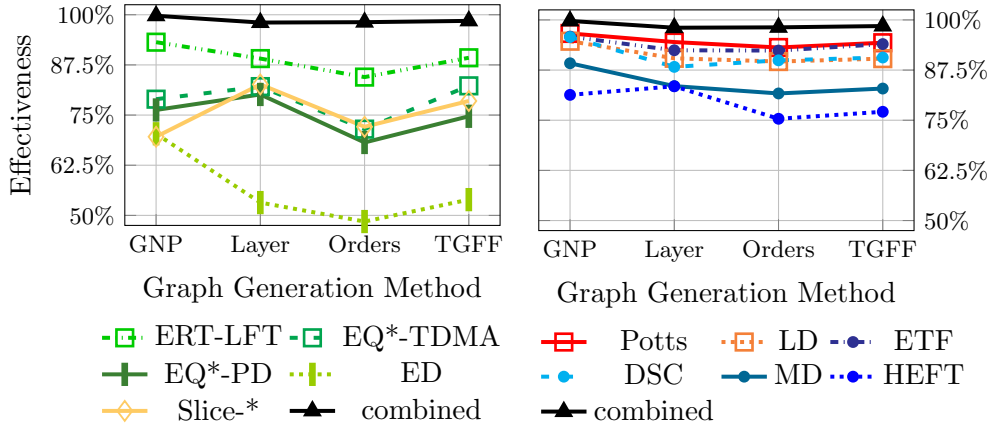
Figure 4.7 examines the effectiveness of our heuristics against the number of edges in the task graph. We have expressed this property as the edge generation chance which denotes the percentage out of all theoretically possible edges being present in a given workflow. An edge generation chance of 100 % would mean a fully connected DAG. This relative attribute reduces the bias that would be introduced by choosing the number of edges directly, because graphs with a high number of edges tend to have a higher number of tasks than graphs with a lower number of edges. The number of edges only has a small influence on the overall effectiveness of the heuristics. When more edges are added to a task graph, the search space for the scheduling problem is reduced because the partial order represented by the task DAG is brought closer to a total order of the tasks contained within the DAG. More precedence constraints therefore mean fewer possible task orderings. However, edges in the task DAG not only represent precedence constraints but also stand for data dependencies, thus the amount of network traffic also rises when adding more edges to the task DAG. As such, the network scheduling becomes harder.

Figure 4.8 also indirectly considers network scheduling as a potential source of difficulty. The figure shows the number of layers in a graph on the x-axis. As a graph becomes “deeper”, there are longer chains of messages and fewer potential parallel executions. The heuristics are not very sensitive to this attribute on its own. The decrease in effectiveness of the EQ* heuristics until 12 levels can be explained by the accompanying rise in the average number of tasks in the graph. It is intuitive that DAGs with a smaller number of tasks have fewer layers in them. The average workflow size in our benchmark set stays constant from 12 levels onward. The following increase in effectiveness can be explained by the decreasing average number of machines, from 4.7 to 3.1 machines, which reduces the complexity and delays caused by network communication. An exception to the previous statement are the slicing heuristics which have almost consistently decreasing effectiveness for graphs with more levels, indicating that the way they distribute deadlines along the critical path of a DAG is ineffective.



(a) Deadline assignment heuristics (b) Distributed scheduling heuristics

Figure 4.8: Effectiveness of heuristics plotted against the number of workflow levels



(a) Deadline assignment heuristics (b) Distributed scheduling heuristics

Figure 4.9: Effectiveness of heuristics plotted against the graph generation method

The small decrease in effectiveness around an edge generation chance of 37.5 % in Figure 4.7 is caused by a large number of graphs generated by the Random Orders generation method. These types of graph seem to be harder for the heuristics than graphs generated with other methods, as shown by Figure 4.9. Totally random DAGs generated by the Erdős-Rènyi $G(n, p)$ method are the easiest class of workflows for most heuristics. Both the Layer-by-Layer and Task Graphs for Free graph generation methods, which are designed to represent typical task graphs, pose a similar level of difficulty for the heuristics.

4.3.2 Information Gain

So far we often had to remark on the relative influence of different attributes on the heuristics' effectiveness to explain anomalies in some of the graphs plotting the heuristics' effectiveness against an attribute. To quantify the influence of individual attributes we have performed an analysis of the *information gain* offered by each attribute. In machine learning, the information gain of an attribute refers to the amount of entropy that is reduced by the inclusion of the attribute in a classification method. It is synonymous with the Kullback-Leibler divergence [69]. Figure 4.10 shows the information gain ratios of all examined attributes, plus the sum of all information gain ratios, for all presented deadline assignment heuristics. The information gain ratio is the ratio of information gain in an attribute relative to the intrinsic information in the class. For example, the ERT-LFT heuristic has the intrinsic information that over 90 % of all test cases scheduled with this heuristic will lead to a feasible result. By using the information gain ratio we are reducing the bias towards feasibility inherent in the data set. The results in Figure 4.10 were obtained by extracting a data set for each heuristic, with the feasibility result ("true" or "false") as the class label. We then evaluated the information gain ratio of the attributes in each data set with the WEKA machine learning framework [41].

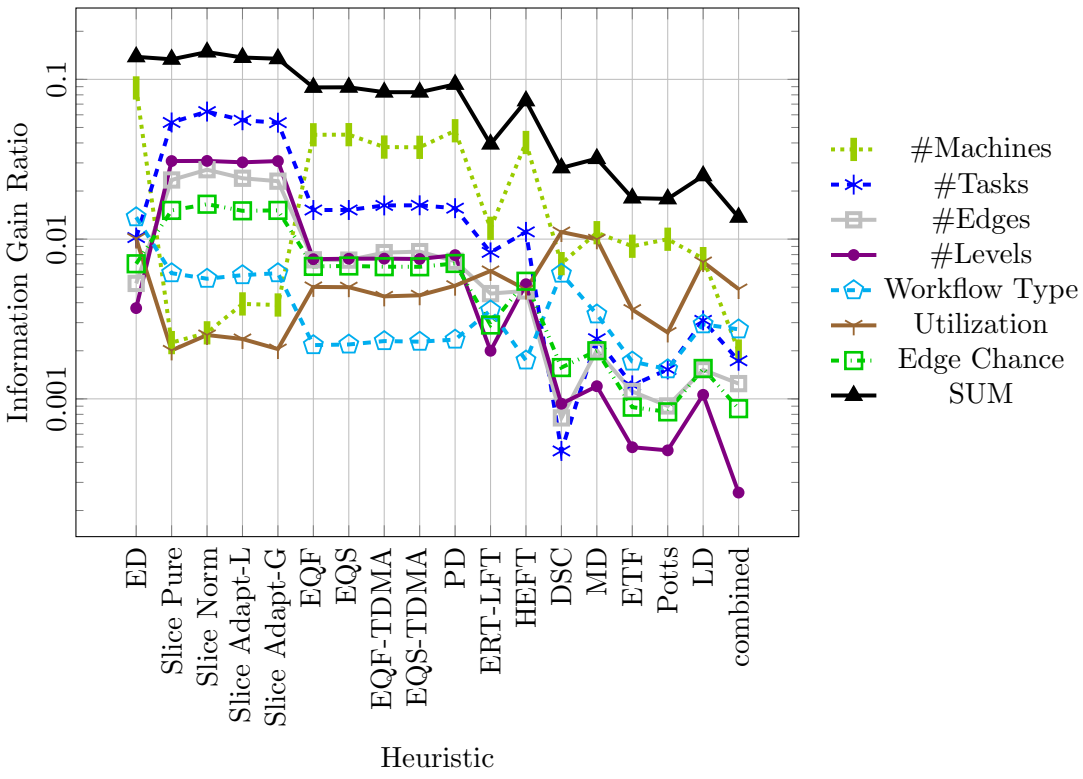


Figure 4.10: The information gain ratio for the measured attributes explains the relative difficulty imposed by each attribute. The Y-axis is scaled logarithmically.

Higher values for information gain of an attribute in Figure 4.10 imply a larger relative difficulty of the attribute for an heuristic. For example, the ED heuristic is highly influenced by the number of machines in the rtSOA scheduling problem. The figure also indicates a high sensitivity of the heuristic to the used workflow generation method, indicating that the heuristic favors certain types of workflows over others. Both observation match our previous evaluation. ED performs poorly as the number of machines increases (c.f. Figure 4.4) and favors graphs generated by the Erdős-Rènyi $G(n, p)$ method over all other graphs (c.f. Figure 4.9).

Figure 4.10 also shows that the number of machines in the execution plan is the most important factor determining the effectiveness of most deadline assignment heuristics, followed by the number of tasks in the workflow. For the more effective heuristic ERT-LFT, the influence of these factors is decreased, but the relative importance of the schedule utilization is rising. In general, the sum of all information gain ratios is a predictor for the effectiveness of a heuristic. Potts' heuristic has the lowest sum, and ED the highest sum. The sum of information gain ratios for a heuristic is negatively correlated with the heuristic's effectiveness. EQS, EQF and PD have a nearly identical sum of information gain ratios, as do EQS-TDMA and EQF-TDMA and the Slicing heuristics. Figure 4.10 also shows that the DSC and MD heuristics are successfully addressing the problems that the simple deadline assignment heuristics are having with a higher number of machines. However, DSC and MD are more sensitive to high schedule utilization than more effective heuristics, such as ETF or Potts. When running all heuristics sequentially, as symbolized by the entry in Figure 4.10 labeled "combined", the influence of the number of machines drops further. The Information gain ratio also explains the difficulties of the Slicing heuristics in our evaluation where they struggle for workflows with many tasks, edges or levels. For the overall schedulability with heuristics, the schedule utilization is the most influential attribute, followed by the graph structure and the number of machines. Other attributes play a minor role and the number of levels in the workflow is nearly irrelevant. This is promising, because it indicates that a combination of heuristics would also have high effectiveness for larger task-graphs running on more machines.

4.3.3 Runtime Comparison

The use of heuristic always implies a trade off between better run-time characteristics and reduced effectiveness. Table 4.4 shows the geometric mean of the heuristics' performance as well as the runtime of the Gurobi MILP-solver. Gurobi did use up to 6 threads in parallel while our heuristics were not parallelized. The measurements were obtained on an Intel Core i7-3930K with 6 physical cores and 64 GB RAM. It is apparent that most heuristics outperform the MILP-solver by two to three orders of magnitude. Even heuristics with a higher run-time complexity, such as the LD heuristic or Potts' heuristic, have a significantly shorter run time than the solver. This situation supports our argument to combine heuristics. Table 4.4 also

[5] Runtimes reported in Table 4.1 may be different because the analyzed task graphs vary.

Algorithm	Valid Workflows					Invalid Workflows				
	16 Tasks	**32 Tasks**	**48 Tasks**	**64 Tasks**	**128 Tasks**	**16 Tasks**	**32 Tasks**	**48 Tasks**	**64 Tasks**	**128 Tasks**
EQS	0.04ms	0.06ms	0.08ms	0.11ms	0.11ms	0.04ms	0.08ms	0.11ms	0.13ms	0.12ms
EQF	0.03ms	0.05ms	0.08ms	0.10ms	0.10ms	0.04ms	0.08ms	0.10ms	0.13ms	0.12ms
EQS-TDMA	0.12ms	0.15ms	0.18ms	0.22ms	0.17ms	0.08ms	0.13ms	0.17ms	0.21ms	0.19ms
EQF-TDMA	0.06ms	0.09ms	0.12ms	0.15ms	0.16ms	0.07ms	0.12ms	0.16ms	0.19ms	0.18ms
PD	0.03ms	0.05ms	0.07ms	0.10ms	0.10ms	0.03ms	0.06ms	0.09ms	0.11ms	0.12ms
Slice Pure	0.18ms	0.60ms	1.76ms	4.56ms	-ms	0.28ms	1.18ms	8.03ms	62.26ms	-ms
Slice Normalized	0.17ms	0.58ms	1.73ms	4.53ms	-ms	0.26ms	1.14ms	7.49ms	54.06ms	-ms
Slice Adapt-L	0.21ms	0.78ms	2.26ms	5.94ms	-ms	0.33ms	1.43ms	9.34ms	68.64ms	-ms
Slice Adapt-G	0.18ms	0.61ms	1.79ms	4.69ms	-ms	0.28ms	1.22ms	8.35ms	63.88ms	-ms
ERT-LFT	0.06ms	0.14ms	0.22ms	0.32ms	0.38ms	0.09ms	0.21ms	0.34ms	0.44ms	0.49ms
HEFT	0.10ms	0.20ms	0.34ms	0.55ms	0.90ms	0.13ms	0.28ms	0.45ms	0.62ms	0.91ms
DSC	0.11ms	0.22ms	0.37ms	0.55ms	0.85ms	0.15ms	0.32ms	0.53ms	0.70ms	0.90ms
MD	0.08ms	0.17ms	0.30ms	0.45ms	0.73ms	0.11ms	0.26ms	0.44ms	0.60ms	0.80ms
ETF	0.09ms	0.24ms	0.46ms	0.75ms	1.40ms	0.13ms	0.35ms	0.62ms	0.88ms	1.49ms
Potts	0.13ms	0.30ms	0.50ms	0.72ms	0.96ms	0.23ms	0.51ms	0.85ms	1.17ms	1.69ms
LD	0.94ms	6.06ms	20.06ms	49.71ms	228.51ms	1.21ms	6.55ms	18.46ms	36.32ms	206.58ms
Gurobi	5.34ms	35.60ms	106.78ms	226.88ms	2,508.05ms	8.51ms	27.57ms	61.21ms	95.62ms	219.30ms

Table 4.4: Geometric averages of the runtime spent by the presented heuristics and the Gurobi MILP solver on scheduling problems of varying workflow size, split into valid and invalid workflows.[5]

shows that most heuristics need slightly more time for offering potential solutions when workflows are invalid, as these graphs tend to be more complex. In contrast, Gurobi is often able to quickly prove the infeasibility of such configurations from the mathematical structure of the resulting MILP. Table 4.4 also shows that the Slicing heuristics are a poor choice for larger and more complex workflows as their runtime requirements rise rapidly. This is caused by the search for the critical path in those heuristics, which necessitates the enumeration of all possible paths from a root to a leaf of the task graph.

The cumulative runtime distribution of the LD and Potts' heuristic as well as the MILP-solver are shown in Figure 4.11. For workflows with 16 tasks, Potts' runtime varies in a relatively narrow band since it only rarely needs to reschedule after identifying interference tasks. Rescheduling explains the bump in the runtime curve of Potts' heuristic for 128 tasks. Around 60 % of the 128-task workflows could be solved without rescheduling. The runtime of the LD heuristic has a generally larger runtime than Potts' heuristic, because LD is relatively complex. The worst observed runtime of Potts' heuristic is still smaller than the best observed runtime of the MILP-approach and the worst observed case of the LD heuristic is smaller than the geometric average of the approach based on Gurobi. The solver has the largest spread, ranging from just over 100 ms to over 100 s for 128 task workflows, because it explores a larger portion of the state space. These figures exclude the time required for transformation of the rtSOA scheduling problem to a MILP representation, which requires an additional few hundred milliseconds.

Overall, heuristics are able to quickly find feasible execution plans, even when running several heuristics in sequence. The MILP formulation we presented in Section 4.1.2 also has reasonable response times in most cases. However, the runtime required to solve larger problem instances with a MILP solver increases faster than that of the heuristics because it has a higher algorithmic complexity - even when tak-

ing into account that a modern solver, such as Gurobi, will employ its own heuristics to find an initial feasible solution. From a runtime perspective, we thus view a combination of effective domain specific heuristics as the better choice for the interactive engineering environment as outlined in Goal 4 in Section 3.1 and shown in Chapter 6. The human attention span for interactive systems is in the order of a few hundred milliseconds to one second [83], which is only reliably achieved by using heuristics instead of a solver based approach. Gurobi required several minutes in the worst examined cases (c.f. Figure 4.11).

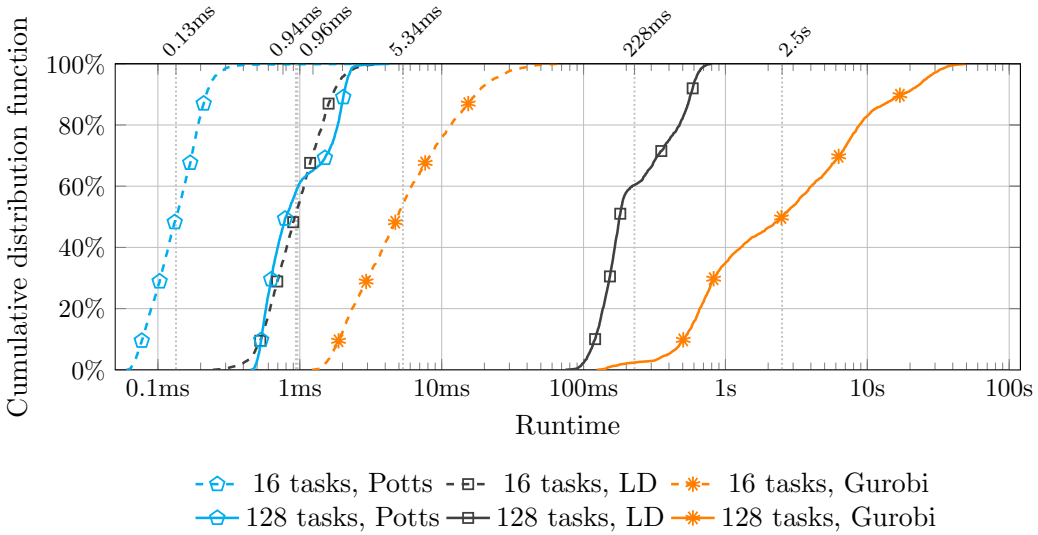


Figure 4.11: Cumulative runtime distribution for feasible problem instances

4.3.4 Conclusions

Table 4.6 shows a pairwise comparison of all heuristics on the entire benchmark set. Even the best heuristics (Potts' or ETF) are unable to find a solution for the scheduling problem in some cases were the overall worst heuristics (Slicing heuristics, EQS, EQF or PD) are successful. This fact is the basis for our approach which uses a combination of different heuristics in succession. This fact also justifies the inclusion of the LD heuristic which has a relatively high run-time cost (Table 4.4). The LD heuristic explores a different area of the search space than many other heuristics, meaning it has the highest number of unique solutions (3398) which are not generated by any other heuristic. Potts' and DSC follow in second and third place with 2753 and 2631 unique solutions, respectively. ETF and ERT-LFT also have a relatively high number of unique solutions with 1342 and 829, respectively. Slice-Pure and EQS have the least unique solutions with 46 and 41, respectively, unique solutions. It is worth having more than one heuristic available as each will explore additional areas of the search space. Overall, the heuristics solved 99 % of all feasible examined scheduling problems. Coupled with a fast execution - below 10 ms or even below 1 ms in nearly all cases - our approach of using heuristics for generating schedule-based execution plans enables interactive planning and reconfiguration without delays.

Heuristics:	Potts'	→	LD	→	DSC	→	ERT-LFT	→	ETF	→	...
Effectiveness:	95.4 %	→	97.5 %	→	98.2 %	→	98.5 %	→	98.7 %	→	99.0 %

Table 4.5: Optimal sequence for executing the heuristics on the benchmark data set

When running heuristics in sequence, it would be ideal to start with an effective heuristic and choose new heuristics that maximize the chance of finding a feasible solution, taking into account which heuristics were run previously. We determined the optimal sequence of heuristics for the benchmark data set by starting with the most effective heuristic (Potts) and successively adding the heuristic that leads to the highest combined effectiveness. The optimal sequence is shown in Table 4.5.

After the first five heuristics, the remaining ones only have negligible delta to the ones before them, thus we do not explicitly list their optimal sequence. Three out of five heuristics in the optimal heuristic sequence are our own heuristics or modified existing heuristics (modified Potts, LD and ERT-LFT). The evaluation of the optimal sequence shows that the LD heuristic should be kept in the set of heuristics despite its higher computational cost, because it already appears in the second spot directly after the overall more effective Potts' heuristic.

	EQS	EQF	PD	SL-Pure	SL-Norm	SL-Adapt L	SL-Adapt G	EQS-TDMA	EQF-TDMA	ERT-LFT	HEFT	DSC	MD	ETF	Potts	LD
EQS	-	↑ 4.44% ← 4.43%	↑ 6.74% ← 7.54%	↑ 9.35% ← 11.03%	↑ 8.86% ← 12.67%	↑ 9.37% ← 10.82%	↑ 9.35% ← 11.02%	↑ 6.54% ← 2.95%	↑ 6.55% ← 2.96%	↑ 16.19% ← 1.41%	↑ 9.66% ← 5.26%	↑ 18.08% ← 1.35%	↑ 12.79% ← 2.66%	↑ 19.12% ← 0.60%	↑ 19.92% ← 0.45%	↑ 17.73% ← 1.18%
EQF	↑ 4.43% ← 4.44%	-	↑ 6.74% ← 7.55%	↑ 9.35% ← 11.03%	↑ 8.84% ← 12.66%	↑ 9.37% ← 10.83%	↑ 9.35% ← 11.03%	↑ 6.53% ← 2.94%	↑ 6.54% ← 2.95%	↑ 16.19% ← 1.41%	↑ 9.68% ← 5.29%	↑ 18.08% ← 1.35%	↑ 12.78% ← 2.65%	↑ 19.12% ← 0.60%	↑ 19.92% ← 0.45%	↑ 17.74% ← 1.19%
PD	↑ 7.54% ← 6.74%	↑ 7.55% ← 6.74%	-	↑ 10.23% ← 11.11%	↑ 9.77% ← 12.78%	↑ 10.27% ← 10.92%	↑ 10.23% ← 11.11%	↑ 9.37% ← 4.98%	↑ 9.37% ← 4.98%	↑ 17.30% ← 1.72%	↑ 8.27% ← 3.08%	↑ 18.69% ← 1.16%	↑ 14.33% ← 3.40%	↑ 20.17% ← 0.85%	↑ 20.92% ← 0.65%	↑ 18.68% ← 1.33%
SL-Pure	↑ 11.03% ← 9.35%	↑ 11.03% ← 9.35%	↑ 11.11% ← 10.23%	-	↑ 3.68% ← 5.81%	↑ 2.99% ← 2.76%	↑ 0.64% ← 0.64%	↑ 12.34% ← 7.07%	↑ 12.36% ← 7.09%	↑ 18.69% ← 2.23%	↑ 13.57% ← 7.50%	↑ 20.77% ← 2.36%	↑ 16.98% ← 5.18%	↑ 21.57% ← 1.38%	↑ 22.16% ← 1.01%	↑ 20.41% ← 2.18%
SL-Norm	↑ 12.67% ← 8.86%	↑ 12.66% ← 8.84%	↑ 12.78% ← 9.77%	↑ 5.81% ← 3.68%	-	↑ 5.81% ← 3.45%	↑ 5.82% ← 3.69%	↑ 14.03% ← 6.63%	↑ 14.03% ← 6.63%	↑ 20.71% ← 2.11%	↑ 15.43% ← 7.23%	↑ 22.81% ← 2.27%	↑ 18.89% ← 4.94%	↑ 23.68% ← 1.35%	↑ 24.27% ← 0.99%	↑ 22.43% ← 2.07%
SL-Adapt L	↑ 10.82% ← 9.37%	↑ 10.83% ← 9.37%	↑ 10.92% ← 10.27%	↑ 2.76% ← 2.99%	↑ 3.45% ← 5.81%	-	↑ 2.71% ← 2.93%	↑ 12.11% ← 7.07%	↑ 12.13% ← 7.09%	↑ 18.47% ← 2.23%	↑ 13.39% ← 7.54%	↑ 20.54% ← 2.36%	↑ 16.75% ← 5.17%	↑ 21.36% ← 1.39%	↑ 21.95% ← 1.03%	↑ 20.18% ← 2.18%
SL-Adapt G	↑ 11.02% ← 9.35%	↑ 11.03% ← 9.35%	↑ 11.11% ← 10.23%	↑ 0.64% ← 0.64%	↑ 3.69% ← 5.82%	↑ 2.93% ← 2.71%	-	↑ 12.33% ← 7.07%	↑ 12.35% ← 7.09%	↑ 18.69% ← 2.23%	↑ 13.56% ← 7.50%	↑ 20.77% ← 2.36%	↑ 16.98% ← 5.18%	↑ 21.57% ← 1.38%	↑ 22.16% ← 1.02%	↑ 20.40% ← 2.18%
EQS-TDMA	↑ 2.95% ← 6.54%	↑ 2.94% ← 6.53%	↑ 4.98% ← 9.37%	↑ 7.07% ← 12.34%	↑ 6.63% ← 14.03%	↑ 7.07% ← 12.11%	↑ 7.07% ← 12.33%	-	↑ 3.64% ← 3.64%	↑ 13.05% ← 1.85%	↑ 7.47% ← 6.66%	↑ 14.90% ← 1.76%	↑ 10.17% ← 3.63%	↑ 15.76% ← 0.83%	↑ 16.51% ← 0.63%	↑ 14.60% ← 1.64%
EQF-TDMA	↑ 2.96% ← 6.55%	↑ 2.95% ← 6.54%	↑ 4.98% ← 9.37%	↑ 7.09% ← 12.36%	↑ 6.63% ← 14.03%	↑ 7.09% ← 12.13%	↑ 7.09% ← 12.35%	↑ 3.64% ← 3.64%	-	↑ 13.06% ← 1.86%	↑ 7.47% ← 6.66%	↑ 14.91% ← 1.76%	↑ 10.15% ← 3.61%	↑ 15.77% ← 0.84%	↑ 16.53% ← 0.65%	↑ 14.61% ← 1.64%
ERT-LFT	↑ 1.41% ← 16.19%	↑ 1.41% ← 16.19%	↑ 1.72% ← 17.30%	↑ 2.23% ← 18.69%	↑ 2.11% ← 20.71%	↑ 2.23% ← 18.47%	↑ 2.23% ← 18.69%	↑ 1.85% ← 13.05%	↑ 1.86% ← 13.06%	-	↑ 2.56% ← 12.95%	↑ 5.57% ← 3.63%	↑ 3.37% ← 8.03%	↑ 5.21% ← 1.47%	↑ 5.70% ← 1.01%	↑ 5.53% ← 3.77%
HEFT	↑ 5.26% ← 9.66%	↑ 5.29% ← 9.68%	↑ 3.08% ← 8.27%	↑ 7.50% ← 13.57%	↑ 7.23% ← 15.43%	↑ 7.54% ← 13.39%	↑ 7.50% ← 13.56%	↑ 6.66% ← 7.47%	↑ 6.66% ← 7.47%	↑ 12.95% ← 2.56%	-	↑ 13.99% ← 1.65%	↑ 10.44% ← 4.70%	↑ 15.21% ← 1.09%	↑ 15.95% ← 0.87%	↑ 14.01% ← 1.86%
DSC	↑ 1.35% ← 18.08%	↑ 1.35% ← 18.08%	↑ 1.16% ← 18.69%	↑ 2.36% ← 20.77%	↑ 2.27% ← 22.81%	↑ 2.36% ← 20.54%	↑ 2.36% ← 20.77%	↑ 1.76% ← 14.90%	↑ 1.76% ← 14.91%	↑ 3.63% ← 5.57%	↑ 1.65% ← 13.99%	-	↑ 1.67% ← 8.27%	↑ 4.36% ← 2.57%	↑ 4.84% ← 2.10%	↑ 3.92% ← 4.10%
MD	↑ 2.66% ← 12.79%	↑ 2.65% ← 12.78%	↑ 3.40% ← 14.33%	↑ 5.18% ← 16.98%	↑ 4.94% ← 18.89%	↑ 5.17% ← 16.75%	↑ 5.18% ← 16.98%	↑ 3.63% ← 10.17%	↑ 3.61% ← 10.15%	↑ 8.03% ← 3.37%	↑ 4.70% ← 10.44%	↑ 8.27% ← 1.67%	-	↑ 9.28% ← 0.89%	↑ 10.28% ← 0.93%	↑ 8.96% ← 2.54%
ETF	↑ 0.60% ← 19.12%	↑ 0.60% ← 19.12%	↑ 0.85% ← 20.17%	↑ 1.38% ← 21.57%	↑ 1.35% ← 23.68%	↑ 1.39% ← 21.36%	↑ 1.38% ← 21.57%	↑ 0.83% ← 15.76%	↑ 0.84% ← 15.77%	↑ 1.47% ← 5.21%	↑ 1.09% ← 15.21%	↑ 2.57% ← 4.36%	↑ 0.89% ← 9.28%	-	↑ 1.74% ← 0.79%	↑ 2.74% ← 4.71%
Potts	↑ 0.45% ← 19.92%	↑ 0.45% ← 19.92%	↑ 0.65% ← 20.92%	↑ 1.01% ← 22.16%	↑ 0.99% ← 24.27%	↑ 1.03% ← 21.95%	↑ 1.02% ← 22.16%	↑ 0.63% ← 16.51%	↑ 0.65% ← 16.53%	↑ 1.01% ← 5.70%	↑ 0.87% ← 15.95%	↑ 2.10% ← 4.84%	↑ 0.93% ← 10.28%	↑ 0.79% ← 1.74%	-	↑ 2.11% ← 5.03%
LD	↑ 1.18% ← 17.73%	↑ 1.19% ← 17.74%	↑ 1.33% ← 18.68%	↑ 2.18% ← 20.41%	↑ 2.07% ← 22.43%	↑ 2.18% ← 20.18%	↑ 2.18% ← 20.40%	↑ 1.64% ← 14.60%	↑ 1.64% ← 14.61%	↑ 3.77% ← 5.53%	↑ 1.86% ← 14.01%	↑ 4.10% ← 3.92%	↑ 2.54% ← 8.96%	↑ 4.71% ← 2.74%	↑ 5.03% ← 2.11%	-

Table 4.6: Cross evaluation of heuristics. The percentages specify the amount of test cases in which one heuristic found a feasible solution while the other heuristic did not. The arrows next to the percentages point to the associated heuristic.

"There is nothing more deceptive than an obvious fact."
- Arthur Conan Doyle, The Boscombe Valley Mystery

CHAPTER 5

Validation

One of the benefits of the rtSOA approach is the generation of execution plans with verifiable real-time properties. This chapter shows different approaches to assessing the temporal behavior of an rtSOA execution plan that may be used depending on the safety requirements of the system. The highest level of confidence can be reached by using an automated formal approach, namely model checking with UPPAAL[1] as described in Section 5.1. However this approach often has high computational requirements and may take a long time to complete. A quicker method is simulation of the schedules through discrete event simulation which follows the same execution semantics but does not offer the same level of confidence. Our simulation approach is described in Section 5.2.

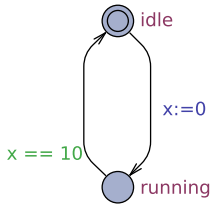
5.1 Model Checking via Timed Automata

> "Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for [...] that model." [3]

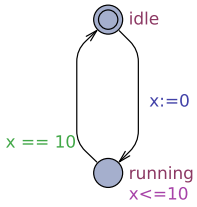
This definition, given by Baier et al. in their book "Principles of Model Checking" [3], summarizes the main benefits of model checking, but also hints at its drawbacks. Compared to other formal approaches, model checking is a highly automated and high level technique. Given a representation of a system it evaluates system properties, such as liveness or safety properties, specified in temporal logic. Temporal logics express statements over the potential execution sequences of a system. Model checking requires no formal proofs and can generate counter examples if the evaluated property does not hold for the examined system.

[1] http://www.uppaal.org/

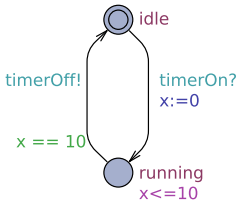
Given the application domain of rtSOA, model checking of its output schedules requires a representation that includes a concrete notion of time. This is different from the classical representation structures for model checking which only include the notion of "happens before" or "happens after". It also poses a challenge, as an unsuitable representation of time may quickly turn the finite-state model into an infinite-state model. The most common ways to model such timed systems are timed Petri nets [10] or timed automata [126]. We chose to perform model checking of rtSOA schedules with UPPAAL [9], a mature tool suite for model checking of networked real-time systems. We will therefore only cover timed automata in the remainder of this section.



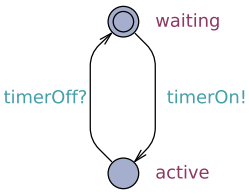
(a) Timer version 1



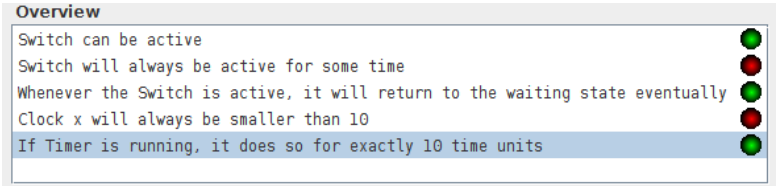
(b) Timer version 2



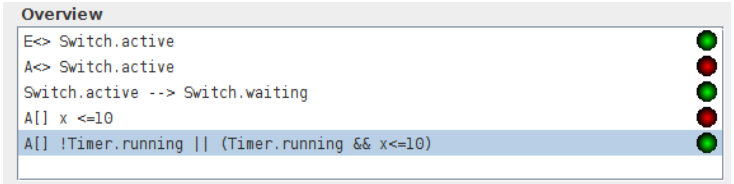
(c) Timer version 3



(d) Switch



(e) Model checked properties of the composition of the models *Timer version 3* and *Switch*.



(f) TCTL formulas corresponding to the properties shown in Figure 5.1e.

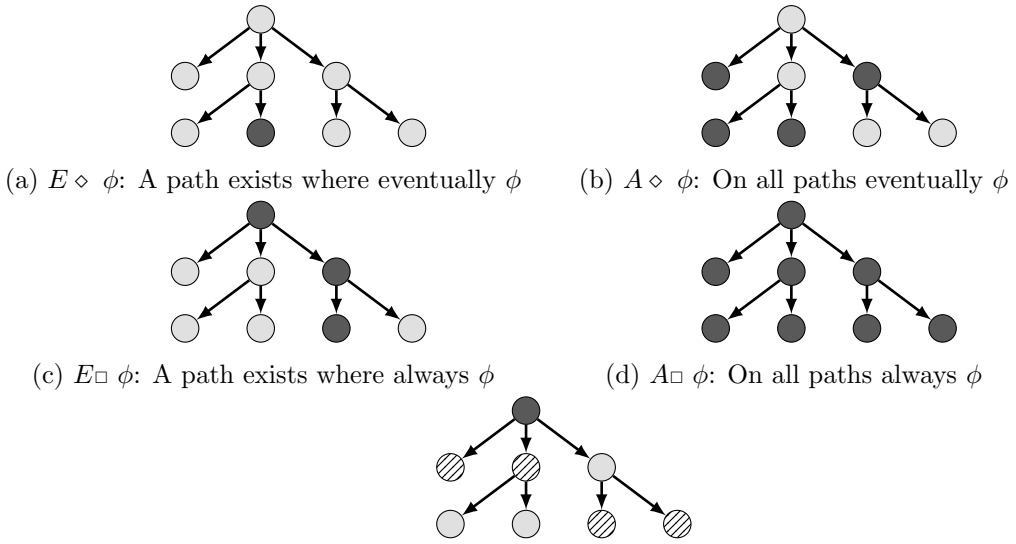
Figure 5.1: Examples demonstrating the properties of timed automata in UPPAAL. Purple text represents state invariants. Green text represents guards, meaning conditions that must be true for the associated transition to be active. Dark blue text shows updates to local or global variables performed during state transitions. Light blue text denotes channels that are used for time synchronization automata.

Figure 5.1a shows a very simple timed automaton modeled with UPPAAL. It features two states, named *idle* and *running* and a clock named x that is reset whenever the automaton enters the state *running*. The guard $x == 10$ enforces that the transition from *running* to *idle* is only active when exactly 10 time units have passed. The displayed model therefore represents a first attempt to model a timer. Unfortunately, progress, defined as the occurrence of further state transitions, is not guaranteed in this model because the automaton may spend an indefinite amount of time in either state. The guard only enables the transition from *running* to *idle*, but it does not enforce the transition and the automaton may instead remain in the state *running* indefinitely. To enforce system progress, UPPAAL offers the possibility to add state invariants over clocks. This refinement is shown in Figure 5.1b where we introduced the invariant $x <= 10$ in the state *running*. The automaton may now only remain in the *running* state as long as the value for clock x is smaller or equal to 10. Combined with the guard $x == 10$ on the outgoing transition, the introduced invariant now causes the automaton to remain in the state *running* for exactly 10 time units.

Our goal is model checking of networked systems, so it would be convenient to have a mechanism which enables communication between multiple automata. Such a mechanism would simplify modeling of the rtSOA execution plans because we would not have to model the execution plan as a single automaton but could instead use a more natural, compositional approach. UPPAAL offers communication between automata in two ways: An asynchronous communication path via shared global variables and a time synchronized path via channels. This is illustrated in Figure 5.1c and Figure 5.1d. We added a second automaton, called *Switch*, which is synchronized with the *Timer* automaton via two channels, named *timerOn* and *timerOff*. When the *Switch* leaves the *waiting* state, it forces the *Timer* automaton to make a synchronous transition, via the *timerOn* channel. This transition must happen at the same time. The *Timer* enforces another synchronous transition after 10 time units have passed via the *timerOff* channel.

Using UPPAAL, we can now modelcheck the composition of these two automata. Figure 5.1e shows some properties of the composition of the *Timer* automaton and the *Switch* automaton. This composition is performed automatically by UPPAAL. It shows that the *Switch* may reach the *active* state, but may also remain in the *waiting* state indefinitely. However, it will not remain in the *active* state forever. In fact, it will do so for exactly 10 time units, as dictated by the synchronization between the *Timer* automaton and the *Switch* automaton and as verified by the last query shown in Figure 5.1e. Figure 5.1f shows the corresponding temporal logic specifications.

UPPAAL allows specification of properties with a subset of the timed computation tree logic (TCTL) [10]. In CTL, a run of a system is represented as a potentially infinite tree of states and transitions between states. To reduce the state space, a model checker will try to detect possible loops in the system execution that will then be added to the tree as annotations. Examples for the allowed TCTL subset in UPPAAL



(a) $E \diamond\ \phi$: A path exists where eventually ϕ

(b) $A \diamond\ \phi$: On all paths eventually ϕ

(c) $E \Box\ \phi$: A path exists where always ϕ

(d) $A \Box\ \phi$: On all paths always ϕ

(e) $\phi \implies \rho$: ϕ always leads to ρ.

Figure 5.2: Different (T)CTL-formulas. A state is shaded darker if condition ϕ holds in that state, hatched states indicate that condition ρ holds.

are shown in Figure 5.2 which visualizes the TCTL properties on a tree. Put simply, A describes properties that hold on all paths in the tree where E only describes that at least one path exists. Similarly, $\Box$ describes that a property must always be true while $\diamond$ only demands that it must be true at some time. Invariant properties are useful for checking safety properties of the system, for example $A\Box\ \mathit{clock} \leq \mathit{deadline}$ could be used to verify that a system will always finish its execution before a deadline has been reached. Properties of the form $A\diamond \mathit{num\_tasks\_finished} = \mathit{num\_tasks}$ can be used to check the eventual correctness or completion of a system's execution, e.g.,in the provided example we are checking that a system will eventually complete all of its tasks. Path invariants (Figure 5.2c) can be used to identify execution paths through the system that satisfy certain properties. For example, $E\Box\ \mathit{clock} \leq \mathit{deadline}$ could be used to identify a valid task ordering that would fulfill all scheduling conditions.

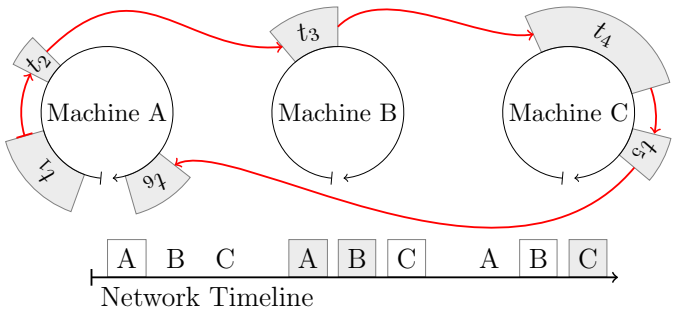
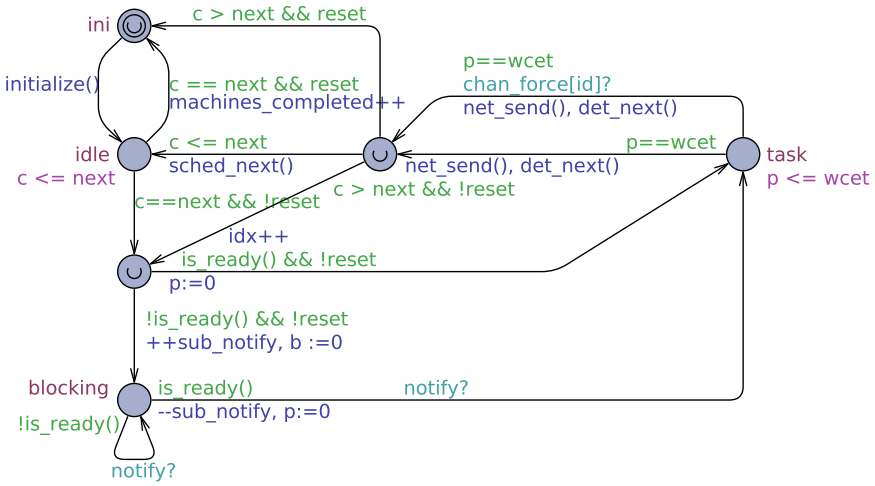


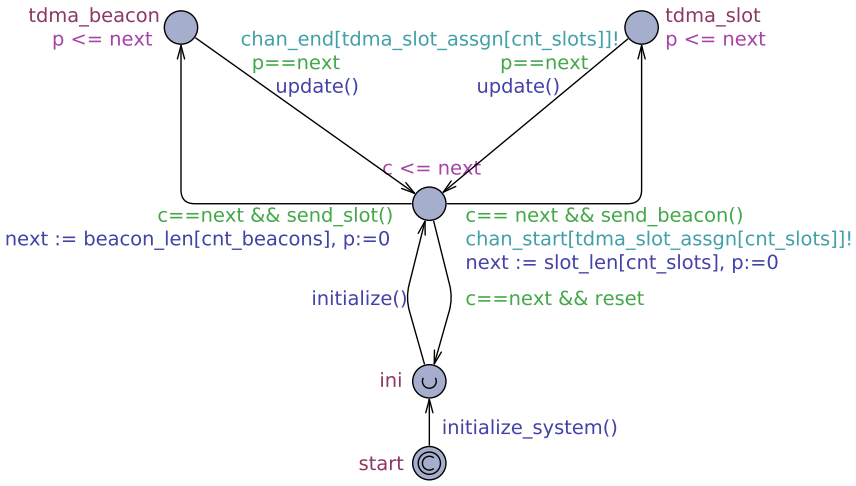
Figure 5.3: The rtSOA execution model based on static cyclic schedules. Gray areas indicate resources used by an example workflow, arrows represent data dependencies.

Figure 5.3 illustrates the rtSOA execution model again. Machines execute static cyclic schedules and communicate over a TDMA-network. The schedules are the output of the rtSOA heuristics or state exploration approaches and are designed to meet a global deadline. More details are given in Chapter 3. The focus of this chapter lies on the validation of these precomputed schedules. We have therefore designed a system model in UPPAAL that can quickly and automatically be adapted to new schedule configurations. Figure 5.4a shows the automaton template that models the behavior of a machine following the rtSOA execution semantics. One instance of this automaton is present for each machine that participates in an rtSOA execution plan. The machine may only spend time in the states named *idle*, *blocking* or *task*. All other states in Figure 5.4a have been marked as urgent (denoted by the ∪ sign), meaning that no time may be spent in those states. The machine starts in the *ini* state and immediately transitions to the *idle* state. During this transition the workflow clock is started and a new workflow cycle starts. The machine now spends time in the *idle* state until the next task is scheduled for execution. If all required messages for the execution of the task have been received, the machine enters the *task* state, representing active execution of the task. Otherwise, the machine enters the *blocking* state where it remains until new all messages from preceding tasks have been received over the network. In our current model, a task will always run for exactly the WCET. This behavior could be changed by modifying or removing the guards on the outgoing transitions of the *task* state if, for example, the influence of jitter were to be studied. The synchronization channel *chan_force* on the outgoing transition from *task* is used to ensure that tasks finish before a TDMA-slot starting at the same time instance. This means that a message from a task completing at time 10 will be included in a TDMA-slot that also starts at time 10 and is allocated to the same machine as the task.

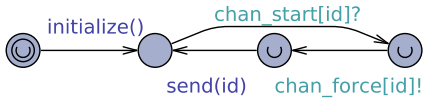
Synchronization channels in UPPAAL cannot carry messages, they only ensure synchronized transitions of two or more automata from one state to another. We are therefore using shared variables that represent messages sent from task t_i to t_j. Our model contains three Boolean matrices named *produced, net* and *have*. When task t_i is completed, the function named *net_send()* sets *produced[i][j]* to *true*, if task t_j is a direct successor of t_i and located on a different machine. The actual delivery of this message is modeled with three additional automata. The TDMA network itself is modeled by the automaton shown in Figure 5.4b. Before entering the *tdma_slot* state, the automaton enforces a synchronization on the *chan_start* channel. When leaving the state, it enforces a synchronization on the *chan_end* channel, ensuring that data from the machines is made available to all receivers directly after a TDMA slot. This automaton merely synchronizes the transitions between the sending automaton (Figure 5.4c) and receiving automaton (Figure 5.4d) at the beginning and end of each TDMA slot. The sending automaton will move the produced message to the *net* matrix, representing a message that is currently in flight. The receiving automaton finally sets *have[i][j]* to *true*.



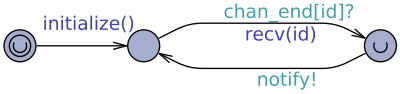
(a) Automaton modeling a machine for verifying an rtSOA schedule with UPPAAL



(b) Automaton modeling the TDMA-network



(c) Automaton for sending messages from the network



(d) Automaton for receiving messages from the network

Figure 5.4: UPPAAL templates for verification of rtSOA schedules

Notice that guards and updates may contain method names, such as *is_ready()* or *sched_next()*. By using these methods we can reuse the same automata templates for verification of all rtSOA execution plans. A concrete model of an rtSOA schedule is configured for verification in UPPAAL by changing global variables representing the schedule times of the tasks, the allocation of tasks to machines, the precedence relation between tasks and the start and end times of the TDMA slots as well as the assignment of TDMA slots to machines. This dynamic part of the UPPAAL model is automatically generated after deriving an execution plan with rtSOA. For example, the method *sched_next()* determines the time that the automaton in Figure 5.4a spends in the *idle* state through a simple lookup of the start time of the next scheduled task on the machine represented by the automaton. This corresponds to a lookup in the array *task_start* in Listing 5.1, which contains a simplified and shortened example for the configuration variables.

```
...
const uint16_t num_tasks = 2;    // number machines
const uint16_t num_tasks = 8;    // number of tasks in the workflow

// cycle time of each machine
uint31_t machine_cycles[num_machines] = {10000000, 10000000};
// start time of each task
uint31_t task_start[num_tasks] = {0, 2555713, 2505713,...};
// WCET of each task
uint31_t task_wcet[num_tasks] = {1092734, 50000, 50000,...};
...
```

Listing 5.1: Example for configuration parameters specifying the schedule times in our UPPAAL model

We formulated the following properties for model checking of rtSOA execution plans:

1. `A[] wf_clock <= deadline`
   This property is fulfilled if all machines always complete their respective schedules before the deadline is reached.

2. `A<> tasks_completed == num_tasks`
   All tasks of the workflow will eventually be completed.

3. `forall(i:id_t) Machine(i).ini -> forall(i:id_t) Machine(i).ini`
   All machines will eventually return to their initial state, meaning all machines will complete their schedules.

4. `TDMA.tdma_beacon -> TDMA.ini`
   The TDMA automaton will always complete its cycle after it has sent a beacon.

5. `A[] forall(i:id_t) Machine(i).blocking imply Machine(i).b <= 0`
   No machine spends any time in the blocking state. Machine(i).b is the clock that measures time spent blocking.

Properties	16 Tasks				32 Tasks			
	2 Machines		4 Machines		2 Machines		4 Machines	
	Runtime	**DNF**	**Runtime**	**DNF**	**Runtime**	**DNF**	**Runtime**	**DNF**
correctness	209 *ms*	1%	1,712 *ms*	27%	448 *ms*	0.6%	3,633 *ms*	9.9%
liveness	258 *ms*	1%	4,043 *ms*	27%	558 *ms*	0.6%	8,786 *ms*	9.9%
non-blocking	90 *ms*	2.2%	722 *ms*	78%	183 *ms*	0.9%	1,217 *ms*	93%

Table 5.1: In the enumeration above, properties 1 and 2 are correctness properties, properties 3 and 4 are liveness properties and property 5 is the non-blocking property. *DNF* stands for "did not finish" and includes timeout after 30 minutes or out-of-memory models.

Table 5.1 shows a quantitative evaluation of the above properties on two or four machines with random workflows that had either 16 or 32 tasks. We generated 1000 instances for each combination of machine number and workflow size with the $G(n, p)$ method (c.f. Section 4.3). We can see that, even for relatively small models, there are some instances where the evaluation did not finish, either due to memory constraints or a timeout after 30 minutes. Our model scales relatively well with the number of tasks but additional machines quickly lead to a state space explosion, as established by the quickly rising number of "did not finished" entries in Table 5.1. This section has proven that model-checking of rtSOA schedules is possible, but often not fast enough or feasible for larger problem instances. We therefore present an alternative approach to validation based on simulation in the next section.

5.2 Discrete Event Simulation

In discrete event simulation, events happen at a concrete point in time and mark a change in the overall system state. Between two events, the system state is assumed to stay constant. This mode of simulation fits well with the representation of rtSOA execution plans. Tasks start and end at a particular time in an execution plan and communication over TDMA also implies that message transfer starts and ends at a particular instance in time. This short description already covers the essence of the rtSOA simulation model. Figure 5.5 shows an illustration of the event stream that represents a simulation of an rtSOA schedule. The events that make up a simulation run are managed in an event queue in which events can be enqueued and removed. As shown in Figure 5.5, we enqueue "missed deadline" events for each task and the overall workflow but remove those events again if the respective tasks or the workflow successfully complete before the "missed deadline" events.

The simulations follows the same basic model as shown in Figure 5.3 and described in Section 3.3. The simulation contains three different entities: nodes, meaning the machines running the rtSOA workflow; the TDMA network; and messages carried by the TDMA slots and sent from node to node. As before, each node has its own cycle. The *NodeCycleStart* event for each machine is executed at simulation time 0 along with the *TDMACycleStart* event. When executing the *NodeCycleStart* event,

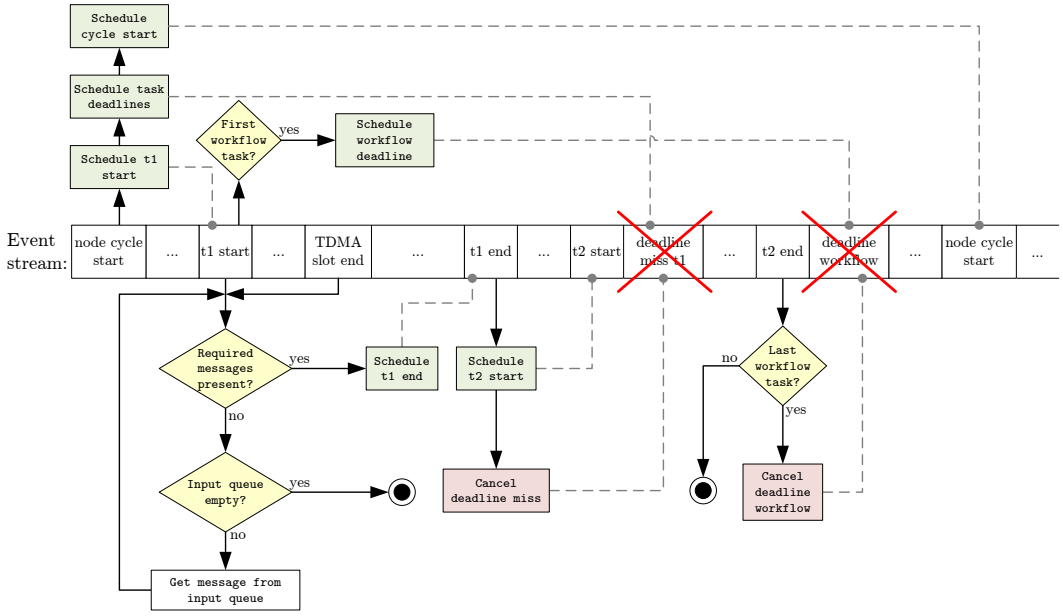


Figure 5.5: Illustration of the event queue in a discrete event simulation of a rtSOA schedule. Arrows on solid edges represent a logic flow that is triggered by the start of an event in the event stream. Dashed edges with rounded tips represent additions or deletions in the event queue.

the first task is chosen from the nodes' schedule and the *TaskStart* event is added to the simulation's event chain. During the *NodeCycleStart* event, events are scheduled that represent the deadline of each individual task. An event representing the deadline of the workflow is scheduled during the first *TaskStart* event of the current workflow execution. Deadline events only fire if the associated task or workflow have missed their deadline, thus indicating a violation of timing constraints. Therefore, deadline events are canceled if the associated task or workflow end event fires before them. Whenever a *TaskStart* event is executed, the simulation checks if all required messages have already been received. If this is the case, the machine state is changed to "running" (c.f. the state transition from *idle* to *task* in Figure 5.4a via intermediate states) and the *TaskEnd* event is scheduled. If the task is still missing some prerequisite messages from other tasks, the simulation first checks whether undelivered messages are present in the node's message input queue. If this is the case, it removes the first message, delivers it to the correct task and then reevaluates the readiness of the scheduled task. If no more messages are waiting in the input queue, the machine is entering the "blocked" state (c.f. the *blocking* state in Figure 5.4a). In this state, each arriving message is immediately delivered and the readiness of the scheduled task is reevaluated. When the *TaskEnd* event is executed, the *TaskStart* event of the next task in the current node's schedule is added to the event queue and the currently ending task removes its deadline event from the queue. Deadline events only fire if the deadline is violated. Should the currently ending task be the

last task in the workflow, it also removes the workflow deadline event from the event queue. The *TaskEnd* event also produces all outgoing messages of the ending task. Messages for successors on the same machine are delivered immediately. Messages for successors on other machines are put in the output queue of the machine.

Message delivery over the network is simulated via TDMA events. Analogously to the *NodeCycleStart* event, the *TDMACycleStart* event represents the start of a new iteration of the network slot schedule. The *TDMACycleStart* event also schedules the first *TDMASlotStart* event as well as the next *TDMACycleStart* event. During a *TDMASlotStart* event messages from the output queue of the machine associated with the TDMA-slot are removed and marked as being in transit. These messages are then delivered to the receiving machines during the next *TDMASlotEnd* event. The *TDMASlotEnd* event also schedules the next *TDMASlotStart*.

Our simulation is implemented based on the simulation framework DESMO-J[2]. In contrast to formal verification with UPPAAL, simulation with DESMO-J is fast and only requires a few milliseconds for each of the cases in our benchmark data set. For our benchmark data set, the simulation time scales with the number of tasks: Workflows with 16 tasks require $2.9\,ms$, 32 tasks require $5\,ms$ and 64 tasks require $10.3\,ms$. This is fast enough to be employed in the rtSOA heuristics pipeline for verification of each schedule result.

To be more precise, simulation time depends on the number of simulated events. This number may become large for workflows which require a long simulation time. To make reliable claims about the validity of a given rtSOA execution plan, the simulation should be run until the relative period of the TDMA-cycle and all machine cycles repeats. This time span is called the hyper period of the execution plan and constitutes the least common multiple of the TDMA cycle, all machine cycles and all workflow cycles. If the system exhibits repeating behavior over multiple hyper periods, then it can be stated with a high degree of confidence that generated schedules will satisfy the timing constraints. The largest drawback of the simulation based approach is the absence of any formal guarantees, which can only be provided by formal verification methods. However, simulation is a fast and efficient way to test candidate execution plans. Plans passing this initial test can then be verified with UPPAAL or other formal approaches.

[2] http://www.desmoj.de

"The whole is more than the sum of its parts."
- Aristotle

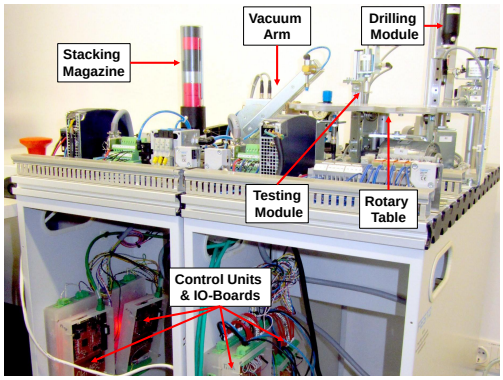
CHAPTER 6

Real-World Prototype

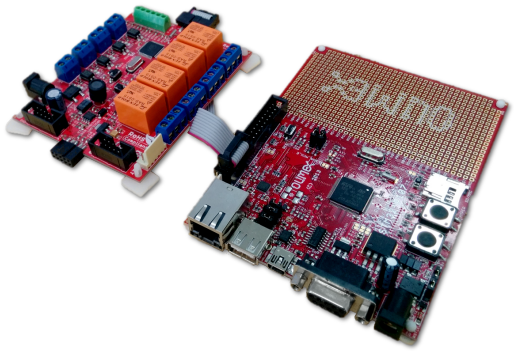
Parts of this chapter have been previously published in [67].

We have implemented a real-world prototype to validate the assumptions given in Section 3.2 and to show the feasibility of the execution model detailed in Section 3.4. The goal in developing the demonstrator was to show the ease of reconfiguring a complex control workflow in a networked real-time environment. The physical setup of our prototype consists of a Festo Modular Production System (MPS)[1] distribution station and processing station as shown in Figure 6.1a. The distribution station features a stacking magazine and swivel arm for work piece distribution to the processing station. Both the magazine and arm are pneumatic actuators. The processing station has four electric actuators: A rotary table, a testing module and a drilling module. It also features an electric sorting gate that is used to remove work pieces from the rotary table. We control all sensors and actuators in this setup through five Olimex STM32-P107 development boards[2] connected to IO-boards[3]. This connection is established via the I^2C (Inter-Integrated Circuit) bus running in standard mode (100 kbit/s). The CPU is a STM32F107 32-bit ARM-based micro controller running at 72 MHz, featuring 256 kB of flash memory and 64 kB RAM. The actuators and sensors are grouped together with a controller in functional units, e.g.,one node controlling the magazine, one for the rotary table and sorting gate and another one for the drill. The development boards are connected via 100 Mbit full-duplex switched Ethernet.

[1] http://www.festo-didactic.com/int-en/learning-systems/mps-the-modular-production-system
[2] https://www.olimex.com/Products/ARM/ST/STM32-P107/
[3] https://www.olimex.com/Products/Modules/IO/MOD-IO/



(a) Hardware setup for our demonstrator



(b) The development- and IO-boards used in our demonstrator

Figure 6.1: The prototype consists of a Festo MPS distribution and processing station (Subfigure a), controlled by Olimex STM32-P107 development boards (Subfigure b).

This chapter is subdivided into a description of the software runtime on the nodes (Section 6.1) followed by an explanation of how services are discovered (Section 6.2) and, in Section 6.3, how new execution plans for this manufacturing system can be automatically derived with rtSOA.

The author of this thesis does not take credit for the full implementation of the software stack running on the physical demonstrator nodes. Much of the scaffolding, the networking stack and the internal message routing (c.f. Section 6.1) has been contributed by the author's industry partner Siemens. The author implemented the timetable mechanics described in Section 6.1, the service description and discovery features explained in Section 6.2 and all interactions with the rtSOA planner and the graphical user interface described in Section 6.3.

6.1 Software Runtime

The software on the nodes is implemented directly in C without a real-time operating system (RTOS). The software architecture is a simple control loop, shown in Figure 6.2. It first updates the sensor / actuator values by communicating with the IO-board over the I^2C bus. After this first step, the software performs message routing between service instances on the node. Services do not communicate directly with each other but via links created between their input- and output ports, as shown between `Service 1` and `Service 2` in the example in Figure 6.2. A message producing service writes its output to the message queue of the node local message routing layer which delivers the message to the consuming services during the `processMessages()` call. The routing layer also performs message distribution over the network, if the user has configured a link to a service instance on a remote node. This process is transparent for the sending and receiving services. Messages sent over the network are encapsulated with the Erbium CoAP implementation [68], adapted for use without the Contiki OS.

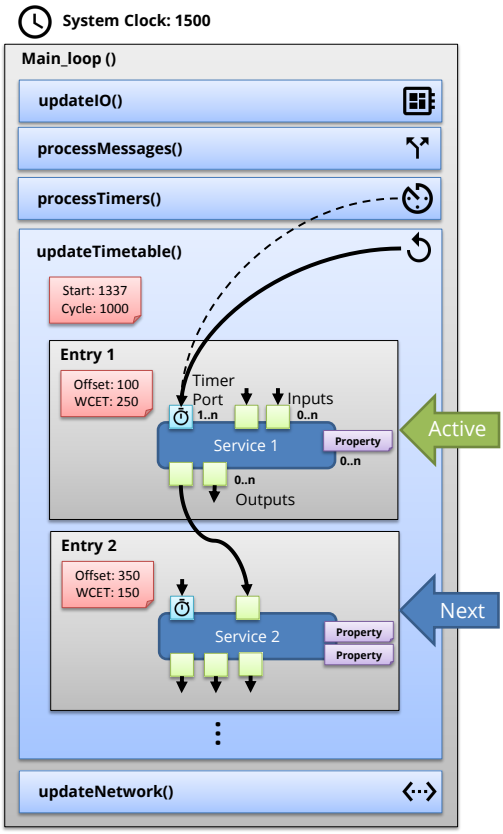


Figure 6.2: Illustration of the main loop on the nodes controlling the demonstrator. The `updateTimetable()` method triggers the execution of all scheduled service instances on the node and is thus the central control point in this time triggered system.

Since our implementation does not support context switching, long running services are encouraged to yield control of the CPU whenever possible. An example would be a service controlling the vacuum arm in our demonstrator. The arm needs several seconds to reach its end position after being instructed to move in a given direction. The service would yield control after triggering the movement and set an internal timer which will reactivate the service after a given duration. The service then queries the sensors for whether or not the arm has reached its resting position so that the service may signal completion to its successors. The `processTimers()` method will check for expired timers and activate the associated service instances. Another important implementation detail is the adherence of services to the data flow semantics detailed in Section 3.4. To ensure that a workflow composed of rtSOA service instances is a "well-behaved" data flow graph, every service must be invoked during each timetable cycle and must emit a message on each output port. This message may indicate a "no-op" if the preconditions of the service were not fulfilled.

Initially, service instances are triggered by the `updateTimetable()` method. The timetable contains the machine local schedule computed by the rtSOA planning heuristics, an example of which is shown in Figure 3.3e. The schedule on the node consists of pointers to all scheduled service instances with a given time offset from the cycle start and a value for the expected WCET of the service instance. After configuring schedules on all nodes in the network the user may choose any node as the master node, which will then trigger the synchronized execution of all schedules in

the network by issuing a start command via network broadcast. It also periodically resends the start signal to re-sync the cycle start times of all participating nodes. Finally, the `updateNetwork()` method sends and receives messages over the Ethernet connection.

6.2 Service Description and Discovery

Semantic description of services and their discoverability when composing workflows is a complex topic and subject to ongoing research [77] and standardization efforts [89]. Although these are important building blocks in a full fledged service-oriented architecture for manufacturing systems, they are out of scope for our current demonstrator. We therefore only implemented a minimal set of features to enable discoverability. Service discovery is performed in two steps: The planner first sends a ping command to the IPv6 address `ff02::fd`, which corresponds to all link local CoAP nodes [99]. Afterwards rtSOA downloads all available service descriptions, from the nodes that responded to the ping, by issuing a GET-request to the URI `coap://[<IP>]:5683/timetable/.installed`. The node responds to this GET-request with a JavaScript Object Notation (JSON) object that contains a description of each service available on the machine. An example for this description is shown in Figure 6.3. The service description contains information about the service's WCET, its input and output ports as well as configurable attributes of the service. In the shown example, the drill service has two input ports (for triggering the service execution via the timer or via the network), two output ports (one that signals completion and another one for status information) and three parameters, for example a configurable duration for how long the drill-bit should be run.

6.3 Changing Execution Plans

The rtSOA approach to software reconfiguration of manufacturing systems follows the steps outlined in Section 3.3. Given a workflow layout with global timing constraints and a manual selection of the nodes which should execute the services the rtSOA planner will generate an execution plan consisting of a schedule for each node. The machine local schedules are then deployed on each node, so that the time table implementation (Figure 6.2) may trigger the service execution at the predetermined time. To instantiate the execution plan, a matching cycle time must be set on all participating nodes before naming the services that should be instantiated with a predetermined offset from the cycle start. Following that, links between the input and output ports of services with data dependencies must be created on the node that contains the data producing service, following a publish-subscribe model. The node-internal message routing is performed by the `processMessages()` method as described in Section 6.1, communication between nodes is encapsulated in CoAP messages. Lastly, configuration parameters are set on the service instances as necessary.

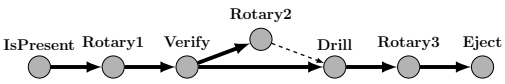
```
{name: „TTDDrill“,
 wcet: 2200,
 inports: {
   num: 2,
   attr: [{
       size: 1,
       type: 1,
       name „In_Drill_Timer“
     },{
       size: 1,
       type: 0,
       name: „IN_Drill_Trigger“
     }]
 }, outports: {
   num: 2,
   attr: [{
       size: 1,
       type: 0,
       name: „Out_Drill_Done“
     },{…}]
 },
 ... continued →
```

```
confAttributes: {
  num: 3,
  attr: [{
      name: „Attr_DrillDuration“,
      min: 0,
      max: 65535,
      default: 0
    }, {…},{…}]
 }
}
```

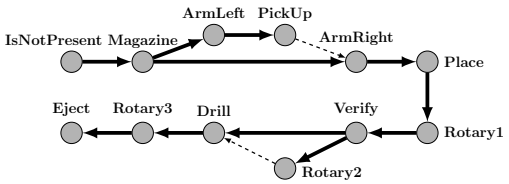
Figure 6.3: Nodes advertise their installed services via a simple JSON description, when queried. A video demonstrating the process of manually configuring a workflow with rtSOA is available at https://youtu.be/Wa5KdHEivOo.

For example, our demonstrator features a service for moving the vacuum arm where a configuration parameter must be set, indicating the direction (left or right) in which the arm should be moved upon service invocation. This naturally means that our implementation supports multiple instances of the same service with different parameters on the same machine. An example video demonstrating the process of manually configuring a service orchestration with our demonstrator can be found at https://youtu.be/Wa5KdHEivOo. This video only serves demonstration purposes, because manually orchestrating more complex workflows in this manner would be too error prone and time consuming. When using the rtSOA planning tool, these configuration steps are performed automatically during deployment.

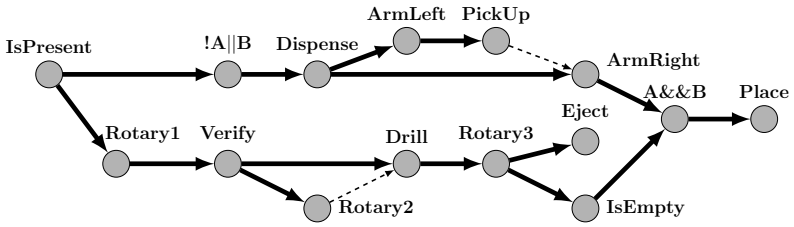
To demonstrate the evolution of a basic workflow we consider the example workflows shown in Figure 6.4. At the first stage, the systems only consists of a single module, the Festo MPS processing station featuring an electric turn table with a testing and drilling module (c.f. Figure 6.1a). A real world example for this system would be an half-automated assembly system with a turntable on which a human operator would place base parts and from which the human would transfer finished parts to storage [117]. A workflow for such a system is shown in Figure 6.4a. Vertices in the displayed graph represent named service invocations, edges represent a successor relationship. While thicker edges represent actual data flow, thin edges only represent a logical precedence relation without actual data flow. After sensing the presence of a work piece with the service named `IsPresent`, the system transfers the work piece to the testing module which performs the tests offered by the `Verify` service. Based on the output of this service, the work piece may be further processed by the `Drill` service or only transported past that module by the `Rotary` service.



(a) Workflow for the processing station of the Festo MPS



(b) Extended workflow which added the distribution station to the processing station



(c) Advanced version of the previous workflow that runs the distribution and processing stations in parallel

Figure 6.4: Evolution of the service composition controlling the Festo modular production system: From a simple processing station, via an intermediate step that extended the system by a second station, to an optimized workflow that runs both station in parallel.

Figure 6.4b represents an evolution of the system by extending the system with the Festo MPS distribution station providing automated supply of parts. All of the services present in the first workflow (Figure 6.4a) have been reused, but some feature different configuration parameters. For example, the service instance named `IsPresent` in Figure 6.4a triggers the further execution of the workflow once the worker has placed a work piece in the starting position. In contrast, the service instance named `IsNotPresent` in Figure 6.4b triggers the transport of a work piece to the starting position if no work piece was detected there.

rtSOA does not perform optimization on the structure of the workflow. For example, the workflow shown in Figure 6.4b is inefficient because it is using all modules of the system in sequence whereas the distribution station and processing station can also be run in parallel. This parallel workflow is shown in Figure 6.4c. It still reuses all of the previous services but requires additional logic services. For example, the service instance named `!A||B` triggers the dispensing of a work piece from the magazine when either no work piece is present on the starting position, or if a work piece was present but has already been moved to the next position by the `Rotary` service. The first code path is used when executing the workflow initially, whereas the second path is executed when the cycle wraps around and is executed repeatedly. Mutually exclusive paths through a workflow are currently not supported by rtSOA and constitute an area for future research.

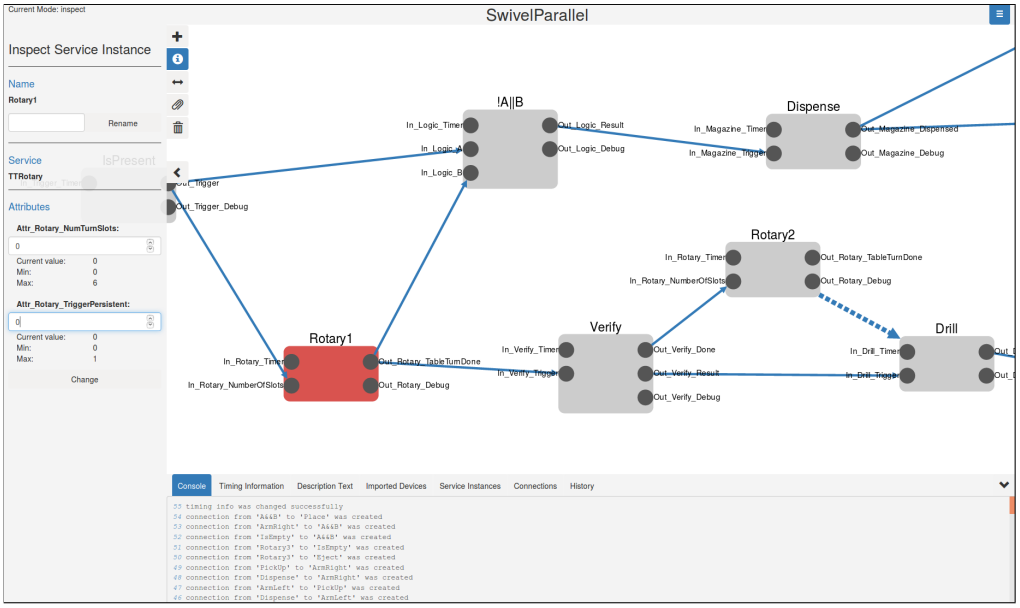


Figure 6.5: The service composition interface of the rtSOA demonstrator. This screenshot depicts a zoomed in portion of the workflow from Figure 6.4c.

Reconfiguring the system can be performed without rebooting or reprogramming the nodes. Our demonstrator offers a graphical user interface (GUI) for the task of (re-)configuration which is shown in Figure 6.5[4]. This view constitutes the service composition GUI and is used by an engineer to design or alter the structure of a workflow represented as a DAG. This view supports multiple modes which correspond to different task in specifying the workflow structure, connections between services instances' ports or specifying service instance attributes. The first mode is the composition mode which can be used to add new service instances to the workflow. Service instances may be created from service descriptions advertised by real devices in the network or from archived device descriptions. Service instances are visually represented by rounded rectangles with their name written above. The circles on the left-hand side of the service instance represent the instance's input ports, the ones on the right-hand side represent its output ports. Connections between ports can be created by putting the GUI into the connection mode and first clicking on the output port and then clicking on the input port of another service instance. These connections are represented by solid arrows in the GUI. Dashed arrows show logical precedence without data transfer, they can be created the same way as port connections by clicking on the two service instances involved. Figure 6.5 shows the GUI in the inspection mode which can be used to display or change individual attributes of service instances. The last relevant mode is the instance assignment mode in which an engineer can specify the concrete device on which a service should be placed during workflow execution. The GUI also offers delete, undo and redo functionality for convenience.

[4]The service composition GUI was developed by Christian Feiler during his Bachelor's thesis titled "A Graphical Editor for Service Choreographies in Industrial Real-Time Control Loops"

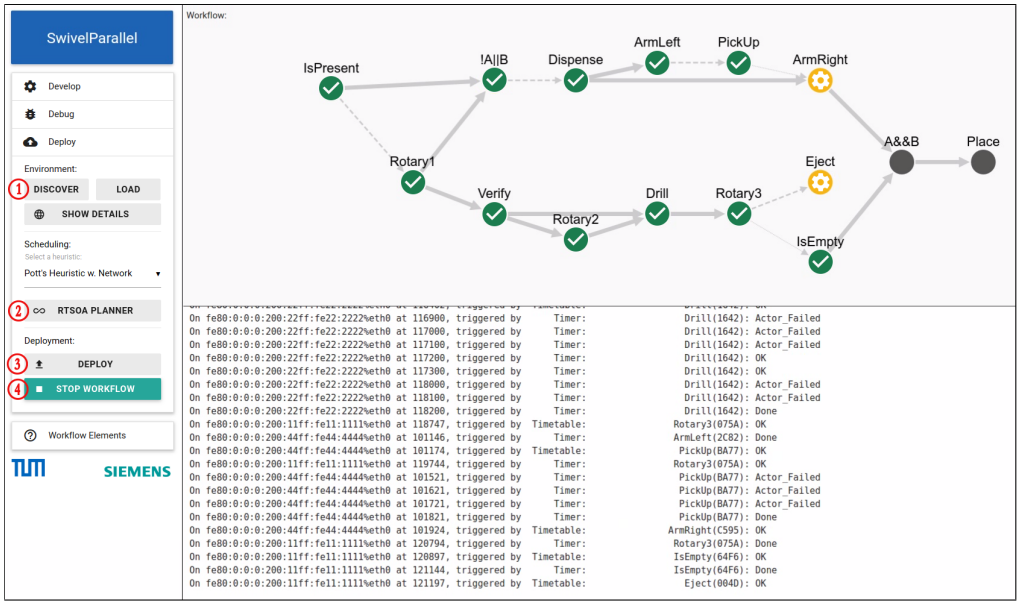


Figure 6.6: After generating and deploying an execution plan by pressing the buttons marked 1-3, the user has started the execution of the workflow with the button marked by the number 4. The GUI is currently displaying a live visualization of this execution. A video showing the process described here can be found at https://youtu.be/WjgEySzpTo8.

For deployment and simulation of a workflow designed with the service composition GUI (Figure 6.5), the user can switch to the appropriate view as shown in Figure 6.6. In this case the parallel workflow shown in Figure 6.4c was loaded. The user can either trigger the discovery of nodes and services, as described in Section 6.2, by selecting the button labeled `Discover` or load prerecorded service descriptions with the `Load` button, if they wish to perform a dry-run of the planning heuristics. If the environment offers all services required by the workflow, the user can create an execution plan by clicking the button labeled `rtSOA Planner` which will generate schedules for all devices in the network. This process follows the heuristics described in Section 4.2 and typically completes within a few milliseconds. After deploying the execution plan to the participating nodes with the `Deploy` button the user can then trigger the workflow execution with the `Start Workflow` button. Figure 6.6 shows a running workflow in which most service instances have already completed their tasks, as indicated by a checkmark. This visualization is generated live, meaning in a timely manner, from debug information sent from the nodes to the PC running the visualization interface. The `ArmRight` service is executed in parallel with the `Eject` service, as indicated by the cogwheel icon, on two different nodes. The remaining service instances are awaiting activation. A video demonstrating reconfiguration with this GUI can be viewed at https://youtu.be/WjgEySzpTo8. The video shows the initial execution plan of the workflow depicted in Figure 6.4b and subsequent reprogramming with the parallel workflow depicted in Figure 6.4c and Figure 6.6 followed by the execution of this new workflow.

CHAPTER 7

Conclusion

In this work we have presented the rtSOA approach for dataflow driven engineering of distributed hard real-time systems. rtSOA is focused on enabling wide-reaching reuse of existing software modules and enabling fast, iterative and incremental development following principles of distributed data processing. The main use case for rtSOA lies within the area of industrial manufacturing which is under market pressure to increase its flexibility and adaptability, both in terms of product volume and product variants. We showed how distributed data processing principles can be used together with service-oriented architectures to achieve rapid reconfiguration of modular production systems, which are positioned to provide the necessary adaptability for manufacturing enterprises. rtSOA execution plans offer deterministic, verifiable real-time properties and can be integrated with event-driven architectures on higher levels of the automation hierarchy. Therefore, we consider the rtSOA approach as an extension to existing research regarding SOAs in industrial environments, which is often either controlled in a centralized service orchestration or has emergent, non-deterministic behavior from a temporal perspective.

The heuristics-based synthesis of time triggered workflow execution plans is an unique aspect of the rtSOA approach. By reducing the time required for the generation of execution plans to a sub-second interval, rtSOA can be integrated in interactive development tools, offering quick response times for design space exploration. Our evaluation shows that a combination of heuristics can solve over 99% of 1.2 million test cases, making heuristics a feasible alternative to exhaustive search methods. The heuristics proved to be, on average, two to three orders of magnitude faster than an approach based on an MILP satisfiability solver. Our own heuristics contribute a high number of unique solutions that were not found by other heuristics from literature. Our prototype implementation of an rtSOA engineering tool and an embedded

runtime has shown that modification of an automation workflow is possible through reconfiguration and rerouting of dataflow while incurring nearly zero downtime. In the following we outline directions for future research based on this work.

Support for decision nodes

The current dataflow semantics of rtSOA do not have any concept of decision nodes which may send output data to different successor nodes depending on internal or external state. Such behavior may be desirable, for example to provide different execution paths under failure conditions. The closest equivalent in the current rtSOA semantics would be nodes with multiple successors that send output data to all successors where the successor nodes would then individually decide whether or not they will act upon the data. Each node could then potentially omit the emission of data tokens of their own, thus deactivating certain execution paths. However, each of the tasks on those execution paths would still be included in an rtSOA execution plan, even if those paths are mutually exclusive. Explicit decision nodes could be used to indicate such mutually exclusive paths, allowing rtSOA to schedule them in an "either / or" fashion, thus reducing unnecessary overhead in such scenarios.

Service placement and composition

In the current state, rtSOA supports engineers in determining a suitable service placement only by providing quick feedback about the feasibility and schedulability of their manual service placement and service composition. Algorithms for determining suitable service placements and service compositions in a (semi-)automatic way, taking semantic descriptions and ontologies of the services and their execution environment into account, would further increase the development velocity of rtSOA-based systems. Domain specific heuristics for task placement could be applied to this problem as well, mirroring the rtSOA principles for deriving schedule based execution plans.

Integration with standard protocols

Chapter 2 has outlined the comprehensive efforts of research and industry in the area of service-oriented architectures for manufacturing systems. The work presented in this thesis has considered rtSOA as an abstract engineering approach with generalized execution semantics. Integration or adaption of the rtSOA principles to existing standards, for example the IEC 61499 standard for distributed automation systems or the use of OPC-UA or DPWS with rtSOA, would further increase the applicability of rtSOA for next generation manufacturing systems or in the Industrie 4.0 initiative.

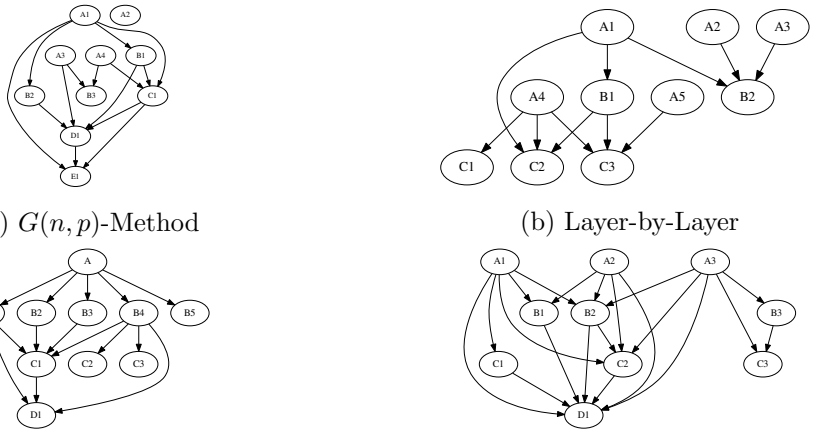
APPENDIX A

Testbench and Benchmark Data Set

Since industrial use cases span a wide range of potential layouts of the resulting task graphs, we rely on synthetic benchmarks, based on several well-known graph generation methods [25]. Section A.1 describes these graph generation methods in more detail while Section A.2 explains how we generated our benchmark data set and explores its properties. The data set described here forms the basis of the evaluation in Chapter 4.

A.1 Graph Generation Methods

We reimplemented the four different task graph generation method described by Cordeiro et al. [25]. Example graphs from the different generation methods are shown in Figure A.1.



(a) $G(n, p)$-Method

(b) Layer-by-Layer

(c) Task Graphs for Free

(d) Random Orders

Figure A.1: Example output of the used graph generation methods

The $G(n, p)$ method

The $G(n, p)$ method is a simple graph generation method, shown in Algorithm 8, that yields every possible DAG with the same output probability [25]. The definition is given by Cordeiro et al. as:

> "For a given n number of vertices, the $G(n, p)$ method generates a graph where each element of the $\binom{n}{2}$ possible edges is present with independent probability p." [25]

Algorithm 8 The $G(n, p)$ method (from [25])

```
1: function G(n ∈ ℕ, p ∈ ℝ)
2:     Let M be an adjacency matrix n × n initialized as the zero matrix.
3:     for all i = 1 to n do
4:         for all j = 1 to i do
5:             if random() < p then
6:                 M[i][j] ← 1
7:             else
8:                 M[i][j] ← 0
9:     return graph represented by M
```

Layer-by-Layer

This method was proposed by Tobita and Kasahara [107] for the evaluation of multiprocessor scheduling algorithms. The definition of a layer is equivalent to our definition of the top-level $t_i.L^t$ of a task t_i as $min(\lceil t_r \prec\!\!\!\prec t_i \rceil)$ for $t_r \in Roots$ (c.f. Chapter 4). Pseudocode for this method is shown in Algorithm 9.

Algorithm 9 The Layer-by-Layer method (from [25])

```
1: function Layer − by − Layer(n ∈ ℕ, k ∈ ℕ, p ∈ ℕ)
2:     Distribute n vertices between k sets enumerated as L_1, ..., L_k
3:     Let layer(v) be the layer assigned to vertex v
4:     Let M be an adjacency matrix n × n initialized as the zero matrix.
5:     for all i = 1 to n do
6:         for all j = 1 to n do
7:             if layer(j) > layer(i) then
8:                 if random() < p then
9:                     M[i][j] ← 1
10:                else
11:                    M[i][j] ← 0
12:    return random DAG with k layers and n nodes
```

Random Orders

This method, proposed by Winkler [119], generates a partial order, i.e.,a DAG, by intersecting two or more total orders. The pseudocode for this method is shown in Algorithm 10.

Algorithm 10 Random Orders method (from [25])

function *RandomOrders*($n \in \mathbb{N}, k \in \mathbb{N}$)
  Initialize graph $G = (V, E)$ with $V = \emptyset, E = \emptyset$
  Generate k random permutations of V
  for all $t_1, t_2 \in V$ **do**
    if $t_1 < t_2$ in all k permutations **then**
      Add edge t_2, t_1 to E
  return DAG with n vertices

Fan-in / Fan-out

Dick et al. [33] published this task graph generation method under the name *Task Graphs for Free*, the name "Fan-in / Fan-out" is the terminology used by Cordeiro et al. However, this name better captures the essence of the method, as the two specifiable parameters are the maximum in-degree and out-degree of a vertex in the task-DAG as shown in Algorithm 11. For example, a binary tree would be generated by this method when using an in-degree of one and an out-degree of two.

Algorithm 11 The Fan-in / Fan-out method (from [25])

function $FanIn - FanOut(n \in \mathbb{N}, ind \in \mathbb{N}, outd \in \mathbb{N})$
  Initialize graph $G = (V, E)$ with $V = \emptyset, E = \emptyset$
  $V \cup t$
  while |V| < n **do**
    if $Random() < 0.5$ **then** ▷ Fan-out phase:
      $t_{source} \leftarrow \arg\max(outd - |Succ(t)|)$ over $t \in V$
      Add up to $outd - |Succ(t_{source})|$ tasks to V
      Add edges from t_{source} to new tasks
    else ▷ Fan-in phase:
      $S = \{t \in V | outd > |Succ(t)|\}$
      Compute subset $T \subseteq S$ with $|T| \leq ind$
      $V \cup t_{sink}$
      Add edges to t_{sink} for all $t \in T$
  return DAG with $\geq n$ tasks having out-degree $\leq outd$ and in-degree $\leq ind$

A.2 Benchmark Data Set

We generated random workflows with either 16, 32, 48 or 64 tasks that we then randomly distributed onto 2, 4 or 8 machines. The tasks were then assigned a random WCET so that the total schedule utilization was between 12.5% and 75%. An utilization of 100% for a scheduling problem with 8 machines means that all 8 CPUs would be busy for 100% of the time. We then used the MILP formulation described in Section 4.1.2 to determine the feasibility of the scheduling problem. This process was continued until we collected 1.2 million feasible test cases. Each combination of machine number and task count is represented 100 000 times in the data set, distributed between the different graph generation methods described above. 50% of the data set is generated with the $G(n, p)$ method, 20% are generated with the Layer-by-Layer method and the Fan-in / Fan-out method, each. The last 10% were generated with the Random Orders method.

Our approach to generating scheduling problems will not lead to feasible workflows with the same probability across all parameters used for the generation methods. Easier problems may be represented more often than harder problems, as we explain in the remainder of this appendix chapter. Unfortunately, we did not posses the computational resources necessary to generate a truly uniformly distributed data set. The computation time for the feasible workflows alone on a 12-core Intel Core i7-3930K was already *over 200 CPU days* with the infeasible scheduling problems adding another multiple on top. The author estimates that the generation of the benchmark set in its current state took over three months of non-stop computation.

The following figures visualize the respective frequency of workflows generated by the graph generation methods described earlier, plotted against the graph generation method parameters.

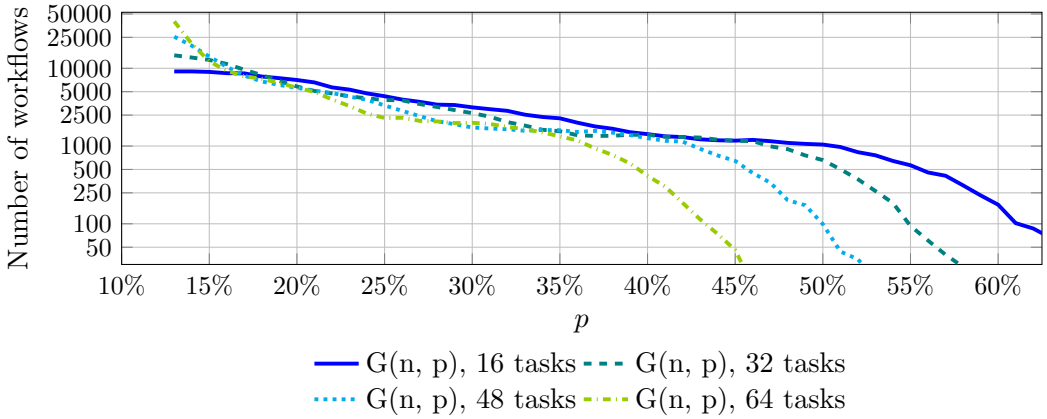


Figure A.2: Distribution of graphs generated by the $G(n, p)$ method, plotted against their frequency in the test data set

Figure A.2 shows the relative frequency of workflows generated by varying the edge probability p in the $G(n, p)$ graph generation method. Most graphs have a edge probability $< 20\%$, which might indicate relatively loosely connected graphs. Figure A.3 studies the relation between the workflows in our data set and their number of edges more closely. It shows that most graphs are indeed well connected and that isolated nodes can only be expected for the 16-task workflows. The reason for this is indicated by the two example graphs in Figure A.4, showing that the relative number of edges to tasks increases in workflows generated with the $G(n, p)$ method when the number of tasks is increased while the edge probability is kept constant. In fact, the number of edges increases quadratically when the number of tasks is increased linearly.

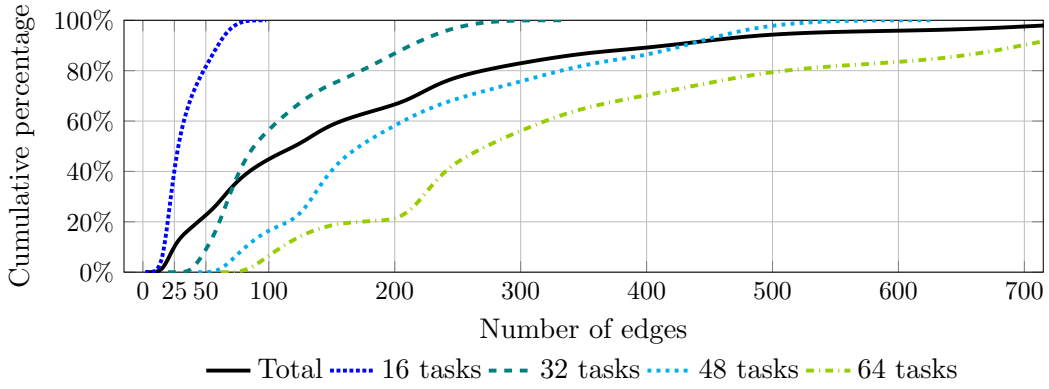
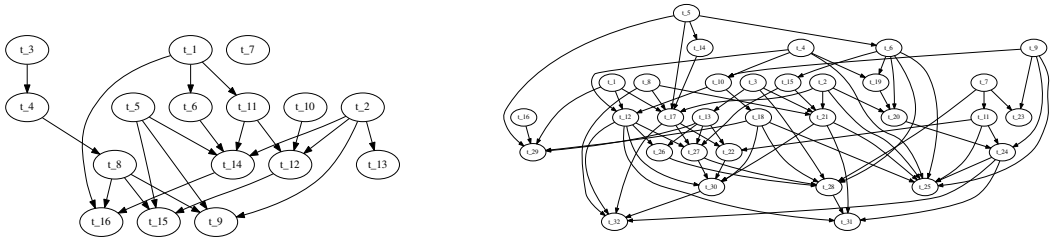


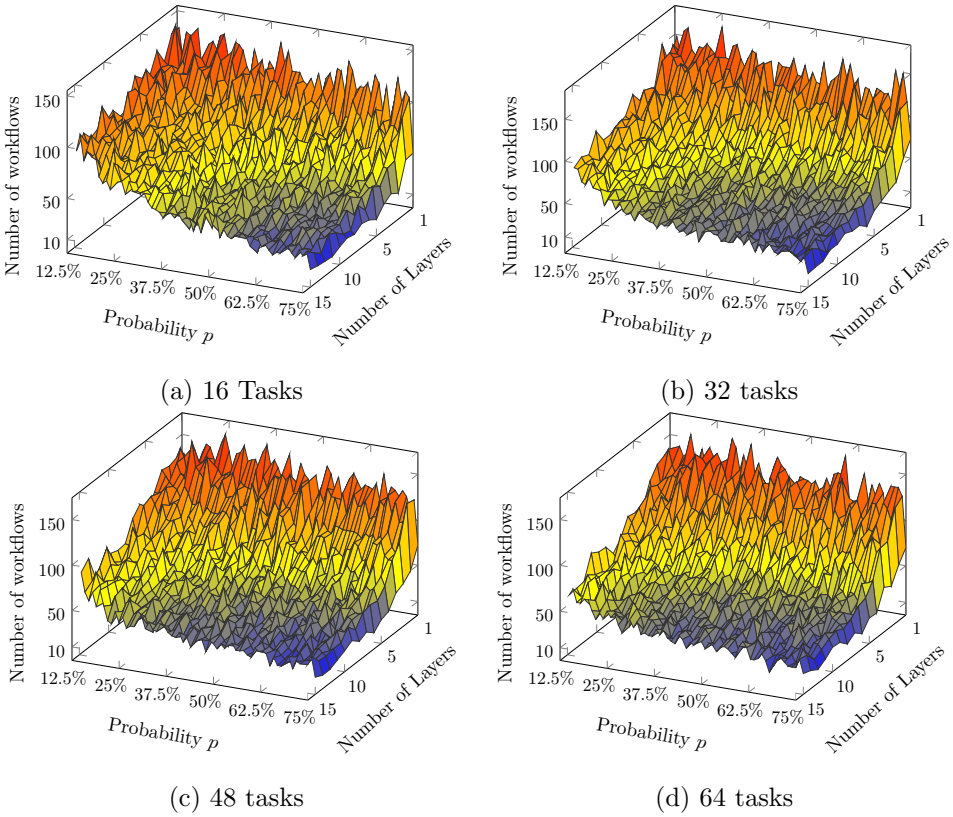
Figure A.3: Cumulative distribution of workflows with a given number of edges in the benchmark set



(a) $G(16, 0.125)$

(b) $G(32, 0.125)$

Figure A.4: Two example graphs generated by the $G(n, p)$ method, showing that the degree of connectedness rises with an increasing number of tasks n when keeping the edge probability p the same.



(a) 16 Tasks (b) 32 tasks

(c) 48 tasks (d) 64 tasks

Figure A.5: Distribution of graphs generated by the Layer-by-Layer method, plotted against their frequency in the test data set

Figure A.5 is analyzing the frequency of test cases generated by calls to the Layer-by-Layer method with different parameters for the number of layers k and the edge probability p. As with the $G(n, p)$ method, the benchmark set contains a smaller number of workflows with high edge probability p. Additionally, the benchmark set contains relatively more test cases with fewer levels than with more levels. Figure A.6 Shows that most workflows have between 4 and 12 levels but with a relatively long tail.

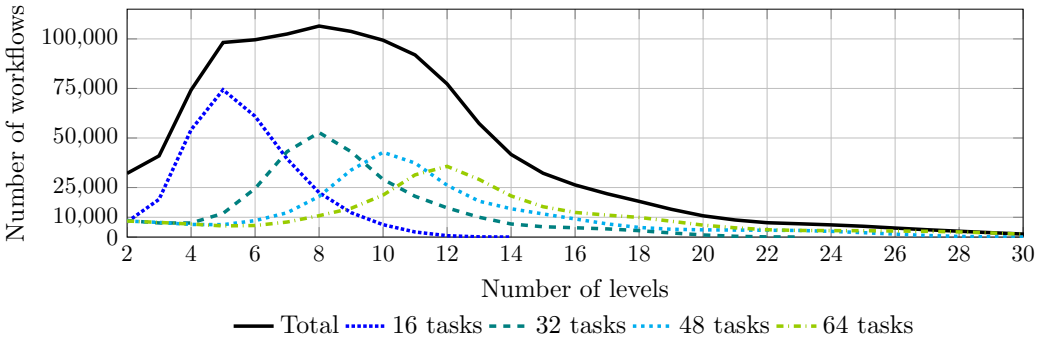
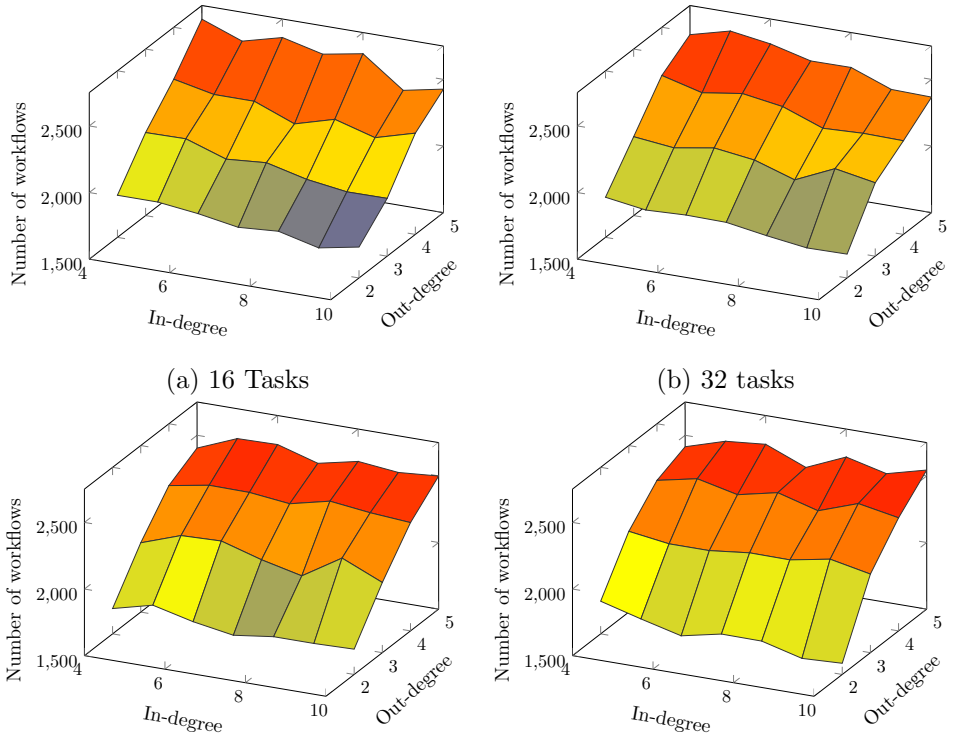


Figure A.6: Number of workflows with a given number of levels in the data set



(a) 16 Tasks (b) 32 tasks

(c) 48 tasks (d) 64 tasks

Figure A.7: Distribution of graphs generated by the Fan-in / Fan-out method, plotted against their frequency in the test data set

Lastly we analyzed the frequency of workflows generated with differing parameters for the Fan-in / Fan-out method as shown in Figure A.7. No such analysis is necessary for the Random-Orders method because we always generated the workflows by intersecting two random permutations, as this lead to the larges variation in the out-degree of the generated tasks. One more interesting characteristic of the data set is the Figure A.8 distribution of scheduling problems in regard to their processor utilization.

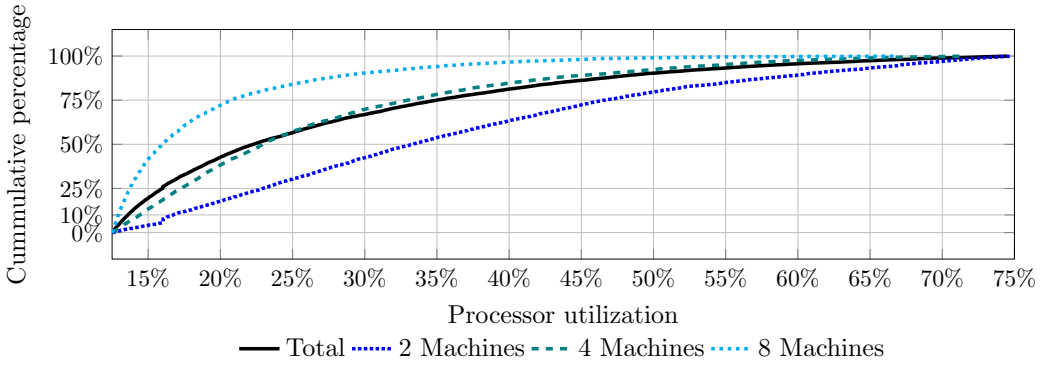


Figure A.8: Cummulative distribution of test cases with a given processor utilization in the benchmark data set. Workflows do not vary significantly from the overall distribution shown here based on the number of tasks.

A.3 Testbench

This section covers the technical details of how we generated the benchmark data set described in the previous section. Figure A.9 shows a domain model of our testbench program which also visualizes the processing pipeline for workflow tasks in the testbench: Initially, a workflow is created that represents a DAG with a global deadline and period. Over time, more and more information is attached to the task. After assignment to a machine, the task is also assigned a WCET. It is then further enriched with a local deadline and release time, after which the task can be scheduled. The final processing step is the verification of the overall schedule through simulation on a given network topology.

By using the testbench, a user can generate workflows with the graph generation methods described in Section A.1, use the methods from Chapter 4 and trigger verification through simulation or model checking as described in Chapter 5. Any output of the processing pipeline can be persisted to a database during any step via the Hibernate object-relational-mapping (ORM) framework. The user can also resume work by loading input from the database. The testbench is configured via a declarative query language which we explain in Section A.3.1. Section A.3.2 lists the commands we used to generate our benchmark set.

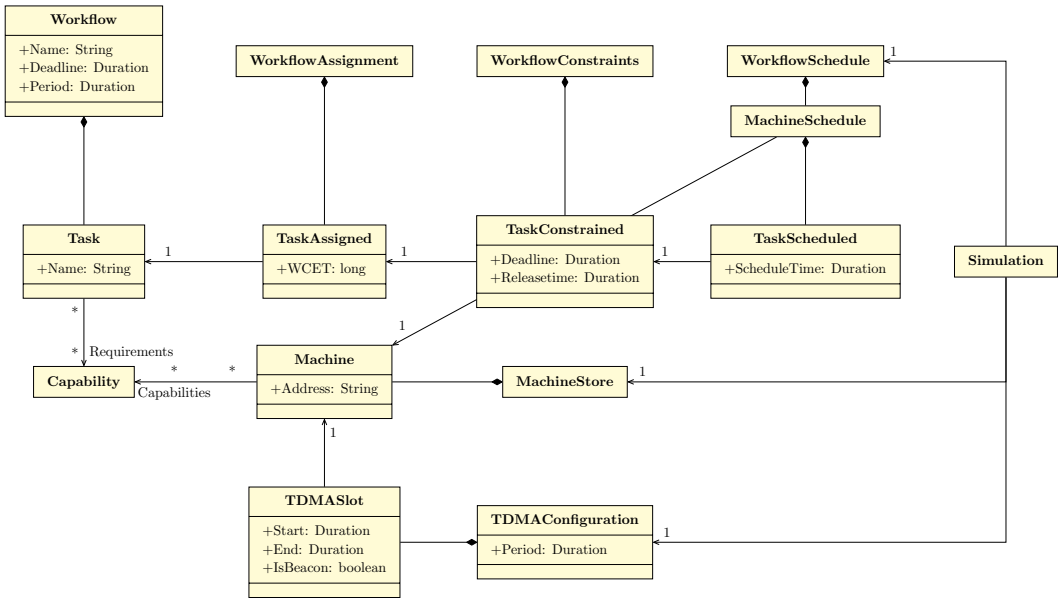


Figure A.9: The domain model for the rtSOA testbench in UML. Internal classes and database tables roughly follow the same model, but are omitted for reasons of complexity.

A.3.1 Specification Language

This section describes the grammar of the testbench configuration language we used to generate our data set. The specification of the data set itself can be found in the next section (Section A.3.2). On a a conceptual level, the specification language declares a number of streams that generate tuples from one of the classes shown in the domain model (Figure A.9). For example, the workflows command in Figure A.14 generates tuples of type "Workflow" which group a number additional tuples of type "Task". For each of the composition classes in Figure A.9 there exists a generator command. The WCET-analysis command (Figure A.13) is an exception as it has no representation in the domain model. This command simply generates a mapping of machine and task to WCET, that is it specified the WCET of a given task on all the possible machines.

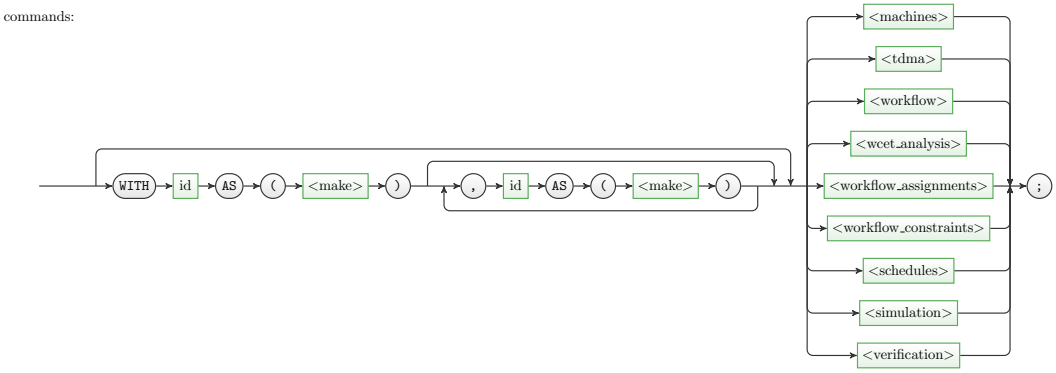


Figure A.10: High-level structure of the test case specification grammar.

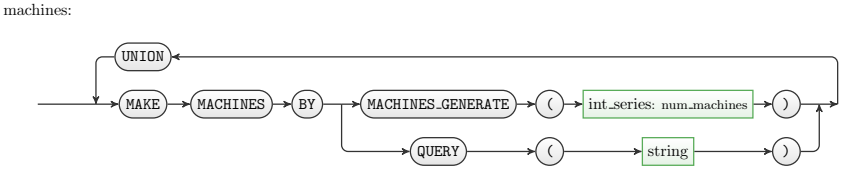


Figure A.11: Grammar for generating Machine objects.

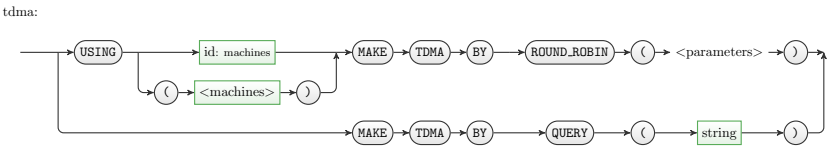


Figure A.12: Grammar for generating a TDMA configuration.

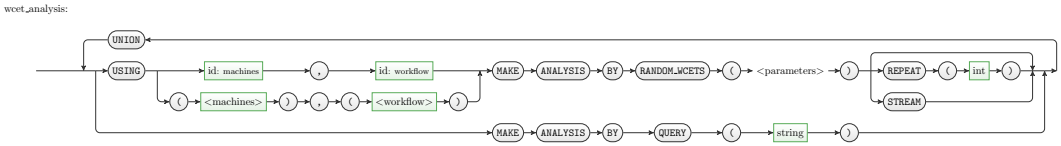


Figure A.13: Grammar for generating random WCET values for tasks.

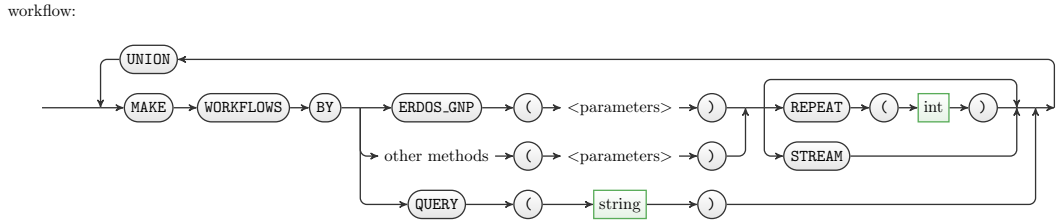


Figure A.14: Grammar for generating workflows.

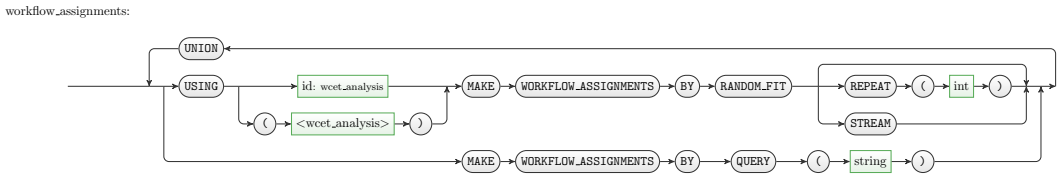


Figure A.15: Grammar for assigning workflow tasks to machines.

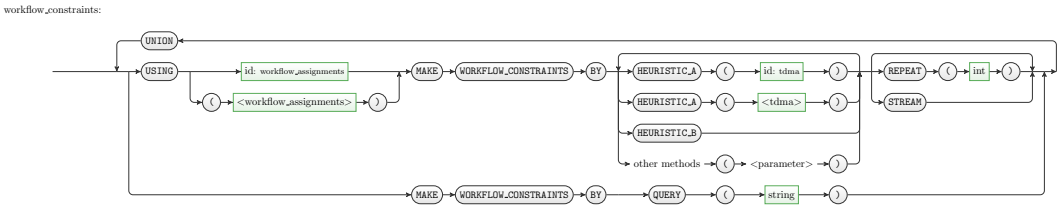


Figure A.16: Grammar for assigning deadlines and release times.

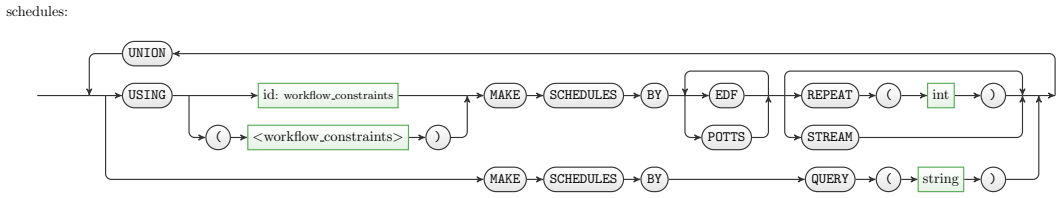


Figure A.17: Grammar for scheduling times.

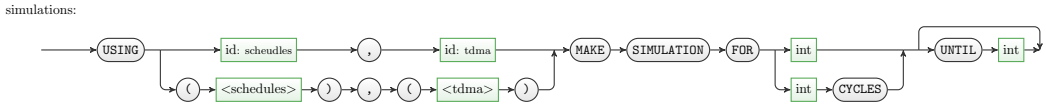


Figure A.18: Grammar for simulating scheduled tasks on a given network topology.

A.3.2 Data Set Specification

This section serves as a reference for how the benchmark data set was generated. Listings A.1 to A.4 specify the declarations used to generate the corpus of feasible problem instances. Listing A.5 specifies how these feasible problem instances were then used to evaluate the efficacy of the implemented heuristics (Section 4.2).

```
WITH
  m    as (MAKE MACHINES BY machines_generate(2)),
  td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)),
  # workflow with 16 tasks and random edge chance between 12.5% and 75%
  wf   as (MAKE WORKFLOWS BY erdos_gnp(16, rnd(0.125, 0.75), 10000000, 10000000) STREAM),
  # generate WCETS with random utilization between 12.5% and 75%
  ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125, 0.75) STREAM),
  a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
  # use Gurobi-solver to identify feasible problem instances
  c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
  sched as (USING c MAKE SCHEDULES BY potts STREAM)
# keep generating problems until we have 50000 feasible instances
USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 50000;

WITH
  m    as (MAKE MACHINES BY machines_generate(2)),
  td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)),
  wf   as (MAKE WORKFLOWS BY erdos_gnp(32, rnd(0.125, 0.75), 10000000, 10000000) STREAM),
  ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125, 0.75) STREAM),
  a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
  c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
  sched as (USING c MAKE SCHEDULES BY potts STREAM)
USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 50000;

WITH
  m    as (MAKE MACHINES BY machines_generate(2)),
  td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)),
  wf   as (MAKE WORKFLOWS BY erdos_gnp(48, rnd(0.125, 0.75), 10000000, 10000000) STREAM),
  ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125, 0.75) STREAM),
  a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
  c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
  sched as (USING c MAKE SCHEDULES BY potts STREAM)
USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 50000;

WITH
  m    as (MAKE MACHINES BY machines_generate(2)),
  td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)),
  wf   as (MAKE WORKFLOWS BY erdos_gnp(64, rnd(0.125, 0.75), 10000000, 10000000) STREAM),
  ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125, 0.75) STREAM),
  a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
  c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
  sched as (USING c MAKE SCHEDULES BY potts STREAM)
USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 50000;

WITH
  m    as (MAKE MACHINES BY machines_generate(4)),
  td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)),
  wf   as (MAKE WORKFLOWS BY erdos_gnp(16, rnd(0.125, 0.75), 10000000, 10000000) STREAM),
  ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125, 0.75) STREAM),
  a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
  c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
  sched as (USING c MAKE SCHEDULES BY potts STREAM)
USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 50000;

WITH
  m    as (MAKE MACHINES BY machines_generate(4)),
```

```
57    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
58    wf  as (MAKE WORKFLOWS BY erdos_gnp(32, rnd(0.125, 0.75), 10000000,
        10000000) STREAM),
59    ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
        , 0.75) STREAM),
60    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
61    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
62    sched as (USING c MAKE SCHEDULES BY potts STREAM)
63  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 50000;
64
65  WITH
66    m   as (MAKE MACHINES BY machines_generate(4)),
67    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
68    wf  as (MAKE WORKFLOWS BY erdos_gnp(48, rnd(0.125, 0.75), 10000000,
        10000000) STREAM),
69    ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
        , 0.75) STREAM),
70    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
71    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
72    sched as (USING c MAKE SCHEDULES BY potts STREAM)
73  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 50000;
74
75  WITH
76    m   as (MAKE MACHINES BY machines_generate(4)),
77    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
78    wf  as (MAKE WORKFLOWS BY erdos_gnp(64, rnd(0.125, 0.75), 10000000,
        10000000) STREAM),
79    ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
        , 0.75) STREAM),
80    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
81    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
82    sched as (USING c MAKE SCHEDULES BY potts STREAM)
83  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 50000;
84
85  WITH
86    m   as (MAKE MACHINES BY machines_generate(8)),
87    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
88    wf  as (MAKE WORKFLOWS BY erdos_gnp(16, rnd(0.125, 0.75), 10000000,
        10000000) STREAM),
89    ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
        , 0.75) STREAM),
90    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
91    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
92    sched as (USING c MAKE SCHEDULES BY potts STREAM)
93  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 50000;
94
95  WITH
96    m   as (MAKE MACHINES BY machines_generate(8)),
97    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
98    wf  as (MAKE WORKFLOWS BY erdos_gnp(32, rnd(0.125, 0.75), 10000000,
        10000000) STREAM),
99    ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
        , 0.75) STREAM),
100   a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
101   c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
102   sched as (USING c MAKE SCHEDULES BY potts STREAM)
103 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 50000;
104
105 WITH
106   m   as (MAKE MACHINES BY machines_generate(8)),
107   td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
108   wf  as (MAKE WORKFLOWS BY erdos_gnp(48, rnd(0.125, 0.75), 10000000,
        10000000) STREAM),
109   ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
        , 0.75) STREAM),
110   a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
```

```
111   c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
112   sched as (USING c MAKE SCHEDULES BY potts STREAM)
113 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 50000;
114
115 WITH
116   m   as (MAKE MACHINES BY machines_generate(8)),
117   td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
118   wf  as (MAKE WORKFLOWS BY erdos_gnp(64, rnd(0.125, 0.75), 10000000,
        10000000) STREAM),
119   ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
        , 0.75) STREAM),
120   a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
121   c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
122   sched as (USING c MAKE SCHEDULES BY potts STREAM)
123 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 50000;
```

Listing A.1: Definitions for the part benchmark data set that was generated with the $G(n, p)$ method. Comments are prefixed with #.

```
1
2   WITH
3     m   as (MAKE MACHINES BY machines_generate(2)),
4     td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
5     wf  as (MAKE WORKFLOWS BY layer_by_layer(16, rnd(2, 15), rnd(0.125,
        0.75), 10000000, 10000000) STREAM),
6     ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
        , 0.75) STREAM),
7     a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
8     c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
9     sched as (USING c MAKE SCHEDULES BY potts STREAM)
10  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
11
12  WITH
13    m   as (MAKE MACHINES BY machines_generate(2)),
14    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
15    wf  as (MAKE WORKFLOWS BY layer_by_layer(32, rnd(2, 15), rnd(0.125,
        0.75), 10000000, 10000000) STREAM),
16    ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
        , 0.75) STREAM),
17    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
18    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
19    sched as (USING c MAKE SCHEDULES BY potts STREAM)
20  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
21
22  WITH
23    m   as (MAKE MACHINES BY machines_generate(2)),
24    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
25    wf  as (MAKE WORKFLOWS BY layer_by_layer(48, rnd(2, 15), rnd(0.125,
        0.75), 10000000, 10000000) STREAM),
26    ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
        , 0.75) STREAM),
27    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
28    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
29    sched as (USING c MAKE SCHEDULES BY potts STREAM)
30  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
31
32  WITH
33    m   as (MAKE MACHINES BY machines_generate(2)),
34    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
35    wf  as (MAKE WORKFLOWS BY layer_by_layer(64, rnd(2, 15), rnd(0.125,
        0.75), 10000000, 10000000) STREAM),
36    ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
        , 0.75) STREAM),
37    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
38    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
39    sched as (USING c MAKE SCHEDULES BY potts STREAM)
```

```
40 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
41
42 WITH
43   m    as (MAKE MACHINES BY machines_generate(4)),
44   td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
       ),
45   wf   as (MAKE WORKFLOWS BY layer_by_layer(16, rnd(2, 15), rnd(0.125,
       0.75), 10000000, 10000000) STREAM),
46   ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
       , 0.75) STREAM),
47   a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
48   c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
49   sched as (USING c MAKE SCHEDULES BY potts STREAM)
50 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
51
52 WITH
53   m    as (MAKE MACHINES BY machines_generate(4)),
54   td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
       ),
55   wf   as (MAKE WORKFLOWS BY layer_by_layer(32, rnd(2, 15), rnd(0.125,
       0.75), 10000000, 10000000) STREAM),
56   ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
       , 0.75) STREAM),
57   a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
58   c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
59   sched as (USING c MAKE SCHEDULES BY potts STREAM)
60 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
61
62 WITH
63   m    as (MAKE MACHINES BY machines_generate(4)),
64   td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
       ),
65   wf   as (MAKE WORKFLOWS BY layer_by_layer(48, rnd(2, 15), rnd(0.125,
       0.75), 10000000, 10000000) STREAM),
66   ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
       , 0.75) STREAM),
67   a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
68   c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
69   sched as (USING c MAKE SCHEDULES BY potts STREAM)
70 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
71
72 WITH
73   m    as (MAKE MACHINES BY machines_generate(4)),
74   td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
       ),
75   wf   as (MAKE WORKFLOWS BY layer_by_layer(64, rnd(2, 15), rnd(0.125,
       0.75), 10000000, 10000000) STREAM),
76   ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
       , 0.75) STREAM),
77   a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
78   c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
79   sched as (USING c MAKE SCHEDULES BY potts STREAM)
80 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
81
82 WITH
83   m    as (MAKE MACHINES BY machines_generate(8)),
84   td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
       ),
85   wf   as (MAKE WORKFLOWS BY layer_by_layer(16, rnd(2, 15), rnd(0.125,
       0.75), 10000000, 10000000) STREAM),
86   ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
       , 0.75) STREAM),
87   a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
88   c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
89   sched as (USING c MAKE SCHEDULES BY potts STREAM)
90 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
91
92 WITH
93   m    as (MAKE MACHINES BY machines_generate(8)),
94   td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
       ),
```

```
95    wf  as (MAKE WORKFLOWS BY layer_by_layer(32, rnd(2, 15), rnd(0.125,
        0.75), 10000000, 10000000) STREAM),
96    ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
        , 0.75) STREAM),
97    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
98    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
99    sched as (USING c MAKE SCHEDULES BY potts STREAM)
100 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
101
102 WITH
103   m   as (MAKE MACHINES BY machines_generate(8)),
104   td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
105   wf  as (MAKE WORKFLOWS BY layer_by_layer(48, rnd(2, 15), rnd(0.125,
        0.75), 10000000, 10000000) STREAM),
106   ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
        , 0.75) STREAM),
107   a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
108   c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
109   sched as (USING c MAKE SCHEDULES BY potts STREAM)
110 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
111
112 WITH
113   m   as (MAKE MACHINES BY machines_generate(8)),
114   td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
115   wf  as (MAKE WORKFLOWS BY layer_by_layer(64, rnd(2, 15), rnd(0.125,
        0.75), 10000000, 10000000) STREAM),
116   ana as (MAKE ANALYSIS BY random_wcets(wf , m, 50000, 10000000, 0.125
        , 0.75) STREAM),
117   a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
118   c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
119   sched as (USING c MAKE SCHEDULES BY potts STREAM)
120 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
```

Listing A.2: Definitions for the part benchmark data set that was generated with the Layer-by-Layer method

```
1
2  WITH
3    m   as (MAKE MACHINES BY machines_generate(2)),
4    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
       ),
5    wf  as (MAKE WORKFLOWS BY fanin_fanout(16, rnd(4, 10), rnd(2, 5), 10
       000000, 10000000) STREAM),
6    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
       rnd(0.125, 0.75)) STREAM),
7    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
8    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
9    sched as (USING c MAKE SCHEDULES BY potts STREAM)
10 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
11
12 WITH
13   m   as (MAKE MACHINES BY machines_generate(2)),
14   td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
       ),
15   wf  as (MAKE WORKFLOWS BY fanin_fanout(32, rnd(4, 10), rnd(2, 5), 10
       000000, 10000000) STREAM),
16   ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
       rnd(0.125, 0.75)) STREAM),
17   a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
18   c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
19   sched as (USING c MAKE SCHEDULES BY potts STREAM)
20 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
21
22 WITH
23   m   as (MAKE MACHINES BY machines_generate(2)),
24   td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
       ),
25   wf  as (MAKE WORKFLOWS BY fanin_fanout(48, rnd(4, 10), rnd(2, 5), 10
       000000, 10000000) STREAM),
```

```
26    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
27    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
28    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
29    sched as (USING c MAKE SCHEDULES BY potts STREAM)
30  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
31
32  WITH
33    m    as (MAKE MACHINES BY machines_generate(2)),
34    td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
35    wf   as (MAKE WORKFLOWS BY fanin_fanout(64, rnd(4, 10), rnd(2, 5), 10
        000000, 10000000) STREAM),
36    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
37    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
38    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
39    sched as (USING c MAKE SCHEDULES BY potts STREAM)
40  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
41
42  WITH
43    m    as (MAKE MACHINES BY machines_generate(4)),
44    td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
45    wf   as (MAKE WORKFLOWS BY fanin_fanout(16, rnd(4, 10), rnd(2, 5), 10
        000000, 10000000) STREAM),
46    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
47    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
48    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
49    sched as (USING c MAKE SCHEDULES BY potts STREAM)
50  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
51
52  WITH
53    m    as (MAKE MACHINES BY machines_generate(4)),
54    td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
55    wf   as (MAKE WORKFLOWS BY fanin_fanout(32, rnd(4, 10), rnd(2, 5), 10
        000000, 10000000) STREAM),
56    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
57    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
58    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
59    sched as (USING c MAKE SCHEDULES BY potts STREAM)
60  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
61
62  WITH
63    m    as (MAKE MACHINES BY machines_generate(4)),
64    td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
65    wf   as (MAKE WORKFLOWS BY fanin_fanout(48, rnd(4, 10), rnd(2, 5), 10
        000000, 10000000) STREAM),
66    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
67    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
68    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
69    sched as (USING c MAKE SCHEDULES BY potts STREAM)
70  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
71
72  WITH
73    m    as (MAKE MACHINES BY machines_generate(4)),
74    td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
75    wf   as (MAKE WORKFLOWS BY fanin_fanout(64, rnd(4, 10), rnd(2, 5), 10
        000000, 10000000) STREAM),
76    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
77    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
78    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
79    sched as (USING c MAKE SCHEDULES BY potts STREAM)
80  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
81
```

```
WITH
  m    as (MAKE MACHINES BY machines_generate(8)),
  td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)),
  wf   as (MAKE WORKFLOWS BY fanin_fanout(16, rnd(4, 10), rnd(2, 5), 10000000, 10000000) STREAM),
  ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000, rnd(0.125, 0.75)) STREAM),
  a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
  c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
  sched as (USING c MAKE SCHEDULES BY potts STREAM)
USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;

WITH
  m    as (MAKE MACHINES BY machines_generate(8)),
  td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)),
  wf   as (MAKE WORKFLOWS BY fanin_fanout(32, rnd(4, 10), rnd(2, 5), 10000000, 10000000) STREAM),
  ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000, rnd(0.125, 0.75)) STREAM),
  a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
  c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
  sched as (USING c MAKE SCHEDULES BY potts STREAM)
USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;

WITH
  m    as (MAKE MACHINES BY machines_generate(8)),
  td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)),
  wf   as (MAKE WORKFLOWS BY fanin_fanout(48, rnd(4, 10), rnd(2, 5), 10000000, 10000000) STREAM),
  ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000, rnd(0.125, 0.75)) STREAM),
  a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
  c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
  sched as (USING c MAKE SCHEDULES BY potts STREAM)
USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;

WITH
  m    as (MAKE MACHINES BY machines_generate(8)),
  td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)),
  wf   as (MAKE WORKFLOWS BY fanin_fanout(64, rnd(4, 10), rnd(2, 5), 10000000, 10000000) STREAM),
  ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000, rnd(0.125, 0.75)) STREAM),
  a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
  c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
  sched as (USING c MAKE SCHEDULES BY potts STREAM)
USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 20000;
```

Listing A.3: Definitions for the part benchmark data set that was generated with the Fan-In / Fan-Out method

```

WITH
  m    as (MAKE MACHINES BY machines_generate(2)),
  td   as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)),
  wf   as (MAKE WORKFLOWS BY random_orders(16, 2, 10000000, 10000000) STREAM),
  ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000, rnd(0.125, 0.75)) STREAM),
  a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
  c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
  sched as (USING c MAKE SCHEDULES BY potts STREAM)
USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 10000;

WITH
  m    as (MAKE MACHINES BY machines_generate(2)),
```

```
14    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
15    wf  as (MAKE WORKFLOWS BY random_orders(32, 2, 10000000, 10000000)
        STREAM),
16    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
17    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
18    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
19    sched as (USING c MAKE SCHEDULES BY potts STREAM)
20  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 10000;
21
22  WITH
23    m   as (MAKE MACHINES BY machines_generate(2)),
24    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
25    wf  as (MAKE WORKFLOWS BY random_orders(48, 2, 10000000, 10000000)
        STREAM),
26    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
27    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
28    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
29    sched as (USING c MAKE SCHEDULES BY potts STREAM)
30  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 10000;
31
32  WITH
33    m   as (MAKE MACHINES BY machines_generate(2)),
34    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
35    wf  as (MAKE WORKFLOWS BY random_orders(64, 2, 10000000, 10000000)
        STREAM),
36    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
37    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
38    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
39    sched as (USING c MAKE SCHEDULES BY potts STREAM)
40  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 10000;
41
42  WITH
43    m   as (MAKE MACHINES BY machines_generate(4)),
44    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
45    wf  as (MAKE WORKFLOWS BY random_orders(16, 2, 10000000, 10000000)
        STREAM),
46    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
47    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
48    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
49    sched as (USING c MAKE SCHEDULES BY potts STREAM)
50  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 10000;
51
52  WITH
53    m   as (MAKE MACHINES BY machines_generate(4)),
54    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
55    wf  as (MAKE WORKFLOWS BY random_orders(32, 2, 10000000, 10000000)
        STREAM),
56    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
57    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
58    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
59    sched as (USING c MAKE SCHEDULES BY potts STREAM)
60  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 10000;
61
62  WITH
63    m   as (MAKE MACHINES BY machines_generate(4)),
64    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
65    wf  as (MAKE WORKFLOWS BY random_orders(48, 2, 10000000, 10000000)
        STREAM),
66    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
67    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
```

```
68    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
69    sched as (USING c MAKE SCHEDULES BY potts STREAM)
70  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 10000;
71
72  WITH
73    m   as (MAKE MACHINES BY machines_generate(4)),
74    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
75    wf  as (MAKE WORKFLOWS BY random_orders(64, 2, 10000000, 10000000)
        STREAM),
76    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
77    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
78    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
79    sched as (USING c MAKE SCHEDULES BY potts STREAM)
80  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 10000;
81
82  WITH
83    m   as (MAKE MACHINES BY machines_generate(8)),
84    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
85    wf  as (MAKE WORKFLOWS BY random_orders(16, 2, 10000000, 10000000)
        STREAM),
86    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
87    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
88    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
89    sched as (USING c MAKE SCHEDULES BY potts STREAM)
90  USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 10000;
91
92  WITH
93    m   as (MAKE MACHINES BY machines_generate(8)),
94    td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
95    wf  as (MAKE WORKFLOWS BY random_orders(32, 2, 10000000, 10000000)
        STREAM),
96    ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
97    a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
98    c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
99    sched as (USING c MAKE SCHEDULES BY potts STREAM)
100 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 10000;
101
102 WITH
103   m   as (MAKE MACHINES BY machines_generate(8)),
104   td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
105   wf  as (MAKE WORKFLOWS BY random_orders(48, 2, 10000000, 10000000)
        STREAM),
106   ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
107   a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
108   c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
109   sched as (USING c MAKE SCHEDULES BY potts STREAM)
110 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 10000;
111
112 WITH
113   m   as (MAKE MACHINES BY machines_generate(8)),
114   td  as (USING m MAKE TDMA BY round_robin broadcast (10000000, 50000)
        ),
115   wf  as (MAKE WORKFLOWS BY random_orders(64, 2, 10000000, 10000000)
        STREAM),
116   ana as (USING m, wf MAKE ANALYSIS BY random_wcets(50000, 10000000,
        rnd(0.125, 0.75)) STREAM),
117   a as (USING ana MAKE WORKFLOW_ASSIGNMENTS BY first_fit STREAM),
118   c as (USING a MAKE WORKFLOW_CONSTRAINTS BY gurobi(td) STREAM),
119   sched as (USING c MAKE SCHEDULES BY potts STREAM)
120 USING sched , td MAKE SIMULATION FOR 2 CYCLES UNTIL 10000;
```

Listing A.4: Definitions for the part benchmark data set that was generated with the Random Orders method

```
1  WITH
2    # Machines are not loaded from database by dependent objects.
       Regenerate
3    m    as (MAKE MACHINES BY machines_generate([2, 4, 8]])),
4    td   as (USING m MAKE TDMA BY round_robin (10000000, 50000)),
5    # Load all workflow assignments from the database
6    a    as (MAKE WORKFLOW_ASSIGNMENTS BY query("")),
7    # Run specified heuristics on the loaded assignments
8    c    as (USING a MAKE WORKFLOW_CONSTRAINTS BY heuristic_a heuristic_b
       ),
9    sched as (USING c MAKE SCHEDULES BY potts)
10 # run simulation to check validity of heuristics' solution
11 USING sched , td MAKE SIMULATION FOR 2 CYCLES;
```

Listing A.5: Definitions for running evaluating the heuristics (Section 4.2) over the known feasible scheduling problems in the data set. The query function in line 4 loads the workflow assignments from the database with the specified filter. Since no filter is specified, all existing assignments are loaded and used as input for generating the workflow constraints. Comments are prefixed with #.

APPENDIX B

Archived Benchmarks of SMT and MILP-Solvers

This appendix chapter archives several benchmarks results of SMT and MILP-Solvers, which were accessed online by the author.

B.1 Benchmark of MILP-Solvers

This is a verbatim copy of the benchmarks performed by Hans Mittelmann, published on http://plato.asu.edu/ftp/milpc.html, accessed on Sept. 26th 2016. The benchmarks show that the Gurobi MILP-solver, empoyed in this thesis, is the fastest commercialy avaliable solver at the time of writing. IBM CPLEX [1] and FICO Xpress [2] are other performant options. CBC [3] is an open source option, but the results show that it is over 20 times slower than Gurobi.

```
15 Jun 2016
=========================================================
Mixed Integer Linear Programming Benchmark (MIPLIB2010)
=========================================================
          H. Mittelmann (mittelmann@asu.edu)

The following codes were run with a limit of 2 hours on the MIPLIB2010 benchmark
set with the MIPLIB2010 scripts (exc Matlab) on two platforms.
1/4 threads: Intel i7-4790K, 4 cores, 32GB, 4GHz, available memory 24GB;
12 threads: Intel Xeon X5680, 12 cores, 32GB, 3.33Ghz, available memory 24GB;
These are updated and extended versions of the results produced for the
MIPLIB2010 paper.
```

[1] https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/
[2] http://www.fico.com/en/products/fico-xpress-optimization-suite
[3] https://projects.coin-or.org/Cbc

```
CPLEX-12.6.3: CPLEX
GUROBI-6.5.0 GUROBI
ug[SCIP/cpx/spx]-3.2.1: Parallel development version of SCIP (SCIP+CPLEX/SOPLEX on
1 thread)
CBC-2.9.8: CBC
XPRESS-8.0.0: XPRESS
MATLAB-2016a: MATLAB (intlinprog)
MIPCL-1.1.2: MIPCL

[...]

Statistics of the problems can be obtained from the MIPLIB2010 webpage.

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Unscaled and scaled shifted geometric means of run times

All non-successes are counted as max-time.
The third line lists the number of problems (87 total) solved.


 1 thr   CBC    CPLEX   GUROBI   SCIPC   SCIPS   XPRESS   MATLAB   MIPCL
-------------------------------------------------------------------------
unscal  1639      86       74      418      517      93     3416      870
scaled    22    1.16        1     5.65     6.99    1.25     46.2     11.8
solved    53      86       86       75       70      86       26       65
-------------------------------------------------------------------------


 4 thr    CBC    CPLEX    FSCIPC   FSCIPS   GUROBI   XPRESS  MIPCL
-----------------------------------------------------------------
unscal     839     46       339      641      39       49      396
scaled    21.8    1.19     8.82      17       1       1.28    10.3
solved     66      86       75       69       87       86      74
-----------------------------------------------------------------


12 thr    CBC    CPLEX    FSCIPC   FSCIPS   GUROBI   XPRESS  MIPCL
-----------------------------------------------------------------
unscal     668     41       327      511      37       44      336
scaled     18     1.11     8.85       14       1       1.20    9.11
solved     69      87       74        70      87       86      76
-----------------------------------------------------------------
MIPCL could only be run with 10 threads
```

B.2 Feasibility Benchmark of MILP-Solvers

This is a verbatim copy of the benchmarks performed by Hans Mittelmann, published on http://plato.asu.edu/ftp/feas_bench.html, accessed on Sept. 26th 2016. The benchmarks show that the Gurobi MILP-solver, empoyed in this thesis, is the fastest commercialy avaliable solver for determining a feasible solution at the time of writing. IBM CPLEX is the only other solver with comparable performance in determining an initially feasible solution. CBC is an open source option, but the results show that it is over 80 times slower than Gurobi.

14 Apr 2016

Feasibility Benchmark

H. Mittelmann (mittelmann@asu.edu)

Logfiles for these runs are at: plato.asu.edu/ftp/feas_bench_logs/

MILP problems mostly from MIPLIB2010 were solved for a feasible point

The following codes were run on an Intel i7-4790K with 4 threads:
CPLEX-12.6.3: CPLEX
FEASPUMP2: as implemented for interactive use at NEOS (utilizes CPLEX)
GUROBI-6.5.0: GUROBI
XPRESS-8.0.0: XPRESS
CBC-2.9.8: CBC

Times given are elapsed times in seconds. A time limit of 1 hr was imposed.
Shifted geometric means of the times are listed. [...]

problem(33 tot)	CPLEX	FP2	GUROBI	XPRESS	CBC
geometric mean	1.14	3.74	1	2.75	80.1
problems solved	33	31	33	30	15
bdry0_79	898	178	99	29	t
bdry1_79	985	83	195	26	578
cdma	1	11	1	7	f
circ10-3	69	1552	564	t	t
ivu06-big	1	522	1	52	t
ivu52	1	33	1	12	t
lectsched-1	68	t	14	8	t
lectsched-3	26	18	8	8	1165
momentum3	1	42	1	17	t
n15-3	4	84	6	55	1138
neos-826650	1	1	4	7	12
neos-849702	28	56	25	627	t
ns1116954	3	913	156	63	1565
ns1354092	205	6	473	45	81
ns1456591	1	13	1	1	t
ns1631475	1	19	1	1	t
ns1685374	1	5	1	1	355
ns1854840	2	28	1	2	t
ns1904248	4	f	2	1	t
ns2122603	1	307	1	1	t
ns506428	235	103	3	70	690
ns848845	18	51	56	223	t
ns894236	1064	13	3	t	t
ns894786	1	28	356	t	t
ns894788	220	6	419	299	3104
ns903616	1	14	36	59	t
rail01	1	86	19	122	2330
rocII-9-11	5	16	3	5	t
satellites3-40	16	681	1	2	1115
satellites3-40-fs	1	581	3	1	2647
shs1023	78	83	8	345	2113
triptim2	4	24	15	1299	202
triptim3	107	29	38	338	210

"t": time limit exceeded; "f": fail

B.3 Benchmark of SMT-Solvers

This is a copy of the results of the 11th International Satisfiability Modulo Theories Competition (SMT-COMP 2016), accessed on Sept. 26th 2016. It was published on http://smtcomp.sourceforge.net/2016/results-QF_LIA.shtml?v=1467876482.

The "QF_LIA" track contains problems with unquantified integer arithmetic, i.e.,the underlying logic used in the SMT-formulation of Section 4.1.1. All Quantifiers therin are resolved by adding individual terms for the bound variables. The results show that Z3 solver, which was used in this thesis, is a state-of-the-art SMT-solver for these kinds of problems. Other potential candidates, such as Yices 2 [4], CVC 4 [5], MathSAT 5 [6] or SMTInterpol [7] have runtime characteristics of the same order of magnitude as Z3. MathSat 5 has thelowest CPU time and the highest amount of correctly solved problem instances, however it ist not listed as winner of the contest below, because it did not participate in SMT-COMP 2016.

QF_LIA (Main Track)

Competition results for the QF_LIA division as of Thu Jul 7 07:24:34 GMT

Benchmarks in this division: 5839
Winners:
Sequential Performances: CVC4
Parallel Performances: CVC4

Results:

	Sequential performance			Parallel performance				Other information
Solver	Error Score	Correctly Solved Score	avg. CPU time	Errors	Corrects	avg. CPU time	avg. WALL time	Unsolved benchmarks
CVC4	0.000	5510.092	167.597	0.000	5510.092	167.669	167.493	168
MathSat5*	0.000	5561.147	138.080	0.000	5561.147	138.144	138.040	120
ProB	0.000	1178.476	1507.090	0.000	1178.476	1508.020	1507.096	5148
SMT-RAT	0.000	3223.364	1098.452	0.000	3223.814	1099.095	1098.449	3450
SMTInterpol	0.000	5456.162	198.425	0.000	5456.666	211.890	190.450	251
Yices2	0.000	5465.835	180.926	0.000	5465.835	181.015	180.885	206
veriT-dev	0.000	2654.377	709.000	0.000	2854.377	709.385	708.966	3578
z3*	0.000	5444.101	187.484	0.000	5444.101	187.577	187.420	254

*Non-competitive.

[4] http://yices.csl.sri.com/
[5] http://cvc4.cs.nyu.edu
[6] http://mathsat.fbk.eu/
[7] https://ultimate.informatik.uni-freiburg.de/smtinterpol/

Bibliography

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] Arvind and D. E. Culler. Dataflow Architectures. *Annual Review of Computer Science*, 1, 1986.

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[4] E. Balas. On the facial structure of scheduling polyhedra. In *Mathematical Programming Essays in Honor of George B. Dantzig Part I*, number 24 in Mathematical Programming Studies. North-Holland, 1985.

[5] T. Bangemann, S. Karnouskos, R. Camp, O. Carlsson, M. Riedl, S. McLeod, R. Harrison, A. W. Colombo, and P. Stluka. State of the Art in Industrial Automation. In *Industrial Cloud-Based Cyber-Physical Systems*. Springer, 2014.

[6] E. Baroth and C. Hartsough. Visual Programming in the Real World. In *Visual object-oriented programming*. Manning Publications, 1995.

[7] T. Bauernhansl. Die Vierte Industrielle Revolution – Der Weg in ein Wertschaffendes Produktionsparadigma. In *Industrie 4.0 in Produktion, Automatisierung und Logistik*. Springer, 2014.

[8] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. UPPAAL-Tiga: Time for Playing Games! In *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*. Springer, 2007.

[9] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL – A Tool Suite for Automatic Verification of Real-Time Systems. In *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*. Springer, 1996.

[10] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*. Springer, 2004.

[11] H. Bohn, A. Bobek, and F. Golatowski. SIRENA – Service Infrastructure for Real-Time Embedded Networked Devices: A Service Oriented Framework for Different Domains. In *Proceedings of the IEEE International Conference on Networking (ICN)*, 2006.

[12] G. Brightwell and P. Winkler. Counting Linear Extensions. *Order*, 8, 1991.

[13] S. Burbeck. The Tao of e-Business Services. IBM Software Group, 2000.

[14] G. Buttazzo. *Hard Real-Time Computing Systems*, volume 24 of *Real-Time Systems Series*. Springer, 2011.

[15] G. Cândido, J. Barata de Oliveira, A. W. Colombo, and F. Jammes. SOA in Reconfigurable Supply Chains: A Research Roadmap. *Engineering Applications of Artificial Intelligence*, 22(6), 2009.

[16] G. Cândido, F. Jammes, J. Barata de Oliveira, and A. W. Colombo. SOA at Device Level in the Industrial Domain: Assessment of OPC UA and DPWS Specifications. In *Proceedings of the IEEE International Conference on Industrial Informatics (INDIN)*, 2010.

[17] A. Cannata, M. Gerosa, and M. Taisch. SOCRADES: A Framework for Developing Intelligent Systems in Manufacturing. In *Proceedings of the IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, 2008.

[18] F. Cassez, J. J. Jessen, K. G. Larsen, J.-F. Raskin, and P.-A. Reynier. Automatic Synthesis of Robust and Optimal Controllers – an Industrial Case Study. In *Hybrid Systems: Computation and Control*, volume 5469 of *Lecture Notes in Computer Science*. Springer, 2009.

[19] R. Checcozzo, F. Rusina, L. Mangeruca, A. Ballarino, C. Abadie, A. Brusaferri, R. Harrison, and R. Monfared. RI-MACS: An Innovative Approach for Future Automation Systems. *Mechatronics and Manufacturing Systems*, 2(3), 2009.

[20] C.-H. Cheng, M. Geisinger, and C. Buckl. Synthesizing Controllers for Automation Tasks with Performance Guarantees. In *Model Checking Software*, volume 7976 of *Lecture Notes in Computer Science*. Springer, 2013.

[21] C.-H. Cheng, A. Knoll, M. Luttenberger, and C. Buckl. GAVS+: An Open Platform for the Research of Algorithmic Game Solving. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2011.

[22] G. Chryssolouris. *Manufacturing Systems: Theory and Practice*. Mechanical Engineering Series. Springer, 2nd edition, 2006.

[23] A. Colombo, T. Bangemann, S. Karnouskos, J. Delsing, P. Stluka, R. Harrison, F. Jammes, and J. L. Lastra. *Industrial Cloud-Based Cyber-Physical Systems*. Springer, 2014.

[24] A. W. Colombo, T. Bangemann, and S. Karnouskos. IMC-AESOP Outcomes: Paving the Way to Collaborative Manufacturing Systems. In *Proceedings of the IEEE International Conference on Industrial Informatics (INDIN)*, 2014.

[25] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner. Random Graph Generation for Scheduling Simulations. In *Proceedings of the International ICST Conference on Simulation Tools and Techniques (SIMUTools)*, 2010.

[26] T. Cucinotta, A. Mancina, G. F. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo, and F. Rusinà. A Real-Time Service-Oriented Architecture for Industrial Automation. *IEEE Transactions on Industrial Informatics*, 5(3), 2009.

[27] W. Dai, V. Vyatkin, J. H. Christensen, and V. N. Dubinin. Bridging Service-Oriented Architecture and IEC 61499 for Flexibility and Interoperability. *IEEE Transactions on Industrial Informatics*, 11(3), 2015.

[28] P. Danielis, J. Skodzik, V. Altmann, E. B. Schweissguth, F. Golatowski, D. Timmermann, and J. Schacht. Survey on Real-Time Communication via Ethernet in Industrial Automation Environments. In *Proceedings of the IEEE International Conference on Emerging Technology and Factory Automation (ETFA)*, 2014.

[29] R. I. Davis and A. Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM Computing Surveys*, 43(4), 2011.

[30] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture notes in Computer Science*. Springer, 2008.

[31] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1), 2008.

[32] J. B. Dennis. First Version of a Data Flow Procedure Language. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*. Springer, 1974.

[33] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task Graphs for Free. In *Proceedings of the IEEE International Workshop on Hardware/Software Codesign (CODES / CASHE)*, 1998.

[34] J. Du, J. Y.-T. Leung, and G. H. Young. Scheduling Chain-Structured Tasks to Minimize Makespan and Mean Flow Time. *Information and Computation*, 92(2), 1991.

[35] M. E. Dyer and L. A. Wolsey. Formulating the Single Machine Sequencing Problem with Release Dates as a Mixed Integer Program. *Discrete Applied Mathematics*, 26(2), 1990.

[36] H. A. ElMaraghy. Flexible and Reconfigurable Manufacturing Systems Paradigms. *International Journal of Flexible Manufacturing Systems*, 17(4), 2005.

[37] P. C. Evans and M. Annunziata. Industrial Internet: Pushing the Boundaries of Minds and Machines. Technical report, General Electric, Nov. 2012.

[38] R. France and B. Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering (FOSE)*, 2007.

[39] M. García Valls, I. Rodríguez Lopez, and L. Fernández Villar. iLAND: An Enhanced Middleware for Real-Time Reconfiguration of Service Oriented Distributed Real-Time Systems. *IEEE Transactions on Industrial Informatics*, 9(1), 2013.

[40] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9), 1991.

[41] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11(1), 2009.

[42] S. N. Han, G. M. Lee, and N. Crespi. Semantic Context-Aware Service Composition for Building Automation System. *IEEE Transactions on Industrial Informatics*, 10(1), 2014.

[43] S. Holmbacka, W. Lund, S. Lafond, and J. Lilius. Lightweight Framework for Runtime Updating of C-Based Software in Embedded Systems. In *Proceedings of the USENIX Workshop on Hot Topics in Software Upgrades (HotSWUp)*, 2013.

[44] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Pearson Education, 2010.

[45] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *SIAM Journal on Computing*, 18(2), 1989.

[46] F. Jammes, B. Bony, P. Nappey, A. W. Colombo, J. Delsing, J. Eliasson, R. Kyusakov, S. Karnouskos, P. Stluka, and M. Till. Technologies for SOA-Based Distributed Large Scale Process Monitoring and Control Systems. In

Proceedings of the Annual Conference of the IEEE Industrial Electronics Society (IECON), 2012.

[47] F. Jammes and H. Smit. Service-Oriented Paradigms in Industrial Automation. *IEEE Transactions on Industrial Informatics*, 1(1), 2005.

[48] F. Jammes, H. Smit, J. L. M. Lastra, and I. M. Delamer. Orchestration of Service-Oriented Manufacturing Processes. In *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2005.

[49] S. Jin, G. Schiavone, and D. Turgut. A Performance Study of Multiprocessor Task Scheduling Algorithms. *Journal of Supercomputing*, 43(1), 2008.

[50] W. M. Johnston, J. R. Hanna, and R. J. Millar. Advances in Dataflow Programming Languages. *ACM Computing Surveys*, 36(1), 2004.

[51] J. Jonsson and K. G. Shin. Robust Adaptive Metrics for Deadline Assignment in Distributed Hard Real-Time Systems. *Real-Time Systems*, 23(3), 2002.

[52] S. Käbisch, D. Peintner, J. Heuer, and H. Kosch. Optimized xml-based web service generation for service communication in restricted embedded environments. In *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2011.

[53] H. Kagermann, W. Wahlster, and J. Helbig. Recommendations for Implementing the Strategic Initiative Industrie 4.0. Technical report, National Academy of Science and Engineering, Apr. 2013.

[54] B. Kao and H. Garcia-Molina. Deadline Assignment in a Distributed Soft Real-Time System. *IEEE Transactions on Parallel and Distributed Systems*, 8(12), 1997.

[55] S. Karnouskos, T. Bangemann, and C. Diedrich. Integration of Legacy Devices in the Future SOA-Based Factory. *IFAC Proceedings Volumes*, 42(4), 2009.

[56] S. Karnouskos, A. W. Colombo, and T. Bangemann. Trends and Challenges for Cloud-Based Industrial Cyber-Physical Systems. In *Industrial Cloud-Based Cyber-Physical Systems*. Springer, 2014.

[57] S. Karnouskos, A. W. Colombo, T. Bangemann, K. Manninen, R. Camp, M. Tilly, P. Stluka, F. Jammes, J. Delsing, and J. Eliasson. A SOA-based Architecture for Empowering Future Collaborative Cloud-Based Industrial Automation. In *Proceedings of the Annual Conference of the IEEE Industrial Electronics Society (IECON)*, 2012.

[58] N. Kaur, C. S. McLeod, A. Jain, R. Harrison, B. Ahmad, A. W. Colombo, and J. Delsing. Design and Simulation of a SOA-Based System of Systems for

Automation in the Residential Sector. In *Proceedings of the IEEE International Conference on Industrial Technology (ICIT)*, 2013.

[59] A. B. Keha, K. Khowala, and J. W. Fowler. Mixed Integer Programming Formulations for Single Machine Scheduling Problems. *Computers & Industrial Engineering*, 56(1), 2009.

[60] A. Kemper and A. Eickler. *Datenbanksysteme: Eine Einführung*. Oldenbourg Verlag, 2015.

[61] H. Kopetz. Should Responsive Systems be Event-Triggered or Time-Triggered? *IEICE Transactions on Information and Systems*, 76(11), 1993.

[62] Y. Koren. *The Global Manufacturing Revolution*. John Wiley & Sons, 2010.

[63] T. Kothmayr, J. Hirscheider, A. Kemper, A. Scholz, and J. Heuer. Comparing Heuristics and Linear Programming Formulations for Scheduling of In-Tree Tasksets. In *Proceedings of the Work in Progress Session of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS WiP)*, 2014.

[64] T. Kothmayr, A. Kemper, A. Scholz, and J. Heuer. Machine Ballets Don't Need Conductors: Towards Scheduling Based Service Choreographies in a Real-Time SOA for Industrial Automation. In *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2014.

[65] T. Kothmayr, A. Kemper, A. Scholz, and J. Heuer. Schedule-Based Service Choreographies for Real-Time Control Loops. In *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2015.

[66] T. Kothmayr, A. Kemper, A. Scholz, and J. Heuer. Synthesizing Schedules Through Heuristics for Hard Real-Time Workflows. In *Proceedings of the IEEE International Conference on Industrial Technology (ICIT)*, 2015.

[67] T. Kothmayr, A. Kemper, A. Scholz, and J. Heuer. Instant Service Choreographies for Reconfigurable Manufacturing Systems - a Demonstrator. In *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016.

[68] M. Kovatsch, S. Duquennoy, and A. Dunkels. A Low-Power CoAP for Contiki. In *Proceedings of the IEEE International Conference on Mobile Ad-Hoc and Sensor Systems (MASS)*, 2011.

[69] S. Kullback and R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1), 1951.

[70] Y.-K. Kwok and I. Ahmad. On Multiprocessor Task Scheduling Using Efficient State Space Search Approaches. *Journal of Parallel and Distributed Computing*, 65(12), 2005.

[71] R. Kyusakov, H. Mäkitaavola, J. Delsing, and J. Eliasson. Efficient XML Interchange in Factory Automation Systems. In *Proceedings of the Annual Conference of the IEEE Industrial Electronics Society (IECON)*, 2011.

[72] E. L. Lawler. Optimal Sequencing of a Single Machine Subject to Precedence Constraints. *Management Science*, 19(5), 1973.

[73] E. A. Lee. Cyber Physical Systems: Design Challenges. In *Proceedings of the IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2008.

[74] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 100(1), 1987.

[75] S.-H. Leitner and W. Mahnke. OPC UA – Service-Oriented Architecture for Industrial Applications. Technical report, ABB Corporate Research Center, 2006.

[76] J. K. Lenstra, A. H. G. R. Kan, and P. Brucker. Complexity of Machine Scheduling Problems. *Annals of Discrete Mathematics*, 1, 1977.

[77] M. Loskyll, J. Schlick, S. Hodek, L. Ollinger, T. Gerber, and B. Pîrvu. Semantic Service Discovery and Orchestration for Manufacturing Processes. In *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2011.

[78] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems*, 30(1), 2005.

[79] J.-F. Martínez, M. López, V. Hernández, K. Jean-Marie, A.-B. García, L. López, C. Herrera, and C.-J. Sánchez-Alarcos. A Security Architectural Approach for DPWS-Based Sevices. In *Proceedings of the CollECTeR Ibéroamérica conference*, 2008.

[80] T. R. Mayer, L. Brunie, D. Coquil, and H. Kosch. On reliability in publish/-subscribe systems: a survey. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5), 2012.

[81] J. M. Mendes, P. Leitão, A. W. Colombo, and F. Restivo. High-Level Petri Nets for the Process Description and Control in Service-Oriented Manufacturing Systems. *International Journal of Production Research*, 50(6), 2012.

[82] J. M. Mendes, P. Leitão, F. Restivo, and A. W. Colombo. Composition of Petri Nets Models in Service-Oriented Industrial Automation. In *Proceedings of the IEEE International Conference on Industrial Informatics (INDIN)*, 2010.

[83] F. F.-H. Nah. A Study on Tolerable Waiting Time: How Long are Web Users Willing to Wait? *Behaviour & Information Technology*, 23(3), 2004.

[84] P. Nappey, C. El Kaed and Armando W Colombo, J. Eliasson, A. Kruglyak, R. Kyusakov, C. Hübner, T. Bangemann, and O. Carlsson. Migration of a Legacy Plant Lubrication System to SOA. In *Industrial Cloud-Based Cyber-Physical Systems*. Springer, 2014.

[85] W. Nie, S. Zhou, K.-J. Lin, and S. D. Kim. An On-Line Capacity-Based Admission Control for Real-Time Service Processes. *IEEE Transactions on Computers*, 63(9), 2014.

[86] OASIS. Reference Model for Service Oriented Architecture (Version 1.0). https://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html, 2006.

[87] OASIS. Devices Profile for Web Services Specification (Version 1.1). http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01, 2009.

[88] L. Ollinger, J. Schlick, and S. Hodek. Leveraging the Agility of Manufacturing Chains by Combining Process-Oriented Production Planning and Service-Oriented Manufacturing Automation. *IFAC Proceedings Volumes*, 44(1), 2011.

[89] OPC Foundation. OPC Unified Architecture (OPC UA) Specifiations. http://www.opcfoundation.org/UA, 2008.

[90] M. Panahi, W. Nie, and K.-J. Lin. The Design of Middleware Support for Real-Time SOA. In *Proceedings of the IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2011.

[91] P. Pietrzak, R. Kyusakov, J. Eliasson, and P. Lindgren. Roadmap for SOA Event Processing and Service Execution in Real-Time Using Timber. In *Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE)*, 2011.

[92] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 2012.

[93] C. N. Potts. Technical Note – Analysis of a Heuristic for One Machine Sequencing with Release Dates and Delivery Times. *Operations Research*, 28(6), 1980.

[94] M. Queyranne and A. S. Schulz. Polyhedral Approaches to Machine Scheduling. Technical report, University of British Columbia & Technische Universiät Berlin, 1994.

[95] F. Reimann, M. Glaß, C. Haubelt, M. Eberl, and J. Teich. Improving Platform-Based System Synthesis by Satisfiability Modulo Theories Solving. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES)*, 2010.

[96] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-Sensitive Operators for Parallel Main-Memory Database Clusters. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2014.

[97] J. Schlick, P. Stephan, M. Loskyll, and D. Lappe. Industrie 4.0 in der Praktischen Anwendung. In *Industrie 4.0 in Produktion, Automatisierung und Logistik*. Springer, 2014.

[98] A. Scholz, I. Gaponova, S. Sommer, A. Kemper, A. Knoll, C. Buckl, J. Heuer, and A. Schmitt. ϵSOA-Service Oriented Architectures Adapted for Embedded Networks. In *Proceedings of the IEEE International Conference on Industrial Informatics (INDIN)*, 2009.

[99] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252, RFC Editor, June 2014. http://www.rfc-editor.org/rfc/rfc7252.txt.

[100] W. Shen and D. H. Norrie. Agent-Based Systems for Intelligent Manufacturing: A State-of-the-Art Survey. *Knowledge and Information Systems*, 1(2), 1999.

[101] J. P. Sousa and L. A. Wolsey. A Time Indexed Formulation of Non-Preemptive Single Machine Scheduling Problems. *Mathematical Programming*, 54(1), 1992.

[102] L. M. S. D. Souza, P. Spiess, D. Guinard, M. Köhler, S. Karnouskos, and D. Savio. SOCRADES: A Web Service Based Shop Floor Integration Infrastructure. In *The Internet of Things*, volume 4952 of *Lecture Notes in Computer Science*. Springer, 2008.

[103] D. Spath, O. Ganschar, S. Gerlach, M. Hämmerle, T. Krause, and S. Schlund. *Produktionsarbeit der Zukunft – Industrie 4.0*. Fraunhofer Verlag Stuttgart, 2013.

[104] G. Starke, T. Kunkel, and D. Hahn. Flexible Collaboration and Control of Heterogeneous Mechatronic Devices and Systems by Means of an Event-Driven, SOA-Based Automation Concept. In *Proceedings of the IEEE International Conference on Industrial Technology (ICIT)*, 2013.

[105] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 Requirements of Real-Time Stream Processing. *ACM SIGMOD Record*, 34(4), 2005.

[106] M. Taisch, A. W. Colombo, S. Karnouskos, and A. Cannata. SOCRADES Roadmap: The Future of SOA-based Factory Automation, 2009.

[107] T. Tobita and H. Kasahara. A Standard Task Graph Set for Fair Evaluation of Multiprocessor Scheduling Algorithms. *Journal of Scheduling*, 5(5), 2002.

[108] H. Topcuoglu, S. Hariri, and M. you Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3), 2002.

[109] L. G. Valiant. The Complexity of Computing the Permanent. *Theoretical Computer Science*, 8(2), 1979.

[110] P. Varga, F. Blomstedt, L. L. Ferreira, J. Eliasson, M. Johansson, J. Delsing, and I. M. de Soria. Making System of Systems Interoperable – The Core Components of the Arrowhead Framework. *Journal of Network and Computer Applications*, 2016.

[111] Verein Deutscher Ingenieure. Fortentwicklung des Referenzmodells für die Industrie 4.0, Apr. 2016.

[112] B. Vogel-Heuser. Herausforderungen und Anforderungen aus Sicht der IT und der Automatisierungstechnik. In *Industrie 4.0 in Produktion, Automatisierung und Logistik*. Springer, 2014.

[113] S. Voss and B. Schatz. Deployment and Scheduling Synthesis for Mixed-Critical Shared-Memory Applications. In *Proceedings of the IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS)*, 2013.

[114] V. Vyatkin. IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review. *IEEE Transactions on Industrial Informatics*, 7(4), 2011.

[115] W3C Recommendation. Efficient XML Interchange (EXI) Format 1.0 (Second Edition). https://www.w3.org/TR/exi/, 2014.

[116] K. N. Whitley and A. F. Blackwell. Visual Programming in the Wild: A Survey of LabVIEW Programmers. *Journal of Visual Languages & Computing*, 12(4), 2001.

[117] H.-P. Wiendahl, H. A. ElMaraghy, P. Nyhuis, M. F. Zäh, H.-H. Wiendahl, N. Duffie, and M. Brieke. Changeable Manufacturing – Classification, Design and Operation. *CIRP Annals - Manufacturing Technology*, 56(2), 2007.

[118] E. D. Wikum, D. C. Llewellyn, and G. L. Nemhauser. One-Machine Generalized Precedence Constrained Scheduling Problems. *Operations Research Letters*, 16(2), 1994.

[119] P. Winkler. Random Orders. *Order*, 1(4), 1985.

[120] M.-Y. Wu and D. D. Gajski. Hypertool: A Programming Aid for Message-Passing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3), 1990.

[121] T. Yang and A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9), 1994.

[122] L. H. Yoong, P. S. Roop, Z. E. Bhatti, and M. M. Y. Kuo. IEC 61499 in a Nutshell. In *Model-Driven Design Using IEC 61499*. Springer, 2015.

[123] E. Zeeb, G. Moritz, D. Timmermann, and F. Golatowski. WS4D: Toolkits for Networked Embedded Systems Based on the Devices Profile for Web Services. In *Proceedings of the IEEE International Conference on Parallel Processing Workshops (ICPPW)*, 2010.

[124] F. Zhang and A. Burns. Schedulability Analysis for Real-Time Systems with EDF Scheduling. *IEEE Transactions on Computers*, 58(9), 2009.

[125] A. Zoitl, G. Grabmair, F. Auinger, and C. Sunder. Executing Real-Time Constrained Control Applications Modelled in IEC 61499 with Respect to Dynamic Reconfiguration. In *Proceedings of the IEEE International Conference on Industrial Informatics (INDIN)*, 2005.

[126] W. M. Zuberek. Timed Petri Nets Definitions, Properties, and Applications. *Microelectronics Reliability*, 31(4), 1991.