

Knowledge-based Dynamic Reconfiguration for Embedded Real-Time Systems

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

Dipl.-Ing. Oliver Höftberger

Matrikelnummer 0325723

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Prof. Dr. Roman Obermaisser

Diese Dissertation haben begutachtet:

(Prof. Dr. Roman Obermaisser)

(Em.O.Univ.Prof. Dr.phil.
Dr.h.c. Hermann Kopetz)

Wien, 17.11.2015

(Dipl.-Ing. Oliver Höftberger)

Knowledge-based Dynamic Reconfiguration for Embedded Real-Time Systems

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Dipl.-Ing. Oliver Höftberger

Registration Number 0325723

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Prof. Dr. Roman Obermaisser

The dissertation has been reviewed by:

(Prof. Dr. Roman
Obermaisser)

(Em.O.Univ.Prof. Dr.phil.
Dr.h.c. Hermann Kopetz)

Wien, 17.11.2015

(Dipl.-Ing. Oliver Höftberger)

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Oliver Höftberger
Hildebrandgasse 30/1/2, 1180 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift Verfasserin)

Acknowledgements

At first I want to express my gratitude to my advisor Roman Obermaisser. He supported my work on this thesis since our first discussions, and even after he moved to the University of Siegen (Germany) he took it for granted to provide me with excellent advices whenever I contacted him. During my work I was incited by many tough questions he raised about the solutions I came up with, and he amazed me with his valuable extensive knowledge in several different fields of computer science.

Furthermore, I would like to thank Prof. Hermann Kopetz for many helpful recommendations and the inspiring lectures during my studies that caught my interest to start an academic career.

Thanks to my colleagues and friends at the Department of Computer Engineering at the Vienna University of Technology, who supported me within countless discussions and suggestions for improvement of my ideas. Especially I want to thank Denise Ratasich, who integrated the framework presented in this thesis into a robot platform, and thereby provided additional evidence for the applicability of the framework.

Last but not least I am also grateful to all my friends and my family for their support during the years of my studies. Particular thanks to my wife Martina, who is my inspiration as she encouraged me to write this thesis and she continuously pushed me to make progress and finalize my work.

– *Oliver Höftberger*

Vienna, November 2015

Abstract

Innovations in many domains like transportation, industrial systems, home automation, health-care and consumer electronics are driven by embedded computer systems in order to attain an unprecedented quality of control in physical processes. While this enables new services as well as the enhancement of existing capabilities based on changing user demands and technological advancements, these systems increasingly depend on the correct operation of electronic devices. As a fault in a safety-relevant application might lead to hazardous events, fault-tolerance mechanisms must be introduced to ensure a continued provision of services even in the presence of the failure of individual components. These fault-tolerance mechanisms are based on a fault hypothesis and exploit redundancy in the system to detect and mitigate faults and their effects. Typically, the system designers explicitly introduce redundancy at design time by the replication of components or computations, which increases the production cost, energy consumption, weight and size of the system. Alternatively, implicit redundancy can be exploited, which is available as a priori knowledge about the correct system behaviour and the relationship between system properties. However, it requires high engineering effort to identify implicit redundancy, and its availability varies during the runtime of dynamically changing systems.

Within this thesis a dynamic reconfiguration framework for embedded real-time systems is presented that automatically identifies redundancy and adapts the configuration of components and their interactions accordingly. This allows to react at runtime to changes in the system or its environment and to recover from service failures, including unanticipated failures that are not covered in the fault hypothesis. In order to prevent the violation of temporal requirements, the reconfiguration has to be completed within a bounded time. To ensure semantic correctness of new configurations, also the semantic relationship between system properties has to be considered when a valid configuration is computed. Furthermore, such new configurations may only be applied if the accuracy of information exchanged between components meets the application's requirements. State-of-the-art solutions are not capable to provide temporal guarantees for reconfiguration. Also, they do not consider the semantics and accuracy of processed information.

The framework provides an architecture for systems with dynamic reconfiguration capability and a modelling language for a knowledge base that describes the semantic relationship of the system's building blocks and its properties. Algorithms have been developed that search for redundant information in the knowledge base and to determine the information uncertainty. Formal analyses and experiments show the real-time capability of the framework and the effectiveness of semantic matching as well as the determination of uncertainty of information. An evaluation of the probability to find substitutes for failed services demonstrates the increase of reliability when dynamic reconfiguration is used as a never-give-up strategy for the system.

Kurzfassung

Eingebettete Computersysteme sind die Grundlage von Innovationen in vielen Bereichen wie dem Transportwesen, der Industrie- und Gebäudeautomation, dem Gesundheitswesen oder der Unterhaltungselektronik. Durch deren Einsatz lässt sich eine noch nie dagewesene Präzision bei der Steuerung physikalischer Prozesse erreichen. Einerseits sind die Entwicklung neuer Dienste sowie die Verbesserung existierender Funktionalitäten eine Reaktion auf sich ändernde Nutzeranforderungen und technologischen Fortschritt. Andererseits sind diese Systeme zunehmend vom korrekten Funktionieren der elektronischen Komponenten abhängig. Da ein Fehler in sicherheitskritischen Anwendungen katastrophale Auswirkungen haben kann, müssen Fehlertoleranzmechanismen zum Einsatz kommen um eine durchgängige Bereitstellung von Diensten zu garantieren, selbst wenn einzelne Systemkomponenten ausgefallen sind. Diese Mechanismen basieren auf einer Fehlerhypothese und nutzen Redundanz im System um Fehler zu erkennen und deren Auswirkungen einzudämmen und zu maskieren. Normalerweise wird Redundanz explizit bei der Systementwicklung durch die Replikation von Komponenten und Berechnungen eingeführt. Dies erhöht allerdings die Produktionskosten, den Energieverbrauch, das Gewicht und die Abmessungen des Systems. Alternativ kann implizit vorhandene Redundanz genutzt werden, welche aus dem vorhandenen Wissen über das korrekte Systemverhalten und die Beziehungen zwischen Systemeigenschaften besteht. Es ist jedoch ein hoher Entwicklungsaufwand notwendig um implizite Redundanz zu finden. Eine weitere Herausforderung ist, dass die Verfügbarkeit impliziter Redundanz in dynamisch veränderlichen Systemen variiert.

In dieser Arbeit wird ein Framework für dynamische Rekonfiguration von eingebetteten Echtzeitsystemen vorgestellt, welches automatisiert implizite Redundanz identifiziert und die Konfiguration von Komponenten und deren Interaktionen entsprechend anpasst. In vielen Fehlerszenarien können somit ausgefallene Dienste wiederhergestellt werden, selbst wenn der zugrundeliegende Fehler nicht in der Fehlerhypothese berücksichtigt wurde. Ebenso kann das System zur Laufzeit autonom auf Änderungen der Systemzusammensetzung und der erbrachten Komponentendienste reagieren. In Abhängigkeit von der Dynamik der Umgebung muss die Rekonfiguration in einer begrenzten Zeit abgeschlossen sein. Um die semantische Korrektheit einer neuen Konfiguration zu garantieren müssen auch die semantischen Beziehungen zwischen Systemeigenschaften bei der Berechnung gültiger Konfigurationen berücksichtigt werden. Außerdem darf eine neue Konfiguration nur eingesetzt werden, wenn die Genauigkeit der ausgetauschten Informationen den Systemanforderungen entspricht. Aktuelle Methoden sind nicht in der Lage zeitliche Garantien für die Rekonfiguration abzugeben und gleichzeitig die Semantik und Genauigkeit der verarbeiteten Informationen zu berücksichtigen.

Kernelemente des vorliegenden Frameworks sind eine Architektur für Systeme mit dynamischer Rekonfiguration, sowie eine Modellierungssprache für eine Wissensdatenbank, welche die semantischen Zusammenhänge zwischen Systembausteinen und deren Eigenschaften beschreibt. Ein weiterer wissenschaftlicher Beitrag sind Algorithmen die in der Datenbank nach redundanten Informationen suchen und die Genauigkeit dieser Informationen ermitteln. Formale Analysen und Experimente evaluieren die Echtzeitfähigkeit des Frameworks, die Effektivität der semantischen Methoden sowie die Bestimmung der Genauigkeit von Informationen. Die Evaluierung der Wahrscheinlichkeit, einen ausgefallenen Dienst zu ersetzen, demonstriert die Steigerung der Zuverlässigkeit wenn dynamische Rekonfiguration als Never-Give-Up Strategie eingesetzt wird.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Problem Statement and Research Gap	4
1.3	Structure of the Work	5
2	Basic Concepts	7
2.1	Real-time Systems	7
2.2	Component-based Design	9
2.3	Dynamic Reconfiguration	11
2.4	System Dependability	12
2.5	Conceptual Modeling	17
2.6	Information Uncertainty	18
3	Requirements and Related Work	21
3.1	Requirements	21
3.2	Related Work	23
4	Dynamic Reconfiguration Framework	35
4.1	Building Blocks of Reconfigurable System	35
4.2	Reconfiguration Process and Component Interaction	43
5	Modeling Cyber Physical Systems (CPSs)	47
5.1	Building Blocks of Ontology	47
5.2	Service-to-Ontology Mapping	55
5.3	Ontology Preprocessing	57
6	Service Orchestration	65
6.1	Process Overview	66
6.2	Composition Search	68
6.3	Transfer Service Generation	83
6.4	Service and Communication Scheduling	87
6.5	WCET of Service Orchestration	88
7	Uncertainty of Service Composition	93

CONTENTS

7.1	Modeling Uncertainty	93
7.2	Propagation of Uncertainty	97
7.3	Combining Composition Trees	105
8	Implementation	109
8.1	System Ontology	109
8.2	Composition Search	124
8.3	Calculation of Worst-Case Uncertainty	126
8.4	Demonstration on robot platform	129
9	Experiments and Simulation	133
9.1	Use Case Description	133
9.2	Runtime of Composition Search	137
9.3	Probability of Finding Service Compositions	138
9.4	Simulation of Propagation of Uncertainty	139
10	Results	147
10.1	Experimental Results and Interpretation	147
10.2	Requirement Fulfillment	151
11	Conclusion and Future Work	155
11.1	Conclusion	155
11.2	Future Work	157
	Bibliography	159
	Acronyms	169
	Index	171

Introduction

In the last decades the influence of embedded computer systems in the control of physical processes continuously increased. Countless applications in different domains like transportation, industrial and home automation, healthcare or consumer electronics benefit from the integration of computational devices in order to enable new services or to enhance existing capabilities. This trend is ongoing and the majority of product innovations depends on electronics. For instance, to realize the envisioned autonomous driving of cars [109], many sensors like radar, laser-based range sensors and stereo cameras are integrated, which provide raw input data for complex object detection and sensor fusion algorithms that generate a model of the vehicle and its environment. This model constitutes the basis to evaluate the current driving situation and to plan the next driving actions, which are performed subsequently. During the lifetime of such a system, users often adapt their demands to new technological trends, calling for new services to be added and the system to be reconfigured (e.g., to integrate a new device into a smart home).

At system level the correct services need to be provided despite the failure of individual components, such as sensors or computing nodes. Particularly for safety-relevant applications (e.g., aircrafts, autonomous cars, chemical plants or medical devices) the efforts required to ensure the dependability of the system often exceed the efforts for developing the prime functionality. Failure rates of 10^{-9} failures per hour, as demanded for ultradependable systems like flight control or automotive steer-by-wire, are not achievable by a single component in the system [80]. Therefore, fault-tolerance and mitigation techniques at system level (e.g., n-modular-redundancy, pair-and-spare systems) are applied to keep the system operational even in the presence of component failures. The selection of appropriate techniques is based on a fault hypothesis [52] that documents the assumptions about the frequency and types of faults occurring in the system. As fault-tolerance might have a significant impact on the system design, the fault hypothesis is devised in an early stage of the system design process. However, the analysis and prediction of the behaviour in the presence of faults are impaired by the ever increasing complexity of such systems [72]. For reasons of cost and efforts, fault scenarios that are assumed to be too unlikely are not considered in the fault hypothesis. This leads to a limited

assumption coverage. Therefore, the probability that the system fails to provide its services, even with the consequence of catastrophic incidents, cannot be reduced to zero.

Most techniques to achieve the required level of fault-tolerance are based on the exploitation of redundancy. This is either done through the explicit introduction of redundancy, as for instance by replicating components, repeated execution of the same component or utilization of a priori knowledge. Alternatively, the system designers can exploit implicit redundancy in the system, which is given if there exists a known relationship between system components (e.g., two drive shafts are mechanically connected by a transmission chain). While explicit redundancy increases the production cost, energy consumption, weight or space requirements of the system, system designers are often not able to identify existing implicit redundancy. This is due to the fact that implicit redundancy typically involves multiple components with a complex physical relationship between them. Tools that support the engineers by automatically identifying implicit redundancy are not available. Furthermore, components are often developed independently by different suppliers, like it is typically done in the automotive industry. As many of these suppliers hide their intellectual property (IP) or the information about integrated sensors is not provided to other suppliers, the components of each supplier often integrate separate sensors in order to capture the same information. This type of redundancy could either contribute to increase the dependability of the system, or it should be reduced in order to save cost. As also proposed in [84], for future safety relevant systems, engineers will strive to find a cost-efficient trade-off between the required explicit redundancy and fault recovery by reconfiguration upon failure.

Within this work a dynamic reconfiguration framework is proposed that enables fault recovery by exploiting the implicit redundancy between components [39, 40]. The framework uses a knowledge base to automatically identify implicit redundancy for a requested service in the system. This enables the substitution of faulty components, as well as the provision of services that otherwise could not be provided because of missing input information. Furthermore, system designers can apply this framework to minimize explicit redundancy in the system in order to reduce production cost, energy consumption or weight. The applicability and performance of the proposed framework are evaluated with experiments using a realistic use case from the automotive domain. In the next section the motivation for this work is given, which is followed by the description of the research problem and a statement about the thesis' contributions beyond the state-of-the-art. The chapter closes with an outline of the structure of the document.

1.1 Motivation

Modern automotive systems integrate an increasing number of features for driver assistance, autonomous driving, safety and comfort. This requires numerous electronic devices. Currently, already up to 80 electronic control units (ECUs) and more than 100 sensors [32] are installed in a car, and their numbers are estimated to increase in the future to reach 200 automotive sensors. As the number of such devices and the complexity of the system increases, also the probability of faults in the system is raised. Component failures in cars do not only diminish the satisfaction of the passengers, but they also constitute a safety hazard. For safety-relevant features (e.g., the

brakes) redundant components are required in order to address these hazards. Simultaneously, car manufacturers strive to reduce the production cost by avoiding any unnecessary components.

On the other hand, due to the high amount of sensors and control units implicit redundancy is created, because the same properties in the system and its environment (e.g., the speed of the car, the air temperature) are measured and processed by multiple sensors and ECUs concurrently, or combinations of these properties allow to deduce the value of other properties. System engineers already exploit implicit redundancy and create components that perform the corresponding deductions. For instance, since a fixed relation exists between the crankshaft and the camshafts of an engine (e.g., by using a metal chain or toothed belts between these parts), the fault of a sensor providing the rotational speed of the crankshaft can be mitigated by switching to the measurements originating from the camshaft. However, the effort to identify implicit redundancy while ensuring the completeness of the solutions increases with the complexity of the relations between the involved components. Furthermore, at design time of the system the engineers cannot consider the dynamic availability of services during runtime. Since a human system engineer is able to identify such relations, the question arises, whether it is also possible to perform this identification automatically by the computer system. With an appropriate system model and tool support the task of the system engineer can be facilitated such that more complex relations can be identified and the number of required components in the system can be reduced. Moreover, when the execution time of the identification process is bounded and this bound is sufficiently low, it is also possible to perform an online search that determines whether a failed component can be substituted by other available components. The system is dynamically reconfigured to mitigate the fault in order to recover the failed service or to reestablish redundancy. For instance, the automotive safety standard ISO 26262 [43] defines the fault tolerant time interval (FTTI), within which the unavailability of services is acceptable and outputs to actuators are usually frozen. If the reconfiguration can be conducted within this interval, the safety of the system is not compromised by the component failure. This increases the safety and availability of the system, such that the owner of the car does not immediately have to tow the vehicle to the service station in order to repair the faulty component, but he can still use it and drive it to the repair station when the time is right.

On June 1st, 2009, Air France flight AF 447 from Rio de Janeiro to Paris crashed into the Atlantic Ocean. According to the final accident report of the *Bureau d'Enquêtes et d'Analyses pour la sécurité de l'aviation civile (BEA)* a sequence of events was responsible for the accident. At the beginning of this sequence there was a *'temporary inconsistency between the measured airspeeds, likely following the obstruction of the Pitot probes by ice crystals that led in particular to autopilot disconnection and a reconfiguration to alternate law'* [21, p. 17]. This type of aircraft uses three heated Pitot probes that measure the total air pressure that is effective at the nose of the aircraft. The total air pressure value is influenced by the static air pressure, the headwind and the speed of the aircraft. It is mainly used to deduce the airspeed, but also the calibrations of the air temperature and altitude are influenced by this value. Due to ice crystals in the atmosphere, a probe can get obstructed until the ice is melted down by the built-in heating element, which might take a few seconds. The values deduced from the probe measurements are forwarded to different replicated flight control computers. For the case of a single probe failure, voting mechanisms are applied to mask the failure until the system recovers from the failure.

When probes fail simultaneously, the system disconnects the autopilot and hands the control over to the pilots.

In case of flight AF 447, at least two probes were obstructed simultaneously for a few seconds, causing the autopilot to disconnect and the aircraft control laws to be switched to reduced protection (i.e., the protection against dangerous flight manoeuvres is disabled) while the aircraft was exposed to strong turbulences. Due to the discrepancy in the airspeed measurements and the correlated invalidity of other important flight parameters (e.g., altitude, vertical speed, angle of attack), the pilots were surprised and overstrained by the situation. Their subsequent false reactions lead to a dangerous stall situation where the aircraft lost uplift and finally crashed.

According to the accident report the safety model included the rare event of a total loss of airspeed information. The assumption was that the pilots are able to rapidly identify the problem and react appropriately in order to keep the aircraft stable. Eventually, the violation of basic assumptions in the safety model contributed to the disaster. In situations where safety-relevant assumptions are violated, the system should not give up to try to solve the problem. For instance, if there is a discrepancy between all speed measurements, the system could automatically consult the GPS receiver, which is usually not actively involved in the airspeed provision, to determine which of the probes provides the most accurate value, and thus, help the pilots to evaluate the situation. Introducing explicit redundancy by replication of the Pitot probes reduces the probability of a total loss of airspeed information. However, since all replicas use the same methodology for measuring, all of them are susceptible to the same systematic error. In particular, if the aircraft is flying through a region containing many ice particles, the same environmental effect can affect all probes simultaneously. Implicit redundancy can solve the problem of systematic errors, because deduced information is usually based on measurements of independent sensors.

1.2 Problem Statement and Research Gap

This thesis contributes to dynamic reconfiguration of embedded real-time systems in order to increase system dependability and support system designers to find a trade-off between the explicit redundancy required in the system and its production cost. The resulting system is able to autonomously identify implicit redundancy for the substitution of a given service in the system, such that faults can be mitigated by replacing a faulty component with a set of available correct components. Especially in cases where the assumptions about faults documented in the fault hypothesis are violated, the system should still react to these unforeseen events and attempt to recover from the failure by reconfiguration. Furthermore, during design time, the automatic identification of implicit redundancy assists the system designer to decide whether additional replication of components is necessary to achieve the specified level of dependability.

In the state-of-the-art of dynamically reconfigurable systems the following research gap exists that has to be addressed in order to solve the problem of dynamic reconfiguration. A system architecture is required within which an application can be decomposed into self-contained components that interact with each other in order to provide the specified services. This architecture has to support the dynamic addition and removal of components and interactions between these components, as well as the integration of new software components at runtime. As the system has to react to unforeseen events, a request for reconfiguration has to be handled autonomously

by the system without the interaction of an operator or the previous definition of fixed reaction strategies. Therefore, an open challenge is the modelling of the system and its environment as well as corresponding algorithms such that the computer system is enabled to automatically identify redundant information in the system. After the identification, the knowledge about redundant information has to be transformed into new configurations of components and their interactions. The reconfiguration system has to ensure that components only interact with each other if the exchanged information is semantically matching and the required minimum data accuracy is provided. To solve this problem, a sound definition and models for the semantics and accuracy of information are missing that can be processed by machines. As the dynamic reconfiguration is applied for real-time systems, the reconfiguration framework has to meet temporal requirements. This results in the challenge to ensure temporal bounds for the execution of reconfiguration actions.

By providing answers to these open research questions, this work contributes to extending the state-of-the-art. The contributions can be summarized as follows:

- ✓ Definition of the building blocks of a scalable system architecture allowing dynamic reconfiguration. Within this architecture, system services are provided by the interaction of self-contained components. Reconfiguration is conducted by modification of component interactions and the automatic generation of software components that deduce missing information from the information provided by groups of other components.
- ✓ Specification of a modelling language for a knowledge base that provides a machine-readable semantic description of the building blocks of the physical system and their hierarchic arrangement, system and environmental properties, as well as interrelations between these properties.
- ✓ Time-bounded algorithms for the identification of implicit redundancy and for service orchestration, which also ensure that only semantically matching information is exchanged between components. The algorithms can be applied during system design to reduce the amount of required explicit redundancy, and online to perform dynamic reconfiguration upon component failures or upon the addition/removal of components to/from the system.
- ✓ A methodology for automatic evaluation of the accuracy of information that is achieved by implicit sources of redundancy. This methodology is based on the probabilistic uncertainty of information resulting from imprecise measurements of system properties and the inexactness of models relating these properties.
- ✓ Evaluation of the proposed framework using realistic use case examples from the automotive domain.

1.3 Structure of the Work

The remainder of this work starts with an introduction to basic concepts related to dynamic reconfiguration of real-time systems.

Afterwards, Chapter 3 specifies the requirements for a dynamically reconfigurable real-time system. Also related work is presented including a discussion of the ability to fulfill the requirements by the approaches described in related publications.

The architecture and building blocks proposed for a system with reconfiguration capability are introduced in Chapter 4.

In Chapter 5 the modelling of a system and its environment is explained. Modelling includes the creation of a knowledge base which can be searched for implicit redundancy, as well as the representation of components and its services in the knowledge base.

The algorithms for service orchestration are presented in Chapter 6. Service orchestration consists of the search for groups of components that are able to interact and which represent a redundant source of information. In addition, the algorithms automatically generate software components that use the services of an identified group of components in order to provide a requested service.

After that, a methodology for determining the worst-case uncertainty of redundant information produced by combinations of components is described in Chapter 7.

Chapter 8 outlines the implementation of the knowledge base containing the system model, as well as the algorithms for searching for combinations of components that represent redundant sources of information and for the calculation of their worst-case uncertainty.

A use case description and details about the setup of experiments conducted in order to evaluate the applicability and performance of the developed algorithms can be found in Chapter 9. The corresponding results are provided and discussed in Chapter 10. This chapter also revisits the requirements and elaborates their fulfillment with respect to the framework introduced in this work.

Finally, Chapter 11 concludes the thesis and provides suggestions for future work.

Basic Concepts

Within this chapter fundamental concepts for the knowledge-based dynamic reconfiguration of real-time systems are described as the basis for the remainder of this work. The introduction of basic concepts like *real-time system*, *service*, *reconfiguration* or *uncertainty*, which are used differently within distinct communities, is essential to clarify the terminology of the thesis.

The chapter starts with an introduction to the types of systems which are of relevance for the dynamic reconfiguration framework. Afterwards, the decomposition of these systems into components is discussed in Section 2.2. In Section 2.3 the meaning of dynamic reconfiguration is elaborated. Concepts related to system dependability are introduced in Section 2.4. Additional background information on conceptual modeling is provided in Section 2.5. Section 2.6 finalizes the chapter with an insight into uncertainty of information.

2.1 Real-time Systems

The reconfiguration mechanism at the core of this thesis targets the automated recombination of information flows and processing in *embedded real-time systems*. Among the various definitions of a real-time system in literature, the following definition is considered the most appropriate one to be applied in this work:

'A real-time computer system is a computer system where the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical time when these results are produced. By system behavior we mean the sequence of outputs in time of a system' [56, p. 2].

Real-time computer systems are embedded into a larger system, where their purpose is to monitor and control physical processes. Examples of such systems are cars, airplanes or industrial plants. The integration of a cyber system and a physical part, that is controlled by that cyber system, is also referred to as *cyber-physical system (CPS)* [73]. At each instant of time, the CPS is in a certain state, which we define as *'the totality of the information from the past that can have an influence on the future behaviour of a system'* [4, p. 30]. In the state distinct attributes

– i.e., characteristic qualities of entities that constitute the system – are subsumed. A valued attribute is referred to as a *property*. The state can be *observable*, which means that the properties constituting the state can be measured by sensors (e.g., the speed of a vehicle), or it can be *unobservable* in case these properties cannot be measured directly with the sensors of a given system (e.g., the curve radius of the vehicle). If the state is *partially observable*, only some of the properties are observable, but from them the value of unobservable (hidden) properties might be estimated. On the other hand, some of the properties can be directly controlled by actuators, and due to interrelations between properties, actuation on one property might influence other properties as well (cf. stigmergic channels in Section 2.2).

To control certain properties of the physical part of the system, which is also denoted as *controlled object*, a *control loop* is implemented, that tries to keep these properties within desired ranges. This loop reads parts of the observable state of the controlled object by sensors, uses control algorithms to processes new setpoints, and applies these setpoints to the actuators. Upon actuation, the system reacts with a changed system state that again can be observed by sensors, which closes the feedback loop. As the physical part of the system changes its state based on its inputs and the progression of time, the cyber part needs to be sensitive to the progression of time. In order to control the physical environment, the cyber system has to react on stimuli from the environment within a bounded time, where the temporal bound is defined by control engineers in order to attain stable control. The latest instant in time, at which the results for this reaction (including the results of intermediate computation steps) have to be produced, is called *deadline*. A deadline is considered *soft*, if the result of a computation is still of utility, even if the deadline was missed. Otherwise, the deadline is classified as *firm*. When the consequence of a firm deadline miss may result in a catastrophic event, it is named *hard deadline*. For instance, a deadline miss when decoding the frames of a video stream might result in a diminished user experience, but the frame can still be displayed. On the other hand, if the control signal for the spark plug of a car engine arrives too late, the engine might even be damaged. A *safety-critical real-time computer system* is a system that must meet at least one hard deadline [50].

Two methodologies are commonly used to implement the control loops: the *event-triggered* and *time-triggered* approaches. In case of the event-triggered paradigm the actions of the control loop (i.e., execution of control tasks, exchange of messages) are started by events denoting a significant change of state in the controlled object or by interaction with a system operator (e.g., pushing the call button of an elevator). In contrast, when the time-triggered approach is used, the activities are initiated based on the progression of time at predefined instants. Many real-time systems perform a periodic activation of communication and tasks, because of better temporal predictability and determinism. Furthermore, the time-triggered paradigm can be applied to prevent temporal and spatial interference of independent tasks [58], which simplifies fault-tolerance, validation and certification of the system. The periods of the control cycles are derived from the dynamics of the environment in order to bound the error at the time of use of information, as well as from the temporal constraints for actuation (e.g., stability of control of a control loop, permitted time interval until safe state is reached). Typically, the deadline equals the period of the control cycle.

In a *distributed real-time system* the functionality of the control application is distributed on several *nodes*. Such nodes are self-contained computers including hardware and software,

that are interconnected by a communication network. In order to achieve a common goal in the system, a consistent global state has to be maintained among the processes that cooperate in a distributed application. This requires the execution of a common set of distributed protocols. While distributed systems are often used to increase the dependability of a system, an inadequate system architecture can also reduce the reliability of the system. Using a time-triggered architecture (as described in [62]) with a synchronized global notion of time and a deterministic communication protocol, which provides temporal guarantees on the exchange of messages, facilitates the implementation and execution of reliable distributed applications. [116] provides more details about distributed systems and adequate architectures.

2.2 Component-based Design

CPSs are typically partitioned into components (i.e., hardware and/or software) in order to manage the complexity and enable independent development. 'A *component* is a *self-contained sub-system that can be used as a building block in the design of a larger system*' [61]. These components provide a *service* to other components by the interaction via well defined interfaces, called *Relied Upon Interfaces (RUIs)* [4, p. 21]. In the context of this work, a service denotes the intended behaviour of a component visible at its RUIs, which provides an added value to interacting components. This consideration implies that the component is working correctly, as otherwise no service, or a reduced service is provided by that component. Conversely, the expression *service failure* refers to the behaviour of a component that is not able to continue providing its service. The focus of this work is on the services provided by components, rather than on a component itself. Example services in a system with the proposed reconfiguration framework are sensors that provide the measured value of a system property to other services, an actuator service that uses the output of other services to control the actuator attached to the component, or processing services using the input from a group of services to produce outputs that are read by other services.

RUIs can further be classified into *Relied Upon Message Interfaces (RUMIs)* and *Relied Upon Physical Interfaces (RUPIs)* [59] as depicted in Figure 2.1. The RUMI implements a direct message-based communication channel (i.e., the *cyber channel*) between two components. On this channel, an explicit exchange of information about past, current or predicted future states is performed. In contrast, the RUPI handles the interaction with the physical environment by sensors and actuators. This interface only allows interactions related to the current environmental state (i.e., determine the current value of a property or modify a property's value), while information about past and future states can only be estimated if environmental models are available.

Distinct properties within the environment are related with each other following mathematical, mechanical, electrical, chemical or other physical laws. As one component modifies some system properties via an actuator, other properties in the system might be influenced as well. For instance, if a heating element in a closed vessel containing water is activated, also the pressure inside the vessel is influenced and increases. By reading the value of a system property that has been influenced directly or indirectly, sensing components are able to reason about the actions

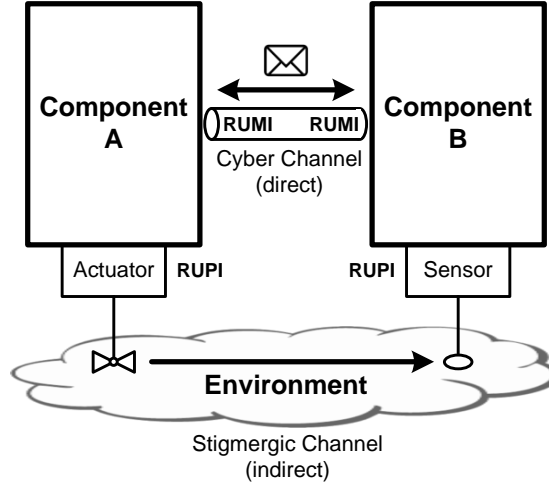


Figure 2.1: Distinction between RUMI and RUPI. Adapted figure from [59].

of the controlling component. This creates an indirect communication channel between these components, which is called a *stigmergic channel*.

Similarly, modifications of properties caused by environmental effects can influence the value of other properties. Two components with the intention to reason about these environmental effects do not necessarily have to observe the same property, but each of them might investigate different properties which are influenced by the same effect. The other way around, multiple actuating components and environmental effects can manipulate the same system property. This technique, for example, is used in biological systems like ant colonies, where each individual ant affects the environment by distributing pheromones on their trails [36]. On routes that are taken by more ants, a higher intensity of these pheromones – which represents the property of interest – is observed, which attracts even more ants to take that route. Environmental effects ensure the decay of the pheromones if a route is abandoned.

While the communication across RUMIs is initiated by the sender of a message with the explicit purpose of information exchange with a certain group of receivers, the primary intention of a component which affects its environment via the RUPI is to control the state of the environment. Typically, a component is not even aware whether there are any other components sensing the properties that are influenced by the control activities.

The reconfiguration framework elaborated in this work focuses on the automatic exploitation of stigmergic information flows between system properties that are accessible via RUMIs. If a required property is not directly observable, stigmergic information flow allows to deduce the value of that property from the observations of distinct properties.

In order to enable the exchange of information and the provision of services, an exact specification of the component behaviour on both types of interfaces is required. An appropriate specification describes all interactions in the temporal and value domain, and allows to abstract from the internal implementation of the component. In case of stigmergic communication, the RUPI specification also requires a detailed description of the environment, containing physical

models of the dependencies between system properties. This facilitates the compositional design [61] of the CPS from independently developed and tested components, and enables dynamic reconfiguration by modification of component interactions and imitation of failed services.

For a system using the reconfiguration framework proposed in this work, a service-oriented approach [24] is assumed, where services either control the environment by actuators or they provide information about system properties, state variables and intermediate results to other services. This information can be acquired from the environment by sensors, or it is aggregated and computed based on information received from other services. The global system behaviour is then given by the composition of individual services and the flow of information among them.

2.3 Dynamic Reconfiguration

A CPS consists of a set of components that are distributed on nodes and which interact with each other at their RUIs in order to obtain the specified system behaviour – i.e., the system service. Within this thesis, the number and arrangement of distinct types of components providing the specified services, together with their particular parameter values and their interactions defined for a given period of time, is referred to as the *system configuration*. The adaptation of the configuration to changed system objectives or to preserve the system services after a component failure, is called *reconfiguration*. It can be achieved, for instance, by the addition of physical nodes and logical components, modification of parameters or component interactions. The authors in [9] distinguish between *programmed reconfiguration* and *evolutionary reconfiguration*. The former is considered in the system design phase and automatically leads to reconfiguration upon the satisfaction of certain conditions during system operation (e.g., recovery after failures or change of system mode). The latter concerns modifications in the system structure independently from application specifications (e.g., maintenance or component upgrades).

In this work, *dynamic reconfiguration* refers to the reconfiguration at runtime, where the process of reconfiguration is triggered as a reaction to changes in the system and its context, and which cannot be anticipated at design time. Therefore, due to the dependence on runtime information, the actions performed as part of the reconfiguration (e.g., adding or removing components, modifying interactions between components) have to be determined during system operation. The changes which are of interest in this thesis mainly concern the availability of services and the flow of information between these services. Changes in the environment, which do affect the availability of information, but which might have an effect on the application (e.g., temperature rising above a certain level, control input of operator), are not relevant for the proposed reconfiguration framework.

The purpose of reconfiguration is to restore the flow of information from components producing information (e.g., sensors) to components that consume information (e.g., actuators or output devices), whenever this flow is disrupted by a component failure (e.g., caused by a damaged sensor) or the intentional deactivation of a component. Alternatively, sources of information need to be identified for new services in the system, which require information that is not yet provided. Thereby, the reconfiguration consists of the activation of required existing services (i.e., starting available components), the automatic generation of software components that pro-

cess available information to provide the requested information, and the coordination of existing information flows by connecting services.

A combination of services that can be used to provide a desired information is also referred to as a *composition of services* – or *service composition*. The process of finding a valid service composition is called *composition search*. The process of *service orchestration* includes the composition search and the automatic generation of a connecting service that coordinates a service composition by performing a computation to generate the value of the desired property from the available information. It is a key objective of this work to provide a framework that automates this service orchestration.

2.4 System Dependability

One major purpose of the reconfiguration framework is to increase the dependability of the CPS. According to [67], *dependability* is the '*property of a computer system such that reliance can justifiably be placed on the service it delivers*'. A list of attributes has been elaborated by the authors in [68], that highlight the distinct aspects of dependability:

- **Reliability:** is a measure for the continuity of service, given as the probability that the system is able to provide its service at time t , when it was operating correctly at $t = t_0$. Assuming a constant *failure rate* of λ *failures/hour*, the reliability at time t can be calculated with the formula:

$$R(t) = \frac{1}{e^{\lambda(t-t_0)}} \quad (2.1)$$

The inverse of the failure rate $1/\lambda$ is called *mean time to failure (MTTF)*, which is given in *hours*.

- **Maintainability:** the probability of repair within a time interval d . For a constant *repair rate* μ , the *mean time to repair (MTTR)* is given by the inverse of the repair rate $1/\mu$.
- **Availability:** denotes the fraction of time that the system is ready for usage. It is based on the assumption, that the correct system operation is disrupted by incidents that prevent the system from providing its service. In case of constant failure and repair rates, the availability A can be expressed as a function of MTTF and MTTR:

$$A = \frac{MTTF}{MTTF + MTTR} \quad (2.2)$$

The sum $MTTF + MTTR$ is also known as *mean time between failure (MTBF)*.

- **Safety:** concerns the prevention of critical failures in the system that lead to catastrophic consequences for the system environment. Safety is reliability with respect to critical failure modes.
- **Confidentiality:** refers to the prevention of unauthorized disclosure of information.

- **Integrity:** is related to the avoidance of improper alternation of information.

Dynamic reconfiguration contributes to improving the first four attributes by automatically orchestrating the remaining components after a failure. The authors in [6] provide a detailed elaboration about the pathology of failures in a system. Accordingly the *failure* of a component means that the behaviour of the component deviates from the service specification (i.e., in the value and/or the temporal domains). This failure is caused by an unintended internal state, called *error*. A *fault* is the adjudged or hypothesized cause of an error. Faults introduced during the development phase (e.g., incorrect program code) may exist in a component without causing an error, and hence, they are dormant. If such a fault is activated, it may lead to an erroneous state and result in a component failure. Operational faults, which arise during system operation, have physical reasons (e.g., radiation) or are caused by wrong inputs. In the standard ISO 26262 [43] the fault tolerant time interval (FTTI) is defined as the '*time-span in which a fault (1.42) or faults can be present in a system (1.129) before a hazardous (1.57) event occurs*'. As a failure within one component can lead to incorrect interactions with other components, a failure might propagate and thereby affect the provision of other services.

Different methods have been developed to counteract failures and attain a high degree of dependability in a system. According to [6], these methods are classified as follows:

- **Fault prevention:** aims at the reduction of the number of faults introduced during the development and production of a system by applying certain software and hardware development patterns, as well as development techniques like testing or formal verification.
- **Fault tolerance:** focuses on the continuation of the system service despite the presence of a fault. Fault tolerance comprises *error detection* and *system recovery* mechanisms. Error detection tries to identify erroneous states in the system, while system recovery transforms erroneous states into correct system states and prevents the reactivation of faults.
- **Fault removal:** is conducted during system development and its operational phase to minimize the number and seriousness of faults. In the development phase, validation and verification techniques are applied to ensure that the system behaviour complies with the specification and to find faults by static analysis and testing. Repair and maintenance actions are taken to remove faults during the operational phase.
- **Fault forecasting:** involves the evaluation of the system w.r.t. the occurrence of faults and their activation. Qualitative evaluations concentrate on the failure modes or combinations of events leading to system failures, as well as on the consequences of failures. Quantitative evaluations estimate the number of faults and aim to predict future incidents.

Dynamic reconfiguration is a strategy to increase fault tolerance. Even though the focus of this thesis is on system recovery, parts of the framework can also be used to identify service compositions that act as a source of information for error detection mechanisms.

In most cases fault tolerance techniques rely on redundant information, where deviations between these sources of information indicate an error. Either a source for redundant information is introduced explicitly in the system by *replication* of components or redundant computations,

or implicit redundancy is obtained by exploiting the knowledge about correct system behaviour and dependencies between system properties (e.g., by using analytical redundancy). The justification of using replicated components to build ultra-reliable systems is based on the assumption, that replicas fail independently [13]. A prominent representative of fault tolerance techniques that uses explicit redundancy is triple modular redundancy (TMR) [5] – or N-modular redundancy in the general case of N replicas. It implements both, error detection and system recovery by *fault masking*, which conceals possible failures of components. Three components, which might have a diverse implementation, provide the same service. The output of them is compared by a *voter*, which decides on the final output based on a majority vote – i.e., more than half of the voter inputs have to be equal – and indicates an error if an input deviates from the decided output. An example for implicit redundancy used for error detection are *input assertions*. A component performs a plausibility check of information it receives from another component. Thereby it can use knowledge about the allowed temporal behaviour, the value ranges or the maximum rate of change of values that cross the input interface. If the input data for the component does not comply with the restrictions according to the prior knowledge, an error is indicated.

In order to apply appropriate fault tolerance techniques for a CPS, assumptions about the type of faults, the frequency of component failures and their failure modes have to be made. These assumptions are known as the *fault hypothesis* [86]. The probability that these assumptions hold in reality is referred to as the *assumption coverage* [94]. If the assumptions of the fault hypothesis are violated, the whole system may fail, as fault tolerance mechanisms might become ineffective.

Within the fault hypothesis, the system is partitioned into subsystems – called *fault containment regions (FCRs)* – which are assumed to fail independently. ‘An FCR is a collection of components that operates correctly regardless of any arbitrary logical or electrical fault outside the region’ [65]. On the other hand, a fault within an FCR cannot cause component failures outside the region. Sharing resources among FCRs (e.g., power supply, common clocks), external faults (e.g., radiation, electromagnetic interferences (EMIs)) or design faults may compromise their independence.

Based on the FCRs defined for a CPS, assumptions about the *failure modes* for these regions are made. Failure modes describe the perception of effects of a failure by the user of a service. The failure mode of an FCR has an influence on the degree of redundancy required to achieve fault tolerance in the system. [18] and [86] provide the following classification of failure modes based on the rigidity of assumptions:

- **Fail-stop failures:** the FCR does not produce any outputs until it is restarted. All correct FCRs are assumed to detect fail-stop failures.
- **Crash failures:** similarly to fail-stop failures, the FCR does not produce any output, but correct FCRs may not detect the failure.
- **Omission failures:** if an FCR outputting messages fails to send a message, or a sent message is not received by the receiver FCR, an omission failure happened. No reaction to input sent to the receiver is produced. These failures might remain undetected.

- **Timing failures:** the FCR produces its output too early or too late with respect to the timing specification.
- **Byzantine or Arbitrary failures:** the effects of these failures, as they are perceived by service users, are not constrained. Messages might even be forged or the same message can be observed differently by distinct users, as described by the problem of Byzantine Generals [66].

In [51] the following additional failure modes are described:

- **Babbling Idiot:** a failure mode where an FCR sends messages that are not compliant with the temporal specification. This behaviour can lead to monopolization of the communication network (e.g., a constant flow of high priority messages in a CAN network).
- **Masquerading:** an erroneous component sends or receives messages by imitating the identity of another component, without the authority to do so.
- **Slightly-off-Specification:** a special kind of byzantine failure, where the value or the temporal behaviour is marginally outside the specification. For instance, the electrical voltage of a signal, that is representing the logical values 0 or 1, is close to the threshold between these values. Two independent recipients might interpret the signal differently. Similarly, if a message arrives marginally outside the allowed temporal interval (e.g., after a timeout) due to imprecise clock synchronization, one of two recipients might see the message as timely, while the other one perceives a timing failure.

Beside the assumption about failure modes, the fault hypothesis has to include a *failure rate assumption* for FCRs. It specifies the number of failures the system has to be able to tolerate without compromising the correct operation. The assumption considers the distinct failure modes as well as the persistence of failures. *Transient failures* are often caused by radiation, EMIs or disturbances in the power supply. They disappear by themselves, such that the system continues to operate after the incident. In contrast, *permanent failures* require an explicit repair action. Transient failures that occur frequently due to a permanent fault are also called *intermittent failures* [50, p. 121].

The maximum duration of time it takes until an FCR is able to provide its specified service again after a failure, is named *recovery interval*. It comprises the time until repair actions are taken after a permanent failure, or the sum of durations it takes for error detection, FCR restart and state restoration, in case of transient failures [86].

The maximum number of simultaneous failures a system has to tolerate determines the required degree of redundancy, and it is therefore part of the fault hypothesis. The failure rate and the recovery interval significantly influence this parameter. The assumption of a single failure only is prevailing in many present-day safety-critical systems [86].

During the development of safety-critical systems a high effort is invested to minimize the probability of component failures leading to catastrophic accidents. This includes the assessment of the system with respect to dependability and risks compromising the safety of the system. Distinct techniques for hazard and risk analysis are presented in the literature [12, 50], like

Fault Tree Analysis (FTA), *Failure Mode and Effect Analysis (FMEA)*, *Hazard and Operability (HAZOP)* or *Event Tree Analysis*. In very rare cases where the fault hypothesis is violated, all measures to ensure safety might fail and the system ends up in an unspecified state. Nevertheless, the system must try to recover and remain operational by implementing a *never-give-up strategy* [56].

The dynamic reconfiguration framework presented in this work provides a contribution to increase the dependability of a CPS at different levels:

- During the development phase of the CPS the semantic models can be used to identify redundant information in the system and ensure the diversity and independence of redundant sources of information. Here, diversity means, that distinct methods are used to acquire the information.
- In the operational phase of the CPS, faulty components – particularly components experiencing a permanent failure – can be automatically replaced by combinations of available components. This reduces the probability of violating the assumptions about the maximum number of failures in the fault hypothesis. Furthermore, it can be considered to be an automatic repair action, which increases the availability of the system.
- As the reaction of the proposed framework to a component failure is determined at run-time, also component failures can be covered that have not been anticipated. Therefore, dynamic reconfiguration can be suitable for the never-give-up strategy.

With respect to the FCRs defined in the fault hypothesis for a CPS with dynamic reconfiguration capability, it has to be assumed that after an FCR failure a reconfiguration needs to be triggered for all components that are comprised by an FCR. If, for instance, a network component fails such that parts of the distributed nodes are not accessible anymore, all services on those nodes have to be restored by reconfiguration. Since failures due to transient faults disappear after a component restart, the reconfiguration framework is focused on the reconfiguration after an error detection mechanism identified a permanent or intermittent failure of a component. Within this thesis no emphasis is put on this error detection mechanism, nor on the deactivation of failed components disturbing the correct operation of other components (e.g., babbling idiot failures). It is assumed that such mechanisms are already given and faulty components have been isolated before reconfiguration starts. Therefore, only the error detection mechanism is sensitive to the different failure modes, while the reconfiguration framework assumes fail-stop failures. Since the reconfiguration can be conducted for only one component at a time, the maximum rate of permanent and intermittent failures is constrained by the worst-case execution time of reconfiguration, which mainly depends on the time required to perform the composition search in the knowledge base (see Section 6.5). For failures that affect more components simultaneously, the allowed failure rate additionally has to be divided by the number of affected components, or concurrent substitutions are established by instantiating the reconfiguration mechanism on distinct nodes. Finally, no failures of the reconfiguration framework are assumed. To relax this assumption, replication-based fault tolerance techniques can be applied to the reconfiguration mechanisms.

2.5 Conceptual Modeling

Dynamic reconfiguration requires the autonomous identification of new valid configurations upon changes in the context of an application. This might be a simple task for a human knowing the dependencies between different components in the system and its environment – provided that not too many of these components and environmental conditions are involved. For a computer system the knowledge about these dependencies has to be modeled and stored in a knowledge base that can be searched automatically to find valid combinations of components. The models within the knowledge base are an abstraction of the relevant properties and their interrelations in the environment. Whether properties or relations are relevant for reconfiguration is highly application specific. In general, a lower level of abstraction leads to more knowledge that is stored explicitly, which provides more possibilities for new configurations. However, the level of abstraction also determines the memory size of the knowledge base and the time required to find a valid configuration.

Conceptual modeling is concerned with the construction of a representation of selected phenomena in a domain [117]. These phenomena are either static, like things and their properties, or dynamic, like events and processes. By abstraction different *categories* are created, which are sets of elements that share common characteristic features [56, p. 32]. Since the elements of a category can be categories as well, a hierarchy of categories (i.e., concrete categories at the bottom and abstract categories at the top) is formed. A category, that is extended by a set of beliefs about its relations to other categories, is called *concept*. The structured network of interrelated concepts, which is also named *conceptual landscape* [56, p. 35], constitutes a knowledge base modelling the domain of interest. An *ontology* is the explicit specification of this conceptual landscape [37]. Traditionally, '*an ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary*' [83]. Besides this terminological focus of ontologies where mainly hierarchical relations between properties are defined (e.g., generalization and specialization relations), within this work an ontology also includes behavioural models about the interrelation of properties.

According to [79], the human mind is only capable of keeping 7 ± 2 concepts in the short term memory, which does not allow a human engineer to reason about complex dependencies between concepts of a CPS without supporting tools. Fortunately, an engineer creating the ontology, which describes a complex CPS, can benefit from distinct simplification strategies [54]:

- *Abstraction*: by omitting irrelevant details of concepts at an higher level of abstraction the cognitive complexity is reduced and the focus can be kept at the properties and relations at that level.
- *Partitioning*: the system can be decomposed into nearly independent parts that can be described with a few concepts and which are studied in isolation.
- *Segmentation*: complex behaviour is temporally decomposed into parts that are described sequentially.

After its creation, an ontology can automatically be analyzed by algorithms in order to identify dependencies between concepts which can hardly be managed by the human mind. Therefore an appropriate system model is not only advantageous for dynamic reconfiguration, but can also be beneficial during the development phase of the system.

An ontology that contains a detailed description of the environment of the CPS can also serve as an *explanation* for data items processed by the cyber system. Data that is augmented by an explanation constitutes *information*, and forms an *itom* (information atom) [57]. '*The explanation of the data establishes the links between data and already existing concepts in the mind of a human receiver or the rules for handling the data by a machine*' [4, p. 28]. In most real-time systems the explanation – also referred to as *semantics* of information – is given implicitly in the program code of an application by the usage of variables. Therefore, the explanation cannot be processed automatically by a computer system. Relating data with concepts in an ontology does not only provide the semantic of the information, but also allows to reason about its context (i.e., its enclosing concepts) in the CPS. By linking the input and output data of components with concepts, sources and sinks of information are defined. It is the task of the dynamic reconfiguration framework to identify combinations of available sources of information which are equivalent to the semantic required by a sink.

2.6 Information Uncertainty

In a CPS, most of the information processed is either based on the measurement of properties of physical entities (e.g., temperature measurement by sensors), or an estimation of the value of such properties (e.g., estimated driving range of a car). Usually, also these estimations use measurements as input. The appropriateness of a source of information for the intended usage is not only a question of semantics, but also of the qualitative correctness of this information. This qualitative measure refers to the *accuracy* of the information, or the *uncertainty* about its correctness, which depends on the quality of measurements. In discussions about the correctness of information, uncertainty will be the preferred term within this work.

With *uncertainty* the possible error in measurement is denoted that results from the deviation of the measured value of a physical entity from its true value. Uncertainty in measurement influences the significance of observations of the world and the reliability of predictions about future happenings. Therefore, a widely accepted science with the single focus on measurement has been founded in the past: *metrology*. This science is especially important for CPSs, as the quality of monitoring and control strongly depends on the accuracy of input data to the control system.

In the domain of metrology, the *Guide to the Expression of Uncertainty in Measurement (GUM)* [44] can be regarded as a norm for handling of uncertainties when measuring an entity. According to the GUM, '*in most cases, a measurand Y is not measured directly, but is determined from N other quantities X_1, X_2, \dots, X_N through a functional relationship f* ' [44, p. 8], which will be referred to as a *measurement function*. Beside definitions w.r.t. uncertainty the authors describe how the uncertainties of input values of a measurement function influence the uncertainty of the final measurand. The first supplement to the GUM – *Propagation of distributions using a Monte Carlo method* [45] – provides additional methods based on the Monte Carlo

technique to numerically determine the uncertainty of measurement functions with a single output and arbitrarily shaped uncertainty distributions of its input quantities.

In general, various sources for uncertainty in measurement can be identified (e.g., inexact definition of measurand, resolution of measurement instruments, undefined environmental conditions, etc.), which are often divided into the two categories *random* and *systematic* errors [44, p. 5]. Random errors are not predictable and usually of stochastic nature (often also called measurement noise), and hence, they cannot be corrected. But since they manifest in variations of the measurand during repeated observations, the degree of variations can be estimated if multiple observations of a constant – or known – physical entity are available. In contrast, systematic errors originate from known, quantifiable effects that influence the measurand (e.g., a pressure sensor that is sensitive to variations in temperature). They cannot be directly eliminated, but correction mechanisms can be applied to the measurand to compensate for this type of errors. Since the GUM introduces the assumption that *'the result of a measurement has been corrected for all recognized significant systematic effects'* [44, p. 5], this assumption is also adopted within this work. Therefore, within this work the term uncertainty only concerns random errors and other errors that cannot be corrected with reasonable effort (e.g., imprecisions in system models).

The objective of the methodologies described in the GUM is to determine the true value of the measurand as accurate as possible, when a defined measurement function is used. Such measurement functions combine several given measurements of distinct measurands to determine the value of the entity of interest. Thereby, the uncertainties of input values to the measurement function have different impacts on the uncertainty of the function output (i.e., the value of the measurand of interest). Within this work, similar techniques are applied to determine the worst-case uncertainty of automatically determined combinations of services, which can be seen as measurement functions. But in contrast to the methods described in the GUM, where the uncertainty of the measurand for a particular given set of input values for the measurement function is of interest, for this work the usage of these methods has to be adapted to analyze the whole ranges of input and output values of the measurement function. The knowledge about the expected worst-case uncertainty can then be used to evaluate whether the measurement function – or a group of several concatenated measurement functions – is appropriate to fulfill the requirements of an application that uses the measurand as input.

Requirements and Related Work

This chapter discusses requirements, which are relevant for the dynamic reconfiguration of a CPS. In addition, an overview of related work is given along with a discussion of the fulfilment of the identified requirements.

3.1 Requirements

The requirements presented in this section are discussed in thematic categories. For the development of a CPS with reconfiguration capability, numerous requirements can be deduced from each category.

3.1.1 R1: High Dependability at Low Cost

Error detection algorithms and the establishment of a high degree of fault tolerance in a CPS require redundancy. For this purpose, in many systems replication or redundant components are introduced (e.g., redundant sensors in a car), which increases the cost of the system [38]. The reconfiguration mechanism should reduce the number of redundant components in the CPS without reducing the provided system services nor the fault tolerance of the system. Explicit redundancy often implies additional hardware that is necessary, like duplicated sensors, communication networks or computational nodes to host redundant software components, or in case of temporal redundancy more time is needed. This does not only increase the production cost of the system, due to extra cost for hardware and its installation, but also raises the weight and consumption of energy, which comes with further expenses and might be of concern for mobile systems. Nevertheless, redundant components do not provide additional system services.

Implicitly redundant sources of information that originate from combinations of existing components should be identified and used instead of adding explicit redundancy.

3.1.2 R2: Never-Give-Up Strategy

A safety-critical CPS has to be able to provide its service also in the presence of faults. Therefore, fault tolerance techniques have to be applied that prevent a fault within a component to cause a failure of a safety-relevant service of the system. If the reaction to faults is defined at design time, only expected faults, which are described in the fault hypothesis, can be handled. Faults that are not considered in the fault hypothesis can lead to a system failure and can result in a catastrophe.

The reconfiguration mechanism can establish a never-give-up strategy and be able to react to unforeseen faults. A never-give-up strategy has to be triggered for any type and combination of faults that violate the fault hypothesis. This requires that new configurations and service compositions are determined online after the fault occurred. In order to flexibly respond to different types of faults, the process of service orchestration cannot only consist of the online selection of predefined service compositions, as it is infeasible for complex CPSs to store all possible compositions.

3.1.3 R3: Real-time Support for Reconfiguration

Per definition real-time systems are sensitive to the progression of time. To ensure stable control of the CPS, the cyber system has to apply new control commands within certain temporal bounds after the observation of the controlled object. These bounds are imposed by the physical process and are usually represented by the deadlines of the control loop, where the miss of a hard deadline could lead to a catastrophe. After the detection of a permanent or intermittent component failure the dynamic reconfiguration framework is triggered. The reconfiguration process either changes the configuration to re-establish the service of the failed component, or it has to provide a negative response to indicate that no new valid configuration can be determined, and other fault mitigation strategies have to be applied (e.g., switching to a safe state). To ensure that no hazardous events can occur, this process has to be completed within the FTTI, which also requires a guaranteed maximum response time. Consequently, dynamic reconfiguration must not be applied to control physical processes, if the FTTI imposed by the process is shorter than the guaranteed worst-case response time.

In order to guarantee a worst-case response time, it must be possible to determine the worst-case execution time (WCET) of the reconfiguration algorithms for a given knowledge base that is modeling the system. It is only feasible to apply dynamic reconfiguration, if the determined WCET is shorter than the maximum response time imposed by the physical process.

3.1.4 R4: Semantic Correctness of Configurations

The control of a complex CPS involves the observation of real-time entities, interpretation of the obtained information, computation of set points and control commands, and the actuation on the real-time object based on these set points and commands. During the execution of this control process, many distinct items are processed and exchanged between components. Obviously, control commands can only be appropriate if they are based on the interpretation of semantically corresponding items. For instance, it usually does not make sense to control the temperature of

a room based on the measurement of the barometric pressure. Combining the wrong items may also lead to catastrophic events.

Therefore, new configurations applied by the reconfiguration framework must be semantically correct, ensuring consistency of connected inputs and outputs of services. This requires that mechanisms and models exist which either guarantee correct configurations by construction, or that allow to prove the semantic correctness of a proposed configuration. A framework combining available components purely based on syntactic description of their inputs and outputs (e.g., combining a component that outputs a temperature within a defined range with another component that requires a temperature in the same range as input) would be insufficient to provide services with correct semantics. In such case, an additional check of semantic correctness (e.g., verifying that both temperatures relate to the same system property – like the temperature of the cooling water of an automotive engine) is a prerequisite for the correctness of configurations.

3.1.5 R5: Accuracy of Information

Despite the semantic and syntactic correctness of items exchanged between components, these items might not be appropriate for the control and monitoring of a real-time object in a CPS. In a computer system, an item corresponding to the observation of a system property is referred to as a *real-time image* [50] of the property. The value of a real-time image must accurately represent the true value of the property, as otherwise inappropriate control decisions can be triggered and monitoring information becomes useless. The accuracy of a real-time image depends on various factors, like the accuracy of measurements, exactness of processing of items or the temporal gap between observation and usage of information (as also discussed in [113]). The value of the property of a real-time entity continuously changes with the progression of time. And since there is some duration of time after the observation, within which the real-time image is not adapted to that change, the property's true value might be different from the real-time image. The maximum divergence is determined by the maximum rate of change of the real-time entity, and can be decreased by state estimation techniques.

The dynamic reconfiguration framework must ensure that the accuracy of information provided in the new configuration meets the requirements of the application. The output of a component may only be connected to the input of another component, if the uncertainty about the deviation between the real-time image and the true value of a property is below a given bound.

3.2 Related Work

In the literature, dynamic reconfiguration of real-time systems is associated with many different objectives and fields of computer engineering. The objectives range from reconfiguration upon component failures or adaptation as a reaction to altered resource availability, to reconfiguration of component interactions due to switched modes of operation and changed system goals. Reconfiguration techniques are applied at distinct application levels. For instance, partial dynamic reconfiguration of FPGA devices for mission critical applications is gaining importance in the last decade [11, 88]. Other publications consider the dynamic activation and scheduling of

tasks [17, 33, 99] and the management of resources [7, 49] based on distinct criticality levels and modes of operation.

For instance, in [30] the self configuration of dependent tasks is described in the domain of reconfigurable automotive embedded systems. Tasks can be executed on different distributed ECUs, if all required input information – that is output by other tasks – is available. At system startup only tasks are activated, for which the condition of available inputs holds. After new devices are attached and associated tasks provide new information at their outputs, tasks which previously have been deactivated because of missing input information can be activated. Similarly, in case of the detachment of devices or hardware failures, corresponding tasks and all subsequently depending tasks are deactivated, since input information is missing. While this procedure allows some flexibility in the resource usage and the provision of system services, only fixed sequences of tasks are activated or deactivated. This does not allow to dynamically reason about solutions for unforeseen events. Failure recovery is only possible, if a redundant (diverse) version of the faulty task is available.

Due to the focus within this work on CPSs that follow a component-based design paradigm, the related work presented in the remainder of this section is also based on components and their services, and reconfiguration mainly refers to the adaptation of component interactions. Related work is not restricted to real-time environments, but also includes reconfiguration within web service architectures managing business processes, since dynamic adaptation and service orchestration are already well established concepts in this field. However, the trivial solution that tries to syntactically match the input of a component with the output of all other components – like it is also described as a solution for web service composition – will not be covered in this section. The number of possible combinations that have to be tested in order to find a sequence of services, which finally provides the required output, increases exponentially with the number of components. The restriction to syntactic matching and the poor scalability for a high number of components make such solutions infeasible.

The remainder of this section continues with the presentation of related publications. At the end of the section a summary about the fulfillment of the aforementioned requirements by these related approaches will be given.

3.2.1 Related Work in the Area of Dynamic Reconfiguration

The following related approaches for dynamic reconfiguration are categorised based on their fundamental idea of reconfiguration. As the assignment is not always clear, in some cases an approach might also fit into another category.

Predefined reconfiguration actions

The authors in [81, 82] present an approach for reconfiguration in web service architectures based on architectural reconfiguration actions, which was developed in the *WS-Diamond* project. The focus of this approach is on the recovery after the occurrence of faults and the prevention of Quality of Service (QoS) degradation of web service applications, like online shopping or digital libraries. Depending on the location of a fault (e.g., in the network, the server or in the application) and its type (e.g., blocking caused by missing data or faulty interactions), components

in the architecture are rearranged or reconfigured. Application scenarios are decomposed into functional building blocks and modeled at the conceptual level (e.g., as UML diagrams), such that reasoning about the requirements of services is performed at a high level of abstraction, while abstracting from the details of service implementations. The proposed reconfiguration actions consist of combinations of basic rules for the addition and removal of services, as well as the connections between them. With ordered sequences of these basic rules duplication and substitution actions for services are performed. Duplication is used for load balancing between services, while substitution is implemented for recovery after a fault.

The basic principles behind these actions can also be applied for real-time systems, where software components are replicated or migrated to other nodes for load balancing, or failed services are replaced by similar services in the system. However, service substitution/replacement is restricted to the combination of components with matching interface descriptions (i.e., compliance of syntactic and semantic description). Thus explicit redundancy is introduced in order to enable reconfiguration.

Predefined configurations

When possible configurations are given at design time, the dynamic reconfiguration mechanism has to select an appropriate configuration at runtime. This allows to quickly react to changes in the system and/or its environment, but it is not possible to react to unforeseen events.

In [96,97] the *Adapt.NET* framework for dynamic reconfiguration of component-based real-time software is presented, which uses Microsofts *.NET* environment as a basis. With the framework adaptation of complex real-time applications to changing environmental conditions is performed within bounded time. Furthermore, meeting of task deadlines is ensured despite the concurrent dynamic reconfiguration of an application. Directed acyclic graphs are used to describe component-based applications, where vertices represent components and edges denote connections between these components. The configuration of an application is a particular set of parametrized components together with their interconnections. The framework requires that distinct adaptation profiles are given. These profiles map selected observable environmental conditions and system states (e.g., load conditions or component states) to dedicated application configurations. The configuration manager reads the profiles and configurations, starts an observer which continuously monitors environmental properties and triggers the reconfiguration if certain predefined conditions are satisfied. Dynamic reconfiguration is performed by adopting a new configuration defined in the adaptation profiles.

Due to a set of predefined adaptation profiles, the flexibility of reconfiguration at runtime is restricted to the selection of matching profiles. Also the possible interactions of components are fixed by the graphs describing the application configurations. Therefore, situations which have not been anticipated cannot be handled with this approach, nor does it contribute to the reduction of explicit redundancy in the system. The described technique for interruption free reconfiguration while guaranteeing task deadlines can be performed within bounded time. The evaluation of the worst-case bound for the reconfiguration process is additionally facilitated by only switching between predefined profiles.

The *MARS* approach for predefined software reconfiguration is described in [1,2,110], which focuses on the resilience of software-intensive embedded systems. It enables a trade-off between availability and costs by exploiting implicit redundancy in the system. The system is built of basic components, which exchange information with each other via input and output ports, and hierarchical components that integrate several sub-components. For each basic component different functional behaviour variants can be implemented, which are referred to as basic configurations. For instance, in one configuration the yaw rate of a car is determined from a dedicated sensor, and in another configuration it is calculated from the lateral acceleration and the longitudinal speed of the car. Each configuration has its requirements on the quality of input signals (e.g., exact measurement by sensor, signal is derived from other signals) and provides its output signals with a certain quality. In order to automatically evaluate quality requirements of components, a type system for the quality of signals is proposed.

Adaptation specifications in *MARS* provide an abstract description of requirements of configurations and conditions that have to hold before and after the adaptation. At design time, hierarchical configurations are created from a set of target configurations, which are selected among all possible combinations of component configurations by evaluating properties given in the adaptation specification. Thereby the set of combinations of configurations is significantly reduced, in order to obtain a feasibly low number of configurations that have to be checked at runtime. The runtime adaptation architecture consists of an environment monitor that detects errors and changes in signal quality, and an adaptation framework which is responsible for reconfiguration. A reconfiguration is conducted by selecting an appropriate top level hierarchical configuration, based on which each basic component changes according to the target configuration.

By providing different configurations of components that are based on diverse input signals, this approach can be used to exploit implicit redundancy in the system and to reduce explicit redundancy of components that is otherwise required for fault tolerance. However, implicit redundancy has to be identified at design time and corresponding configurations have to be created by the system designer. Furthermore, target configurations that are fixed while the system is developed allow to trade-off between the availability of the system and its cost. Therefore, the system can only select predefined configurations at runtime, but it cannot dynamically identify solutions (i.e., configurations) for events that have not been anticipated. The proposed quality type system ensures that only those configurations are available at runtime, for which the quality requirements of input and output ports of components are satisfied.

Applications modelled as directed graphs

Several frameworks use directed acyclic graphs to model the dependencies between services or components constituting an application. These graphs are either created at design time and remain static during system operation, or they are generated at runtime which allows to find new configurations when additional services are integrated.

Dynamic service composition for distributed real-time systems is supported by the architecture proposed in [25, 26]. It is targeted at online updating and reconfiguration of services for the purpose of dynamic QoS management by online software and hardware updates, fault

tolerance by substitution of failed services, and load balancing between nodes. Applications are dynamically created by composition of available services, while ensuring the specified temporal behaviour of the application.

The architecture is built upon the flexible time triggered communication paradigm implemented in FTT-Ethernet [91]. This centralized master/multislave protocol enables flexible time-triggered and event-triggered communication between services. The master creates a schedule for each elementary cycle, which consists of a synchronous time-triggered and an asynchronous event-triggered communication window, and enforces timed message exchange and collision avoidance by explicitly imposing message send instants to each slave at the beginning of an elementary cycle. During the asynchronous window, each slave has to be polled by the master in order to be able to send sporadic messages (e.g., alarm messages). To do so, the master has to keep a database of all message streams and tasks in the system. Before new messages are allowed to be scheduled, a schedulability test has to be performed by the master. Besides scheduling of messages, the master can also trigger the service tasks, either dependent or independent of the transmission of messages.

In this architecture, applications are modeled as directed graphs of services, that are provided by periodically executed tasks, and interactions between them in form of messages. Multiple implementations of services may exist with different execution times and other QoS metrics, where the system has to decide which version is used for an application. This passive replication approach is exploited to substitute services in case of failures or to balance the load of services which are frequently requested. Two algorithms are proposed for dynamic service composition which consider the real-time requirements of the application and optimize other quality criteria (e.g., utilization factor of nodes). The first one exhaustively tests all possible sequences of service implementations according to the application graph. Among all combinations of tasks which satisfy the temporal requirements of the application, the combination with the highest quality score is selected. The complexity of this algorithm is given by $\mathcal{O}(k_{max}^n n^2)$, where n denotes the number of services and k_{max} is the maximum number of implementations per service. The term n^2 results from the schedulability test that is applied for each combination. Since the execution time of exhaustively testing combinations might not be feasible for real-time applications, also an improved algorithm that uses heuristics to reduce the number of tests is presented. In the worst-case this improved algorithm has a complexity of $\mathcal{O}((k_{max}/2)^n n^2)$, which – according to the authors – makes it feasible to be applied for real-time systems.

Event though applications can be dynamically created by compositions of available services, the exact sequence of services is already predetermined by the application graph. Fault tolerance relies on the fact, that redundant services exist as standby to substitute failed services. The proposed architecture does not provide the flexibility to react to unexpected events by combining services which are not connected in the application graph. The architecture can be suitable for real-time systems, because an upper bound for the execution time of the service composition algorithms can be determined. However, if more than two implementations are allowed for a service (i.e., $k_{max} > 2$), the bound scales exponentially with the number of services. Therefore, an offline evaluation of this bound might not be feasible for systems with many dynamically changing services.

The dynamic creation of application graphs using the planning graph model known in the artificial intelligence domain is proposed in [120]. The described methodology focuses on syntactic matching and composition of web services which model business processes. Web services are described by their input and output parameters, which are identified by names. If the names of all input parameters appear in the list of output parameters of another service or a group of other services, these services can be connected. A valid service composition consists of a sequence of services, which expects a given set of input parameters and provides a defined set of output parameters. In the trivial case, the input parameters of each service have to be compared with the output parameters of all other services to create such a sequence. Since this approach requires the evaluation of exponentially many combinations, a planning graph is constructed, which only needs polynomial time. Figure 3.1 shows an example of such a planning graph, which consists of action levels A_i and proposition levels P_i . Web services w_k are mapped to the action levels and parameters (i.e., input and output parameters) to the proposition levels. The input parameters for the whole composition are given in level P_0 , while the required output parameters have to be included in the last proposition level.

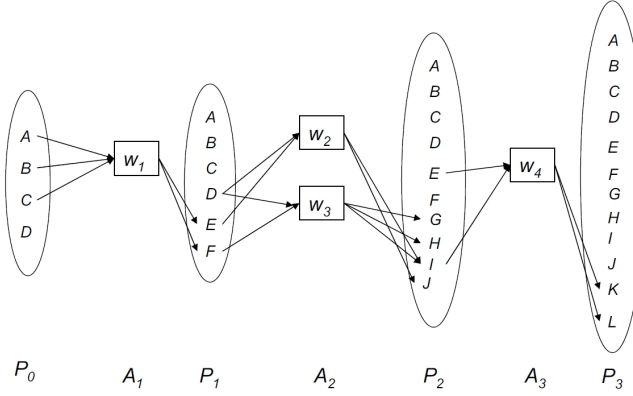


Figure 3.1: Example planning graph from [120].

The planning graph is created iteratively, starting with proposition level P_0 . In the i -th step the new level A_i is created by adding those services which have not been added to any level A_k , $\forall k < i$ and for which all required input parameters are in level P_{i-1} . The new proposition level P_i is obtained from the lower level P_{i-1} together with the outputs resulting from the services in A_i . This iterative expanding proceeds until either all requested output parameters are contained in the proposition level P_i , or a fixed point is reached, which means that $P_{i-1} = P_i$. In the first case, the solution for the requested service composition consists of the sequence of services of the action levels A_1 to A_n , with n denoting the level where all requested output parameters are contained in P_n . Otherwise, no solution for service composition exists.

As the presented methodology is not based on fixed dependencies between services, it provides a high flexibility in the dynamic composition of services, and allows to find compositions even with dynamically changing availability of services. Therefore it can be used to react to unforeseen events. However, it is only possible to find solutions if a complete sequence of available services is given. No services can be generated automatically, which would transform an

input to the requested output. The runtime estimation to generate the planning graph depends on the number of available services, which might change during the mission time of the system, making it hard to provide temporal guarantees at development time.

Knowledge-based approaches

Reconfiguration frameworks including ontologies or other knowledge bases for service orchestration instead of predefined graphs of service sequences provide more flexibility to find appropriate service compositions. The knowledge bases are used to identify semantic equivalences between requested and provided services.

A framework for dynamic resource management in ubiquitous indoor environments, which has been developed in the *CHIL* project, is described by the authors in [89, 105]. Ubiquitous computing environments (like smart rooms) are characterized by their context awareness, referring to the ability to react to changes in the environment (i.e., the context), and dynamicity, which relates to continuous modifications in the system by joining and leaving sensors, actuators, mobile devices or services.

The presented framework uses an agent-based platform, where core agents provide a set of installation-independent basic services for the system. These services include, for instance, a communication mechanism for distributed entities, management of user profiles, observation of context and situation detection or knowledge base access. Basic service agents, which are plugged to the core agents, are tightly coupled with the infrastructure and integrate basic low level services like sensor and actuator control or situation tracking. An ontology at the core of the framework provides a common vocabulary for the information exchange, a mechanism for dynamic integration, registration, discovery and invocation of components, and enables the interchangeability between these components. Two components are interchangeable, if the information is delivered using the same language (i.e., an equal interface specification exists). In order to ensure the same language of similar sensors and actuators, proxy agents are used as wrappers between low level access and an interchangeable high level representation. The following four components are distinguished, which can be registered to the ontology:

- *Installation-specific components*: Sensors, actuators and other devices that are automatically registered after their installation with information about their interfaces, capabilities, APIs, and other information.
- *Perceptual components*: These components process sensor input. They are grouped according to the information required at their inputs and provided at outputs, which are additionally related to concepts in the ontology.
- *Situation models*: Describe the application of perceptual components and infrastructure elements for different situations (i.e., context states). These models are parsed during run-time in order to dynamically react to context switches.
- *Services*: Are actions that are triggered upon the identification of contextual changes.

Also, wrappers exist that group all services with the same functionality. Such a service proxy handles requests for a specific service and decides which service implementation is effectively used. Requests are handled at run-time wherefore information about available services is retrieved from the knowledge base. The runtime management of the knowledge base allows that services dynamically join or leave the system. It is the purpose of the Web Ontology Language (OWL)-based ontology to offer a vocabulary for the different kinds of information exchanged by components, as well as for their data format, and to describe the information provided by perceptual components. This enables the lookup of perceptual components based on the input and output data types at their interfaces. Similarly, situations and actions are modelled within the ontology. Upon the detection of a new situation, situation actions are performed, which may include the adaptation of component interactions to the new situation. By using OWL features like *inverse properties*, *transitive properties* or *nominals* [89], available components, which provide the requested properties, can be identified.

Even though the ontological lookup for matching components increases the flexibility of dynamic reconfiguration, only components with matching interface specifications can be combined. Proxy agents allow the selection of a concrete component implementation among the given alternatives. However, dynamic reconfiguration is restricted to situations defined by situation models, which does not allow to react to unforeseen events. Complex recursive OWL rules (e.g., transitive properties) complicate the worst-case analysis of the framework, and thus, prevent the usage of this approach for safety-relevant applications.

The application of semantic web service technologies for dynamic reconfiguration in factory automation is advocated in [69]. It targets the autonomous orchestration of manufacturing processes by reasoning about the classification and capabilities of these processes. The processes in manufacturing systems are usually composed and executed in predefined sequences, which define more complex processes at higher levels of abstraction. The system is modelled using the service-oriented architecture approach and web semantics to provide the knowledge about components in a format that can be interpreted by machines. This knowledge includes a specification of information exchanged, as well as the syntax and semantics of interactions. With machine-based reasoning also components are enabled to interact, which were not known to each other at design time.

For the purpose of knowledge representation of components and their relationships the OWL description logic (*OWL-DL*) is proposed. In description logics, at first the terminology for the application domain is defined, which consists of the relevant concepts in the domain. Afterwards, properties of concepts and the individuals, which are instantiations of concepts, are specified for the domain. Inference is based on the classification of concepts and individuals, which describes the hierarchical relationship between concepts (i.e., subconcept and superconcept) in the terminology. Furthermore, classification allows to infer information about properties of an individual by investigating whether the individual is the instantiation of a given concept. Figure 3.2 presents an example knowledge base depicting the terminology of concepts in the domain of field devices used for manufacturing, as well as the hierarchies between the concepts and their instances.

The semantics defined in the ontology are used to reason about production and conversation capabilities of components in the factory. Composition of components is based on the seman-

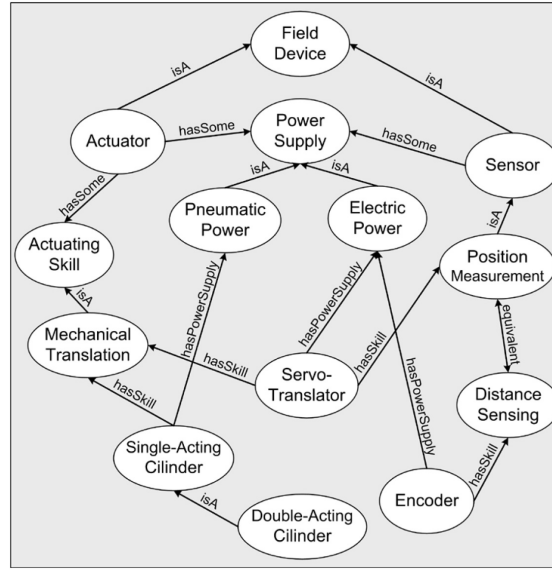


Figure 3.2: Example partial ontology describing the terminology for domain of field devices from [69].

tic match of capabilities required by some process and the available capabilities inferred from the knowledge base. The OWL-DL classification inference pattern is used for semantic match-making, which is performed by checking whether an available capability is an instance of the required capabilities.

While this semantic web service based approach is not designed for dynamic reconfiguration with real-time constraints, it already provides high flexibility in the composition of services. *Note* that manufacturing processes often include control loops with stringent temporal requirements, but the orchestration of such processes is usually not inherently sensitive to the progression of time. The framework might also infer solutions for compositions where unanticipated events occur. However, the complexity of reasoning with OWL-DL¹ (i.e., instance checking) is *NEXPTIME-complete*, which makes this approach infeasible if real-time behaviour of dynamic reconfiguration is required. Furthermore, it is not possible to combine several capabilities of components to obtain the requested capability. Therefore, in order to substitute a component, another redundant component with similar capabilities is required. Since some solutions might also include the connection of components with incompatible accuracy values of input and output items, additional information about achieved and required accuracy has to be added to their interface descriptions, which is checked for each solution.

Similar approaches for semantic matchmaking based on ontologies are also described in other publications, like [19, 76, 111, 112, 118].

¹http://www.w3.org/TR/owl2-profiles/#Computational_Properties (August 2015)

3.2.2 Fulfillment of Requirements in Related Work

Table 3.1 summarizes the capabilities of the described approaches to fulfill the requirements discussed in Section 3.1.

None of the evaluated approaches is capable to fulfill all defined requirements for dynamically reconfigurable CPSs. Implicit redundancy can only be exploited if it has been considered by the system designer at design time, but it cannot be identified automatically from a system model. Only approaches that are not based on predefined interconnections between components are able to react to unforeseen events by rearranging available services. However, none of these approaches is capable to automatically create transformation services, which are required if the interface specifications of available services do not match. While some approaches are dedicated to real-time systems, approaches related to business processes often provide more flexibility in the dynamic interconnection of components, but their temporal behaviour is infeasible for CPSs. Approaches that only allow the selection of predefined combinations of components inherently provide semantically correct configurations. For the other approaches, the semantic correctness depends on the correctness of the specifications in the ontology or the exactness of the interface specification (i.e., names given to itoms). Most approaches do not consider the accuracy of itoms exchanged between dynamically connected components. If components can only be connected according to a predefined schema, it has to be assumed that this issue is solved by the system designer in advance. For the other approaches, either quality type systems provide a coarse classification of the accuracy of itoms, or an exact specification has to be included in the interface specification. In this case, only components with suitable accuracy can be combined in a service composition.

Table 3.1: Fulfillment of requirements with different approaches.

Approach	R1	R2	R3	R4	R5
WS-Diamond	no	no	no runtime information given	yes, given by pre-defined application scenarios	no information available
Adapt.NET	no	no	yes	yes, given by pre-defined configurations	no information available
MARS	yes, if predefined by system designer	no	yes	yes, given by pre-defined configurations	partially, due to quality type system
FTT-based architecture	no	no	bound given, but infeasible for large CPSs	yes, given by pre-defined application graphs	no information available
Planning Graphs	no	partially, if matching services are available	only feasible for low number of services	yes, if parameter names are unique	no information available
CHIL	no	no	no	depends on correctness of ontology	no information available
Semantic Web Services	no	partially, if matching services are available	no	depends on correctness of ontology	possible, if accuracy is part of interface specification

Dynamic Reconfiguration Framework

This chapter presents the architecture and the model of reconfiguration for the dynamically reconfigurable system. The dynamic reconfiguration framework is intended for CPSs which comprise a set of components that provide services to other components in order to realize the intended system behaviour (e.g., control of a physical process). The presented framework is designated to facilitate the design and implementation of such CPS. On the one hand, the robustness of those applications can be improved by automatic reconfiguration of services in case of a failure of the component providing a service. On the other hand, if a service is started that requires an input which is not provided by any other service in the system, the framework can be used to create service compositions offering this required input.

At first an overview of the different building blocks contributing to dynamic reconfiguration are presented in Section 4.1. Afterwards, in Section 4.2 the proposed system reconfiguration process is described.

4.1 Building Blocks of Reconfigurable System

CPSs using the proposed framework typically consist of distributed computing nodes that are interconnected by some communication network. Nodes are self-contained computers that consist of hardware and software. A node incorporates one or several components, which provide their services to other components via the exchange of messages. The intended system behaviour is obtained by the combination of available services. Although no general constraints on the communication system are defined within this work, the application of a time-triggered network between the computing nodes can be advantageous (e.g., TTP/C [60] or TTEthernet [55]). When using time-triggered communication, messages are exchanged according to a communication schedule at predefined instants in time, which prevents any interference between messages of different senders and allows to temporally align the execution of components.

A system supporting dynamic reconfiguration comprises several building blocks that are required for reconfiguration and management of components. The schematic overview of these systems is presented in Figure 4.1.

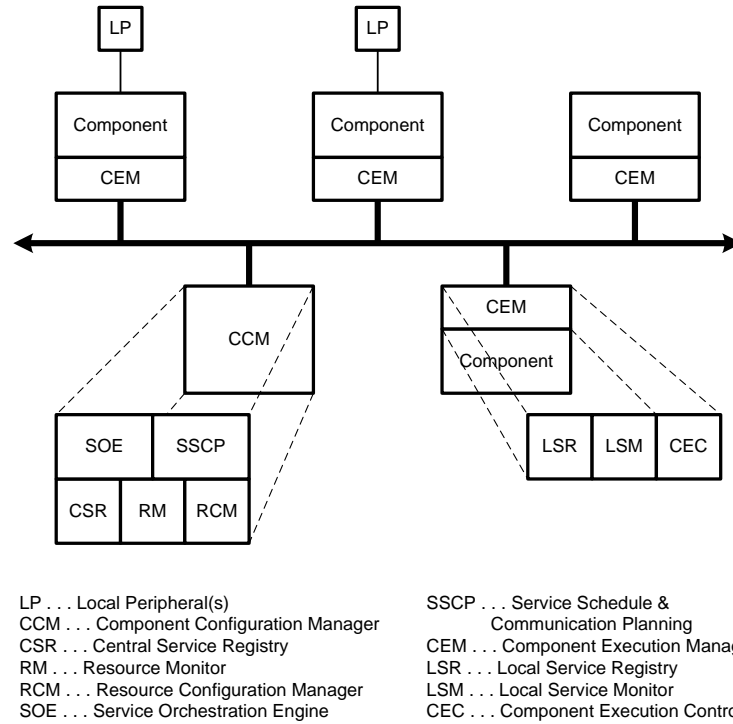


Figure 4.1: Schematic overview of system with reconfiguration capability.

The core building block is a dedicated node responsible for service composition and reconfiguration management. This node is called the Component Configuration Manager (CCM). Beside this node, normal computing nodes are used to provide services that are part of different real-time applications. These can be data processing services, or services interfacing with local peripherals (LPs) like sensors or actuators. In the latter case sensor readings are provided as input for other services or the output of another service is used to control the physical environment by accessing actuators. On each computing node a middleware software is deployed, that is responsible for component management on this node. This middleware is referred to as Component Execution Manager (CEM). Available services have to be registered at the CEM before the central CCM can use them for service composition. The CEM also has to control the provision of services on the local node by activating services that are requested but not yet active, or deactivating unused services in order to save computational resources. A further important task of the CEM is to monitor the provision of local services in order to detect service failures and initiate the reconfiguration of the system.

In the following paragraphs, a detailed description of the components of a system with reconfiguration capability is presented. Some of the components can further be divided into sub-components that enclose a dedicated functionality.

4.1.1 Component Execution Manager (CEM)

The CEM has to be instantiated at each computing node in the system – except for the node of the CCM. It is the middleware that coordinates the execution of components at an individual node. Each service a node is able to provide to other services has to be registered at the CEM in order to enable the Service Orchestration Engine (SOE) (see subsection below) to use this service for service orchestration. Services provided by a node have to be monitored in order to detect the loss of a service, and as a consequence, to initiate the reconfiguration process. Due to reconfiguration, new components have to be started on a node or stopped in case the service is not used anymore and computational resources are needed for other components.

The task of the CEM is divided into three subtasks implemented by three subcomponents within the CEM, which are described in the following paragraphs.

Local Service Registry (LSR)

The Local Service Registry (LSR) keeps track of the services that are available at a specific node and communicates changes in the service availability to the Central Service Registry (CSR) – that is part of the CCM. Each new service has to register at the LSR when the service becomes available. This is especially important for services that are bound to a specific node, which is the case when LPs are connected to that node. In contrast to services that interface with external devices (i.e., LPs), pure computational services can be migrated or replicated from one node to another node, such that computational or communication resources can be used more efficiently. In order to successfully register, a service has to provide a semantic service description of itself. This semantic service description links the input and output items of the service with defined concepts of the system ontology (see Section 5.1). Furthermore, all necessary services that are required by the registering service must be specified along with other resources needed (e.g., processing time or memory requirements). Only after all those required services and resources can be provided to that service, the component providing the new service can be started.

A service that is stopped deliberately, or a component that detects the fault of an external device or an internal error, deregister from the LSR. In case of a component failure that cannot be detected by the component itself, the Local Service Monitor (LSM) is responsible to stop the component and deregister the corresponding service. Hence, if the service has been used by another service, the CCM has to find a substitute for the lost service.

Upon registering or deregistering of services, the LSR has to inform the CSR about new services or services that became unavailable. The CSR then has to decide which actions need to be taken. In case a service has been deregistered that is used by other active services in the system, a reconfiguration request is generated in order to find a substitute for the missing service.

Local Service Monitor (LSM)

The task of the LSM is to monitor the provision of services. As components can experience a fault that cannot be noticed by the component itself, the LSM acts as an independent observer that might deactivate the component and trigger the substitution of its service by the CCM. Therefore the LSM can use different techniques to detect the failure of a component. For instance, a watchdog mechanism can be used to detect stuck components. Another possibility is,

to define input or output assertions for components such that a faulty behaviour can be recognized.

As the detection of faults is an extensively studied field not only in computer science (e.g., [50, p. 126ff], [56, pp. 152, 234ff], [41, 42, 48, 104, 115]), the detailed functionality of the LSM is not in the focus of this work. Within this work, the LSM is considered a generic building block which might employ different existing algorithms for the detection of failed services.

Component Execution Control (CEC)

Whether services on a computational node, that are ready for activation, will actually be activated is controlled by the Component Execution Control (CEC). After the central CCM found matching services that interact with each other, it can also define an appropriate schedule for each component in order to temporally align the service provision in the system. The CEC has to start components providing services that are needed by other services and to stop those components that are not required anymore. Furthermore, the CEC has to trigger the actual execution of a component. In case of time-triggered systems, components are executed according to a schedule that has to be defined by the CCM.

As the CCM might instruct that a service has to be migrated from one computational node to another node, the CEC locally controls the process of migration. This includes stopping the corresponding component at the original node, copying the code and state variables to the target node, and finally starting the component again at the target node. This is especially interesting for transfer services (see Section 6.3) that define how to compute the required input value for services based on the output of a group of other services that do not directly provide this value. These transfer services are automatically generated from the system model ontology and the list of available services in the CSR. When the CCM finds a valid service composition and creates the transfer service, it has to send the code of the transfer service to the CEC of a target node, which is then able to provide this service to other services in the system.

4.1.2 Component Configuration Manager (CCM)

The CCM is an integral part of the proposed architecture for reconfigurable systems, which coordinates the whole reconfiguration process. Whenever there is a change in the registered services (e.g., new services, service failure, changed service description), the CCM checks if this change affects other services that are active. If this is the case, a matching service composition has to be found to compensate for that change. Figure 4.2 depicts the replacement of a sensor service by a composition of other services. The dashed line between the *Transfer Service* and the *Application Service* shows the new flow of information. In case of a service substitution due to a failed service, the service composition has to be found within a bounded duration. Otherwise, when only services are affected that are ready but not yet used, the CCM also searches for matching services in order to keep the service registry up to date with ready services. As this is just a housekeeping task, it may be pre-empted by an urgent reconfiguration request caused by a service failure.

For the purpose of reconfiguration, the CCM comprises a centralized service registry, where the services of all computational nodes are represented. Furthermore, a resource monitor ob-

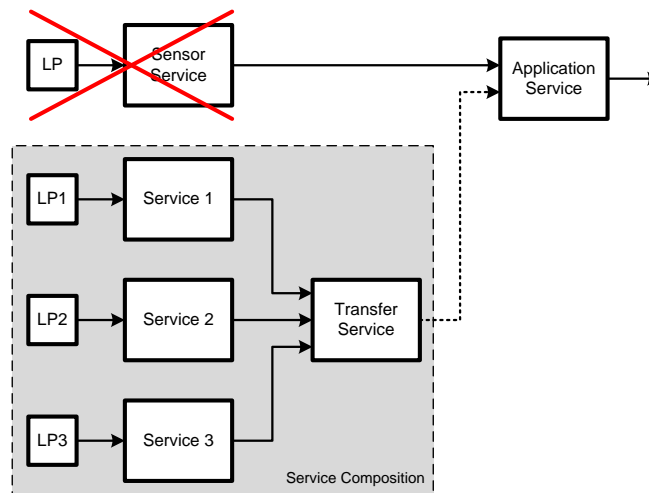


Figure 4.2: Using a service composition instead of a single sensor service.

serves the correct functionality of computational nodes and communication networks. The core part of the CCM is the SOE that uses the system model ontology and the CSR to find matching service compositions. After an appropriate service composition has been found, the corresponding components have to be scheduled such that no temporal requirement is violated. This involves the assignment of services to computational nodes, as well as scheduling of communication in systems that use time-triggered communication. Finally, the Resource Configuration Manager (RCM) has to transfer the code for transfer services – that implement the service composition (see Section 6.3) – to the target computational node, and instruct the CEC at that node with which period and phase a component has to be executed.

Conceptually, the CCM exists only once, which could make this building block a potential single point of failure. In order to prevent this risk, replication techniques (e.g., TMR) can be applied, where reconfiguration is based on voting among the decisions of the replicated CCMs.

To perform its tasks, an CCM is divided into five interacting components, where each of them is dedicated to a specific task of the CCM. In the following paragraphs, these components are described in detail.

Central Service Registry (CSR)

The CSR contains a copy of all service descriptions in the LSRs of the system. Thus, in the CSR all relevant information about available services in the system, which is needed by the SOE to find matching services, is stored. This information includes for each component:

- a semantic service description
- required input services
- temporal properties and accuracy information

- services that dependent on the service due to service orchestration
- the state of the component (i.e., ready, blocked, active, running, error)
- the computational node at which this service is registered
- for time-triggered systems a component schedule

The semantic service description is mainly an *service-to-ontology mapping (SOM)*, which relates the information produced and/or consumed by each service with the corresponding concepts in the system ontology, and thereby, models the semantics of the data exchanged by the service.

After a change in an LSR, this change is also communicated to the CSR. The CSR can then decide if this change also affects other services, and whether a reconfiguration process has to be triggered, or not.

Resource Monitor (RM)

It is the purpose of the Resource Monitor (RM) to observe the correct functioning of resources (i.e., computational nodes, communication infrastructures, etc.). In case of a computational node that experiences a fault, the LSR of that node might not be able to inform the CSR about the loss of services. Hence, an independent component checks whether there are failed nodes, in which case all services of the failed node are deregistered from the CSR and a reconfiguration is triggered. The same applies if there is a fault in parts of the communication infrastructure and more than one node is not accessible anymore. Additionally, the RM keeps track of the resource usage, which is needed by the Component Schedule & Communication Planning (CSCP) building block to find free computational and communication resources for components.

Simple strategies for detecting faulty components or parts of the communication network could be based on periodic life sign messages or rely on detection mechanisms provided by the communication system itself (e.g., the membership service of the Time-Triggered Protocol [60]). However, the concrete fault detection mechanisms within the RM building block are not in the focus of this work.

Resource Configuration Manager (RCM)

The RCM has to apply the new configuration by migrating program code (i.e., application or transfer service code) to the target node, and for time-triggered systems by sending new schedules for component execution and communication to the corresponding nodes. After the new configuration and schedules have been calculated by the SOE and the CSCP building block, the RCM creates the required transfer code from the system model ontology (see Section 6.3) and instantiates a component at the target node. The RCM also coordinates the reallocation of software components from one node to another node by migrating the program code and the component state, and starting and stopping the component execution at the respective computational nodes. In systems with time-triggered communication infrastructure, the last step is to reconfigure the schedule of the communication system. The same applies in case that due to the reconfiguration components have to be stopped and communication resources are freed.

When resource usage has been reconfigured, the RCM has to update the resource usage database of the RM, such that the system is always aware of free available resources.

Service Orchestration Engine (SOE)

The core building block of the proposed system with reconfiguration capability is the SOE. It uses a system model ontology (see Chapter 5) and the CSR to find a mapping of requested and provided services (i.e., component inputs and outputs) in order to combine those services to achieve the required service composition. For service orchestration, the SOE can rely on components that are already active or that do not require the services of other components (e.g., a component providing a sensor service), and hence, can be started immediately when they are needed. In contrast, a component that only reads inputs but does not provide any output for other components (e.g., a component providing an actuator service) always has to wait for other components that provide the required services. The system ontology is used by the SOE to identify semantic dependencies between the available services and services that depend on other services which are not yet available.

Therefore, at startup an ontology preprocessor within the SOE reads the ontology description and creates an optimized memory representation of the system ontology, which enables the service composition search within temporal bounds. After that the system ontology remains static over the runtime of the system – except if the structure of the system itself is changed.

Upon a reconfiguration request for a service, the SOE checks the semantic service description of the required services in the CSR and locates a knowledge concept within the system model ontology that corresponds to the semantic service description. For instance, if the service description requires the rotational speed of the left front wheel of a car, then an adequate concept in the system model ontology is found that represents this value. If in the CSR another service is directly linked to this concept, it is checked whether that service also fulfils the temporal and accuracy requirements. In the positive case it means that this service directly provides the required information, and the service is selected for orchestration. Otherwise, beginning from the concept in the system model ontology, the SOE walks through the ontology and tries to find a group of services that matches the service description of the required service, or from which the required information can be obtained with a defined accuracy. This can also mean that several other services have to be combined in order to derive the one service that is required, or that a transfer service has to be created in order to calculate the desired value (e.g., conversion from degrees into radian measure).

After all required input services have been found according to the service description of a service, the SOE hands over the service mapping information to the CSCP building block for temporal alignment of the orchestrated services, and it updates the CSR with actualized component states and dependencies.

Component Schedule & Communication Planning (CSCP)

As the temporal alignment of services [85] is an important measure to extend the duration of validity of sensor data and calculations, a dedicated building block exists in time-triggered systems with the purpose to align the execution of components and the information exchange between

the corresponding services. The building block responsible for this task is the CSCP. This building block has to solve scheduling problems in order to fulfil the temporal constraints of control applications (e.g., time between measurement by sensors and the output of control commands to actuators). In order to achieve this goal, the CSCP uses the service composition graph from the SOE with the timing constraints given in the service descriptions in the CSR. Additionally, it has to find a target node for each service which is not bound to a computational node. Afterwards, it has to define an appropriate period and phase for each component, such that temporal requirements can be fulfilled. Finally, the communication schedule has to be calculated by the CSCP. During scheduling it has to be taken care that new schedules do not unnecessarily affect the provision of already available services.

The final schedules are passed to the RCM which has to accomplish the reconfiguration by sending the necessary information to each affected computational node and the communication infrastructure. Information on resource allocation and component schedules is also stored in the CSR and forwarded to the RM in order to monitor the temporal provision of services and the resource utilization.

Since the topic of task and communication scheduling has already been studied extensively (e.g., [14,20,63,95,106]), it is not in the focus of this work to propose actual scheduling methodologies. It is rather assumed that such algorithms can be chosen from existing literature.

4.1.3 Components & Local Peripherals (LP)

Application services are provided by components on computational nodes. On each node, as many components can be executed as allowed by the amount of computational resources. This implies that there does not have to be a one-to-one mapping between services and computational nodes. How the services are actually implemented on a node is not defined in this work. But it is essential for the correct functioning of the system that a well defined message-based service interface exists (i.e., a RUMI), that components have a periodic behaviour after they have been initialized, and that their provision can be controlled by the local CEC. For instance, a node can run a standard operating system where each service is provided by an individual process which is blocked until the CEC triggers the execution of that process.

Components can be started by different means, but basically a component is either started when the system starts up (e.g., services bound to a dedicated node, like sensor services or actuator services), or due to a request from the CEC, when the service is migrated from one node to another one or in case of a transfer service that implements the composition of services. At first, a component has to be initialized in order to obtain and configure all necessary local resources and to register its service at the LSR. After the initialization, the following component states can be distinguished:

- *Ready*: The component has been initialized and all necessary resources and required services are available, but the component is not yet scheduled for periodic execution.
- *Blocked*: The component has been successfully initialized but at least one resource or a required service is not available.

- *Active*: The component is initialized and scheduled for periodic execution, but it is not yet running.
- *Running*: The node's CPU is actually assigned to the component which is thus now executed.
- *Error*: An error has been detected which prevents the component from providing its service.

In the proposed system architecture, LPs are represented by services as well, which implement the interaction at the RUPI. These services are bound to the computational node to which the LP is connected. At system start-up the corresponding components are immediately started in order to provide input data or to control actuators.

4.2 Reconfiguration Process and Component Interaction

In the proposed system with reconfiguration capability, after start-up each component providing a service is located on a dedicated node. This is also true for components that interact with LPs, like sensors or actuators, on a node. Typically, the last active configuration and assignment of components to nodes will be used.

The reconfiguration process is then triggered by different events that cause a change in the databases. The most important one is a change in the service registries (i.e., LSRs and the CSR), which means that either services are not available anymore (e.g., due to a fault), new components have been installed, or the service description for a service changes – i.e., entries in the SOM change (see section 5.2). In case some resources experience a fault (e.g., a computational node or parts of the communication network) several services might be affected, and thus, become unavailable due to this incident. A third possibility for triggering the reconfiguration process is a change in the system model ontology, which is considered to be static during normal operation of the system. The commonality of all these cases is, that there exist service dependencies in the system which are not satisfied (i.e., some of the required input services cannot be obtained from existing services). Hence, the reconfiguration process has to be triggered in order to recover the overall service of the system.

Figure 4.3 depicts the information exchange between components in a system with reconfiguration capability. Each LSR updates the CSR when there are changes w.r.t. the locally available services. The RM additionally observes the availability of resources (i.e., computational nodes and communication resources) and updates the resource repository with actual availability information. In case of failed resources, the affected services in the CSR are marked as erroneous and the reconfiguration process is triggered.

The service orchestration algorithm of the SOE uses the following three databases to resolve service dependencies:

- *System Model Ontology*: contains a detailed semantic description of the domain and the system with its environment that is controlled by the control application. In the system model ontology the interactions between individual parts and properties of the system

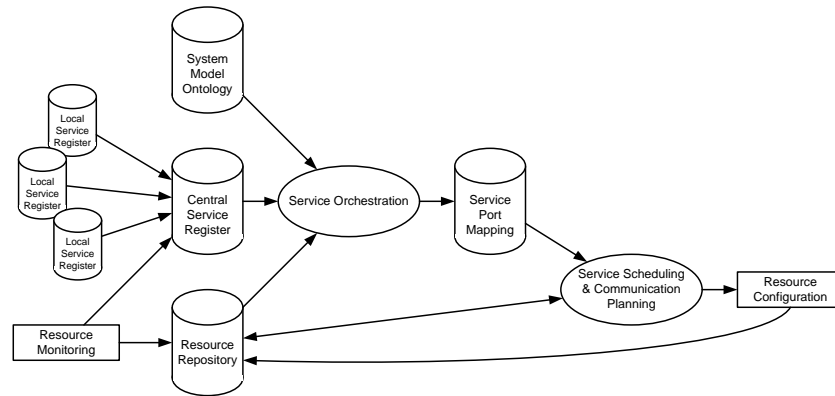


Figure 4.3: Information exchange between components.

under control are defined, which are used by the SOE to find a service composition, and if required, to automatically create a transfer service.

- *Central Service Register*: comprises service descriptions for all services that are available in the system.
- *Resource Repository*: contains information about the resource usage and availability of resources (i.e., computational nodes and communication resources) in the system.

The central service orchestration algorithm starts upon a reconfiguration request that either tells the SOE to find a substitute for a failed service, or to find a new service composition for some – possibly new introduced – service. The SOE uses the service description of the service for which some input is missing to identify services that match this service description. Service descriptions of two services match, if for both services a relation to the same ontology concepts is defined in the SOM. As it cannot be expected that the service description of the required service exactly matches an existing service, the SOE uses the system model ontology to find a composition of services that provides the required service. This is done by identifying the concepts in the system model ontology that match the description of the desired service (e.g., if the rotational speed value of the left front wheel of a car is required, a concept in the system model ontology is found that represents this value). Beginning from this concept, a composition search algorithm (see Section 6.2) is applied such that other concepts are reached which are linked with existing services. If this group of services fulfils the temporal and quality requirements (cf. uncertainty of service composition in Chapter 7) set by the service that uses this group as input, a valid service composition is found. Whenever the service composition does not directly provide the required values – i.e., if the composition consists of more than one service or a value transformation has to be applied (e.g., transformation of degrees into radian measures) –, a transfer service is automatically created to perform those transformations. This process has to be repeated until all failed services have been substituted or all necessary input services for some service have been found.

After all required services have been found and the transfer services have been created, the SOE outputs a port mapping of inputs and outputs (i.e., input and output messages) of services, as well as the code for the transfer services.

The CSCP component uses the output of the SOE and tries to find a schedule for those services that are not yet active. At first, for each component which is not yet initialized an appropriate node must be found by using the resource repository. Then the scheduler has to compute an appropriate period and phase for the execution of each of the components. With the output of the CSCP component the resource configuration is finalized by communicating the changes in the schedule to the CEMs of all affected nodes.

Modeling Cyber Physical Systems (CPSs)

In order to enable the SOE to identify implicit redundancy in the system and to automatically find an appropriate set of services that allows to infer a missing service, knowledge about the semantic structure, properties, their interrelations, etc. of the CPS is essential. This knowledge has to be modeled such that it can be processed by algorithms. Using a system ontology, the expert knowledge about the structure, relations and interactions between components and properties of the system that is controlled by the cyber part of the CPS is formally described. In case dynamic reconfiguration of service interactions is required (e.g., due to a fault within one service), the SOE is triggered. It then starts to traverse the system ontology in combination with the CSR and to search for groups of services that are semantically equivalent to the currently unavailable service.

The system ontology mainly comprises the static knowledge about the CPS, which normally does not change during the operation of the system. Dynamic changes in the system, caused by failure, removal or addition of services, are modeled by updating the mapping of services to the ontology (see Section 5.2).

The following sections explain the building blocks and the construction of the system ontology. Examples from the automotive domain are provided to illustrate how the ontology is set up.

5.1 Building Blocks of Ontology

System ontologies are commonly composed of entities that are classified into the following groups of basic building blocks: *instances*, *concepts* or *relations*. Instances represent individual attributed entities in the ontology. For example the *left front wheel* of a car can be an instance in an automotive ontology. As many instances share similar properties or structures, these instances are assigned to classes of entities that share the same properties and structures. These classes are

called concepts. For instance, all members of the concept *wheel* feature the property *diameter*. An instance – possibly including further properties – can be a concept itself, which can then be further instantiated. This way an hierarchical order of concepts is described. Relations are used to define the relationship between concepts, instances, or concepts and instances. With different types of relations between concepts hierarchical and functional dependencies of concepts can be expressed.

The proposed ontology description framework restricts the design space compared to other existing description languages for knowledge representation (e.g., $\mathcal{SHOIN}^+(\mathcal{D})$ [118] or DAML [90]) in order to reduce the complexity of reasoning algorithms. Only concepts and relations are defined, that can be effectively used to technically describe the interactions in a CPS. For instance, no relations implementing logical negation, disjunction or qualified number restrictions are supported. By using a restricted set of allowed concepts and relations, a predictable worst-case runtime of service orchestration is enabled, which is required under real-time constraints (i.e., finding a service substitution upon the failure of a component within a bounded time). Details about the defined basic building blocks and their usage are provided in the next subsections.

Furthermore, the dynamic reconfiguration framework described in this work does not actively differentiate between concepts and instances of concepts – each instance itself is treated as a concept. Hence, in the rest of the document concepts and instances are used synonymously.

5.1.1 Concepts

In general, three types of concepts are distinguished: *structure concepts*, *property concepts* and *transfer concepts*. The two additional concepts *property type concept* and *transfer type concept* have been introduced to define meta models for property and transfer concepts. During ontology preprocessing (see Section 5.3) these meta models are automatically instantiated where applicable. In the following paragraphs the purpose of the distinct concept types will be defined. Furthermore, a graphical representation for these concepts, which will be used in this work, is presented.

Structure Concepts

Structure concepts describe the structural decomposition of a CPS into different parts with defined properties, connections and interdependencies. For example, the *engine*, *gear boxes* and *wheels* are physically connected parts of the power train of a car. Using relations (see Section 5.1.2), these interdependencies and connections between the structural parts can be modeled. Thereby, structural concepts can be organized hierarchically, where one concept is the instance of another concept. The system ontology requires partial ordering of structure concepts, as it ensures termination of ontology processing algorithms, and circular membership is not a reasonable model for real CPSs. On the other hand, total ordering is not demanded, because a structure concept can also be the instantiation of multiple other structure concepts. Instances of structure concepts automatically inherit properties of higher hierarchy levels. The kind of property inheritance between concepts and their instances depends on the different types of relations. Figure 5.1 shows the graphical representation of a structure concept using the *wheel* of a car as an example.

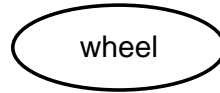


Figure 5.1: Graphical representation of structure concept *wheel*.

Property Concepts

Property concepts (e.g., *speed*, *acceleration* or *curve radius*) denote measurable or quantifiable static or dynamic states (i.e., properties) in the system. An example for a static property is the distance between the wheels, since its value does not change during the lifetime of a car. In opposition to that, the speed of the car constantly changes over time, and thus, it is a dynamic property. The value range of properties in the system can either be:

- *continuous* (e.g., the speed of the car),
- *discrete* (e.g., counters, like the milage counter at the dash board of a car), or
- an *enumeration* of possible values (e.g., the clutch is opened or closed, or color values).

The difference between discrete properties and enumerations is, that discrete properties are quantifiable, which allows mathematical operations and ordering of values. The possible values of discrete properties are not defined explicitly, but as a range between the minimum and the maximum value. In contrast, all possible values of enumerations have to be listed and no ordering of values is given. Therefore, processing of these values usually requires an explicit distinction of cases.

Property concepts represent those states in the system that a service relies upon (i.e., their value is an input to the component providing the service). Hence, the input and output parameters of a service directly map to property concepts in the system ontology. In the CSR a dedicated table specifies this mapping between services and property concepts, which dynamically changes as services are added to or removed from the system (see service-to-ontology mapping in Section 5.2). If a service requires a property concept as an input, that is not yet available (e.g., because the service providing this property experienced a fault), another property concept – or a group of property concepts – has to be found that allows deducting the required property concept. Figure 5.2 presents the graphical representation of a property concept using the *dynamic radius* of a wheel as an example.



Figure 5.2: Graphical representation of property concept *dynamic radius*.

Regarding the availability of the value of property concepts in the system, the following distinctions can be made. Property concepts are either:

- *constant* values in the system (e.g., distance between wheels),

- *provided* by sensors or other services (e.g., angular speed of wheels), or
- *unavailable* (e.g., curve radius, if it is not provided or estimated by some service).

Unavailable property concepts are those which cannot be directly obtained from a memory (like constants) or from a communication interface (i.e., not provided by services). Even though not all property concepts in the ontology represent states that are directly measurable by some sensor, in the optimum case also all unavailable property concepts can be calculated from the knowledge available in the system ontology.

As the value of a property concept can be provided by different services simultaneously and each service might use a distinct method to determine that value, the accuracy of the provided value strongly depends on the actual service that provides the value. Hence, not only one single accuracy value for a property concept might exist, and therefore, the accuracy of the provided value is not part of the property concepts in the ontology. Information about the accuracy of property concepts has to be given for each providing service individually in the service-to-ontology mapping.

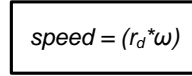
Transfer Concepts

Transfer concepts provide a formal definition of the relationship between properties in the system by describing the transformation from input property concepts to output property concepts. The transfer behaviour (i.e., the dependency) between properties is an intrinsic characteristic of the CPS. It constitutes the implicit (i.e., stigmergic [59]) information flow between two or more properties that are correlated. This correlation can, for instance, be a mechanical, electrical, chemical, physical, mathematical or data relationship between those properties (cf. Section 2.2). In contrast to relations (cf. Section 5.1.2), which are binary functions between concepts, each transfer concept requires at least one property concept as input and at least one distinct property concept as output. Thus, transfer concepts can also connect more than two property concepts.

As part of the transfer behaviour, transfer concepts contain a transfer delay time that measures the duration needed until the transfer behaviour model output reaches 90% of its stable value after a step-change at the transfer behaviour model input [56, p. 7]. This allows to model delays between the application of new input values to the system and the observation of a reaction to that change. On one hand, information about delays can be used by the search algorithms to validate the feasibility of a found solution by checking that the sum of delays is lower than the maximum tolerable delay. On the other hand, state estimation algorithms can then use this information to optimize their estimates.

Furthermore, each transfer concept is assigned sensitivity indices that define the influence of the transfer behaviour on the uncertainty of the output value. These sensitivity indices relate the uncertainty of each input to the transfer behaviour with the output uncertainty. An additional process uncertainty value included in the transfer concept provides an estimation of the error term of the transfer behaviour model which originates from an imperfect model (e.g., terms with negligible influence in an equation are omitted to reduce computational effort). Both, sensitivity indices and process uncertainty are used by the search algorithm to estimate the overall uncertainty of a solution. For detailed information about handling uncertainties, see Chapter 7.

In the rest of this work, the transfer behaviour of a transfer concept will also be referred to as transfer function. Conceptually, the implementation of the transfer functions is not restricted, as long as it can be interpreted by the specific SOE. Hence, various implementation choices of the transfer function of transfer concepts are possible, like simple 1-to-1 mappings of similar properties, difference equations, complex mathematical functions, finite-state machines, look-up tables, or transformation codes using a programming language. Figure 5.3 shows the graphical representation of a transfer concept with the transfer function $speed = (r_d * \omega)$, that relates the *speed* of a vehicle with the dynamic radius r_d of its wheels and the angular speed ω .

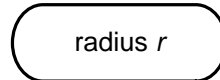


$$speed = (r_d * \omega)$$

Figure 5.3: Graphical representation of transfer concept with transfer function $speed = (r_d * \omega)$.

Property Type Concepts

These concepts specify the type of a property concept (e.g., acceleration in m/s^2), but not every property concept requires a property type. Property type concepts can be used to define classes of property concepts which define common relationships to other property types, like a formulary in mathematics or physics. For instance, from the radius of a circular entity its circumference or its circular area can always be calculated, independent of the nature of the entity (e.g., the radius of a wheel or the turning radius of a car). By assigning property types to property concepts, a meta-level of properties and their relations is created that automatically extends the search space for the composition search algorithm. Section 5.3.1 explains how property types extend the search space. The graphical representation of property type concepts is depicted in Figure 5.4 using the example *radius*.

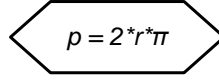


$$radius\ r$$

Figure 5.4: Graphical representation of property type concept *radius*.

Transfer Type Concepts

Transfer type concepts are used similarly as property type concepts to model meta relations defined between property types. If a transfer type is modeled between two or more property types, and all of these property types are instantiated by property concepts that are connected to the same structure concept, then also the transfer type can automatically be instantiated between these property concepts. Details about the automatic instantiation of transfer types are described in Section 5.3.1. Figure 5.5 presents the graphical representation of transfer type concepts with the transfer function $p = (2 * r * \pi)$ as an example. Here, the formula for calculating the perimeter p based on the radius r is modeled.



$$p = 2 * r * \pi$$

Figure 5.5: Graphical representation of transfer type concept with transfer function $p = 2 * r * \pi$.

5.1.2 Relations

Relations are binary connections between two concepts (i.e., structure, property, transfer, property type or transfer type concepts). For the construction of the ontology, different types of relations with different semantics are distinguished. The type of relation between two concepts has a significant impact on the inheritance of properties between structure concepts and on the search algorithm. Hence, only a restricted set of well-defined relations is allowed to construct the system ontology. More elaborate and complex relations between concepts (e.g., relationships between more than two concepts) have to be modeled using transfer concepts (cf. Section 5.1.1). In the graphical representation of a system ontology, relations are depicted as arrows which are annotated with the relation type. The following paragraphs describe the different types of relations allowed in the system ontology.

Input Relation

This type of relation is used between property concepts and transfer concepts, as well as between property type and transfer type concepts. It specifies that the property concept acts as input parameter for the transfer function of the transfer concept. In the graphical representation of this relation (see Figure 5.6), the arrow indicates the flow of information, and hence, it points towards the transfer concept. The search algorithm uses this relation in the opposite direction – i.e., coming from the transfer concept towards the property concepts. Thereby it has to ensure, that all input relations heading to a transfer concept can be provided directly or indirectly (i.e., by using other transfer functions) by some service. Otherwise, the path containing that transfer concept cannot be used to find an appropriate aggregation of services.

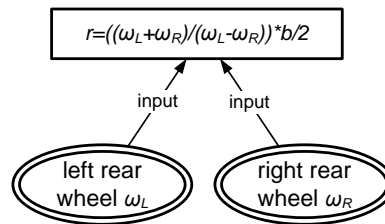


Figure 5.6: Example of two property concepts acting as *input* for a transfer function.

Output Relation

Similarly to *input* relations, an *output* relation defines the information flow between property and transfer concepts, as well as between property type and transfer type concepts. This relation specifies that the transfer function returns the value of the connected property concept. The arrow in the graphical representation of *output* relations (see Figure 5.7) points to the property concept. During the search for service compositions, this relation is used in the opposite direction to find transfer functions that output the property concept of interest.

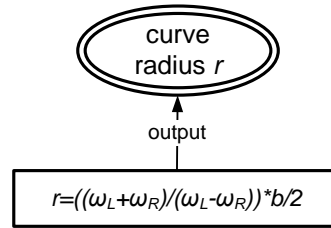


Figure 5.7: Example of property concept acting as *output* for a transfer function.

Has-Property Relation

The *has-property* relation assigns different property concepts to a structure concept. In the graphical representation of this relation (see Figure 5.8), the arrow points to the property that is connected to the structure concept. When the structure concept is instantiated, also its properties are inherited according to the type of the instantiation (i.e., *is-part-of* or *isA* relations, as defined below). In a hierarchically structured system, the *has-property* relation for properties that are common to several structure concepts at different levels should always be added to the structure concept at the highest hierarchy level for which the property is valid. This way, all instances inherit the same property instead of creating multiple distinct versions of properties which are semantically the same.

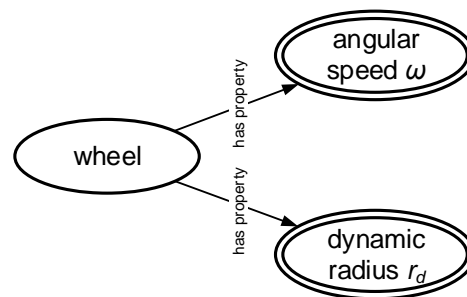


Figure 5.8: Example of two *has-property* relations.

***Is-Part-Of* Relation**

The *is-part-of* relation – or *partOf* relation – is an important membership relation to model the architectural structure of a system as it defines a hierarchy between structure concepts, as well as the inheritance of properties between these concepts. The relation states that one structure concept (i.e., the *child concept*) is a subsystem of another structure concept (i.e., the *parent concept*). This implies, that the properties of the higher level concept – which are defined by the *has-property* relation – are identically valid for the lower level concept. For instance, if the car has a certain amount of lateral acceleration, then also all of its parts (e.g., the wheels, the engine) have the same lateral acceleration. As a consequence, all parts of a structure concept have a 'virtual' *has-property* relation to the same property concept. In the graphical representation of this relation (see Figure 5.9), the arrow points to the higher level concept.

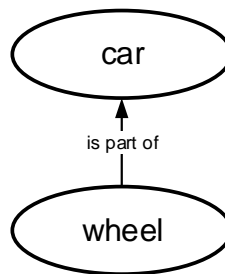
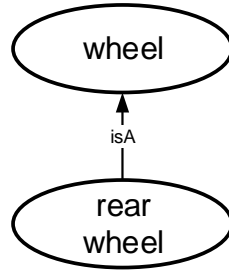


Figure 5.9: Example of *is-part-of* relation.

The *partOf* relation defines a partial order between structure concepts, which results in a tree structure in the ontology graph. Hence, each structure concept must not specify more than one concept as its parent.

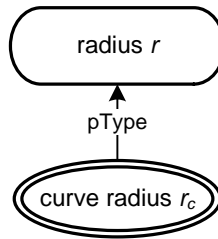
***IsA* Relation**

Similarly to the *is-part-of* relation, the *isA* relation allows to hierarchically structure the system ontology. It is a membership relation with the semantics of a generalization (e.g., *wheel* is the generalization of the *left rear wheel* of a car). The higher level concept is also referred to as *parent concept*, and the lower level concept as the *child concept*. In the graphical representation (see Figure 5.10), the arrow points to the higher level structure concept. With respect to this relation, the higher level concept defines a set of properties and subsystem relations (i.e., *is-part-of* relations) that are inherited by lower level structure concepts. But in contrast to the *is-part-of* relation, this relation clones the properties and subsystem relations of higher level concepts. This means, that for each instantiation the cloned property can have a different value, while in case of *is-part-of* the inherited property has the same value for all instances. Also the *isA* relation defines a partial order between structure concepts. But unlike *is-part-of* relations, each structure concept may specify more than one concept as its parent.

Figure 5.10: Example of *isA* relation.

***pType* Relation**

These relations are needed if a property type should be defined for a property concept. Like *isA* relations, the *pType* relation is a membership relation with the semantics of a generalization (e.g., *radius* is a generalization of the *curve radius* of a vehicle). An example for the graphical representation of this relation is presented in Figure 5.11. In case the property type concept related by the *pType* relation is also linked with a transfer type concept, the transfer function of that transfer type concept is inherited by the property concepts that are connected with the *pType* relation (see inheritance rules in Section 5.3.1).

Figure 5.11: Example of *pType* relation.

5.2 Service-to-Ontology Mapping

Normally, during the operation of the CPS, the system ontology is static. Changes in the ontology are only made during maintenance, if the structure of the system changes. Examples are the addition of new hydraulic lines to a machine or a thermal protection shield that is mounted, which prevents the loss of temperature, and thus, the original transfer function of a pipe changes. But in order to be able to find appropriate service compositions, the CCM requires knowledge about property concepts that are actually provided. This depends on the currently accessible services, which might dynamically change during the operation of the system. Such dynamic changes appear due to intentionally starting and stopping of services, or due to faults in pro-

cessing components or communication links. To model this dynamic part of the system the available information has to be mapped to the static knowledge in the system ontology, i.e., with the property concepts. Similarly, the information that is required as input by services needs to be matched with property concepts. For this purpose a *service-to-ontology mapping (SOM)* register is introduced.

In the SOM, individual input and output data items at a service interface – also referred to as information atoms, or *itoms* [57] – are matched with the corresponding property concepts in the system ontology. As services can require several input itoms and they are able to produce distinct output itoms, a service can have multiple entries in the SOM. Likewise, a property concept can be provided or required by more than one service, and thus, multiple relations to the same property concept may exist. For instance, when diverse sensors measure the value of the same physical entity. This relation of service itoms with the system ontology provides an explanation for the meaning of the itom, and hence, implements a semantic service description. Figure 5.12 depicts the mapping of property concepts of the system ontology to the itoms of the service interfaces.

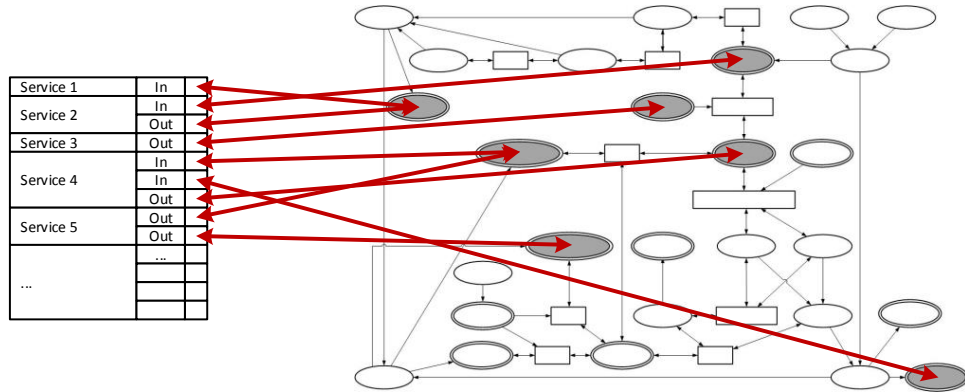


Figure 5.12: Mapping of property concepts in an exemplary system ontology to itoms at service interface.

The implementation of the SOM has to enable fast matching of services and property concepts in both directions. Upon a change in the SOM, affected information flows have to be identified by investigation of the property concepts that are currently used by other services, and hence, these services might have to stop or the quality of information provided to them could be affected. Additionally, a change in the SOM can also enable the execution of a service that was previously blocked by missing itoms. During service composition search, for each property concept within a search tree it is checked whether an entry in the SOM exists, which indicates the provision of that property concept by a service (cf. Chapter 6). Efficient mapping of property concepts to services (with constant access time) allows to reduce the average time for composition search and to find tight bounds for the worst-case execution time of the composition search algorithm.

5.3 Ontology Preprocessing

The system model ontology is specified using a declarative description language – for the implementation of this work RDF was selected (see Section 8.1). As the form of such a description is not very efficient for dynamic reconfiguration, especially if guarantees on the temporal bounds of reconfiguration are requested, the ontology description files have to be preprocessed when the system is started or after the ontology has been changed. Thereby, the *ontology graph* is created, which is a directed graph representation of the system ontology in the memory of the CCM. This task is performed by the ontology preprocessor, which has to resolve different types of inheritances between concepts and optimize the ontology such that parts of the ontology that are not relevant during service composition search are removed.

5.3.1 Expansion of Inherited Concepts and Relations

Inheritance and instantiation of concepts facilitate the design process of the system ontology by minimizing the number of concepts that have to be expressed manually, which contributes to avoid design faults during this process. Within this work, inheritance concerns the automatic expansion of relations defined in the system model ontology during the creation of the expanded ontology graph. Thereby, certain types of relations trigger the creation of multiple vertices and edges in the ontology graph, which represent instances of concepts and simpler relations between them. The expansion process is transitive, which means that inheritance of concepts is further propagated to lower ontology levels. In order to inherit concepts, the ontology preprocessor is responsible to copy the concepts from the higher level to lower levels or to create relations between them.

Expansion of *pType* Relations

pType instantiation allows to automatically create standard transfer functions between property concepts that belong to the same structure concept. Due to *pType* instantiation, the ontology preprocessor inherits those transfer type concepts, which are defined between property type concepts, to normal property concepts. The following conditions have to hold in order to enable this type of inheritance.

- Property concepts that inherit transfer functions need to be defined as instantiations of a property type.
- A transfer type concept exists, which is only related with property type concepts that are further instantiated by property concepts belonging to the same structure concept.

For *pType* instantiation, the transfer function of the transfer type concept is copied by the ontology preprocessor to the new transfer concept. Such situation is illustrated in Figure 5.13.

The second condition demands that transfer concepts might only be created between property concepts belonging to the same structure concept. This is important, as otherwise transfer concepts could be created erroneously, that transform the properties of structure concepts which do not have a valid correlation (e.g., transfer the curve radius of a car to the perimeter of the

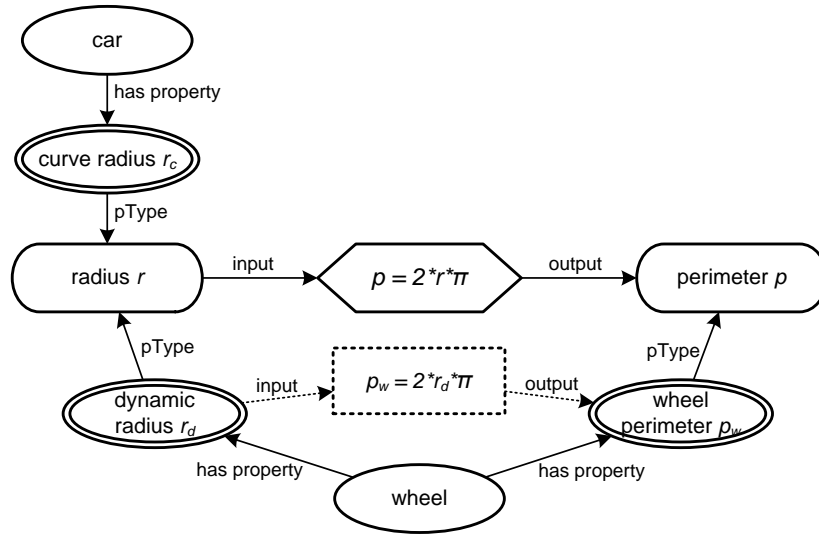


Figure 5.13: Example of *pType* instantiation; a transfer concept is derived from a transfer type concept.

wheels). In the figure, the transfer concept and the input and output relation between the dynamic radius and the perimeter of the wheel – shown in dashed lines – are created, but there exists no relation between the curve radius and the wheel’s perimeter, even if both radiuses share the same property type concept.

***partOf* Inheritance**

Whenever one structure concept (i.e., the lower level concept) is part of another structure concept (i.e., the higher level concept), all properties of the higher level concept are also valid for the lower level concept. For instance, if a car is driving with a certain speed, also its parts (e.g., the chassis and the seats of the car) are moving with the same speed. Due to this type of inheritance, care has to be taken when defining the ontology. Properties that are rather valid for the parts of a system component than for the whole component have to be added to the lowest possible level in the hierarchy. For instance, if the property *color* of a car is added to the top level structure concept *car*, then all parts of the car (e.g., the tyres) are assumed to have the same color – which is not true. In most cases the color of the car is defined by the color of the chassis. In Figure 5.14 the creation of *virtual* has-property relations is depicted.

This type of inheritance is especially useful, if the ontology is created from different independently described parts. A component supplier (e.g., the manufacturer of seats) can model its parts without requiring any knowledge about the later use of its component. For instance, it can be referred to the *speed of the seat* instead of directly relating to the *speed of the car*. Thus, the ontology description need not be changed if the seat is installed in a harvester or a truck. Due to the *partOf* inheritance, these properties are automatically linked. This link can be created explicitly by the ontology preprocessor, or implicitly during service composition search by following up the hierarchies in the system model ontology.

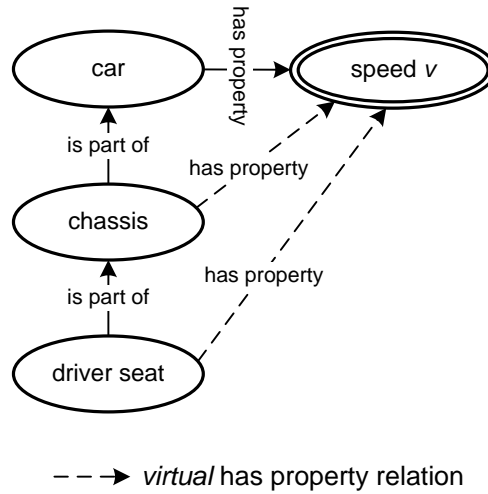


Figure 5.14: *partOf* inheritance causes *virtual* has-property relations.

isA Instantiation

In case an *isA* relation exists between a higher level and a lower level structure concept (i.e., the lower level concept *isA* higher level concept), the property concepts and also *partOf*-subgraphs of the higher level concept have to be instantiated for the lower level concept. This means that new property concepts and complete subgraphs are created for the lower level concept. An example of such an *isA* instantiation is illustrated in Figure 5.15.

In contrast to *partOf* inheritance, where the same property concept is used with different structure concepts, and hence, the same physical value applies for all these structure concepts, here different property concepts are used for distinct structure concepts. Therefore each property can have a different value. For instance, the property *angular speed* can be defined for the concept *wheel*. And as a car has four wheels, four child concepts are derived from the structure concept *wheel*. When the car is driving in a curve or some wheels are slipping on the ground, the *angular speed* value of each of the four wheels might be different.

Since with the *isA* relations more than one parent concepts of a structure concept can be defined, it may happen that the same concepts appear at different parents. Then these concepts can only be inherited from one of those parents. Such a case is illustrated in Figure 5.16.

5.3.2 Ontology Optimization

The second major task of the ontology preprocessor is the optimization of the memory representation of the system ontology. Different objectives of optimization can be pursued, like minimization of memory consumption, simplification of the composition search algorithm, or reduction of the number of concepts and relations examined during the search of service compositions, in order to minimize the maximum search time. Some of these objectives are in conflict with other objectives, such that a tradeoff between them has to be found. For instance, the reduc-

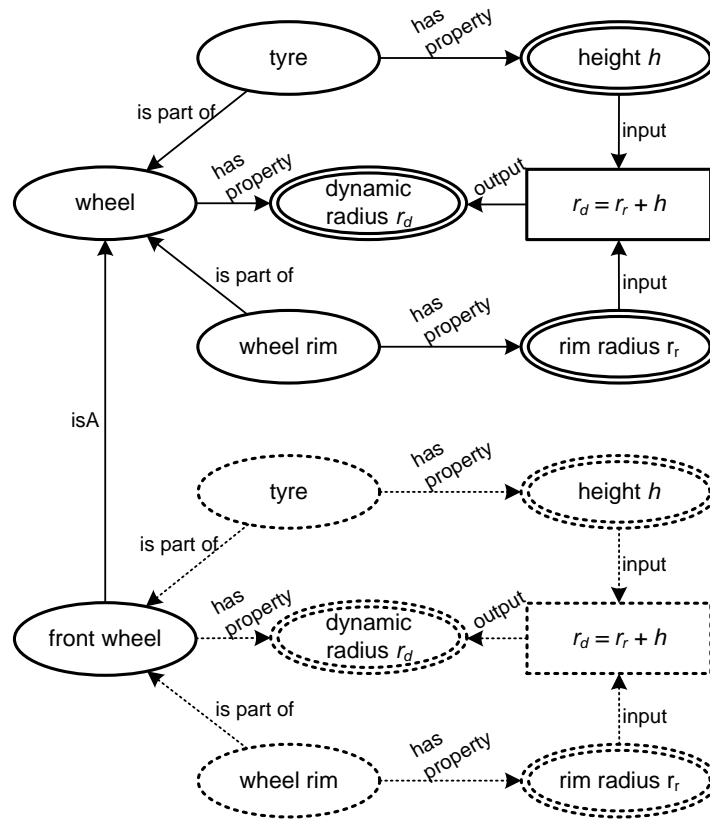


Figure 5.15: Example of isA instantiation; a complete subgraph is cloned.

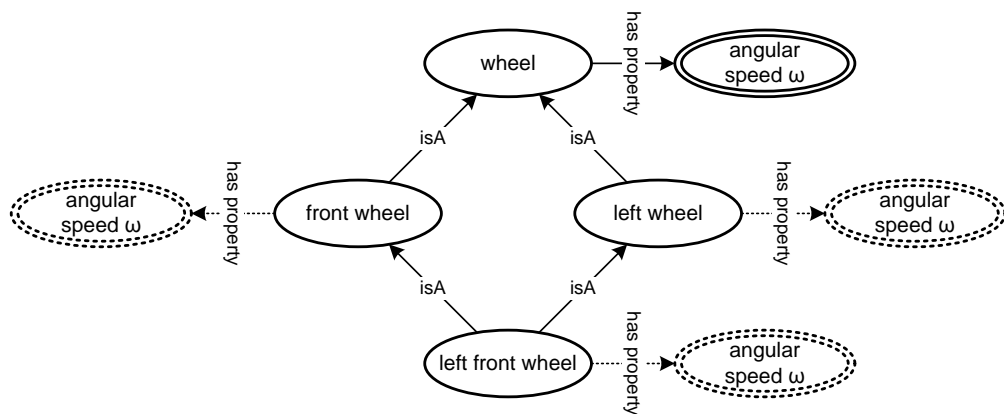


Figure 5.16: Example of inheritance of the same property concept from different parents.

tion of memory requirements will increase the algorithmic complexity of the search algorithm. Various techniques for ontology optimization can be applied, including the following:

- **Ontology matching:** Large CPSs are often composed of components originating from different manufacturers who are responsible for developing an ontology that is describing these components. Unless a predefined vocabulary for the concept names of the distinct ontologies exists – which is likely the case when standard components are applied – the integration of these ontologies can be done manually by a system engineer, or automated using ontology matching techniques.

The automated matching of ontologies from different sources is a research topic on its own, for which many publications exist (e.g., [71, 103]). In [29] the matching process is described as the determination of an alignment between two ontologies. Thereby, correspondences between the concepts of the ontologies have to be identified, such that the relation between these concepts can be specified. In the best case, there is an equivalence relation between two concepts (c_r and c_k) and only one of the concepts (i.e., c_k) has to be kept in the ontology. All relations that connect the removed concept c_r with other concepts in the ontology can be added to concept c_k . Otherwise, if the two concepts are not equivalent, both of them have to be kept and the relation between them becomes part of the ontology graph.

Simple techniques for the detection of similarity can be based on matching of linguistic components of the concept names, as described in [71]. While for the purpose of ontologies for dynamic reconfiguration this method is useful in case of perfectly matching names, concept names with small deviations cannot be related without further investigation. Structural matching of concepts can improve the soundness of correlations based on linguistic matching, but also relations can be found if the names are not matching at all. This is achieved by analyzing the relations among a group of several concepts. If the types of relations between the concepts of the group are the same, or some relations can be decomposed into combinations of inheritance relations (i.e., isA or partOf relations), the concepts and relations of both ontologies are matched. As a consequence, only one set of concepts and relations has to be kept in the ontology graph.

- **Structure concept removal:** During the search for a composition of services that satisfies a dependency of another service, property concepts in the ontology graph are identified, which are related with the services of the composition as well as with the dependent service (see service-to-ontology mapping in Section 5.2). For this operation only property and transfer concepts are needed. In order to reduce the memory size of the ontology graph, after all property and transfer concepts have been instantiated and added to the ontology graph, the ontology preprocessor removes the structure concepts and their relations. Nevertheless, information about the generalization and specialization of property concepts – which are specified by isA relations between structure concepts – must not get lost, as the composition search algorithm uses this information to find appropriate property concepts. Therefore, the ontology preprocessor has to copy the isA relations to the corresponding property concepts before deleting the structure concepts.

- **Unused graph removal:** While the ontology description itself should contain as many details about the system as possible, parts of the resulting ontology graph, which are actually not linked to services (i.e., they do not appear in the service-to-ontology mapping), are not relevant for service composition search. Keeping them in the ontology graph and traversing them only extends the maximum duration for composition search. In systems, within which the available services are changed rarely (i.e., services are usually not added or removed during the operation), it is useful to remove these parts of the ontology graph.

Property concepts in the ontology graph that are not linked with any service, and which are connected in the ontology graph to the next provided property concept (i.e., it is linked to a service) via at most one single property concept, are removed from the graph. Furthermore, all transfer concepts related to the removed property concept are also not required. In such cases, no meaningful information flow can be generated that incorporates the removed property concept, and hence, these concepts are not useful within any service composition. Figure 5.17 illustrates the structure of an ontology graph, where a subgraph can be removed. Property concepts linked to a service are indicated by the gray background color. Three property concepts and two transfer concepts at the bottom can be removed, as in this subgraph no property concept is linked to a service, and it is only connected to the rest of the ontology graph by a single property concept.

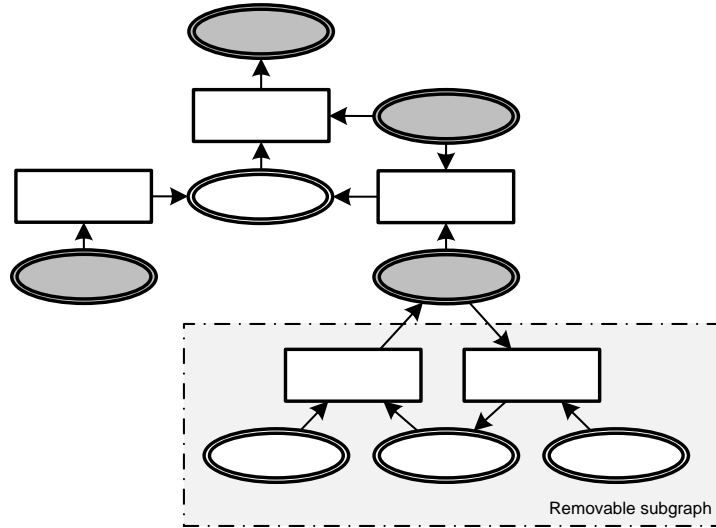


Figure 5.17: Example structure of a subgraph that can be removed. Property concepts linked to a service are shown in gray.

- **Merging of transfer concepts:** Transfer concepts with unary transfer functions (i.e., these functions only have one input parameter, like the *sine* functions) can be merged with other transfer concepts, if the following conditions hold:
 - The input property concept p_i , which will be removed due to merging, is not linked with a service.

- p_i has at most two input relations r_x and r_y with the transfer concepts t_x and t_y (i.e., p_i is the input for the transfer concepts), which both have to be unary functions. The output of t_x and t_y leads to the property concepts p_x and p_y , respectively. In case p_i has more than two input relations, it is not possible to perform the merge of transfer concepts, as p_i is also required as input for other transfer concepts which are not subject to merging. Hence, removing p_i from the ontology graph would leave these inputs unconnected.
- If the second relation r_y exists, p_i is allowed to have at most two output relations to transfer concepts (i.e., t_x^i and t_y^i), which have to implement the inverse transfer function of t_x and t_y . This means, that there is a path connecting p_i to p_x via transfer concept t_x^i , and another path connecting p_i to p_y using the inverse transfer concept t_y^i . The same is true for p_y together with t_y . Then, the transfer concepts t_y^i and t_x , as well as t_x^i and t_y can be merged. This situation is shown in Figure 5.18, where the original graph is presented at the top and the merged graph can be seen at the bottom.

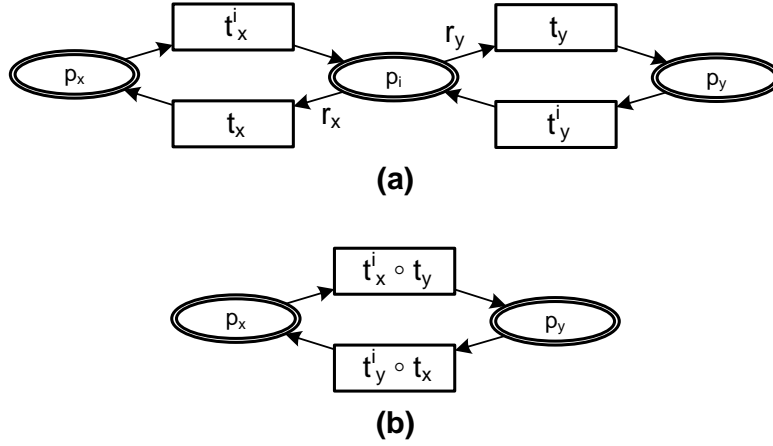


Figure 5.18: Merging of transfer concepts: (a) original graph, (b) merged graph.

- If r_y is not part of the ontology graph (as depicted in Figure 5.19), the transfer function t_x can be merged with all transfer concepts having an output relation towards the property concept p_i . The only exception is the inverse transfer concept t_x^i . As this function could only be used during composition search in a loop back to property concept p_x , t_x^i can also be removed from the ontology graph.
- **Search complexity optimization:** Basically, service composition search (see Section 6.2) is a graph search problem. In a fully expanded ontology graph, where all property and transfer concepts have been copied by the inheritance rules, the service composition search algorithm only has to follow the direction of the edges in the graph until the requested service composition is found. Using a fully expanded ontology graph additionally allows

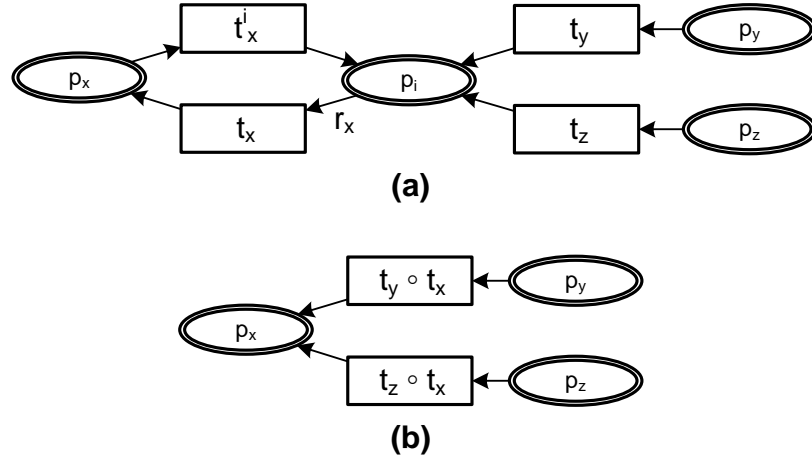


Figure 5.19: Merging of transfer concepts when r_y is not given: (a) original graph, (b) merged graph.

to estimate the worst-case search time based on the number of concepts and relations of the graph.

However, if the ontology contains deep hierarchies between structure concepts, a full expansion can lead to a considerable overhead in memory consumption. In such a case it might be better if the inheritance relations are kept in the ontology graph, and not all of the inherited concepts are instantiated in the graph. Then, the composition search algorithm is responsible to implement some of the inheritance rules (i.e., climb up and down the hierarchies to find appropriate property and transfer concepts). The disadvantage of this runtime expansion is, that the search algorithm becomes more complex and the worst-case bound for the composition search time cannot simply be estimated by counting the number of concepts and relations in the ontology graph.

Service Orchestration

Service orchestration is the central purpose of the presented dynamic reconfiguration framework, which is conducted by the SOE. It is invoked after a service disappears from the SOM in the system or after a new service is integrated into the system. Services disappear for instance, when there is a fault in a component of the cyber part of the system (e.g., the failure of a single service, the loss of a whole computational node hosting several services, or a breakdown of a part of the communication system), or in case services are removed intentionally. For each removed SOM entry, the service orchestration mechanism has to determine, if the removed service is still required by other services. If this is the case, it has to decide, whether the semantics of the removed service can be substituted by combining other services available in the system. This is done by finding a tree-shaped sub-graph in the system ontology graph. At the root of this tree is the property concept which was provided by the removed service. The branches of the tree consist of relations to property concepts that are provided by available services.

Then, from the found sub-graph a *transfer service* is created, that composes the information provided by this group of available services, in order to provide a semantically equivalent information as it was provided by the missing service. The process of replacing a missing service by a group of other services will be referred to as *service substitution*. Thereby, the challenge is that for real-time systems the process of service substitution has to be conducted within the temporal bounds predetermined by the real-time system. Similarly, when a new service is integrated, the service orchestration mechanism is responsible to figure out if the new service can be used to enable services that were blocked due to missing input information. Furthermore, the new service could also be used to improve the quality of input information (cf. Chapter 7) for other services.

Finally, new services (i.e., the transfer services and available services that have been unused) are allocated to computational nodes and a schedulability test has to be performed. In case of a time-triggered communication system, the involved services and communication between these services additionally have to be scheduled.

In this chapter, at first Section 6.1 provides an overview of the service orchestration process. Then details about *composition search* are presented in Section 6.2, which denotes the process of

searching for a group of services that provide information with certain semantics, either caused by removed or added services. Section 6.3 explains the automatic generation of transfer services from sub-graphs of the system ontology. Service allocation and scheduling of component execution and communication will be discussed in Section 6.4. The chapter concludes with considerations about the worst-case execution time (WCET) of the service orchestration process in Section 6.5.

6.1 Process Overview

The service orchestration process conducts the following steps in order to substitute a failed service or enable blocked services after the integration of new services:

1. It has to be determined which information is required by blocked services in the system. The required information is represented by property concepts in the system ontology. While a service might provide information about different system properties, upon the failure of the service, a substitute source of information only has to be found for those properties, that are actually required as input for other services. On the other hand, to exploit newly integrated services, the required properties are obtained from the list of services which are blocked due to missing information.
2. For each of the missing properties the system ontology is used to identify an appropriate group of services allowing to deduce the required property.
3. In order to combine and transform the information from this group of services such that the required property is obtained, a transfer service has to be created. This is done by concatenating the transfer functions of the transfer concepts between the services in the group.
4. Finally, required resources of the platform have to be allocated for the services that are concerned by the service substitution. In case of time-triggered communication this also involves scheduling of component execution and communication between services.

In Algorithm 1 the high level service orchestration algorithm is presented that combines these steps. The inputs to the algorithm are the set A of available services in the system, and a pointer ps to the service in the CSR that has to be substituted. If the service orchestration is triggered by the integration of a new service, ps is a *NULL* pointer. At first, by calling *getReqProperties*, a set R of required property concepts is identified that need to be deduced from a group of available services. In case this set is empty, the output of the failed service – to which ps is pointing – had not been used as input for another service, or no services in the system are blocked due to missing input information. Hence, the algorithm returns without taking any further actions.

Otherwise, the empty set F is created, that holds those property concepts for which no substitution has been found. For each of the property concepts in the set R , the algorithm has to find a combination of services allowing to deduce the corresponding property concept. This is done

Algorithm 1 High level service orchestration algorithm**Input:** Set A of available services in the systemPointer $*ps$ to service in the CSR that has to be substituted, or $NULL$ in case service orchestration is triggered after a new service was integrated**Output:** Set of NOT deducible properties

```

1:  $R \leftarrow getReqProperties(A, ps)$ 
2: if  $R == \emptyset$  then
3:   return  $\emptyset$  ▷ no missing property concepts
4: end if
5:  $F \leftarrow \emptyset$ 
6: for all  $r \in R$  do
7:    $[S, T] \leftarrow findComposition(r)$ 
8:   if  $S == \emptyset$  then
9:      $F \leftarrow F \cup r$  ▷ property concept  $r$  cannot be deduced
10:  else
11:     $C \leftarrow generateTransferService(T)$ 
12:     $scheduleServices(S, C)$ 
13:  end if
14: end for
15: return  $F$ 

```

by calling the function *findComposition* for each required property r . The function returns the set S of services that contribute to the deduction of r , and a tree structure T constituted by a subgraph of the system ontology, which represents a valid solution for the deduction of r . This tree – also referred to as *composition tree* – consists of transfer concepts which connect the required property concept r with property concepts that are provided by the services in S . In case no appropriate group of services has been found, the set S will be empty and the required property is added to the set F of not deducible properties.

If S is not empty, a transfer service is generated from the tree T by calling the function *generateTransferService*. Afterwards the procedure *scheduleServices* schedules the transfer service, as well as those services which are used as input by the transfer service, such that the temporal properties of all involved services can be guaranteed. This includes the scheduling of communication in time-triggered environments. When the search for compositions for all required property concepts has been finished, the algorithm outputs the set of not deducible property concepts. If this set is empty, the service orchestration was completely successful. Otherwise, this set is used to stop the affected services and update the list of blocked services in the system.

6.2 Composition Search

A *service composition* is a group of services that act together in order to reach a certain goal. In the context of the dynamic reconfiguration framework presented in this work, the goal is the provision of information with specified semantics. It is the purpose of the composition search mechanism to automatically identify such service compositions. The semantics of the required composition is defined by those services in the system, which are blocked due to missing information. Hence, a service composition is searched, which allows to provide this missing information.

Service composition search is performed in two steps: determination of required properties and identification of semantic equivalences.

6.2.1 Determination of Required Properties

It is the purpose of the dynamic reconfiguration framework to provide new sources of information about system properties for services, which would be blocked due to missing input information. Hence, semantic equivalences are only needed for those properties, which are actually required as input for some service. If a service fails that provides a property (i.e., it outputs information about the value of that property) that is not used by another service, no equivalence search and no reconfiguration have to be conducted. Similarly, after a new service is integrated, the search and reconfiguration processes are only triggered, if there exist services which might benefit from the new service, since they are currently blocked due to missing input properties.

Algorithm 2 formally describes the method to determine those properties for which service compositions have to be found. The function requires the set A of available services in the system – either active or blocked – and a pointer ps to the CSR. The pointer is either a reference to the failed service or a *NULL* pointer which indicates that a new service was integrated. With this pointer, different entries in the SOM are accessible that provide a list of property concepts, which denote the output properties of the failed service. At the beginning the set R is created that holds the required properties found so far. In case the algorithm was triggered upon a failed service, the set P of output properties is obtained from the SOM by dereferencing the pointer ps . These output properties denote those property values, which had been provided at the interface of the failed service. For all the properties in P it is checked if the property is actually used as an input property for any of the other available services in the system. If this is the case, the property has to be deduced by a set of other properties that are actually provided, and hence, it is added to the set R of required properties.

On the other hand, when the function is called upon the integration of a new service, and hence a *NULL* pointer was given to the function, all available services in the set A are examined. Only if a service is marked to be blocked because some input information is missing, it is determined which input property was not provided so far. These missing input properties are added to the set R of required properties. Finally, the function returns the set R for which a composition search should be performed.

Algorithm 2 Determination algorithm for required properties

```

1: function getReqProperties( $A, *ps$ )
   Input: Set  $A$  of available services in the system
           Pointer  $*ps$  to service in the CSR that has to be substituted, or NULL in
           case service orchestration is triggered after a new service was integrated
   Output: Set of required property concepts

2:    $R \leftarrow \emptyset$ 
3:   if  $ps \neq \text{NULL}$  then
4:      $P \leftarrow (*ps).outputProperties$ 
5:     for all  $prop \in P$  do
6:       for all  $a \in A$  do
7:         if  $prop \in a.inputProperties$  then
8:            $R \leftarrow R \cup prop$  ▷ property concepts is missing
9:           break
10:        end if
11:      end for
12:    end for
13:   else
14:     for all  $a \in A$  do
15:       if  $a.isBlocked == \text{TRUE}$  then
16:         for all  $prop \in a.inputProperties$  do
17:           if  $prop.isProvided == \text{FALSE}$  then
18:              $R \leftarrow R \cup prop$  ▷ property concepts is missing
19:           end if
20:         end for
21:       end if
22:     end for
23:   end if
24:   return  $R$ 
25: end function

```

6.2.2 Search for Semantic Equivalences

The core algorithm for the dynamic reconfiguration framework is the search algorithm for semantic equivalences of property concepts in the system ontology. The primary input to the algorithm is a property concept that has to be deduced by a set of other property concepts in the system. These properties have to be provided by an available service in the system that is not blocked. Such services are, for instance, a sensor service, a computational service combining different system properties, or a service providing constant system values. The algorithm performs a graph search in the directed system ontology graph to identify appropriate services that can be composed in order to calculate the value of the missing property. While structure, property type and transfer type concepts in the system ontology are mainly used for inheritance

and instantiation during the parsing process, only property concepts and transfer concepts are relevant for the search of semantic equivalences. Figure 6.1 presents an example ontology graph for the equivalence search algorithm, where the blue path marks a possible solution.

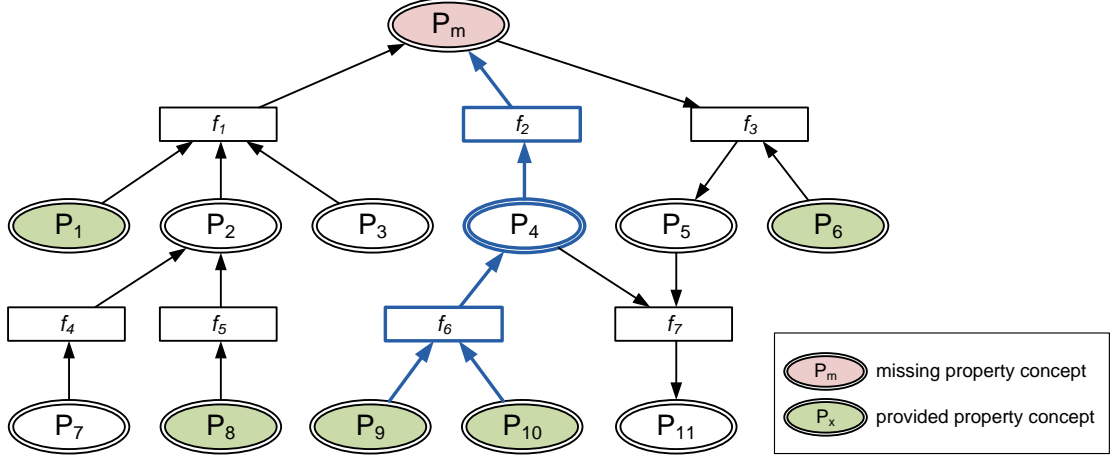


Figure 6.1: Illustration of ontology graph with a path that is semantically equivalent to the missing property.

Basic Search Procedure

Starting at the missing property concept P_m (represented by the red property concept in the figure) the algorithm may use every relation that has a data flow directed towards the missing property concept (as indicated by the arrows). In other words, this means that the neighbouring transfer concept has an output that is semantically equivalent to the required property concept. If no such relation is present, no solution for the equivalence search problem exists. Thus, the algorithm is finished and a negative answer is returned to the caller of the equivalence search algorithm.

When reaching a transfer concept, the algorithm has to check if all input relations to that transfer concept lead to property concepts that are provided. For instance in the figure, the transfer concept f_1 uses three property concepts as input, where only P_1 is provided by a service. In case an input property is not provided (e.g., P_2 or P_3), the algorithm continues to search with this property concept as a starting point. Hence, in the example it tries to find a semantic equivalence for property concept P_2 by examining the transfer concept f_4 with P_7 as input, and f_5 with P_8 , respectively. In contrast, if for a non-provided input property concept no further transfer function exists that outputs this property concept (see P_3), the dependencies of the transfer concept that uses this property concept as input cannot be satisfied. Therefore, the whole transfer concept cannot be part of the solution for the equivalence search. For example, the transfer concept f_1 cannot be used to deduce the property concept P_m , as the input property concept P_3 cannot be provided.

On the other hand, when all inputs of a transfer concept are provided (like for f_6), this transfer concept can be used together with the input property concepts to deduce the value of

the property concept at the output of the transfer concept. This situation is shown in the figure, where the properties P_9 and P_{10} are used as input for the transfer concept f_6 to obtain the value of the property P_4 . Hence, the algorithm backtracks and it can also mark the property concept at the output relation (i.e., P_4) as provided. This property can now be considered as a provided input to the next transfer concept, and thus, probably enable that transfer concept to provide another property concept. The algorithm can now go back on its search path until it reaches a transfer concept where inputs are not yet marked to be provided. From there on, the algorithm also checks the rest of the inputs of that transfer concept until it either finds one input that cannot be provided, or all of the inputs are found to be provided. In the first case, the concept at the output relation of the corresponding transfer concept cannot be provided by the current transfer concept. This is the case for the left part of the system ontology in the figure (i.e., f_1 and related concepts), where the property concept P_3 cannot be provided. Hence, another sub-graph has to be inspected or the algorithm has to backtrack without a valid solution. In the other case, the algorithm can further backtrack with a valid solution. This recursive check for provided input properties of transfer concepts is performed until at least one transfer concept is found, which outputs the missing property concept at the root, or all transfer concepts connected to the missing property concept have been investigated without success. If at least one transfer concept, which outputs the missing property concept, can be satisfied by provided inputs, a solution for the semantic equivalence search and service composition has been found.

Note: For the algorithm to be able to use a transfer concept for the solution, all inputs to that concept must be provided in order to satisfy its transfer function. In contrast, it is sufficient for a property concept to be provided by at least one transfer concept. This is for instance true for property concept P_2 of the figure, which cannot be provided by f_4 but by f_5 , and thus, it is considered as a provided input for the transfer concept f_1 .

The inheritance rules applied on structure concepts create a hierarchy of property concepts during ontology preprocessing. While the algorithm walks through the ontology graph it also encounters property concepts which are organized in such an hierarchy, where one property concept is the instantiation of the other property concept. Figure 6.2 shows an exemplary ontology graph that contains such hierarchies of property concepts. The property concept *angular speed* is defined for the top level structure concept *wheel*, and is inherited to *left wheel*, *right wheel*, as well as to *left rear wheel* and *right rear wheel*. If the algorithm reaches a property concept, which is the parent concept of another property concept that is connected by an *isA* relation, it might also use the child property concept to provide the input for the transfer concept. This is equivalent to using a specialization of the parent property. On the other hand, it is not allowed to use the hierarchy in the opposite direction, i.e., going from a child property to the parent concept, which denotes a generalization. When the *curve radius* in the example has to be deduced, it can be calculated from the difference in the angular speed of the left and the right wheels of a vehicle. The algorithm can use the specializations of the angular speed of the left wheel, and the right wheel respectively, which lead to properties that are actually provided by sensors.

Basically, the system ontology can be considered as an arbitrary graph which is very likely to contain redundant paths or cycles. Redundant paths mean that a particular property concept can be reached from another property concept via more than one path in the ontology graph. A cycle denotes a path that starts at a certain property concept, and which is coming back to the

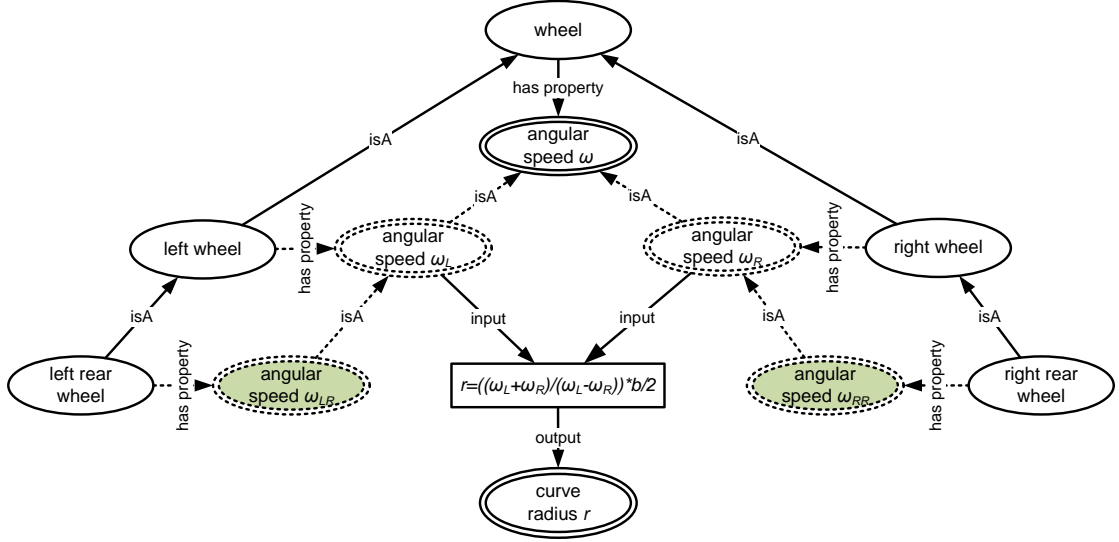


Figure 6.2: Example ontology graph containing hierarchies of property concepts.

same property concept after encountering at least one other property concept (see Figure 6.3). Due to redundant paths and cycles, the algorithm might encounter one property concept several times. If this is the case because of a redundant path, the algorithm reuses the marker, which indicates whether the property concept can be provided or not, of the previous time the property concept was visited. Consequently, the ability to provide a concept has to be evaluated only once. Similarly, a marker exists which is used to detect cycles in the ontology graph and to prevent search solutions containing circular dependencies between system properties. This marker is set when the algorithm enters a property concept while it moves away from the missing property. Hence, the set of currently investigated paths in the system ontology is expanded. The marker is reset, when the algorithm backtracks and moves in the ontology graph back towards the missing property – i.e., the set of investigated paths is truncated. In the example in Figure 6.3 the blue concepts are already marked as part of the current search path. As the property concept P_1 is already part of the current search path, the transfer concept f_2 is not used for the service composition. Due to both markers, the number of concepts that have to be checked for provision steadily decreases over the execution time of the algorithm and guarantees its termination.

In contrast to the arbitrary structure of the system ontology graph, a valid solution of the search always consists of a tree-shaped sub-graph of the system ontology graph (i.e., the composition tree), where the root of the tree is the property concept for which the semantic equivalence has to be found. The leafs of this tree only consist of property concepts that are actually provided by a service (e.g., by a sensor service, by a computational service, or a service providing system constants). The tree's inner nodes alternately comprise transfer concepts and property concepts, which allow to deduce the value of the property concept at the root.

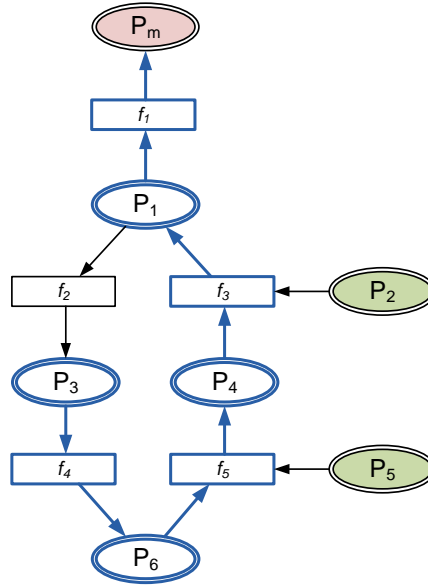


Figure 6.3: Circular search path in ontology graph.

Definition 6.1 (Semantic Distance) *The semantic distance denotes the number of transfer concepts between two property concepts within the system ontology graph.*

Definition 6.2 (Semantic Depth) *The semantic depth of a service composition tree is the maximum distance between the root property concept and any other property concept in the tree.*

Note: The semantic distance and depth are no measures that are inherent to a system, but they depend on the system engineer designing the ontology. The engineer decides how many intermediate steps are used to describe the transfer behaviour between system properties (i.e., if a transfer function is divided into several separate transfer functions). However, these measures can give a rough estimate about the computational complexity of a service composition.

Similarly, the number of structure concepts used in a system ontology strongly depends on the ontology engineering, where the same system can be structured into different subsystems and hierarchy levels. However, from the semantic point of view structuring is not relevant, as it does not influence the semantic distance of a service composition.

Definition 6.3 (Semantic Width) *The semantic width denotes the number of provided property concepts required for a service composition.*

In contrast to the semantic distance and depth, the semantic width is a system inherent property that does not depend on the granularity of the system description within the ontology. The flow of information between system properties is predetermined by the physical part of a CPS (e.g., due to mechanical connections between different drive shafts of a vehicle). Hence, the combinations of properties required to deduce another property are given by the system itself. Also the semantic width is a useful measure for the computational complexity of a service composition.

Requirements for Service Compositions

In addition to the pure search for semantic equivalences, different non-functional requirements can be defined for service compositions. Such requirements may include, for instance, requirements on the quality of the provided values (see Chapter 7), semantic depth or width of the service composition, computational or energy costs, etc. Even though a found solution is semantically equivalent with the missing property concept, the resulting service composition might violate some of the imposed requirements. For example, a service that uses the missing property as input requires a certain accuracy of the property value in order to be able to correctly control the CPS. If this value is deduced from a set of sensors by using different transfer functions to calculate the value, the accuracy of the obtained value can be worse than the required accuracy. Thus, the service controlling the CPS might fail to provide its intended behaviour.

Additional information can be included in the system ontology and service descriptions that allows the search algorithm to check whether the current search path violates one of the given requirements. Whenever one of the requirements cannot be met, the algorithm ceases to explore the current path and it backtracks until no requirement is violated. Thereby, unnecessary computational effort can be prevented. Including the computational cost of a transfer function in the description of the transfer concept allows, for instance, to sum up all computational costs in the current search tree, and if it exceeds the maximum allowed delay, the algorithm chooses another direction in the ontology graph.

Similarly, the algorithm can use the added information to optimize the service composition w.r.t. one or more non-functional criteria. At property concepts that are output by several transfer concepts, the algorithm checks for all of these transfer concepts whether they can be used to provide the property concept. Afterwards, that transfer concept is selected for the service composition, which optimizes the desired criteria.

For the requirements and optimization criteria almost no limitations concerning their type and number exist. However, checking the compliance of requirements and performing an optimization – possibly with more than one criterion – increases the execution time of the equivalence search algorithm, which might not be feasible for some types of CPSs (e.g., when a very short reconfiguration time is required).

Search Strategies

The equivalence search in the system ontology is similar to reachability problems in graph theory [34]. Hence, graph search algorithms and their properties can also be applied for finding equivalences. In the following paragraphs, prevalent representatives of graph search algorithms are described in the context of equivalence search and their advantages and disadvantages are discussed:

- **Depth first search:** The algorithm begins at the missing property concept and selects one neighbouring transfer concept or inherited property concept. In case of an inherited property concept, the algorithm recursively starts again to search from the inherited property concept. Otherwise, from the transfer concept it follows a path until provided property concepts are reached, or a termination condition is encountered (see Figure 6.4). If such a

termination condition was encountered (see subsection on *Requirements for Service Compositions*), the algorithm has to backtrack up to a property concept that is output by several other transfer concepts (i.e., different search branches exist). From this concept it takes another possible branch and tries to continue the path into another direction. In Figure 6.4 this behaviour is illustrated at property concept P_2 , where the algorithm first tries to deduce the value of P_2 by using the transfer concept f_4 . As the input of f_4 (i.e., P_7) is not provided, beginning at P_2 another branch is investigated (i.e., using transfer concept f_5) in order to obtain the value for P_2 . The whole procedure is executed recursively until all conditions that signal a valid service composition are satisfied, or the algorithm reaches the root after all branches have been taken without finding a solution.

Advantages and disadvantages of the depth first search are:

- + Due to the recursive expansion of search paths, the algorithm is simple to implement.
- + When requirements or optimization criteria are given for service compositions, the probability of not satisfying non-functional requirements (e.g., computational effort, accuracy of property value) increases on long paths and will quickly lead to abandon paths and to backtrack. Hence, paths which do not lead to a solution will quickly be omitted.
- Long search paths are explored first which may result in composition trees with a high semantic depth, and thus, inefficiently complex transfer services with high computational demands and increased uncertainty of the deduced property value.
- For large system ontologies, a simple recursive implementation of the depth first search requires a high amount of stack memory.

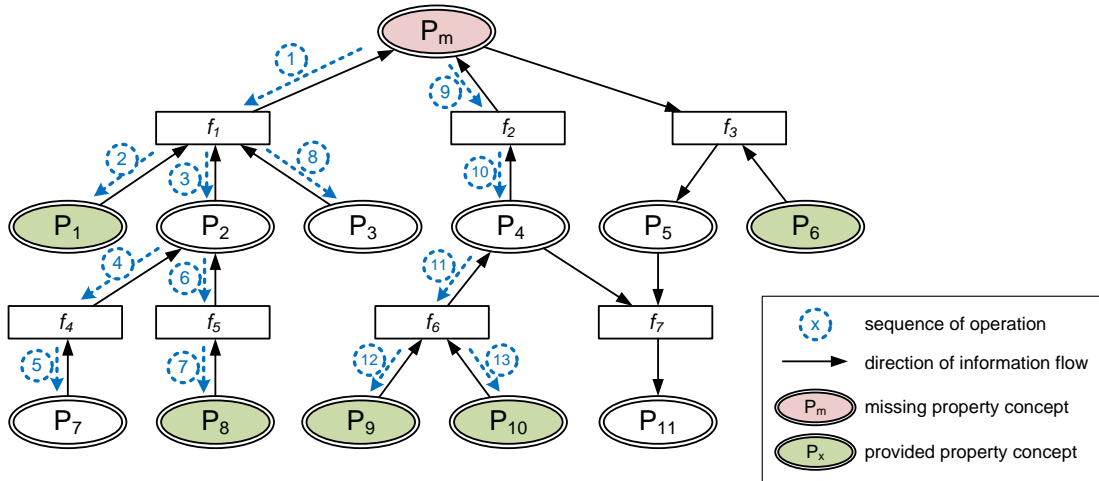


Figure 6.4: Illustration of depth first search algorithm.

- **Breadth first search:** In contrast to the depth first search, the breadth first algorithm explores the system ontology graph within rounds of iterative deepening, where all possible paths are investigated simultaneously until either a valid service composition was

found, or the entire ontology was browsed without solution. Each round comprises the exploration of concepts on distinct paths, where these concepts have the same semantic distance to the missing property. Whenever the algorithm reaches a property concept that allows several branches (i.e., the property concept is at the output of more than one transfer concept), it expands all possible paths before further investigating one particular path. Hence, the system ontology graph is scanned 'layer-by-layer'. An efficient implementation of this algorithm keeps the concepts at the frontier of the expanded paths in a queue, such that the computational overhead for the selection of the next concept to investigate is insignificant. Figure 6.5 illustrates the behaviour of the breadth first search algorithm. Before the left sub-graph is explored completely (i.e., the transfer concepts connected below property concept P_2), the transfer concept f_4 in the center is inspected. The right path in the figure cannot be explored for composition search, as the neighbouring transfer concept (i.e., f_3) has no output relation towards the missing concept.

Advantages and disadvantages of the breadth first search are:

- + This algorithm will always find composition trees with the least possible semantic depth, which reduces the computational complexity of the resulting transfer service and service composition.
 - + The probability of a smaller semantic breadth is higher for smaller composition trees, which increases the probability for improved quality measures of the solution (e.g., reduced resource and energy consumption, better value accuracy).
 - Keeping track of the expanded search paths increases the memory requirements of the algorithm. Especially if the ontology graph contains a high number of transfer concepts with many input relations or property concepts that are output by several transfer concepts, the number of expanded paths strongly increases.
 - Paths that do not lead to a solution – either because property concepts cannot be provided or non-functional requirements cannot be fulfilled – are abandoned comparatively late during the search process, such that the number of simultaneously expanded paths remains high.
- **Guided search:** The guided search approach is similar to the breadth first search, but instead of expanding search paths with the same semantic depth, an objective function is used to select the next path. This objective function provides a value indicating the likelihood of success of a search path. Before following any transfer concept to check if its inputs are provided, the objective function is evaluated and the one transfer concept is selected, which optimizes the metric that is used for the function. Various single metrics or combinations of them can be applied, where the main limitation is the computational effort to evaluate the next path to be taken.

Examples for such metrics for the objective function are:

- *Computation cost:* The cost for computation considered during composition search originates from the execution of the transfer function of transfer concepts in order

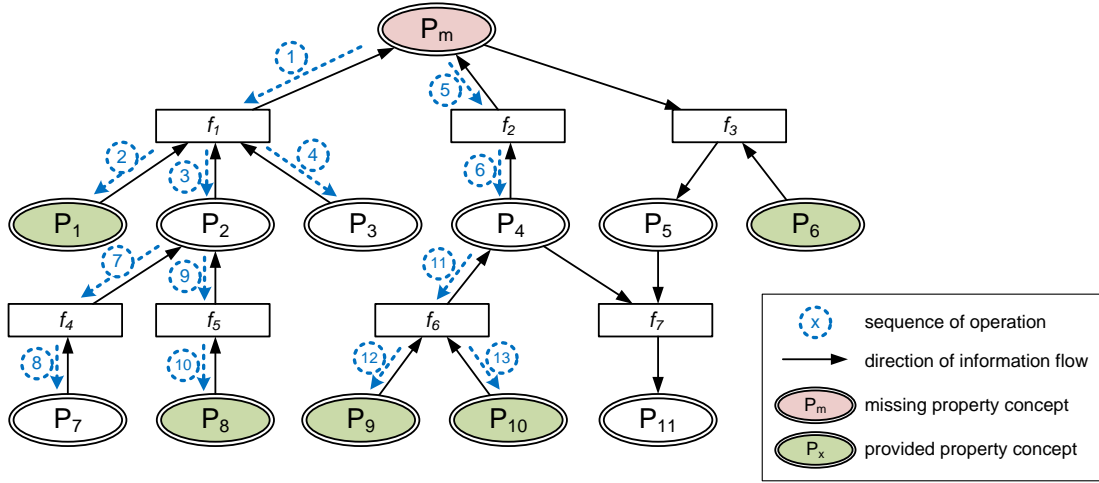


Figure 6.5: Illustration of breadth first search algorithm.

to deduce the value of the missing property, and from the services that are needed in addition to those services which are already running. If the platform specific WCET for these transfer functions and service executions is given in the system ontology, the algorithm prefers paths on which the computational cost of the service composition is minimized.

In each step of the path expansion, the algorithm selects the transfer concept to be explored next, for which the sum of computational cost on the current path plus the computational cost of the transfer function is minimal. Especially at property concepts, which are at the output of more than one transfer concepts, this might lead to several simultaneously expanded (i.e., currently investigated) search paths. In contrast, at each transfer concept that is part of an expanded search path, at most one input path can be explored at once. This means, that it is not possible to expand the search on more than one input of the transfer concept. If a provided property concept is reached (i.e., either directly or indirectly provided), the algorithm backtracks and adds the computational cost of the provision – i.e., the computation cost of the involved services and the transfer functions connecting these services – to the computational cost of the transfer concept which uses that property concept as input. After that, the algorithm may explore the next input of this transfer concept. When all inputs of a transfer concept can be provided, the computational cost of the property concept at the output is the sum of the transfer function cost and the provision cost of all input property concepts. If a property concept is output by more than one transfer concept, the path with the minimum computational cost is selected. After a valid service composition has been found, the algorithm still has to continue to investigate all other expanded paths which are currently marked with a lower computational cost as the found solution. Only if no further path exists with a lower cost, the service composition is returned as the solution.

The behaviour of this guided search is demonstrated in Figure 6.6. The computation cost is given for the expansion phase (i.e., when the algorithm is moving away from the missing property), as well as for the contraction phase (i.e., when the algorithm is tracking back). The first value is used as the metric to select the next path to be expanded. The actual computation cost is calculated from the leafs towards the root of the service composition tree, which is given by the second value. If a property concept is at the output of several transfer functions (e.g., P_2), during backtracking the path with the lowest computation cost is selected. For P_2 only one path exists which provides a valid result. Since each input for the transfer concepts is investigated one after the other, the provision cost of those inputs which have already been investigated, can be added to the expansion cost of the rest of the inputs – see f_6 , where the cost of P_9 is already known and added when the path towards P_{10} is investigated.

Note: The provision of P_{10} is checked before the path f_5 is explored, although its expansion cost is already higher (the cost is 9) than for the path to P_8 (the current cost is 8). This is only the case, because P_{10} is directly provided. If this was not the case, the other path would be preferred.

Note: Even if the algorithm found a valid service composition with computation cost 11 (i.e., the path in the center of the figure), all other expanded paths with a lower computation cost (i.e., by using f_5) still have to be investigated, because a service composition with lower computation cost could exist.

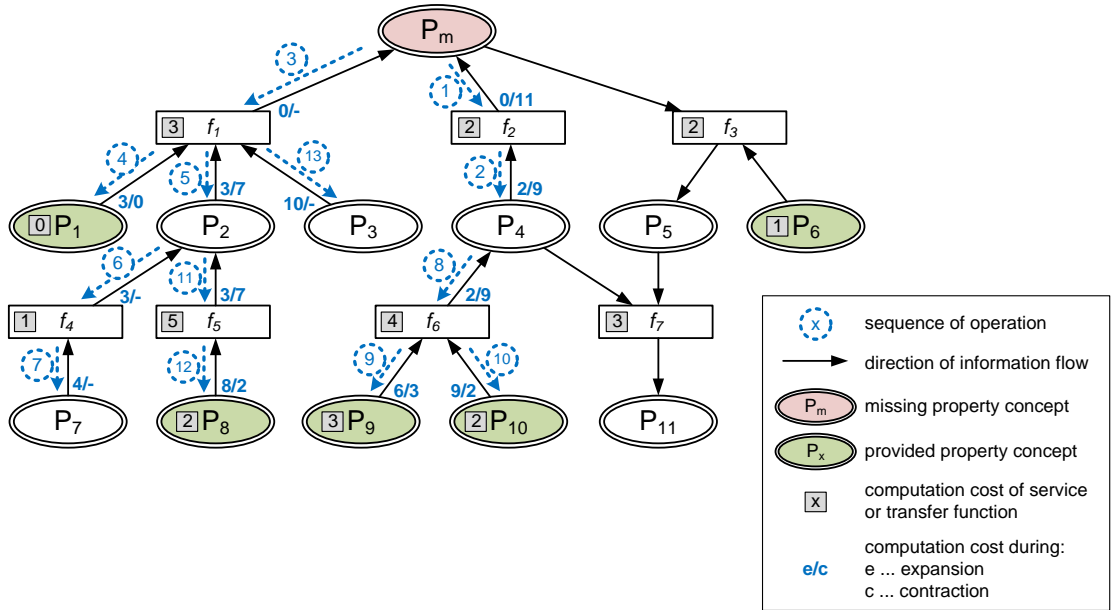


Figure 6.6: Illustration of guided search algorithm with computational cost as search metric.

- *Resource requirements*: Similarly to the computation cost, resource requirements like energy consumption, communication efforts or others can be used as metric.
- *Achievable value accuracy (uncertainty)*: Since the achievable value accuracy for a deduced system property depends on the accuracy of the involved services, as well as the transfer functions combining the values of these services, some service compositions allow to calculate more accurate property values than other compositions (see Chapter 7). If the required information about the accuracy of the services and the influence of the transfer function on the accuracy is given in the CSR and the system ontology, the search algorithm prefers paths on which the achievable accuracy is maximized – or where the value uncertainty is minimized, respectively.
- *Semantic breadth*: When the semantic breadth is used as metric, the search algorithm prefers paths where less branches are required. A higher number of branches usually requires more distinct services as input, which often results in a worse value accuracy and the probability of a failing service may be increased. Furthermore, the computational effort for deduction is increased and the temporal coordination of services gets more complex.
- *Number of required transfer concepts*: Like using the semantic breadth, the number of transfer concepts that are required for a service composition can be used as metric, where a lower number is preferred. This metric is based on the assumption, that using more transfer concepts for a service composition provides a higher probability for increased computational cost and a degraded value accuracy.

Similarly to the breadth first search, the selection of the next concept to explore can be implemented efficiently with an ordered queue. When a new concept is added to the queue, the objective function is evaluated and the concept is placed in the queue such that the queue is ordered with the concept having the highest score at the beginning. The next concept to be explored is the first one in the queue.

Advantages and disadvantages of the guided search are:

- + If a service composition is identified, this solution is optimal with respect to the used metric. In order to achieve optimality with depth or breadth first, the whole system ontology has to be investigated and the best service composition be selected among all found solutions.
- Keeping track of the expanded search paths increases the memory requirements of the algorithm. Especially if the ontology graph contains a high number of property concepts that are output by several transfer concepts, the number of expanded paths strongly increases.
- The evaluation of the objective function increases the computational effort for composition search, particularly if many paths are expanded simultaneously. In cases where a service composition has to be found within a short bounded interval, this additional effort might not be feasible.

- Some objective functions require that the metric value of all transfer concepts in the system ontology, as well as of services have to be determined and stored in the ontology, and CSR, respectively. This increases the memory requirements for the dynamic reconfiguration system.

Anytime Implementation

While the solutions for service compositions using depth or breadth first search might be highly suboptimal, the application of guided search suffers from the high overhead for the evaluation of the selected quality metric and the additional memory requirements. For real-time systems which demand a reconfiguration decision within a bounded time (e.g., within the FTTI) in order to recover from a service failure before the system has to be brought into a safe state, or where dynamic reconfiguration is implemented as a never-give-up strategy [56, p. 154], the guided search might not be feasible and the solution found by the other techniques might fail to fulfill the quality requirements of the system. To overcome these problems, the computationally less demanding depth or breadth first search algorithms can be combined with an anytime approach [74, 121].

Using the anytime approach, the best possible service composition that can be found within a predefined time interval is taken as the solution for the reconfiguration. Instead of returning the first service composition that is identified, the remaining time is used to improve this solution with respect to a defined quality metric. Similarly to the guided search, additional information has to be stored in the system ontology and CSR. But in contrast to guided search, the evaluation of the metric only takes place after a semantic equivalence for the missing property has been found. Therefore, the computational effort for this evaluation is not required for paths in the system ontology that do not lead to a valid solution. However, it is still possible that no service composition is found at all within that interval, even if a solution exists. In the case that no solution is found until the end of the interval, a standard recovery action (e.g., entering degraded or fail-safe mode) has to be taken.

Example Algorithm

The recursive algorithm shown in Algorithm 3 abstracts from details of the actual implementation of the composition search. The simple algorithmic structure of the depth first search methodology is used to demonstrate the search for service compositions, but the basic mechanisms for the identification of semantic equivalences of the missing property remain the same for all types of search algorithms.

The input to this service composition search algorithm is the missing property concept P_m , for which semantically equivalent information has to be found in the system ontology. In order to find a valid composition, the algorithm iterates over all neighbouring concepts of P_m (i.e., transfer or property concepts in the system ontology that have a direct relation to P_m) until either one neighbour leads to a valid composition, or all of them have been checked without success. Each property concept pc is represented by a structure that holds the following information:

- *isA_child*: a set of property concepts that are *isA* children of pc

Algorithm 3 Composition search algorithm

```

1: function FINDCOMPOSITION( $P_m$ )
   Input:  $P_m$  missing property concept
   Output: Set of services required for service composition; Tree-shaped ontology
           sub-graph combining these services to deduce value of missing property
2:   for all  $p \in P_m.isA\_child$  do                                     ▷ Check provision of child
3:     if  $p.provided = true$  then
4:        $p.visited \leftarrow true$ 
5:       return  $[p.service, p.tree]$ 
6:     end if
7:   end for
8:   for all  $n \in P_m.outputs$  do                                     ▷ Try to deduce the missing property
9:      $S \leftarrow \emptyset$ 
10:     $T \leftarrow \{n\}$ 
11:     $provided \leftarrow true$ 
12:    for all  $i \in n.inputs$  do
13:      if  $i.provided = true$  then
14:         $i.visited \leftarrow true$ 
15:         $S \leftarrow S \cup i.service$ 
16:         $mergeTree(T, n, i, i.tree)$ 
17:      else
18:        if  $i.visited = false$  then
19:           $i.visited \leftarrow true$ 
20:           $[s, t] \leftarrow findComposition(i)$ 
21:          if  $[s, t] = \emptyset$  then
22:             $provided \leftarrow false$ 
23:            break                                     ▷ No service composition with  $n$ 
24:          else
25:             $i.provided \leftarrow true$ 
26:             $i.service \leftarrow s$ 
27:             $i.tree \leftarrow t$ 
28:             $S \leftarrow S \cup s$ 
29:             $mergeTree(T, n, i, t)$ 
30:          end if
31:        else
32:           $provided \leftarrow false$ 
33:          break                                     ▷ No service composition with  $n$ 
34:        end if
35:      end if
36:    end for
37:    if  $provided = true$  then
38:      return  $[S, T]$ 
39:    end if
40:  end for

```

```

41:   for all  $p \in P_m.isA\_child$  do                                ▷ Try to deduce the value of a child
42:       if  $p.visited = false$  then
43:            $p.visited \leftarrow true$ 
44:            $[s, t] \leftarrow findComposition(p)$ 
45:           if  $[s, t] \neq \emptyset$  then
46:                $p.provided \leftarrow true$ 
47:                $p.service \leftarrow s$ 
48:                $p.tree \leftarrow t$ 
49:               return  $[s, t]$ 
50:           end if
51:       end if
52:   end for
53:   return  $\emptyset$                                                     ▷ No service composition found
54: end function

```

- *outputs*: a set of transfer concepts that output the value of pc
- *provided*: variable signalling whether pc can be provided
- *visited*: variable signalling whether pc has already been visited during the search
- *service*: set of services required to provide pc
- *tree*: composition tree providing pc

At first, all child property concepts for P_m , which have been instantiated automatically due to the *isA* relation between structure concepts, are examined (lines 2ff). If anyone of these children is already provided, the service that provides the value of this child property, or the composition tree and services that can be used to deduce this property, are returned as the solution for the service composition. For instance in an automotive system ontology, if the missing property P_m is the *angular speed*, which is defined for the structure concept *wheel* and which is inherited to the *left wheel* and *right wheel*, the value of the wheel sensor of any left or right wheel can be used as the solution.

In case no provided child property exists, the algorithm tries to deduce the missing property by using neighbouring transfer concepts that have an *output* relation towards P_m . The empty set S , which is used to hold all services that are involved in the service composition, as well as the tree T , with the current transfer concept n as root, are created. Afterwards, it has to be checked whether all of the input property concepts i to the transfer concept n are provided. If i has already been marked to be provided (lines 14ff), then the set of services $i.service$, on which the value of i depends, is added to the set S of services on which P_m will depend after a service composition has been established successfully. Furthermore, the composition tree T , which holds the transfer concepts that connect the services in S , is updated. This is done by calling the function $mergeTree(T, n, i, i.tree)$, which appends the tree $i.tree$ to the input i of transfer concept n in the tree T . When i is directly provided by a service (e.g., that outputs the readings from a sensor), T will be empty, as no transformation of information has to be performed.

For the case that i has not yet been marked as provided (lines 17ff), two situations have to be distinguished:

1. **Property concept i has not yet been visited:** The algorithm is called recursively with i as missing property concept. If the recursive call returns an empty set, it means that it is not possible to provide i (line 21), and hence, it is also not possible to deduce the value of P_m by using the transfer concept n . Otherwise, i is marked as provided and the list of services and the composition tree are updated accordingly (lines 25–29).
2. **Property concept i has already been visited:** As the composition search algorithm has already been applied for concept i , and no successful composition had been found, the algorithm need not be called again with i as input. Consequently, due to the fact that an input for the transfer concept n exists that cannot be provided, n cannot be used to deduce the value of P_m .

Only if all inputs of the transfer concept n are provided, the algorithm returns the service composition in the form of a list of services S needed to deduce P_m , as well as the composition tree T combining these services (line 38).

The last possibility to achieve a valid service composition is to deduce the value of a child property concept of P_m (lines 41ff). For each child property p that has not yet been visited, the algorithm is called recursively with p as missing property. If the function call returns with a valid service composition (i.e., the return value is not an empty set), this composition is also a valid solution for the deduction of P_m , and hence, the same set of services s and the tree structure t are returned. In the case, that not even for a child property a valid composition can be found, the algorithm returns an empty set to indicate that no valid service composition exists.

6.3 Transfer Service Generation

After a valid service composition for deducing a missing property concept has been found, a transfer service is generated automatically from the services and the composition tree identified during composition search. The transfer service combines and transforms the information, which is output by these services, by using the transfer functions of the transfer concepts in the composition tree. Figure 6.7 shows an example composition tree for transfer service generation, which is based on the example ontologies in the previous sections.

Each node within the tree is a transfer concept that describes a specified *transfer behaviour* which transforms the information at the inputs of the transfer concept into information about the property concept at the output. At the leaves of the tree, transfer concepts are located, which only have property concepts as input that are directly provided by a service (e.g., by sensor services). The output of the tree's root represents the value of the missing property concept. Hence, starting at the leafs, the transfer functions of the found composition tree are combined such that the output of a transfer function acts as the input for the next transfer function until the root is reached. The transfer functions at the leafs of the tree get their input from the services returned by the composition search algorithm. Finally, a single transfer function at the root of the tree outputs the value of the missing property. For instance, in the figure the provided property

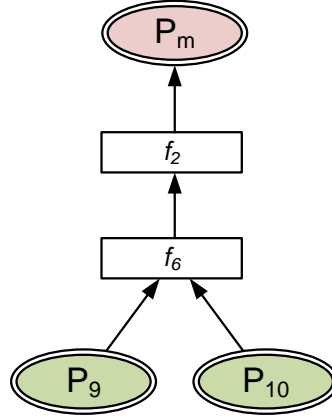
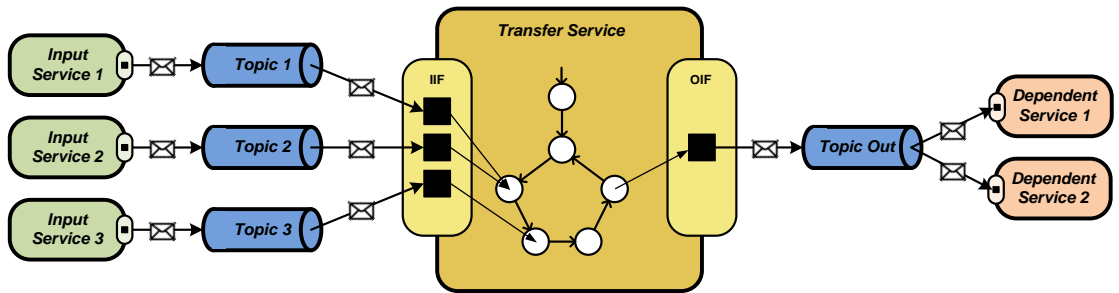


Figure 6.7: Example composition tree resulting from Figure 6.1.

concepts P_9 and P_{10} are combined by transfer concept f_6 , the output of which is transformed by transfer concept f_2 in order to obtain the value of the missing property concept.

The automatically generated program code for the transfer service, which consists of the combination of the transfer functions, must be scheduled and executed to act as mediator between the other services. By using platform-specific interaction mechanisms, the input information for the transfer service is received from the corresponding services and the output value of the missing property is sent to the services that require this information. For instance, when using a *publish-subscribe* mechanism ([27]), additional program code has to be generated that *subscribes* to all *topics* (i.e., information channels) of the input services, and that *publishes* the deduced value to the output topic. The integration of the transfer service into the service framework is illustrated in Figure 6.8. The transfer service subscribes to three topics of distinct input services, such that the information published on these topics is available at the input interface (IIF) of the transfer service. This information is read during the execution of the transfer behaviour code, in order to calculate the value of the missing property, which is then stored in the output interface (OIF). The publishing mechanism reads that value and sends it to the dependent services using a dedicated output topic.

Figure 6.8: Embedding transfer service into *publish-subscribe* framework.

The whole transfer service is then instantiated on a node that is able to fulfill the resource requirements (i.e., computation, memory and communication requirements) of the transfer service. As the output of the transfer service matches the specification of those services which require the missing property as input, these services are not able to distinguish if the values are provided directly or by a transfer service.

Algorithm 4 presents the algorithm for the generation of a transfer service. It consists of the two functions *createCode*, which concatenates the transfer functions to obtain the transfer behaviour, and *generateTransferService*, that starts the creation of transfer service code, which is then integrated into the platform. The input to the algorithm is the root T of the composition tree. It recursively creates the program code implementing the transfer behaviour of the transfer service by first going down to the leafs of the tree, and then assembling the transfer functions of the transfer concepts until the root of the tree is reached. Thereby it is distinguished between initialization code *tci* (e.g., used to subscribe to topics or activate receive interrupts) and main program code *tcm*, that actually calculates the value of the missing property.

For each input i of the current node T it is checked whether the input is a service or another transfer concept (line 5). In case i being a service, it means that this input to the transfer concept is already available in the system and it is provided as output of the service i . Therefore, by calling *getInput*(i) additional initialization code ci is generated (line 6), that causes the output value of service i to be loaded into variable v . For instance, in case of the publish-subscribe mechanism, program code to subscribe to the topic corresponding to service i is created such that the value is stored in the given variable.

Otherwise, if i is a transfer concept, the function is called recursively with the sub-tree rooted at i as input parameter. This causes the generation of the main program code cm , which calculates the output value of the transfer concept i . That value is stored into variable v . The main code cm is then appended to the overall transfer code tcm . Now the program code tcm integrates the transfer behaviour required to provide the inputs for the transfer function of the transfer concept at node T – i.e., program code to provide or calculate the input values for that transfer function. Additionally, the recursive call returns with initialization code ci , which is appended to the overall initialization code tci . As in both cases the variable v is an input to the transfer function of T , the variable has to be bound to the code of the transfer function. This means, whenever the transfer function – given as $T.tfunc$ – reads the output value of cm , the variable v is accessed. Binding of variables to transfer functions is achieved by calling the function *bindVariable*($T.tfunc, i, v$), which binds the variable v to the input i of the transfer function $T.tfunc$.

After all inputs of T have been processed, the transfer function of T is appended to the overall program code tcm . Using the function *getOutput*($T.tfunc$) returns the memory address to which the output of the transfer function is written. Then the function *createCode*(T) returns the initialization and main program code that is necessary to calculate the output value of T , and the variable to which this value will be stored. When the complete program code of the transfer service has been created, output code, that is responsible to send (i.e., publish) the result to services depending on that value, is generated (line 20). In the last step, the actual transfer service code C is built by integrating all parts of the code (i.e., initialization, main and output code).

Algorithm 4 Transfer Service Generation algorithm

```
1: function CREATECODE( $T$ )
   Input: Current transfer concept  $T$  within composition tree
   Output: Main program code and initialization code of transfer service
           Output variable to which the transfer service stores the result
2:    $tci \leftarrow \{\}$ 
3:    $tcm \leftarrow \{\}$ 
4:   for all  $i \in T.input$  do
5:     if  $i.is\_service = true$  then
6:        $[ci, v] \leftarrow getInput(i.topic)$ 
7:     else
8:        $[cm, ci, v] \leftarrow createCode(i)$ 
9:        $tcm \leftarrow concatCode(tcm, cm)$ 
10:    end if
11:     $tci \leftarrow concatCode(tci, ci)$ 
12:     $bindVariable(T.tfunc, i, v)$ 
13:  end for
14:   $tcm \leftarrow concatCode(tcm, T.tfunc)$ 
15:   $o \leftarrow getOutput(T.tfunc)$ 
16:  return  $[tcm, tci, o]$ 
17: end function

18: function GENERATETRANSFERSERVICE( $T$ )
   Input:  $T$  root of composition tree
   Output: Program code of transfer service
19:    $[cm, ci, v] \leftarrow createCode(T)$ 
20:    $co \leftarrow createOutputCode(v)$ 
21:    $C \leftarrow integrateCode(ci, cm, co)$ 
22:   return  $C$ 
23: end function
```

Note: Conceptually, there are no restrictions about the structure and programming language of the transfer functions, as well as the transfer service implementation. It depends on the platform what kind of implementations are supported, and whether the service code has to be compiled online or interpreted at the target component. In practice, restrictions on the transfer functions (e.g., limiting the allowed system calls) might be advisable in order to keep the transfer services as simple as possible.

6.4 Service and Communication Scheduling

In case of a time-triggered system, a static schedule for all services (and particularly for the transfer service) that are concerned by the new service composition, has to be created such that the temporal requirements of all services (e.g., maximum duration between sensor reading and issuing commands for actuators) are met. The purpose of scheduling in the dynamic reconfiguration framework is to guarantee a bounded end-to-end delay based on the application requirements (e.g., for stability of control) between the acquisition of information about the environment by sensors, and the application of control commands that are based on the acquired information at actuators. Implementing the concept of *pulsed data streams* [53] facilitates the minimization of the end-to-end delay by assigning a static schedule to services in a distributed system and the communication between these services. A time-triggered communication system (e.g., TTP/C [60] or TTEthernet [55]) between the computational components supports guarantees on the timeliness of communication. Hence, when using time-triggered communication systems, service scheduling also involves scheduling of communication channels.

As service and communication scheduling is a research topic on its own and countless papers are already available on static and dynamic real-time scheduling (e.g., [14, 20, 63, 95, 106]), within this work a solution to the scheduling problem is assumed to be available. Also for the static assignment of services to processors different algorithms have been proposed in the past (e.g., [46, 87, 119]), which are also applicable for assigning services to computational nodes. The objective of this section is to give a general idea what additional considerations have to be taken into account when implementing the proposed dynamic reconfiguration system.

At this point of service orchestration it is assumed that the set of services that have to be scheduled satisfies the temporal requirements of the missing property concept. These requirements include that deadlines must not be missed due to the additional sum of execution times and delays (e.g., of communication or scheduling) that are caused by the service composition. Furthermore, the update rate of all data, which is output by the involved services, has to be high enough, such that the information about the physical part of the CPS does not get outdated on the data processing path until it affects the environment (i.e., actuation on the environment must not be based on invalidated information). This also requires that the transfer service and communication between services is considered as part of the delay of the data path. Checking this timeliness and data validity properties has to be performed during service composition search (e.g., by using the computation cost as optimization criterion), as it determines whether a service composition can provide semantically equivalent information within the required temporal bounds.

When scheduling services, for each service the following two cases have to be distinguished:

- *Service not active:* The service has not yet been activated and the corresponding component has to be instantiated at a computational node. During scheduling, the scheduler has to find a computational node, to which the service is assigned, with sufficient free computational resources as well as required local devices (e.g., sensors, actuators, IO devices). The service is then temporally aligned with the other services required for the service composition, in order to optimize the temporal validity of information processed

by the service composition. The component for the transfer service is scheduled such that it starts execution when all data from input services is available.

- *Service active*: The service is already activated and provides input to other services. In order to prevent that other services are affected by rescheduling (which is important for certification) and to minimize the computational cost for scheduling, if possible, the timing of running services shall not be changed when a new service composition is installed. New services have to be temporally aligned with these running services.

To satisfy the precedence requirements of services, the scheduler must start to align the deadline of the transfer service to the services that require the output of the transfer service. Then the communication and the deadlines of the other services can be aligned to the execution start of the transfer service. Figure 6.9 illustrates the temporal alignment of three input services (i.e., $S1$, $S2$ and $S3$), the transfer service and a dependent service DS , which are distributed on three computational nodes. The deadline of the transfer service and the data communication are aligned to the start of execution of DS . Similarly, the deadlines for the services $S2$ and $S3$ are aligned to the execution start of the transfer service. As $S1$ and $S2$ are executed on the same node, the deadline for $S1$ is aligned to the start of $S2$.

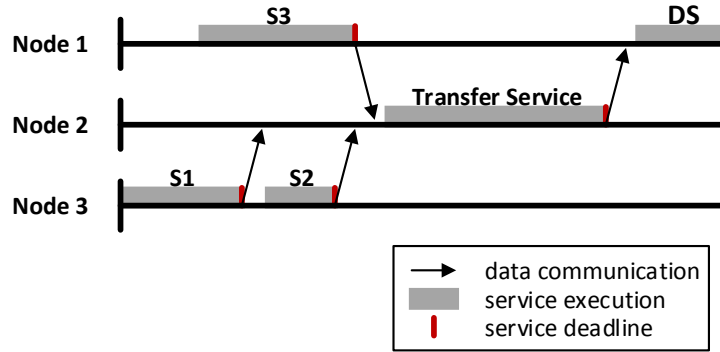


Figure 6.9: Temporally aligned services of service composition with existing unchanged service DS .

When the target nodes and schedules for the services and the communication between them have been determined, the service code needs to be migrated to the target node and the schedules have to be applied. The local CEMs on each computing node are responsible to receive the program code from the CCM and to periodically execute the service.

6.5 WCET of Service Orchestration

In order for the dynamic reconfiguration framework to be applicable for online reconfiguration of embedded real-time systems, the WCET of the reconfiguration process has to be determined.

Otherwise, if no bound for the WCET can be determined, it is impossible to guarantee that deadlines can be met and that reconfiguration is completed within the temporal constraints of the application. For instance, if the dynamic reconfiguration system is used to try to substitute a failed sensor service in a CPS before the system has to enter a safe state, the maximum time between the detection of a failure and the latest instant at which the system enters the safe state is of interest.

Compared to a CPS without dynamic reconfiguration capability, which comprises a failure detection mechanism that triggers the switch to a safe state or degraded mode, the temporal overhead of the following processes has to be considered for dynamically reconfigurable CPSs:

- Determination of required properties
- Composition search
- Transfer service generation
- Service and communication scheduling
- Deployment of new code and configuration of platform

Generally, the WCET of these processes highly depends on their actual implementation. However, a rough estimation of the expected computational complexity of the individual processes is presented in the following paragraphs. *Note:* If no valid service composition can be found, the system can switch to the safe state (in case of fail-safe systems) or a degraded mode (for fail-operational systems) immediately after the composition search, and hence, the rest of the processes do not prolong the time between the failure detection and entering the safe state or degraded mode.

Determination of required properties: The computational complexity of this process can be derived from Algorithm 2. This algorithm is either called to substitute a service that was removed from the system (i.e., due to a fault or intentionally), or after a new service is integrated to enable services that were blocked due to missing input properties. In both cases, for each of the active services in the system, all of the input properties have to be tested. But in the case of substitution, this test has to be performed as many times as the number of output properties of the removed service. Hence, for S active services in the system, where each service has at most P_i input, and P_o output properties, the computational complexity is given as follows:

$$\mathcal{O}(S \cdot P_i \cdot P_o) \quad (6.1)$$

However, if for each service a list of dependent other services is maintained, as well as a list for all properties required to activate a blocked service, the actual online effort to determine the required properties is constant (i.e., $\mathcal{O}(1)$). The effort for the mentioned algorithm is then shifted from the critical phase between the detection of a fault and entering the safe state, to a maintenance task outside this critical path.

Composition search: Intuitively, composition search strongly depends on the size and complexity (i.e., number of relations between concepts) of the system ontology, as well as the choice of the search algorithm. Finding service compositions in the system ontology can be reduced to reachability and depth-first search problems in directed graphs, which can be performed in linear time [108].

Generally, in order to find a composition, each property concept in the ontology is visited at most as often as it appears in the input relations of any transfer concept in the ontology, plus the number of *isA* relations between property concepts. In the worst case it has to be checked for each transfer concept if it can be used for service composition, which means that all of the input relations of a transfer concept have to be tested for provision. This is done by checking the provision of the property concept that is connected to that input relation. This might also include a check for provision of the child concept of that property concept. If the property concept is not provided directly by some service, the provision of this property concept has to be evaluated only when it is visited the first time. For each subsequent encounter of this property concept, the ability to provide this concept is already known. This implies, that each transfer concept is visited only once, and thus, the same applies for each input relation to a transfer concept. As a consequence, an algorithmic complexity of $\mathcal{O}(N + M)$ is given, where N is the total number of input relations of transfer concepts and M is the total number of child relations due to *isA* inheritance. From this reasoning follows, that the complexity only depends on the size and structure of the ontology, but not on the available provided services.

Algorithm 3 demonstrates that a simple depth first search technique can be used to achieve a linear runtime complexity. However, with an efficient implementation for the selection of the next concept to be explored (e.g., using queues), also breadth first and guided search can be performed in linear time.

Transfer service generation: During the generation of transfer services the recursive function is called exactly once for each transfer concept in the composition tree and an input code is generated for the input services. Therefore, the algorithmic complexity of the algorithm is $\mathcal{O}(T + S)$, where T is the number of transfer concepts in the composition tree and S the number of input services.

If no restrictions on the semantic depth and/or semantic width of a composition tree are given during composition search, in the worst-case all transfer concepts of the system ontology, as well as all available services in the system contribute to a service composition.

Service and communication scheduling: The complexity of the service and communication scheduling algorithm strongly depends on the actual implementation of the schedulers. When searching for an optimal schedule for multiple services on multiple computational nodes [23, 70, 92], the scheduling problem is NP-hard. Different schedulability tests, often based on processor utilization [10, 75], and various complexity evaluations of scheduling algorithms (e.g., [16, 100]) have been published.

Fortunately, for the dynamic reconfiguration complex scheduling algorithms can be avoided. As services that are already active shall not be influenced by a new service composition, only services belonging to the new composition, and which are currently not active, have to be sched-

uled. Given that enough computational resources are available in the system, the only restriction on the service schedules is that the transfer service has to be scheduled after it received its input from the other services. Therefore, a simple first fit scheduler can be used that first assigns a time slot for the input services, and afterwards the transfer service is placed after the receive instant of the last input information.

Finally, the reconfiguration system can be restricted to only allow services for a service composition, which are already active, or for all services in the system that are ready, an allocation to a computational node and an execution time slot are assigned in advance, which must not be changed. This is important, for instance, for safety-relevant applications, where the schedules of services and communication are certified and must not be altered at runtime. In this case the effort for scheduling is reduced to the search of a feasible allocation of a node and time slots (i.e., for execution and communication) for the transfer service. This effort includes the determination of the start instant for the transfer service, which can be started after all input information is available. The start instant is obtained by finding the last deadline at which the information of an input service of the service composition arrives. This is done by comparing the predefined deadlines of all services of a service composition and selecting the highest arrival time. Resulting in a complexity of $\mathcal{O}(S + C)$, where S is the number of services contributing to the composition and C is the number of computational nodes.

Deployment of new code and configuration of platform: The deployment of service compositions involves sending the service program code and the schedule for each individual component to its corresponding computational node. It is assumed that no additional startup time for individual resources (like sensors or other peripherals) is required, and hence, the same effort is needed for each service. The validity of this assumption can be enforced by keeping all services active, which interact with such a resource. For D services that have to be deployed, the total effort is $\mathcal{O}(D)$.

When optimizing the reconfiguration system for a minimized WCET of the reconfiguration process, the above steps – except for the first one – have to be considered for each of the R individual property concepts that are missing. An estimation of the resulting computational complexity is given by the following equation:

$$\mathcal{O}(1 + R(N + M + T + S + C + D)) \quad (6.2)$$

R	...	number of required property concepts
N	...	total number of input relations to transfer concepts in system ontology
M	...	number of <i>isA</i> relations in system ontology
T	...	maximum number of transfer concepts in composition tree
S	...	maximum number of input services for service compositions
C	...	number of computational nodes
D	...	maximum number of services that have to be deployed

Uncertainty of Service Composition

The proposed composition search technique is intended to be applicable for monitoring and control of CPSs. Obviously, in order to achieve the objective of monitoring and controlling such systems, it is necessary that the input values to the management applications are accurate real-time images of system states. While the methodology presented in the previous chapter is capable of identifying semantically equivalent sources of information in the system, no statement about the quality (i.e., accuracy or uncertainty) of the information obtained by a service composition has been made so far. Clearly, the quality of different solutions strongly depends on the accuracy of the input values provided by services, as well as the precision of the functions used for the transfer service. When a service composition has to be used as input for a control application of a CPS, the value accuracy of the output of the transfer service must meet the requirements of the controller, as otherwise the CPS might not be operated correctly.

For this purpose, this chapter elaborates on a quality measure for service compositions that is based on the worst-case uncertainty of the values deduced by service compositions. This quality measure can automatically be calculated from the transfer functions and input services involved in a service composition, and afterwards be compared with the uncertainty constraints of the services that use the composition as input. Only if the worst-case uncertainty is within the allowed range, the deployment of the service composition can be accomplished.

The first section starts to explain the model of uncertainty and which sources of uncertainty have to be considered. Section 7.2 describes how uncertainty at the inputs of a transfer service propagates to its output and how the worst-case output uncertainty can be calculated for a service composition. At last, in Section 7.3 the improvement of the worst-case uncertainty, by combining composition trees, is investigated.

7.1 Modeling Uncertainty

This section presents the quality metric for service compositions based on the concept of uncertainty. During the creation of a real-time image of a system entity (i.e., measurement or state

estimation), deviations of the real-time image from the true value of the represented entity are introduced. These deviations either are of systematic or random nature. Similar to the procedures described in the GUM [44], within this work it is assumed that all measures have already been taken to compensate for systematic errors during the creation of real-time images. Hence, the quality measure only considers random errors including those systematic errors which cannot be compensated with reasonable techniques.

7.1.1 Uncertainty of Composition Trees

In general, transfer concepts in a composition tree can involve continuous and discrete properties of the system ontology. The model of uncertainty within this chapter only concerns continuous properties. For the sake of simplicity, discrete properties (e.g., the state of the clutch in a car, or the selected gear) are assumed to have no uncertainty and they always provide an accurate representation of the measurand. This assumption is valid for many technical systems, where the discrete property can directly be measured by a sensor, which ensures that no forbidden logic states are given. However, in future work the assumption of perfect discrete property values will be relaxed.

In order to model the worst-case uncertainty of a composition tree, the deviation of the measured value of the measurand from its real value is represented by using a random variable X . A probability density function (PDF) is used to assign a probability to each possible value of X , which denotes the probability that the deviation from the true value of the measurand equals this value. In this work, a normal (i.e., Gaussian) distribution [93] for uncertainties has been chosen, because this distribution adequately describes many physical systems and it is a commonly used model that is also favoured by the GUM.

The normal distribution $N(\mu, \sigma)$ is characterized by the two parameters: *mean* $\mu = E(X)$ and the *standard deviation* σ . The mean denotes the expected value of the random variable X , while the standard deviation defines the region around the mean, for which the probability of X having a value inside this region is 68.2%. The higher the standard deviation is, the higher is the probability of X having a value farther away from the mean value. Equation 7.1 presents the PDF of the normal distribution.

$$y = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (7.1)$$

In Figure 7.1 the curve corresponding to the PDF of the normal distribution is shown. A higher standard deviation σ results in a wider shape of the curve, and thus, a higher uncertainty. When modeling uncertainty within this work, the mean value μ – representing the mean error of a measurement or composition tree – is assumed to be 0. This implies, that if the service composition is executed repeatedly while the system does not change at all, the output value of the composition varies with a certain probability (given by σ), but on average the deviation from the true value is 0.

An additional measure of the spread of the random variable X is the *variance* of X – usually denoted by $Var(X)$. It is the expectation about the squared deviation from the mean $E[(X - \mu)^2]$, which is also defined as the squared standard deviation: $Var(X) = \sigma^2$.

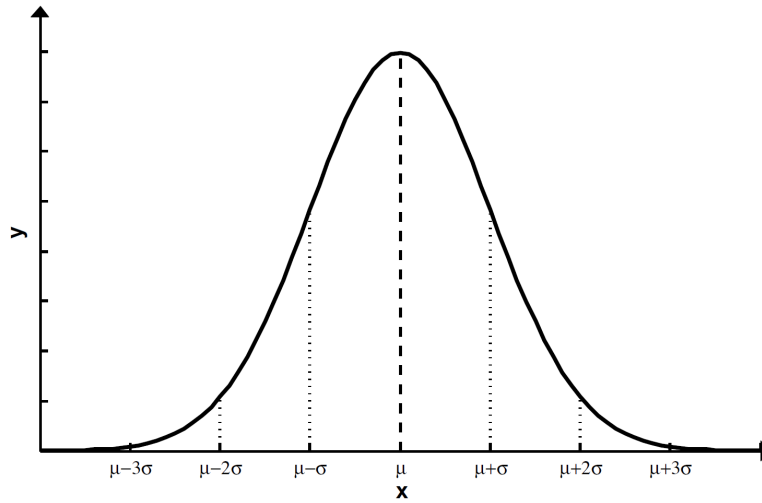


Figure 7.1: Probability distribution function of *Normal Distribution*.

If not stated otherwise, in the remainder of this work the standard deviation σ will be used as the measure of uncertainty.

For modeling the uncertainty of composition trees in dynamically reconfigurable systems, two sources of uncertainty are of interest:

- *measurement uncertainty* originating from sensors or computing nodes,
- *process uncertainty* introduced by inaccuracies in the system models.

7.1.2 Measurement Uncertainty

A measurement uncertainty – or measurement noise – is an uncertainty about the value of the measurand. It is introduced by the measurement instrument (i.e., the sensor), or in case the measurand is determined by a measurement function, it includes uncertainties about the input values of the measurement function that manifest in the output of the function. Such uncertainties can have various reasons, which are typically indeterministic and might lead to different values of subsequent measurements, even if in reality the measurand did not change. Therefore, no correction mechanism that is based on a single measurement can be applied. Examples of sources of measurement uncertainties include:

- Imprecisions in the definition of the measurand (e.g., if the temperature of a liquid for a volume measurement is not defined), measurement methodology and measurement setup.
- Discretization errors due to limited resolution of the measurement device.
- Environmental disturbances.
- Latencies and jitter in communication of the measurement value.

Since the assumption is that all systematic errors have been corrected, the mean value of the uncertainty is $\mu = 0$. This means, that if a sequence of n measurements of a constant measurand is conducted, the sum of positive and negative measurement errors converges towards 0, when n approaches ∞ . It should be noted, that the above assumption implies, that the measurement noise is statistically independent from the actual value of the measurand, since such correlations must have been compensated according to the assumption. The standard deviation σ and its square value (i.e., the variance) are the effective representation of the uncertainty in measurement, as they define the variability of the measurements, when the measurand remains constant.

For many sensors an uncertainty value is given in the data sheet. Even if different formalisms of representing uncertainty are applied (e.g., providing uncertainty intervals, percentage values for maximum errors or standard deviations), these values can usually be adapted such that they meet the requirements within this work. In cases where no such values are available, a method to determine the sensor's uncertainty experimentally is to perform a series of measurements with predefined measurand values. Afterwards, the variance of the measurement error

$$Var(X) = E[(X - \mu_p)^2]$$

is calculated, where μ_p is the predefined value of the measurand. Besides these common methods, the GUM suggests further possibilities to obtain the uncertainty of a sensor.

7.1.3 Process Uncertainty

In contrast to measurement uncertainty, process uncertainty – or process noise – originates from imprecise definitions of system (or process) models. This could be, for instance, intentional imprecisions where higher orders in equations are neglected as they do not improve the system model significantly, or environmental influences to the system (e.g., temperature effects), which are neglected or cannot be determined exactly.

In this work, process uncertainty is treated similarly as it is handled in state estimation techniques, like the Kalman Filters [47], where imprecisions in the process model impact the quality of the estimation of the next state. This means, that the uncertainty of any composition tree does not only depend on the uncertainty of the input values for system models, but also on the accuracy of the system model itself. As it is typical for state estimation techniques [98, 114], the process noise is introduced as the random variable $w \sim N(0, Q)$, which is normally distributed with the mean 0 and covariance Q . The zero mean is based on the assumption that disturbances leading to nonzero mean are handled within the system model itself.

The process uncertainty is considered to be a purely additive term w.r.t. the system model equations. Equation 7.2 formalizes the additive nature of process noise using a system model function f . This function calculates the next state x_n , based on the old state x_{n-1} and the current input u_n . In case of a state-less model – which applies for functions performing pure input-output transformations – only the vector u_n is of interest. Similarly, functions that only use the old state without input can be used. The process noise w_n is an indeterministic component that has to be determined (e.g., experimentally) when developing the system model, then it is added to the function value of f .

$$\mathbf{x}_n = f[\mathbf{x}_{n-1}, \mathbf{u}_n] + \mathbf{w}_n \quad (7.2)$$

For simplicity reasons it is assumed, that the process noise is statistically independent from the value of the measurand, as well as from the measurement noise.

7.2 Propagation of Uncertainty

Generally, the worst-case uncertainty of a composition tree depends on the uncertainty of the services – in many cases sensor services – at the leafs of a composition tree and the transfer functions that combine the output values of the individual services to obtain the required property concept value. Using the terminology of the GUM, these transfer functions are referred to as measurement functions. A measurement function

$$Z = f(X_1, \dots, X_N) \quad (7.3)$$

uses N random variables (X_1, \dots, X_N) as input values to produce the output value Z , which is a random variable itself. For the sake of simplicity, we assume that the input variables of the measurement function are statistically independent. Different publications related to uncertainty of measurement functions (e.g., [31]) already provide approaches for relaxing this assumption. It will be part of future work to extend this work with an appropriate method.

7.2.1 Measure of Sensitivity

The influence of the uncertainty σ_{x_i} of each individual input quantity X_i onto the uncertainty σ_z of the output value Z strongly depends on the measurement function f . Figure 7.2 illustrates the influence of the measurement function on the propagation of uncertainty with an arbitrary, partially linear function $f(x)$ that has a single input quantity x . The figure shows two different inputs $X \sim N(\mu_x, \sigma_x)$ and $Y \sim N(\mu_y, \sigma_y)$ for function $f(x)$, with $\mu_x \neq \mu_y$ and $\sigma_x = \sigma_y$. In case variable X is the current input, the output is $Z_X \sim N(\mu_{zx}, \sigma_{zx})$. Otherwise, when Y is applied to f , its result is $Z_Y \sim N(\mu_{zy}, \sigma_{zy})$.

It can be easily seen from the example in the figure, that after the input has been transformed by the measurement function, also the standard deviations of both random variables are transformed accordingly, such that $\sigma_{zx} < \sigma_{zy}$. In the simple case, where the measurement function is linear around the mean μ of the input value, the transformation of the uncertainty depends on the first order derivative $\frac{\delta f}{\delta x}(\mu) = f'(\mu)$ of f at the position of the mean value μ .

Equation 7.4 formalizes the relation between input uncertainty σ_x , first order derivative f' and output uncertainty σ_z for a linear measurement function with a single input quantity.

$$\sigma_z^2 = \left(\frac{\delta f}{\delta x} \right)^2 * \sigma_x^2 \quad (7.4)$$

Here the general rule for scaling the variance of a random variable X with the constant a , as presented in Equation 7.5, is used.

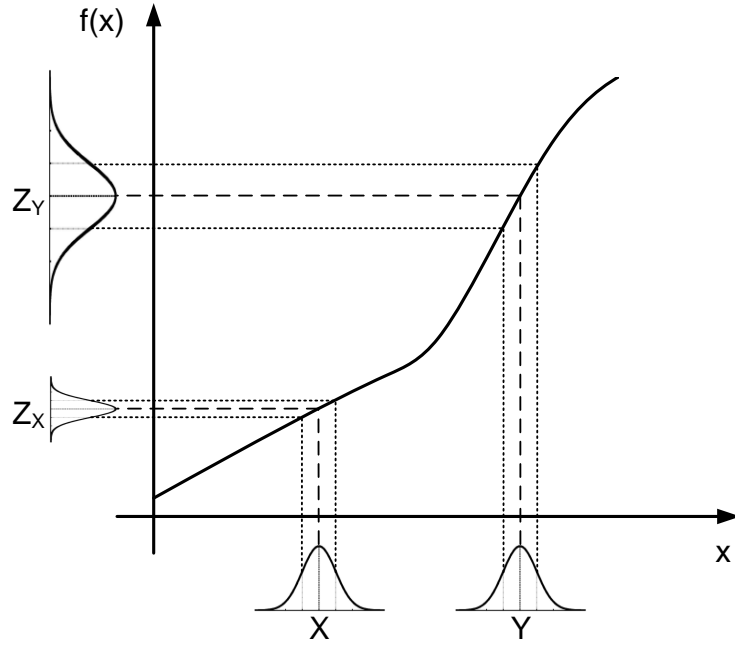


Figure 7.2: Influence of measurement function on propagation of uncertainty.

$$Var(aX) = a^2 Var(X) \quad (7.5)$$

As a dedicated topic in research and engineering, sensitivity analysis [3, 101] is concerned with the assignment of contribution factors – also called sensitivity indices – to input quantities of a measurement function, depending on their impact on the uncertainty of the function’s output quantity. Distinct measures for sensitivity of measurement functions have been proposed, like variance based methods [101], variance gradients [15] or regression analysis [102]. Due to its simplicity, one of the most common techniques following the law of propagation of uncertainties is based on partial derivatives, which is also an approach proposed by the GUM.

According to the law of propagation of uncertainty [64], the variance of a measurement function can be calculated, as expressed in Equation 7.6, from the partial derivatives of the measurement function and the uncertainties of the corresponding input quantities.

$$\sigma_z^2 = \left(\frac{\delta f}{\delta x_1} \sigma_{x_1} \right)^2 + \left(\frac{\delta f}{\delta x_2} \sigma_{x_2} \right)^2 + \dots + \left(\frac{\delta f}{\delta x_N} \sigma_{x_N} \right)^2 \quad (7.6)$$

The first supplement to the GUM [45, Appendix B] suggests, that the sensitivity indices for a measurement function can be determined by holding all but one inputs of the function fixed, and performing a Monte Carlo approach with the one input quantity, for which the sensitivity index should be determined. This is also known as *One-factor-at-a-time (OFAT)* method, which belongs to the group of *local* methods, since the sensitivity is evaluated at fixed points of the input space. In contrast, *global* sensitivity methods (e.g., variance based methods) examine

the entire input space to obtain a global sensitivity index. Therefore, obtaining global indices requires a computational effort, which can be orders of magnitude higher than for local methods. But on the other hand, global indices can be used over the whole range of input values for an input quantity, while local indices are only valid if the measurement function is used with a certain set of input values.

7.2.2 Determination of Sensitivity Indices

In order to determine the worst-case value uncertainty of a composition tree, different transfer functions – where each of them can be seen as its own measurement function – have to be concatenated to obtain a global measurement function for the composition tree. As the worst-case bound of a composition tree has to be valid for the total range of each input quantity, global sensitivity indices are required. Additionally, since the composition tree is determined at runtime, no global sensitivity analysis for the whole tree can be determined in advance. Such an analysis is only possible for the individual transfer functions in the system ontology. The sensitivity indices of the transfer functions that form the composition tree, together with the uncertainty values of the input quantities at the leafs of the composition tree, can then be combined to calculate the worst-case uncertainty of the whole tree.

Suitable single indices for the determination of the worst-case uncertainty are the absolute maximum values of the partial derivatives for each input quantity. The usage of absolute maxima is based on the fact, that the contribution of partial derivatives to the uncertainty is quadratic (cf. Equation 7.6), and the comparison of the sensitivity of each input quantity is facilitated if only positive indices are used. In the simple case of a linear model, the partial derivatives will be constants, which equal the sensitivity indices. For non-linear models, the maximum gradient of the function can be calculated analytically by finding the maximum in the first order derivative, or the whole input space of the function has to be examined numerically to find the indices. A simple method to obtain the required global sensitivity indices is to apply the OFAT method not only for a certain set of values for the input quantities, but to run the measurement function repeatedly until all possible input values of the input space have been tried. Thereby, the search can be reduced to those input values that are technically feasible. For instance, since a normal car will not be able to drive faster than 300km/h , a function using the car's velocity as input need not be tested with higher values.

The problem with a single sensitivity index for each input quantity of non-linear transfer functions is that the resulting worst-case bound will be very conservative. Especially, in cases where two non-linear transfer functions are concatenated such, that regions with high gradients within one function are matched with regions of the other function, which have very low gradients. This is illustrated in Figure 7.3 with the transfer functions e^x and $\ln(x)$, and their combination $\ln(e^x)$.

The graphs (a) and (c) in Figure 7.3 show the $\ln(x)$ function and its corresponding derivative $1/x$ in the range $0 \leq x \leq 10$. Obviously, the gradient of the \ln function has its highest value in the proximity of $x = 0$, where

$$\lim_{x \rightarrow 0} \frac{1}{x} = +\infty \quad \text{and} \quad \lim_{x \rightarrow \infty} \frac{1}{x} = 0$$

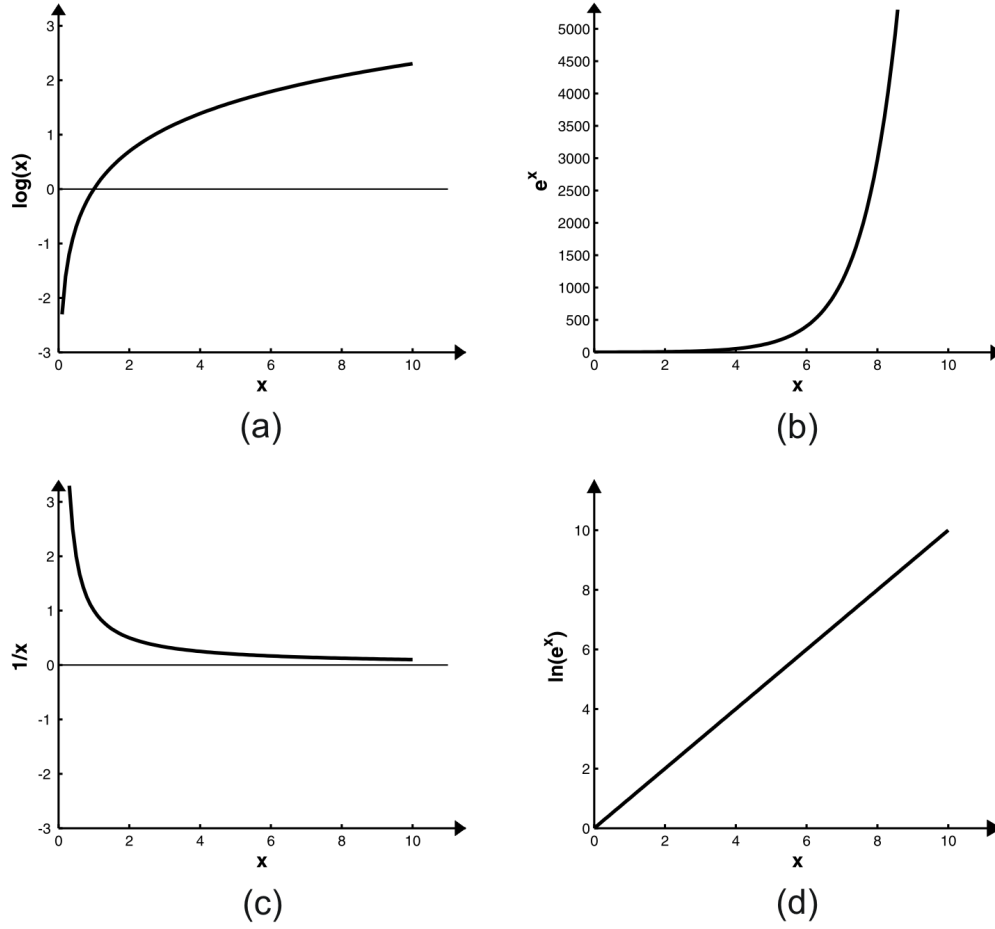


Figure 7.3: Different transfer functions with single input quantity: (a) $\ln(x)$, (b) e^x , (c) first order derivative of $\ln(x)$, (d) concatenation of e^x and $\ln(x)$.

The above methodology to determine the worst-case sensitivity index of the function will identify $+\infty$ at input value $x = 0$ as the value for the index. Since this is the only global index, this value also applies for all other input values of the function. This is particularly the case for the opposite side of the function, where $x = \infty$ and the gradient converges to 0. As the worst-case uncertainty is required, this might be reasonable for a single function. But if a second non-linear function (e.g., e^x as shown in Figure 7.3 (b)) is combined, which compensates – or at least reduces – the non-linearity of the first function, the uncertainty will be massively overestimated. The combined function $\ln(e^x)$ (as depicted in Figure 7.3 (d)) is a linear function with the constant gradient 1. Due to the fact, that the exponential function e^x – its derivative is also the function e^x – has a gradient function that is exactly the opposite of the gradient of $\ln(x)$, the non-linearity is dissolved. But in contrast, the isolated sensitivity analysis for e^x also results in a sensitivity index of $+\infty$, and finally, the combined uncertainty for $\ln(e^x)$ is evaluated to be $+\infty$.

One solution to that problem is combining both functions to get a single transfer function, and then the sensitivity indices are determined for the new function (cf. merging of transfer concepts in Section 5.3.2). This can reasonably be done as an offline preprocessing step for sub-trees in the system ontology, where no alternative combinations of transfer concepts can be created. For all other trees, this reduces the flexibility when searching for a service composition.

A trade-off between a single sensitivity index for each input quantity and fully matching two functions (online, while searching a composition tree that achieves the required worst-case uncertainty) is, to create several intervals on the total range of the input quantity. Afterwards, for each interval the sensitivity index is determined. Apparently, the optimum worst-case bound can be found, when infinitely many intervals are available that exactly represent the shape of the derivative of the transfer function. This would be equal to analyzing the combined transfer function. But unfortunately, it is computationally infeasible to evaluate as many sensitivity intervals within a short time (e.g., during the substitution of a failed service in the system). Thus, a trade-off between computational effort and tightness of the worst-case bound, has to be found.

7.2.3 Selection of Sensitivity Intervals

Basically, the size of each individual sensitivity interval and their distribution can be selected arbitrarily (e.g., using a uniform distribution or even different sizes for distinct intervals). Thereby it is possible to optimize the worst-case bound of the uncertainty in dedicated regions of the transfer function. But the price of a bad choice for the intervals is an overestimation of the worst-case bound. As a consequence, some composition trees might not be recognized as a valid composition, even though their real uncertainty lies within the required range. Therefore, depending on the shape of the function, a different number of intervals is required in order to achieve a reasonable approximation of the function. For instance, a linear function can be approximated by using a single interval, while for a highly non-linear function, more than a hundred intervals might be required. Random functions, like white noise, or hash functions cannot be modeled efficiently, as infinite sensitivity indices have to be assumed for every possible interval.

Figure 7.4 illustrates the partitioning of a transfer function with a single input quantity into distinct sensitivity intervals. Since the sensitivity indices have a quadratic contribution in the propagation of uncertainties (cf. Equation 7.5), the partitioning is based on the squared first order derivative of the transfer function. To obtain the sensitivity intervals, the squared derivative function is partitioned into equally sized intervals, that will be called ϵ -intervals. The size of these intervals determines the maximum quantization error ϵ_{max} , which can be seen as the maximum factor for worst-case bound overestimation. Those values on the x-axis, where the squared derivative function equals the border of an ϵ -interval, are used as the boundaries of the sensitivity intervals. The maximum derivative value within a sensitivity interval is its corresponding sensitivity index.

Transfer functions with more than one input quantity can be handled similarly. The only difference is, that the derivative function g for an input quantity x_i is given by the maximum value of the partial derivative, according to the following equation:

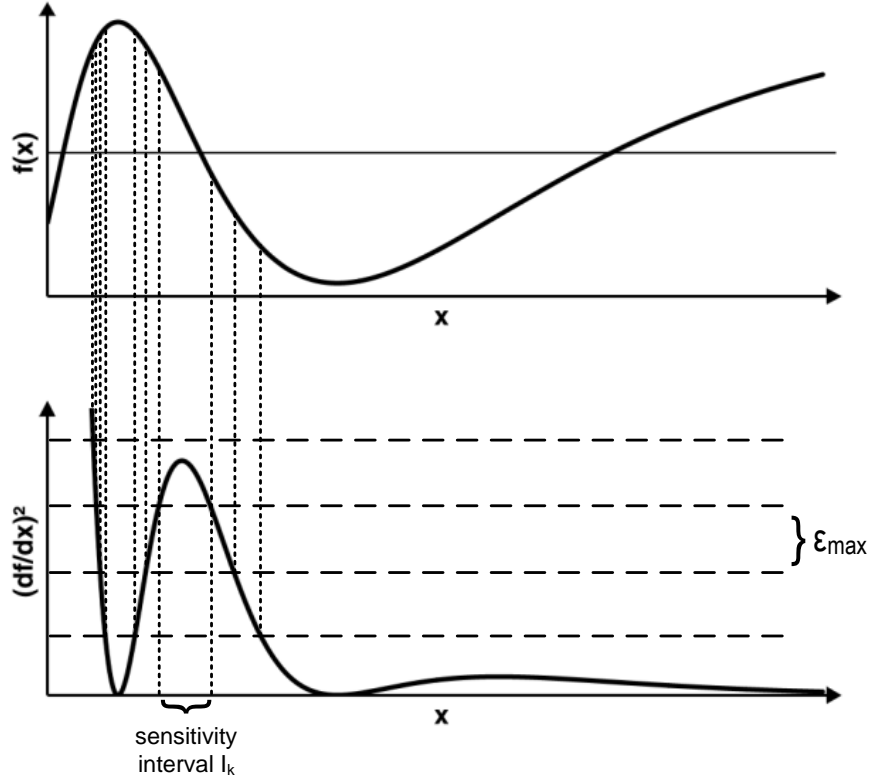


Figure 7.4: Splitting of a transfer function into sensitivity intervals based on the squared first order derivative.

$$\begin{aligned}
 g(x_i) &= \max \left(\frac{\delta f}{\delta x_i} (x_1, \dots, x_i, \dots, x_n)^2 \right) \\
 \forall x_k \in \mathbb{R} : x_{k_{min}} &\leq x_k \leq x_{k_{max}} \\
 k &= 1, \dots, i-1, i+1, \dots, n
 \end{aligned}
 \tag{7.7}$$

This means, that for each value of x_i , the derivative function is the maximum of the squared partial derivative, where the maximum is built by variation of all possible values for all input quantities except for x_i . This equals a search on the whole input space of the transfer function.

But not only the input quantity, for which the sensitivity indices are determined, can be discretized to obtain a better approximation of the transfer function. Also the output quantity is partitioned into intervals, which enables to calculate individual uncertainty values for different regions of the output quantity. These uncertainties are then used as input uncertainties for the next transfer function, and thus, the worst-case bound for the uncertainty is optimized in the distinct intervals. In the best case, the sensitivity intervals of the output quantity of a transfer concept equal the sensitivity intervals of the input quantity of another transfer function which both are related by a common property concept in the system ontology.

After the sensitivity intervals of an input quantity and the output quantity are defined, a matrix is obtained that contains the sensitivity indices to calculate the worst-case uncertainty bound. For linear transfer functions, the elements of the matrix either contain values that are equal to the slope of the function (i.e., its first order partial derivative, which is a constant value) or the element is zero. In case the number of input intervals of the transfer function equals the number of output intervals, non-zero values are only in the main diagonal of the matrix.

7.2.4 Propagation in Transfer Functions

Calculating the worst-case bound for the uncertainty of a transfer function requires one sensitivity table for each of the input quantities, as well as the corresponding standard deviation values σ as a measure for the uncertainty of the input quantity. In cases where the input quantity is directly determined by a sensor, the uncertainty values can either be taken from the datasheet, or they are evaluated experimentally. Additionally, the GUM provides several suggestions, how to get information about the uncertainty of sensors. In the other case, where the input quantity originates from another transfer function in the composition tree, the output of the worst-case uncertainty of that function is used as input uncertainty of the current calculation. As the initial input uncertainty can only be obtained from sensors or for constant system parameters, the propagation of uncertainty has to be started at the leafs of the composition tree.

The calculation of the worst-case bound for the uncertainty of a transfer function is depicted in Figure 7.5. The uncertainty values σ_j (with $j = 1, \dots, N$) of the N distinct input quantities are defined for certain intervals of the measurement domain of the corresponding j -th quantity (as described in the previous subsection). The number K_j of sensitivity intervals can be different for each of these input quantities, but still K_j has to match the number of columns of the sensitivity matrix corresponding to that input.

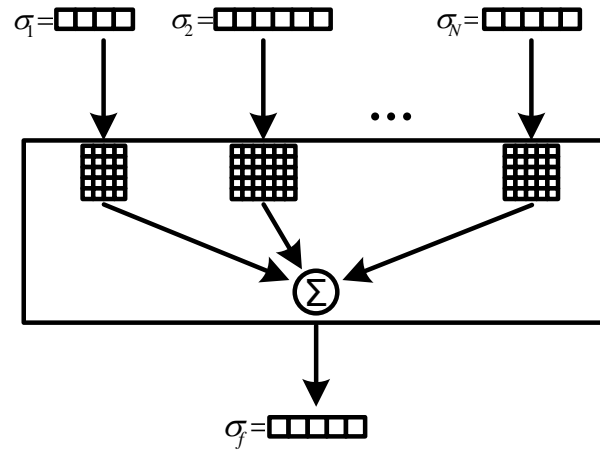


Figure 7.5: Calculation of worst-case uncertainty of a transfer function.

The output uncertainty σ_f of the transfer function f is also divided into different intervals, where the number of intervals M equals the number of rows of the sensitivity matrices for the

input quantities. Note that the number of rows is the same for all sensitivity matrices, and hence, the dimensions of the j different sensitivity matrices is $M \times K_j$. Equation 7.8 presents, how the output uncertainty σ_f is calculated. For each output interval i , the variance (i.e., $\sigma_{f,i}^2$) is calculated by summing up the distinct maximum contributions to the uncertainty of each of the N input quantities. In the equation,

- $i \dots$ is the index of the output interval, which uncertainty is currently calculated,
- $j \dots$ is the index for the input quantities, and
- $k \dots$ is the index for the input interval of the input uncertainty vector of an input quantity.

Because within each combination of input-output intervals – represented by the elements of the sensitivity matrix – the transfer function is interpreted as linear, and the partial derivative of the linearized function (i.e., the sensitivity index) is a constant, the contribution of an input quantity to the output variance is obtained by multiplying the input's variance with the corresponding squared sensitivity index (see Equations 7.4 - 7.6) – which is denoted as ∇^2 . This has to be done for all possible K_j sensitivity intervals of the input. Since at each point in time the input quantity can only have one certain measurement value, only the maximum of all possible contributions is of interest for the output uncertainty of the transfer function. Finally, a constant term θ^2 for the uncertainty about the transfer function itself (i.e., the process noise) is added to the output uncertainty. This term subsumes all deviations of the model (i.e., the transfer function) from the real world process.

$$\sigma_{f,i}^2 = \sum_{j=1}^N \max_{1 \leq k \leq K_j} (\sigma_{jk}^2 \nabla_{jik}^2) + \theta^2 \quad (7.8)$$

The example in Figure 7.6 illustrates how the contribution of an input quantity to the uncertainty of the output quantity is calculated. For efficiency reasons, the input uncertainty is already given as a vector of variances σ_j^2 . The same applies for the sensitivity indices in the table, which are given as ∇_{ik}^2 values. For each row, a field in the input uncertainty vector is multiplied with the corresponding field in the current row of the matrix. Then the maximum of the results of the multiplications in a row is selected as the contribution for the uncertainty of the transfer function. The fields in the matrix that influence the uncertainty contribution are indicated by the gray background color. After the contributions to the output uncertainty of one input quantity are calculated, the same has to be done for all further input quantities. Finally, all individual contributions of the input quantities are summarized for each sensitivity interval i of the output uncertainty σ_f , to get the total uncertainty vector of the transfer function.

7.2.5 Handling of Discrete Signals

Until now, only the uncertainty of continuous signals was covered. In many cases, the model of the system requires connections in the system ontology that represent discrete signals, like gear selection or clutch state (i.e., open or closed) of a vehicle. As already discussed in Section 7.1.1, in the course of this work, discrete signals are considered as certain (i.e., no additional

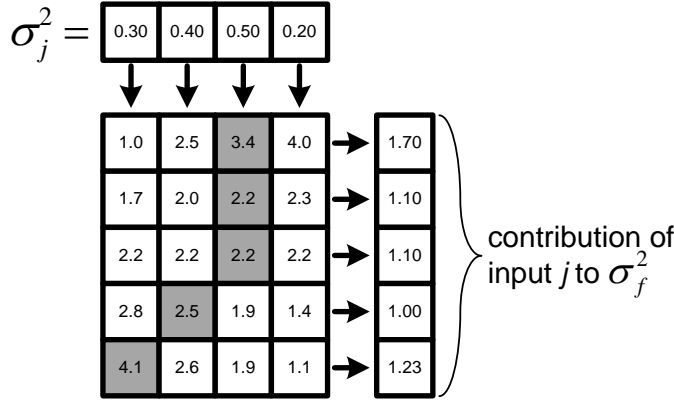


Figure 7.6: Example of calculation of worst-case contribution to uncertainty of an input quantity.

uncertainty is introduced by discrete signals). Hence, the output of system models (i.e., transfer concepts) that only use discrete signals is also assumed to have no uncertainty.

On the other hand, system models may exist where discrete and continuous signals are combined. These transfer concepts can be interpreted as hybrid systems [35]. In hybrid systems, discrete signals model the distinct modes of the system – or parts of the system. In each mode, the dynamic behaviour of the system can be different, and thus, the model for the continuous flow of system states – represented by continuous signals – differs between modes. For instance, the discrete gear selection signal of a vehicle specifies the ratio of the rotational speed of the drive shaft before and after the gear box. In case of the gear selection, the continuous flow models are simple constant factors, while in other cases differential equations will be necessary to describe the change of the system state.

Since the system model in the distinct modes might differ significantly, and hence, also the sensitivity indices, the uncertainty has to be evaluated for each mode individually by using Equation 7.8. As in each point in time only one of the modes can be active, only one mode at a time can contribute to the uncertainty of the composition tree. In order to calculate the worst-case uncertainty of a composition, for hybrid system models the highest uncertainty among all the different possible modes has to be selected for each individual interval.

7.3 Combining Composition Trees

In many cases, different solutions exist to calculate the individual property concepts within the composition tree, i.e., a node in the composition tree (representing a property concept) contains more than one incoming edges coming from a sub-tree with a valid composition. This means, that the composition search algorithm has to decide, which of the sub-trees results in a better composition. The decision can be based on various optimization criteria, implying that all required information to find the optimum is available in the system. Such optimization criteria could be, for instance, the worst-case uncertainty achievable by the sub-tree, computational and

resource requirements of the composition, measurement update rates of sensors, etc. Figure 7.7 shows an example of a composition tree where the property concept *curve radius* can be calculated by using two different equations and the corresponding input property concepts.

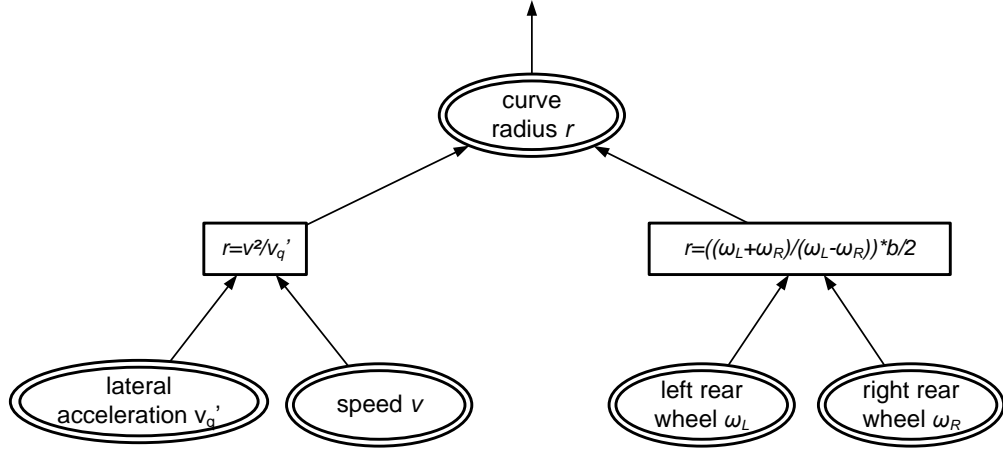


Figure 7.7: Example of composition tree where two distinct sub-trees can be used to calculate the value of the same property concept.

On the other hand, distinct solutions to calculate the value of the same property concept allow to reduce the worst-case uncertainty of a composition by fusing two, or more, sub-trees to obtain a single property concept. However, the improved uncertainty has to be paid by indirect redundant computation, and hence, higher computational and resource efforts. It depends on the overall optimization criteria, if the fusion of composition trees is feasible.

One way to fuse independent composition trees is, to take the average of the input values. In Equation 7.9 the relation between the input uncertainty of N distinct inputs and the uncertainty of the average value is given for the different range intervals i of the property concept. It can be seen, that the overall uncertainty decreases proportionally with the number of independent inputs, if all inputs have about the same uncertainty. Otherwise, when the uncertainty of an input is about N times larger than the lowest input uncertainty, the corresponding term will increase the overall uncertainty to a value that is higher than the best achievable uncertainty of a single input.

$$\sigma_i^2 = \sum_{j=1}^N \frac{\sigma_{ji}^2}{N^2} \quad (7.9)$$

To overcome the problem of increased overall uncertainty, weighted averaging can be used. Thereby, the input uncertainties are weighted and the overall output uncertainty is obtained by combining all weighted uncertainties. This procedure is illustrated in Figure 7.8.

The weights for each sub-tree have to be calculated based on the uncertainty of that tree. Since these uncertainties are defined for different intervals of the value range of the property concept, individual weights have to be defined for each interval. Equation 7.10 shows how the

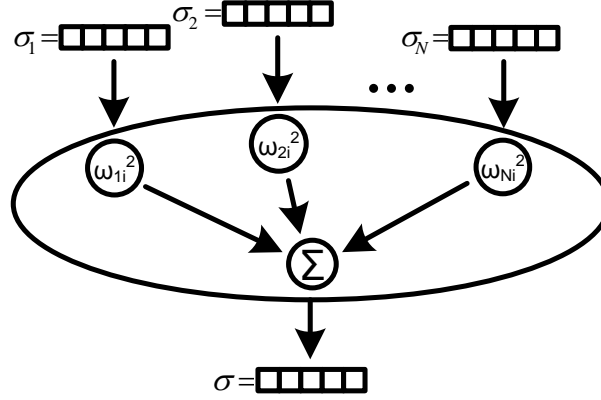


Figure 7.8: Calculation of worst-case uncertainties when applying weighted averaging.

weights ω_{ji} for composition tree fusion are calculated for each interval i of composition tree j , when N independent trees are fused. The higher the uncertainty of a composition tree, the lower is the weight for that tree. The formula ensures that the sum of the weights for an interval is normalized to 1. It also implies, that no input with zero uncertainty may exist, as otherwise the weight would be undefined.

$$\omega_{ji} = \frac{\frac{1}{\sigma_{ji}^2}}{\sum_{j=1}^N \frac{1}{\sigma_{ji}^2}} \quad (7.10)$$

The calculation of the uncertainty for the i^{th} interval that is applicable for the general case of any, but normalized, weights, is presented in Equation 7.11. The individual weighted input uncertainties are simply summed up. The equation is based on the common rules for variances:

- $Var(aX) = a^2 Var(X)$ – (scale by constant), and
- $Var(X + Y) = Var(X) + Var(Y)$ – (sum of random variables)

$$\sigma_i^2 = \sum_{j=1}^N \omega_{ji}^2 \sigma_{ji}^2 \quad (7.11)$$

If the weights are obtained by using Equation 7.10, then the combined calculation of the output uncertainty σ_i^2 can be simplified according to Equation 7.12. This formula also implies, that the output uncertainty is less than – or equal in case of a single input – to the smallest input uncertainty, which means that the output uncertainty is always an improvement compared to the input uncertainties.

$$\sigma_i^2 = \frac{1}{\sum_{j=1}^N \frac{1}{\sigma_{ji}^2}} \quad (7.12)$$

As a simple proof, the following discussion can be considered: Among the N possible inputs the one input $j \in N$ with the smallest uncertainty is chosen, such that

$$\sigma_{ji}^2 \leq \sigma_{ki}^2 \quad \forall k \in N$$

In case only this single input is used for the averaging algorithm, the output uncertainty is

$$\sigma_i^2 = \frac{1}{\frac{1}{\sigma_{ji}^2}} = \sigma_{ji}^2$$

If any further input $k \in N \setminus \{j\}$ is added, the value in the denominator increases, and thus, the corresponding output uncertainty decreases, provided that $\sigma_{ki}^2 < \infty$.

$$\sigma_i^2 = \frac{1}{\frac{1}{\sigma_{ji}^2} + \frac{1}{\sigma_{ki}^2}} < \sigma_{ji}^2$$

Note: A strong overestimation of the worst-case uncertainty of one input to the weighted averaging compared to other inputs will lead to a lower weight of that input than with a tighter bound. As a consequence, the weights might be suboptimal and a smaller uncertainty could be achieved with other weights.

Implementation

In this chapter the implementation of the central parts of the dynamic reconfiguration framework is documented. The implementations concern the components required to conduct the tasks of the CCM. Specifically, the system ontology, the service composition search and the calculation of the worst-case value uncertainty have been implemented. While the implementation of these building blocks is generic and does not depend on a specific platform, this chapter demonstrates the integration of the dynamic reconfiguration framework into a service-oriented platform running on a mobile robot.

The chapter starts with the specification of the selected ontology description language in Section 8.1. Then the implementation of the ontology preprocessor, which is responsible to load the ontology specification from input files and create the ontology graph in the system memory, as well as the resulting data structures are detailed in the same section. The service composition search that is operating on these data structures can be found in Section 8.2. In Section 8.3 the focus lies on the calculation of sensitivity indices and the determination of the worst-case uncertainty of service compositions. Finally, the integration of the dynamic reconfiguration framework into a demonstrator is presented in Section 8.4.

8.1 System Ontology

The system ontology is the main source of information in the dynamic reconfiguration framework to find semantic equivalences for system properties. This information is specified and structured using an ontology description language. The syntax and allowed definitions are detailed in this section. In order to make use of this knowledge, the input files are loaded by an ontology preprocessor creating an appropriate memory representation of the system ontology – referred to as ontology graph. This representation already contains all required connections between concepts, such that a fast traversal of the knowledge base by the composition search algorithm is facilitated. Hence, after the description language was presented, this section also provides insight into the data structures within the memory as well as the implemented mechanisms for ontology preprocessing.

8.1.1 Ontology Description Language

The content of the system ontology for the implementation of the reconfiguration framework is loaded from one or more ontology files, within which the concepts and relations are specified using a declarative description language. For this purpose the *Resource Description Framework (RDF)* with the serialization syntax *Turtle* (Terse RDF Triple Language) [77] is used, since this language is widely accepted and also used as a standard by the World Wide Web Consortium (W3C). Furthermore, tools for ontology editing and modeling are already available (e.g., Protégé¹). The Internet offers many detailed descriptions of the general RDF language². However, since not all possible statements are useful for the system ontology, this section only relates to statements that are actually interpreted by the ontology preprocessor.

Basic statements in RDF consist of the triple *subject*, *predicate* and *object*, which are separated by whitespace characters. The character '.' signals the end of each statement.

- **Subject:** A subject defines the resource that is described by the statement. In most statements these resources are concepts in the system ontology. The other cases comprise RDF specific statements to declare other resources (e.g., basic declarations about the allowed concepts and predicates).
- **Predicate:** Each predicate expresses a relationship between subject and object. Except for RDF specific predicates, these are translated to relations between ontology concepts (as defined in Section 5.1), or are used to assign properties to transfer concepts.
- **Object:** Beside RDF specific objects, an object is either a concept type, an already known ontology concept, or the value that is assigned to the property of a transfer concept (e.g., the transfer function).

Each of these three components is expressed by an RDF Uniform Resource Identifier (URI) reference, or in case of objects also literals can be used. An example URI from the automotive domain could be:

```
http://ti.tuwien.ac.at/reconfig/ontology/car#engine
```

As the permanent usage of the full URI references in RDF statements leads to bad readability of the ontology description files, two directives can be used to abbreviate these references. **@prefix** allows to declare a short prefix name that can be used instead of the long prefix name for URI references, which are repeated multiples times. Thus, namespaces for groups of ontology concepts can be defined. An URI reference is then utilized by writing the short prefix name followed by the ':' character and the resource suffix name. A prefix declaration for the above example could have the following form:

```
@prefix car: <http://ti.tuwien.ac.at/reconfig/ontology/car#>.
```

¹ Available at: <http://protege.stanford.edu/>

² For instance at: <http://www.w3.org/TeamSubmission/turtle/>

Whenever the resource of the above example has to be referenced, only the following qualified name is required instead of the full URI reference:

`car:engine`

The second possibility for abbreviation is the *@base* directive, which sets the namespace for the whole ontology file. Once this namespace has been defined, the qualified name of a resource starts with the ':' character followed by the resource suffix name. The following namespace definition is an example for an ontology file describing the powertrain of a car:

`@base <http://ti.tuwien.ac.at/reconfig/ontology/powertrain>.`

The standard ontology concept identifiers supported by the implementation are listed in Table 8.1. The possible predicate identifiers are given in Table 8.2, where the *domain* indicates the type of concepts that can be used as subject in combination with the predicate, and the *range* defines the type of concepts allowed as objects in an RDF statement together with that predicate.

Table 8.1: Supported concept identifiers.

Concept Identifier
<i>StructureConcept</i>
<i>PropertyConcept</i>
<i>TransferConcept</i>
<i>PTypeConcept</i>
<i>TransTypeConcept</i>

Table 8.2: Supported predicate identifiers.

Predicate Identifier	Domain	Range	Comments
<i>isA</i>	StructureConcept	StructureConcept	
<i>a</i>	StructureConcept	StructureConcept	Short version of <i>isA</i>
<i>isPartOf</i>	StructureConcept	StructureConcept	
<i>hasProperty</i>	StructureConcept	PropertyConcept	
<i>inputs</i>	TransferConcept TransTypeConcept	PropertyConcept PTypeConcept	List of input properties to transfer (type) concept
<i>output</i>	TransferConcept TransTypeConcept	PropertyConcept PTypeConcept	Output property of transfer (type) concept
<i>transfer</i>	TransferConcept TransTypeConcept		Transfer function for transfer (type) concept
<i>pType</i>	PropertyConcept	PTypeConcept	

These basic identifiers are specified in a terminology file, within which the namespace is defined as the URI prefix *http://ti.tuwien.ac.at/reconfig/term*. The following simple example RDF

statements demonstrate the usage of these identifiers by defining a new property concept *carWeight* and assigning it to the structure concept *car*. Due to the *@base* directive, the namespace is set for all resources starting with the ':' character, which includes the new property concept.

```
@prefix rot: <http://ti.tuwien.ac.at/reconfig/term#>.
@base <http://ti.tuwien.ac.at/reconfig/ontology/car>.

:carWeight    a                rot:PropertyConcept.
:car          rot:hasProperty   :carWeight.
```

In many cases the same concept appears several times as the subject of an RDF statement. For instance, when multiple properties are assigned to a structure concept. This might unnecessarily blow up the size of the description file and compromise the readability of the document. Therefore, statements concerning the same subject can be combined to a single statement with multiple predicates and objects. Instead of the '.' character, each triple – except for the last one – is finalized with the ';' character.

The following example RDF statement declares the structure concept of a *crankshaft*. This concept is part of the *engine* of a car and it possesses the properties *angularSpeed* and *angularPosition*. Each line in the statement specifies an individual triple, where the subject in the first triple (i.e., crankshaft) is also used as subjects for all further lines until the last line ends with a '.' character.

```
:crankshaft    a                rot:StructureConcept;
                rot:isPartOf     :engine;
                rot:hasProperty   :angularSpeed;
                rot:hasProperty   :angularPosition.
```

As property concepts of different structure concepts might have the same name – especially in the case of inheritance of properties – property concepts are usually addressed via the corresponding structure concept using the '#' character as separator (e.g., *crankshaft#angularSpeed*).

Table 8.2 indicates, that the predicate *inputs* requests a list of input properties to the transfer function of a transfer concept as the object of the statement. The properties of this list have to be declared in parenthesis with one property in each line.

In the next RDF statement a transfer concept is declared, the transfer function of which calculates the *curveRadius* of the vehicle from the *trackWidth* and the *angularSpeed* of the left and the right wheel. These property concepts are referred to in the transfer function as \$0 – \$3, with \$0 being the output concept and \$1 – \$3 are the input properties in the order as they appear in the input declaration list. The transfer function itself is expressed in the target programming language of the transfer service.

```
:_t1_1         a                rot:TransferConcept;
                rot:inputs       (
                                :undercarriage#trackWidth
                                :leftWheel#angularSpeed
                                :rightWheel#angularSpeed
                                );
```

```
rot : output      : undercarriage#curveRadius ;  
rot : transfer    "$0 = (($2 + $3)/($2 - $3)) * ($1 / 2)".
```

8.1.2 Data Structures in Memory

The data structures for the ontology graph in memory have a strong influence on the efficiency and runtime of the composition search. A trade-off between memory requirements and fast access of related concepts has to be found. For this implementation the focus was put on a fast search, which requires that relations between concepts are stored in both directions, at the expense of higher memory usage. Since the implementation of the search algorithm is based on the *ANSI C* programming language, the ontology graph is built with linked lists of concepts and lists of pointers to realize the relations between these concepts.

Some of these structures are mainly necessary for the management of concepts – e.g., to find referenced concepts while the ontology graph is built, or to clean up the ontology graph – whereas other structures actually model the relations between concepts, and hence, are used during composition search. Since the ontology graph contains static information about the system which rarely changes, in order to reduce the memory utilization, the management data structures – including structure concepts – can be removed from the online representation of the ontology graph.

Management Structures

The top-level structure of the implementation contains a reference to the whole ontology graph that consists of a list of namespaces, which themselves comprise individual lists for the distinct concept types. These structures are depicted in Figure 8.1. Even though the concepts are connected in the lists, no relationship between these concepts is assumed, except that they use a common namespace. Dedicated data structures for these concepts store the actual information about the concepts and their relations.

Relation Structures

The relations between ontology concepts are modeled by lists of pointers to the connected concepts. For every type of relation a different list exists, which improves the speed of traversing the ontology. Due to different types of allowed relations between concepts, the content of the distinct concept structures (represented by the information fields in Figure 8.1) includes different relation lists. However, the organization of these lists is the same for all concepts, and hence, only details about the property concept lists are presented as a placeholder for all other types of concepts.

The relation lists of property concepts are schematized in Figure 8.2. The concept structure and links between the property concepts on the left side correspond to the list of property concepts in the previous figure. Beside the postfix name of the property, a namespace and a structure concept are associated with each property concept in order to be able to uniquely identify the concept. The linked structure concept models the inverse direction of the *has-property* relation. Another element of the structure is holding the property type which might be assigned to the

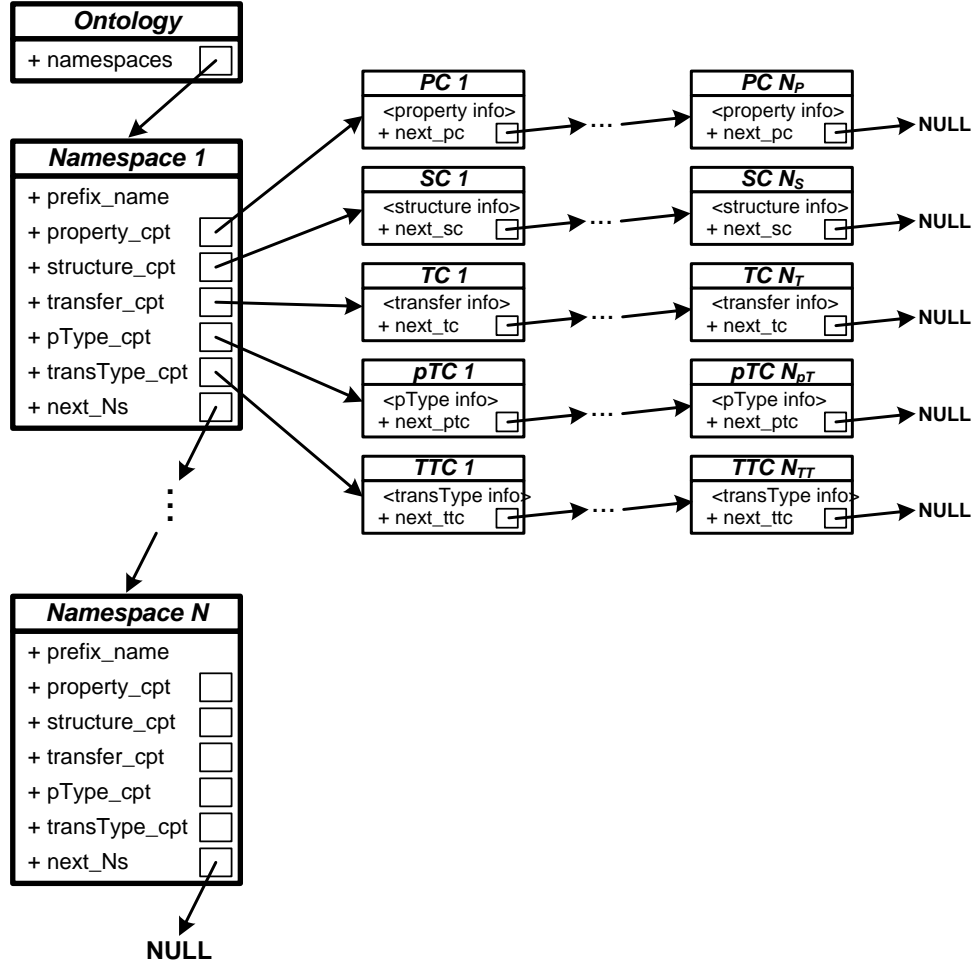


Figure 8.1: Data structures for management of ontology concepts.

property concepts. The remaining relations between concepts are organized in linked lists. Such lists exist, for instance, for the *parents* and *children* of property concepts, which represent *isA* relations inherited from structure concepts during ontology preprocessing. This means, that because of an *isA* relation between structure concepts, the property concepts of the higher level structure concept are copied to the lower level structure concept, and also a parent-child relation is added between the properties. The elements in the list only contain a pointer to the next list element and a pointer to the actually related property concept structure. The latter points to a property concept that already appears in the management structure, but there the list represents an orthogonal view point. A similar list exists to store every transfer concept for which the property concept acts as input, and another one to hold transfer concepts, which output the value of the given property concept.

While the presented linked lists represent the static information about property relations, dynamic information about the provision of the property by services or by derivation from transfer

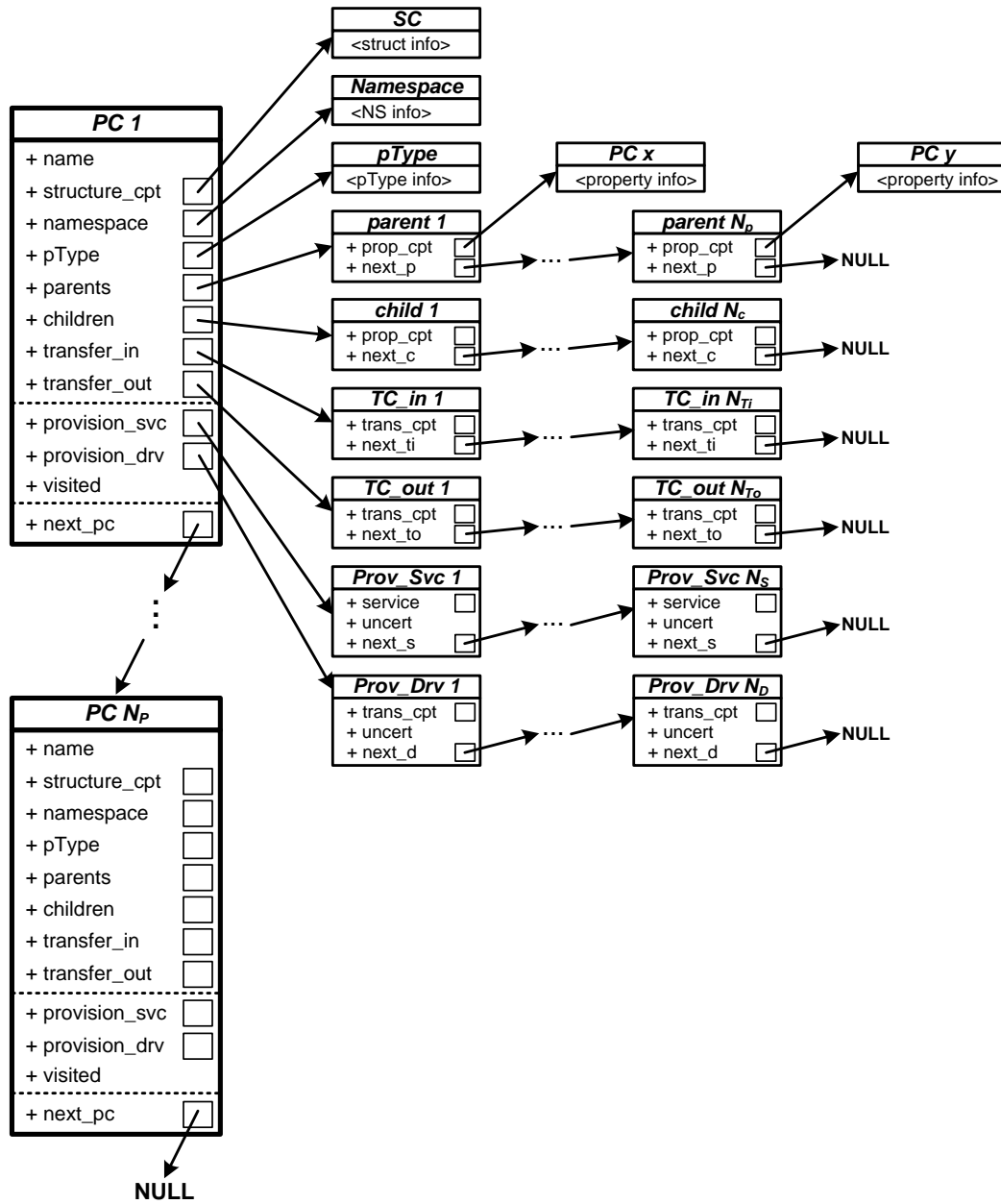


Figure 8.2: Data structures modeling relations of property concepts in ontology graph.

concepts is given in two separate lists. In the first case, the list directly points to the entry in the CSR describing the corresponding service. In the second case, the pointer references the transfer concept that is able to provide information about the property. Both lists are ordered such that the service or transfer concept with the lowest uncertainty is located at the first element of the list. Whenever services are added or removed from the system, these lists have to be adapted to the changes by adding or removing elements.

Figure 8.3 shows the data structure used for structure concepts. Beside the *name* and *namespace*, which are required as unique identifiers, each structure element contains five linked lists that are similar to those presented above:

- *partOf_prt*: pointers to the 'partOf-parents' structure concepts – i.e., the higher level concepts from which property concepts are inherited.
- *partOf_chd*: pointers to the 'partOf-children' structure concepts – i.e., the lower level concepts to which property concepts are passed.
- *isA_prt*: pointers to the 'isA-parents' structure concepts – i.e., the higher level concepts from which property concepts are copied.
- *isA_chd*: pointers to the 'isA-children' structure concepts – i.e., the lower level concepts to which property concepts are copied.
- *properties*: pointers to property concepts owned by the structure concept.

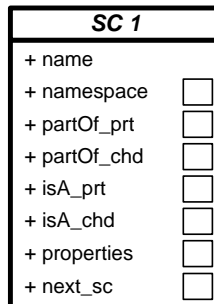


Figure 8.3: Data structure of structure concepts.

Transfer and transfer type concepts use almost the same data structures, as it is depicted in Figure 8.4. Again *name* and *namespace* are given for these concepts, followed by a pointer to the transfer function. Implementing the transfer functions with a separate data structure allows to share the same transfer function code – which might consist of a complex mathematical function, program code, lookup table, etc. – between several transfer and/or transfer type concepts. Particularly, if multiple transfer concepts inherit their function from the same transfer type concept, the memory requirements are reduced. The transfer function structure includes a counter to indicate the number of references to that function. A list of input properties and a pointer to the output property concept in the transfer (type) structure model the corresponding relations of the

transfer function to the system properties. In the field *opt_crit* a numerical optimization criterion (e.g., the computational effort to process the transfer function) is stored, which is evaluated during composition search.

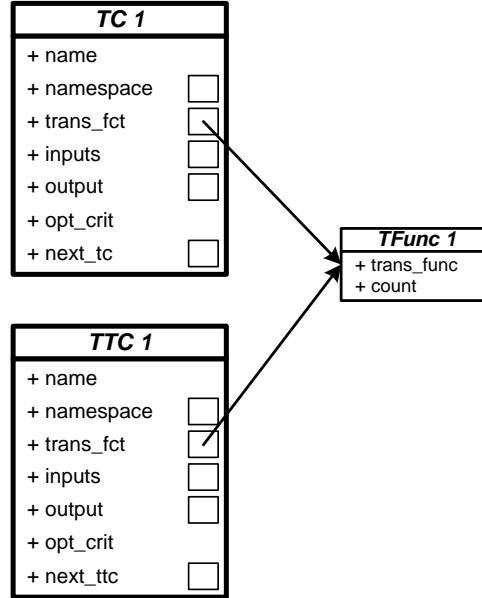


Figure 8.4: Data structures for transfer and transfer type concepts.

The simple data structure of pType concepts can be seen in Figure 8.5. Also this structure contains a *name* and a *namespace* field. Additionally, a list of pointers to property concepts is given, which connects the properties, which are assigned to that type. The two lists *ttype_in* and *ttype_out* point to transfer type concepts for which the property type acts as input and output, respectively.

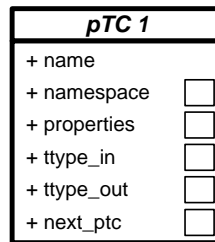


Figure 8.5: Data structure of property type concepts.

8.1.3 Ontology Preprocessor

The ontology preprocessor for this work was realized using the *ANSI C* programming language. It reads several RDF ontology files to create the ontology graph in system memory. This implementation includes the following features:

- *pType* Instantiation,
- *partOf* Inheritance,
- *isA* Instantiation, and
- a simple form of ontology matching.

Details about these features are presented in Section 5.3. The next paragraphs highlight how these features have been realized with the memory structures of the given implementation. The focus of the preprocessors was put on composition search time instead of memory usage. Hence, where possible, the preprocessor creates copies of instantiated concepts instead of resolving inheritance during the search operation.

***pType* Instantiation**

Whenever the ontology preprocessor encounters the assignment of a property type PT to a property concept P , it checks whether the instantiation condition is satisfied. Therefore, every transfer type concept connected to PT is examined. If all input and output property types of a transfer type concept TT are assigned to a property concept belonging to the same structure concept S , to which P is allocated, then a type instantiation is performed. A similar test is conducted when a new transfer type concept is created between property type concepts, but in this case only with the single transfer type and all associated combinations of property and structure concepts.

The corresponding pseudo code for checking the possibility of pType instantiation is presented in Algorithm 5. This code is called for a specific transfer type concept tt , as well as for a concrete structure concept sc . For all property concepts p_o that are associated with the output property type of tt it is tested whether p_o belongs to sc . If no such p_o is found, no pType instantiation can be performed with tt and properties related to sc . Otherwise, the same test is made for all input property concepts p_i that are associated with the input property types in_lst . Only if a p_i can be found for each of the members of the input property type list in_lst , the function indicates that a pType instantiation is valid.

Once the possibility for instantiation has been identified, a new transfer concept element is created that is linked to the involved input and output property concepts, as well as to the transfer function of the transfer type concept. The overall pseudo code for pType instantiation is shown in Algorithm 6, which takes a property concept as input and iterates over all transfer type concepts associated with the pType of this property – i.e., it either appears in the input or output list of the transfer type.

The property type of a property concept P related with some structure concept S is also inherited to the corresponding property concepts \hat{P} of *isA*-child concepts \hat{S} of S . Due to this inheritance, the instantiation of the transfer type concept is automatically passed to lower level concepts of P .

Algorithm 5 Pseudo code to check possibility for pType instantiation

```

1: function CHECKPTYPEINstantiation(tt, sc)
   Input: Transfer type concept tt tested for instantiation;
           Pointer to structure concept sc for which the test is performed
   Output: Possibility of instantiation TRUE or FALSE

           ▷ Check all property concepts associated with output of transfer type concept
2:  $p_o \leftarrow tt \rightarrow output \rightarrow properties$ 
3: while  $p_o \neq NULL$  do
4:   if  $p_o \rightarrow prop\_cpt = sc$  then
5:     break
6:   end if
7:    $p_o \leftarrow p_o \rightarrow next\_pc$ 
8: end while
9: if  $p_o = NULL$  then           ▷ No associated property concept which is related to sc
10:   return FALSE
11: end if

           ▷ Check all property concepts associated with input of transfer type concept
12:  $in\_lst \leftarrow tt \rightarrow inputs$ 
13: while  $in\_lst \neq NULL$  do
14:    $p_i \leftarrow in\_lst \rightarrow properties$ 
15:   while  $p_i \neq NULL$  do
16:     if  $p_i \rightarrow prop\_cpt = sc$  then
17:       break
18:     end if
19:      $p_i \leftarrow p_i \rightarrow next\_pc$ 
20:   end while
21:   if  $p_i = NULL$  then           ▷ No associated property concept which is related to sc
22:     return FALSE
23:   end if
24:    $in\_lst \leftarrow in\_lst \rightarrow next\_ptc$ 
25: end while
26: return TRUE
27: end function

```

Algorithm 6 Pseudo code for instantiation of property and transfer types

```

1: function INSTANTIATEPTYPE(pc)
   Input: Property concept pc, which has been assigned a property type

2:   for all tt ∈ pc − > pType.ttype_in do                                ▷ Inputs to transfer type concepts
3:     if checkPTypeInstantiation(tt, pc − > structure_cpt) = TRUE then
4:       createTransferTypeInstance(tt, pc − > structure_cpt)
5:     end if
6:   end for
7:   for all tt ∈ p − > pType.ttype_out do                                ▷ Outputs to transfer type concepts
8:     if checkPTypeInstantiation(tt, pc − > structure_cpt) = TRUE then
9:       createTransferTypeInstance(tt, pc − > structure_cpt)
10:    end if
11:  end for
12: end function

```

***partOf* Inheritance**

A *partOf* relation between structure concepts does not require any concepts to be copied. It mainly tells the ontology preprocessor that the properties of a higher level structure concept are also valid for the lower level structure concepts. Instead of creating an explicit reference from the lower levels to a property concept *P* of the higher level, the preprocessor evaluates the applicability of *P* whenever it encounters an RDF statement that refers to *P* at the lower level. For instance, the property *speed* of a *car* is inherited to its *chassis*. Upon the statement *chassis#speed*, the preprocessor does not directly find the *speed* in the property list of the *chassis*, and hence, it recursively walks up the *partOf* hierarchy of structure concepts to find this property. If it is found as the property at any higher level structure concept, then this property is the one referenced by the statement.

Algorithm 7 presents the pseudo code for the identification of property concepts inherited by *partOf* relations. The function checks if the referenced property is assigned to a structure concept *sc*. In the positive case, a reference to that property is returned. Otherwise, the function is called recursively with the higher level structure concepts as input.

The advantage of this approach is the reduced memory overhead by decreasing the number of references, for the cost of an increased temporal effort during ontology preprocessing. However, once the ontology graph is set up, the *partOf* relations are not needed by the composition search, and therefore, the search time is not influenced by this practice.

***isA* Instantiation**

An *isA* relation in an RDF statement triggers the recursive copying of property concepts and *partOf* relations, including the related *partOf*-child structure concepts and transfer concepts between properties (as depicted in figure 5.15). First, the property concepts are instantiated for the lower level structure concept *lsc* and recursively for all *isA*-child structure concepts of *lsc*.

Algorithm 7 Pseudo code to find property concepts inherited by *partOf* relation

```

1: function GETINHERITEDPROPERTY(sc, name)
   Input: Structure concept sc to be checked;
           name of property concept
   Output: Reference to property concept that is inherited

2:   for all pc  $\in$  sc  $\rightarrow$  properties do                                 $\triangleright$  Check local properties
3:     if pc  $\rightarrow$  name = name then
4:       return pc
5:     end if
6:   end for
7:   for all hsc  $\in$  sc  $\rightarrow$  partOf_prt do                                 $\triangleright$  Recursively visit partOf parents
8:     if (prop  $\leftarrow$  getInheritedProperty(hsc, name))  $\neq$  NULL then
9:       return prop
10:    end if
11:  end for
12:  return NULL
13: end function

```

During the copy operation *parent-child* relations are also created between the properties themselves, in order to facilitate the composition search (i.e., faster exploitation of provided lower level properties). Afterwards, the *partOf* relations and the complete sub-trees – consisting of structure and property concepts – connected by these relations are copied to the lower levels. Finally, the transfer concepts between the newly created concepts are instantiated. A pseudo code for the function managing the recursive copy operation is presented in Algorithm 8.

Algorithm 8 Pseudo code to recursively copy the sub-tree structures of a structure concept

```

1: function RECURSIVESTRUCTURECOPY(lsc, hsc)
   Input: Lower level structure concept lsc as target for copy operation;
           Higher level structure concept hsc as source for copying

2:   copyStructureParts(lsc, hsc)
3:   copyTreeTransferConcepts(lsc, hsc, lsc, hsc)
4:   for all csc  $\in$  lsc  $\rightarrow$  isa_chd do                                 $\triangleright$  Recursive copy to isA-child concepts
5:     recursiveStructureCopy(csc, lsc)
6:   end for
7: end function

```

Algorithm 9 shows two functions, which recursively copy the structure and property concepts of sub-trees connected by *partOf* and *isA* relations. In both functions, first a copy of the structure concept, which is linked by the *partOf* relation (and *isA* relation, respectively), is made with a unique name. This unique name is created by adding the address of the higher level concept to the original name of the concept. Afterwards the properties of the original structure concept are duplicated. At last the functions are called recursively with the original structure

concept and its clone to copy the structure and property concepts of the levels below the original structure concept.

Algorithm 9 Pseudo code to recursively copy sub-tree of *partOf* and nested *isA* relations

```

1: function COPYSTRUCTUREPARTS(lsc, hsc)
   Input: Lower level structure concept lsc as target for copy operation;
           Higher level structure concept hsc as source for copying
2:   for all part  $\in$  hsc –  $\rightarrow$  partOf_chd do                                 $\triangleright$  Copy all partOf-child concepts
3:     Create copy nsc of part with unique name and partOf relation
4:     Duplicate properties of part to nsc
5:     copyStructureParts(nsc, part)
6:     copyIsAStructure(nsc, part)
7:   end for
8: end function

9: function COPYISASTRUCTURE(lsc, hsc)
   Input: Lower level structure concept lsc as target for copy operation;
           Higher level structure concept hsc as source for copying
10:  for all isa  $\in$  hsc –  $\rightarrow$  isA_chd do                                 $\triangleright$  Copy all isA-child concepts
11:    Create copy nsc of isa with unique name and partOf relation
12:    Duplicate properties of isa to nsc
13:    copyStructureParts(nsc, isa)
14:    copyIsAStructure(nsc, isa)
15:  end for
16: end function

```

Once the structure and property concepts have been copied, Algorithm 10 is triggered to create the transfer concepts between the new property concepts. It starts at the corresponding roots of the original and duplicated sub-trees – *root_hsc* for the source, and *root_lsc* for the duplicate – and identifies all property concepts that are associated with a transfer concept (i.e., it appears in the input list or as the output of the transfer concept). For each such property, a copy of the original transfer concept is created with an unique name. The name is constructed from the original name plus a string of addresses of the contributing property concepts. The function *getRelatedProperty*(*root_lsc*, *root_hsc*, *pc*) browses two sub-trees iteratively and returns the property concept in *root_lsc*, which corresponds to the original property concept *pc* in *root_hsc*. The found property concept is then set as the output value of the transfer concept, or added to the input list, respectively. Finally, when the transfer concepts at the current level have been duplicated, the algorithm is called recursively for all *partOf* and *isA* children of the current level. In order to make this recursive call, an additional check is performed to ensure that the children of the original and the duplicated sub-tree are the corresponding complements.

Algorithm 10 Pseudo code to recursively copy transfer concepts between new property concepts

```

1: function COPYTREETRANSFERCONCEPTS(lsc, hsc, root_lsc, root_hsc)
   Input: Lower level structure concept lsc as target for copy operation;
           Higher level structure concept hsc as source for copying;
           Root lower level concept root_lsc at which isA instantiation started;
           Root high level concept root_hsc at which isA instantiation started

2:   for all pc ∈ hsc − > properties do                                ▷ Iterate over all properties
3:     for all tc ∈ pc − > transfer_out do                                ▷ Copy transfer concepts in output list
4:       Create copy ntc of tc with unique name
5:       lpc ← getRelatedProperty(root_lsc, root_hsc, pc)
6:       Set lpc as output for ntc
7:       for all ipc ∈ tc − > inputs do                                ▷ Connect input properties to ntc
8:         lpc ← getRelatedProperty(root_lsc, root_hsc, ipc)
9:         Add lpc to input list of ntc
10:      end for
11:    end for
12:    for all tc ∈ pc − > transfer_out do                                ▷ Copy transfer concepts in input list
13:      Create copy ntc of tc with unique name
14:      lpc ← getRelatedProperty(root_lsc, root_hsc, tc − > output)
15:      Set lpc as output for ntc
16:      for all ipc ∈ tc − > inputs do                                ▷ Connect input properties to ntc
17:        lpc ← getRelatedProperty(root_lsc, root_hsc, ipc)
18:        Add lpc to input list of ntc
19:      end for
20:    end for
21:  end for
22:  for all hp ∈ hsc − > partOf_chd do                                ▷ Recursive call for all parts
23:    for all lp ∈ lsc − > partOf_chd do                                ▷ Find corresponding concept in lsc
24:      if hp − > name ≐ lp − > name then                                ▷ Check if unique names correspond
25:        copyTreeTransferConcepts(lp, hp, root_lsc, root_hsc)
26:      end if
27:    end for
28:  end for
29:  for all hp ∈ hsc − > isA_chd do                                ▷ Recursive call for all isA-instances
30:    for all lp ∈ lsc − > isA_chd do                                ▷ Find corresponding concept in lsc
31:      if hp − > name ≐ lp − > name then                                ▷ Check if unique names correspond
32:        copyTreeTransferConcepts(lp, hp, root_lsc, root_hsc)
33:      end if
34:    end for
35:  end for
36: end function

```

Ontology Matching

The ontology preprocessor implements a simple form of ontology matching such that ontologies can be defined in isolation within different files and combined during preprocessing to a single ontology graph. Matching points are identified by the names of concepts, which is supported by the usage of distinct namespaces. Connecting ontologies by *partOf* relations additionally allows to reference local properties, which are automatically translated into common global properties. For instance, in an ontology describing the chassis (with all its components) it is possible to refer to the speed of the chassis, which is translated to the speed of the car, after both ontologies have been combined.

8.2 Composition Search

The composition search mechanism navigates through the data structures of the system ontology graph, in order to identify semantic equivalences of a given property concept. A depth first search approach was chosen for the implementation due to smaller overhead for managing expanded search paths, without changing the worst-case complexity of the algorithm compared to other methods. Similar to the data structures and the ontology preprocessor described above, *ANSI C* was used for the implementation of the composition search.

Due to the bi-directional relations within the data structures of the ontology graph, valid service compositions are identified by recursively checking the provision of input properties of transfer concepts. A pseudo code for the implemented procedure is presented in Algorithm 11. It uses an optimization criterion – here the computational effort is assumed as criterion, which can easily be exchanged by other criteria – to select the best provision among different solutions. First it checks whether the property can directly be provided by a service or by deduction, which can be the case if the property has already been checked on another path in the ontology graph.

If the property has not yet been marked as provided, the ability to provide any *isA*-child of the property is checked. Function *checkChildProvision* recursively walks down the hierarchy and examines the provision lists of the child concepts. In the positive case the provision of the child (either direct or by derivation) is copied to the provision list of the searched property. Otherwise, in case the property concept has not yet been visited, the algorithm tries to deduce the property by exploiting transfer concepts. Therefore, all transfer concepts with an output relation towards the property *pc* are investigated. For each suitable transfer concept, the provision of all input property concepts is checked by a recursive call to the algorithm. If at least one input property cannot be provided, the transfer concept cannot be used for deduction, and the next transfer concept is inspected. Since different solutions for deduction might exist, when a solution is found, the optimization criterion is compared with the currently best solution. A solution with lower computational effort is preferred and it replaces the old one. Finally, the return value of the function indicates the possibility of providing the property.

In order to obtain the service composition tree, after a positive provision check, a recursive function collects the services and transfer functions beginning at the leafs of the tree. The corresponding pseudo code, which outputs the composition tree for a solution, is shown in Algorithm 12. If the property is already a leaf of the composition tree, the providing service is output

Algorithm 11 Pseudo code to recursively check the provision of a property concept

```

1: function CHECKPROVISION(pc, crit)
    Input: Property concept pc for which an equivalent is requested;
    Optimization criterion crit returned for selection of concurrent paths
    Output: Returns the ability to provide the property: TRUE or FALSE

2:   crit  $\leftarrow$  -1
3:   if pc  $\rightarrow$  provision_svc  $\neq$  NULL then                                 $\triangleright$  Direct provision by service
4:     crit  $\leftarrow$  pc  $\rightarrow$  provision_svc.effort
5:   else if pc  $\rightarrow$  provision_drv  $\neq$  NULL then                             $\triangleright$  Provision by deduction
6:     crit  $\leftarrow$  pc  $\rightarrow$  provision_drv.effort
7:   else
8:     if checkChildProvision(pc, chd_crit) = TRUE then
9:       crit = chd_crit
10:      pc  $\rightarrow$  provision_drv.effort  $\leftarrow$  chd_crit
11:      Set pc  $\rightarrow$  provision_drv to provision of child
12:    else if pc  $\rightarrow$  visited = FALSE then
13:      pc  $\rightarrow$  visited  $\leftarrow$  TRUE
14:      for all tc  $\in$  pc  $\rightarrow$  transfer_out do                                 $\triangleright$  Check transfer concepts that output pc
15:        tc_crit  $\leftarrow$  tc  $\rightarrow$  opt_crit                                 $\triangleright$  Set optimization value for transfer concept
16:        for all ipc  $\in$  tc  $\rightarrow$  inputs do                                 $\triangleright$  Check if inputs to tc are provided
17:          if checkProvision(ipc, drv_crit) = FALSE then
18:            tc_crit  $\leftarrow$  -1
19:            break                                                         $\triangleright$  Input not provided, transfer concept cannot be used
20:          end if
21:          tc_crit  $\leftarrow$  tc_crit + drv_crit
22:        end for
23:      if (tc_crit  $\geq$  0) and (pc  $\rightarrow$  provision_drv.effort > tc_crit) then
24:        pc  $\rightarrow$  provision_drv  $\leftarrow$  tc                                 $\triangleright$  Store that pc is provided by tc
25:        pc  $\rightarrow$  provision_drv.effort  $\leftarrow$  tc_crit
26:        crit  $\leftarrow$  tc_crit
27:      end if
28:    end for
29:  end if
30:  if crit < 0 then                                                         $\triangleright$  Check if pc cannot be provided
31:    return FALSE
32:  end if
33: end if
34: return TRUE
35: end function

```

along with the current depth of the tree. For the other case, the algorithm moves further towards the leafs by visiting the property concepts at the inputs of the providing transfer concept. After the inputs to that transfer concept have been printed, the transfer function, which combines the inputs printed above at $depth + 1$, is also displayed and may be interpreted as provider for the solution or the input for the next transfer function at $depth - 1$.

Algorithm 12 Pseudo code to recursively collect service composition tree

```

1: function COLLECTTREE(pc, depth)
    Input: Property concept pc to be deduced;
           Current depth within composition tree
2:   if pc → provision_svc ≠ NULL then                                ▷ Check if concept is leaf of tree
3:     Output pc → provision_svc and depth
4:   else
5:     for all ipc ∈ pc → provision_drv.inputs do                      ▷ Visit all inputs of providing transfer concept
6:       collectTree(ipc, depth + 1)
7:     end for
8:     Output pc → provision_drv.trans_fct → trans_func and depth
9:   end if
10: end function

```

8.3 Calculation of Worst-Case Uncertainty

In order to calculate the worst-case uncertainty of service compositions, sensitivity indices for transfer functions have to be obtained. An algorithm for the determination of these indices has been implemented in MATLAB. The algorithm analyzes a transfer function by systematically testing the whole input space of the function and determining the worst-case indices for distinct input-output intervals. The simulation of propagation of uncertainty, as presented in Section 9.4, will be based on these indices. At the end of this section a description of the MATLAB implementation to obtain the weights for combining sub-trees by weighted averaging is included.

8.3.1 Sensitivity Indices

In general, transfer functions can be multivariate and non-linear, and hence, the determination of sensitivity indices is not as trivial as for linear functions. For linear functions it is sufficient to fill the sensitivity matrix with the first order partial derivative of the analyzed input, which is a single constant value. Non-trivial functions can either be analyzed analytically by finding the maximum first order derivative of the transfer function for each sensitivity interval using curve sketching techniques. Or alternatively, a numerical approach can be used, where the maxima within the intervals are found by testing the derivative function with a set of input values and selecting the maximum value obtained within the observed interval. The numerical approach

requires that the whole input space of the transfer function (i.e., all possible combinations of input values) is tested. Within this work, the numerical approach has been chosen.

The MATLAB implementation analyzes one input of a transfer function at a time. Thereby, it iterates over the whole input space of the function, where parameters can be provided that constrain the input space (e.g., restricting the maximum possible speed of a vehicle). The step width of the iterations depends on the value range of the parameter and the transfer function. For instance, in the simulations (see Chapter 9), the step width has been chosen such, that for each input at least 2500 distinct values have been tested. Even though, due to the usage of a discrete step width to analyze the transfer function, the absolute maximum value of the derivative function could be located within the step width, it is not useful to analyze the transfer function with a significantly smaller step width than the resolution of the input values (e.g., smaller than the resolution of the actual sensor). This is due to the fact, that ontology based reconfiguration is only useful if the transfer functions in the system ontology show an almost continuous behaviour (i.e., no spikes in the transfer function) for input values of a reasonably small distance (e.g., a sensor resolution that is technically feasible).

Algorithm 13 presents the pseudo code to determine the sensitivity indices numerically for a dedicated input parameter of the transfer function. First, the sensitivity matrix *sm* is initialized with zeros, where the parameters *m* and *n* denote the number of input intervals and output intervals, respectively. Then, as long as the input space has not yet been completely examined, and thus, a new input vector *iv* is available, the sensitivity of the next input vector is evaluated. The function *getInputVector(...)* simply iterates over the value ranges of all input parameters until the whole input space is explored. The validity of the input vector is checked to ensure that the current set of input values meets the specification of the parameters and physical constraints. For example, with a usual car it is not possible that the left wheel is rotating forward, while the right one is going backwards. If any parameter violates a constraint and the check returns *false*, the loop starts again with a new input vector. Otherwise, since the sensibility matrix relates the intervals of the analyzed input to output intervals, the output interval has to be determined by executing the transfer function with the input vector. Also the output value has to comply with the specification (e.g., the calculated curve radius has to be within a technically feasible range), otherwise this set of input values is not considered for the sensitivity indices. This check is performed by the function *checkFuncOutput(...)*, which returns *false* in case of non-compliance and the algorithm goes back to the head of the loop. In case of compliance, after the input has been validated and the indices of the input and output intervals are determined (with the functions *getInputIndex(...)* and *getOutputIndex(...)*), the first order derivative function is evaluated by calling *evaluatePartialDerivative(...)* with the current input to obtain the absolute value of the slope of the transfer function. If this value is higher than the previously stored sensitivity value for the current input and output intervals, then the old sensitivity index is replaced by the new value. When all sets of input values have been tested, the sensitivity matrix, which contains the maximum sensitivity indices for each combination of input and output interval, is returned.

Due to the diversity of possible transfer functions and input value ranges, the current implementation has to be configured manually for each function and set of input values. To obtain the required derivative function, the first order partial derivative has to be calculated with respect to

Algorithm 13 Pseudo code to determine sensitivity indices

```
1: function DETERMINESENSITIVITY(m, n)  
   Input: Number of input intervals m;  
           Number of output intervals n  
2:   sm  $\leftarrow$  zeros(m, n)  
3:   while iv  $\leftarrow$  getInputVector() do  
4:     if checkInputVector(iv) == false then  
5:       continue  
6:     end if  
7:     result  $\leftarrow$  executeTransferFunction(iv)  
8:     if checkFuncOutput(result) == false then  
9:       continue  
10:    end if  
11:    idx_in  $\leftarrow$  getInputIndex(iv)  
12:    idx_out  $\leftarrow$  getOutputIndex(result)  
13:    sens  $\leftarrow$  evaluatePartialDerivative(iv)  
14:    if sm[idx_in, idx_out] < sens then  
15:      sm[idx_in, idx_out]  $\leftarrow$  sens  
16:    end if  
17:  end while  
18:  return sm  
19: end function
```

the observed input variable. Also the ranges for the input values and possible parameter restrictions have to be set within *getInputVector*, *checkInputVektor* and *checkFuncOutput*.

As it is numerically not possible to evaluate positive or negative infinity as input values – which is a valid input for many transfer functions, like the function in Equation 9.4 described as part of the use case simulation – a maximum value for such functions has to be defined. This is only valid for functions $f(x_1, \dots, x_n)$ with *n* input values, where the limit is defined and

$$\left| \lim_{x_i \rightarrow \infty} f(x_1, \dots, x_i, \dots, x_n) \right| < \infty \quad \forall i = 1, \dots, n$$

Otherwise, for the corresponding intervals a sensitivity index of positive infinity has to be assumed. The maximum value for the analysis depends on the tolerable error when the corresponding transfer function is applied in the range of positive or negative infinity. Exemplary considerations when selecting such a maximum value will be presented in Section 9.4.

8.3.2 Selection of Weights

The weights for input intervals of the weighted averaging algorithm are selected during the calculation of the worst-case uncertainty of a composition tree. Whenever the composition tree includes the fusion of composition trees by weighted averaging, the weights are calculated for each interval using Equation 7.10 and the worst-case uncertainty of each fused subtree. The

weights are used to obtain the worst-case uncertainty of the composition and for weighted averaging during the experimental evaluation of the uncertainty bounds.

8.4 Demonstration on robot platform

The dynamic reconfiguration framework was integrated into a demonstrator for the *ARTEMIS* project *EMC² – Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environments*³.

The demonstrator is based on the *Adept® MobileRobots® Pioneer 3-AT* robot (see Figure 8.6), using the Robot Operating System (ROS)⁴ as execution framework on top of three distributed computing nodes. In contrast to what its name suggests, ROS is actually not an operating system on its own, but a middleware platform running on operating systems like *Linux*, *Windows*, *Mac OS* or others. It provides hardware abstraction, process management and message-based *publish-subscribe* communication between independent software processes that might be distributed on networked computing nodes. A *publisher* process announces a *topic* to which it sends information that is transparently distributed to *subscriber* processes which are interested in this information. Thereby it is possible that several publishers exist for the same topic. Similarly, multiple processes can subscribe to one topic. With ROS as platform, dynamic reconfiguration is performed by automatic generation of a software process that implements the transfer service. This process subscribes to the topics which provide the required inputs for the transfer service, and it publishes the value of the required property to the corresponding topic. Due to the publish-subscribe pattern, for all other existing processes the changes of subscription as well as publication are transparent.

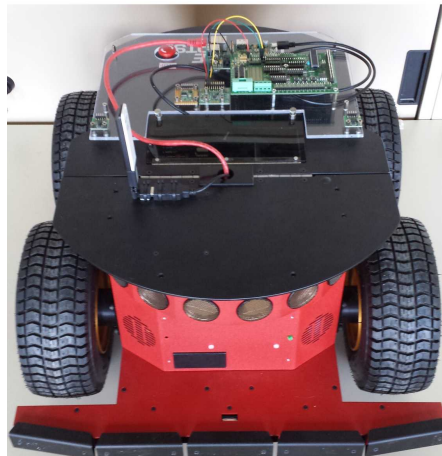


Figure 8.6: *Pioneer 3-AT* mobile robot.

³More information about this project: <http://www.artemis-emc2.eu/>

⁴More details available at: <http://www.ros.org/>

The demonstrator implements a simple battery management system for the robot, which keeps track of the remaining capacity of the battery. The main input for the battery management are measurements of the power consumption. During the demonstration, the current measurements are deactivated and reconfiguration is triggered. This leads to the generation of a transfer service that publishes the information about the power consumption based on the measurements of other properties (i.e., velocity, yaw rate, etc.).

8.4.1 Hardware Configuration

Figure 8.7 illustrates the hardware configuration of the demonstrator. It features the following three computing nodes:

- **SECO CARMA GPU Development Kit:** this board is equipped with a *NVIDIA Tegra 3 ARM Cortex A9 Quad-Core* processor and a *NVIDIA Quadro 1000M* GPU. Each of these processors has access to 2 GB RAM. The GPU board is internally connected to a micro controller that interacts with the motors of the wheels, the wheel encoders, bumpers and sonar sensors. As operating system, the Ubuntu Linux is used on this computer.
- **Raspberry PI (Model B):** the Raspberry PI has an ARM11 CPU that is operating at 700 MHz and it has 512 MB SDRAM. It is connected via Ethernet to the GPU board, and to different sensors, like gyroscopes, accelerometers or an amperemeter, via *I2C*, *SPI* and analog interfaces. On this computer the Raspbian Linux distribution is used.
- **Laptop:** in order to enhance the computational power of the demonstrator, a laptop is connected via IEEE 802.11 WIFI to the computing nodes on the robot. Thereby, the GPU board acts as a gateway between the Ethernet and the WIFI networks, such that all nodes appear to be in the same network.

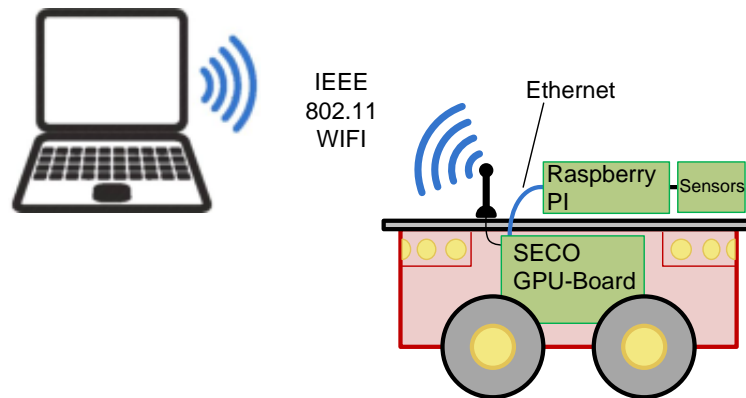


Figure 8.7: Hardware configuration of demonstrator.

8.4.2 Component Configuration

On all of the three nodes the ROS is running. This enables that information which is published on one node can also be subscribed on a another node. Each of the nodes hosts a set of software components that are implemented as independent processes and which implement access to sensors, actuators or they process information received from other components. In order to communicate with other components, a component has to subscribe to the information it requires as input, and it publishes sensor readings or the results of its computations.

In Figure 8.8 the distribution of components that are relevant for the demonstration and the flow of information between them are depicted. Each ellipse represents a software component and an arrow denotes the direction of the information flow between these components. Dashed lines signify the components and interactions that are created upon reconfiguration.

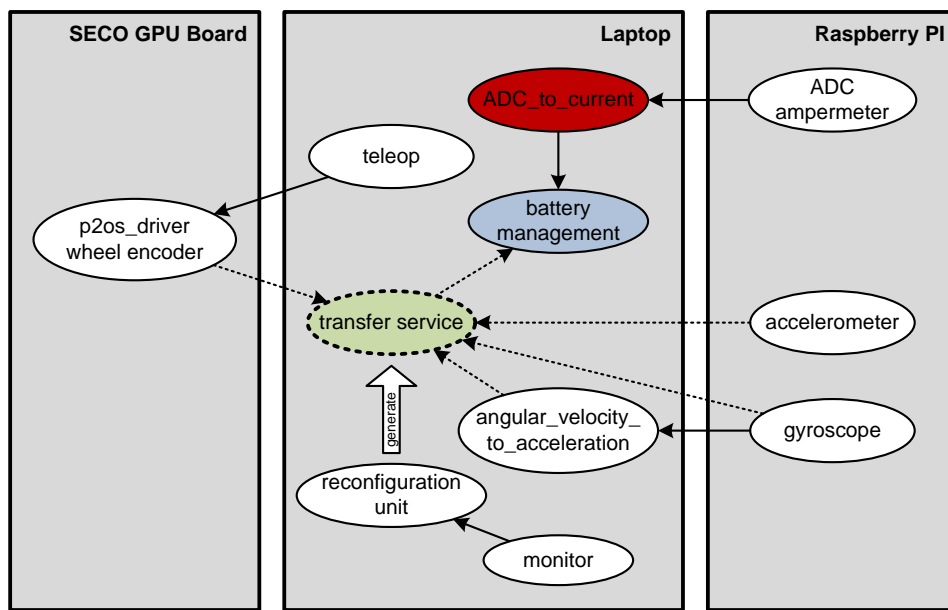


Figure 8.8: Flow of information between components of demonstrator.

The purpose of these components is as follows:

- *p2os_driver, wheel encoder*: this component interacts with the robot hardware in order to drive the wheels and to provide information about the current speed of the robot, which is derived from the wheel encoders.
- *teleop*: a component that provides drive commands to the *p2os_driver*, which are read from the keyboard of the laptop.
- *ADC amperemeter*: interacts with an analog-to-digital converter (ADC) in order to read the current power consumption.
- *ADC_to_current*: converts the value of the *ADC amperemeter* to a value representing the electric current.

- *battery management*: component that keeps track of the remaining capacity of the battery.
- *transfer service*: this component is generated upon the reconfiguration of the system. It calculates the power consumption from the available sensors (i.e., velocity, acceleration, yaw rate).
- *accelerometer*: interacts with a sensor that measures the acceleration in three dimensions.
- *gyroscope*: interacts with a sensor that measures the angular velocity in three dimensions.
- *angular_velocity_to_acceleration*: derives the angular acceleration from the measured angular velocity.
- *reconfiguration unit*: this component is responsible to perform the reconfiguration. It implements the required building blocks of the Component Configuration Manager (CCM): the Service Orchestration Engine (SOE), Central Service Registry (CSR) and the Resource Configuration Manager (RCM). For the composition search it uses the algorithms described in the sections above and comprises a system ontology, which is tailored for the mobile robot.
- *monitor*: implements a simple service and resource monitor that detects service failures by observing the provision of information (i.e., to detect that a component stops publishing the required information). Upon the detection of a failure, the *reconfiguration unit* is triggered, which performs the reconfiguration.

8.4.3 Reconfiguration Process

After the occurrence of a service failure – which is initiated during the demonstration by stopping the *ADC_to_current* component (the red ellipse in Figure 8.8) – the *monitor* component sends a request for reconfiguration to the *reconfiguration unit*. This request contains the topic that is not published anymore, and hence, another component has to be found or created that publishes this topic. The topic corresponds to a property concept in the system ontology. This property concept is the input to the composition search algorithm, which at the end outputs a composition tree with the solution.

The composition tree consists of the input property concepts for the transfer service, as well as the transfer concepts required to calculate the value of the required property. The transfer functions of the transfer concepts contain a program code using the *Python* language. These transfer functions are embedded into a program skeleton in the correct order. With this program skeleton the transfer functions are executed periodically and at the end of each cycle the result of the computation is published to the required topic. In the initialization section of this program the topics corresponding to the input property concepts are subscribed. Finally, ROS specific commands are used to execute the obtained program code in a new process, in order to start the *transfer service* component (the green ellipse in Figure 8.8). As the transfer service provides its information at the same topic as the failed service, the *battery management* (the blue component in the figure) automatically continues its work as soon as new information is published.

Experiments and Simulation

To evaluate the applicability of the dynamic reconfiguration framework w.r.t. the requirements of distributed real-time systems presented in Chapter 3, a use case has been developed with which a series of experiments and simulations were conducted. Based on the use case from the automotive domain, a simple system ontology has been defined. The ontology acts as the input for runtime evaluations and experiments analyzing the probability to find service compositions in the system model. With a simulation in MATLAB the determination of the worst-case uncertainty by propagation of uncertainty in service compositions is shown.

Section 9.1 starts with the description of the use case that builds the basis for the remainder of the chapter. Afterwards, the idea and setup of the runtime experiment and the evaluation of the probability of finding service compositions are discussed in Sections 9.2 and 9.3. Finally, the simulation of propagation of uncertainty in service compositions is presented in Section 9.4.

9.1 Use Case Description

A practical use case example from the automotive domain has been chosen, since in modern cars a high number of sensors, actuators and networked ECUs can be found [22]. Furthermore, due to the envisioned autonomous driving [8, 28] and the possibility to share information with the environment of vehicles [107], additional sources of information will be available in the future. Even though safety and availability of automotive systems are of utmost importance, manufacturers are pushed to apply cost-efficient solutions for their products. The proposed reconfiguration framework can be employed in this application domain – in the development phase as well as online during the operation of a vehicle – to identify and exploit implicitly redundant information available in the system, and thus, to increase the system’s reliability and/or to reduce the cost of implementing redundant system components. Implicit redundancy allows to create service compositions that derive a system property (e.g., the steering angle) by a set of other system properties (e.g., the angular speed of the left and right wheel). The simulation in Section 9.4 includes a detailed description of such a composition.

Also in [2, 110] the authors issued the reasonableness of runtime reconfiguration in the automotive domain. But since their proposed solutions use predefined configurations that can be switched at runtime to provide the desired behaviour, these mechanisms lack the ability to counteract unforeseen faults, nor can they be used at design time to reduce production cost by identifying redundant sources of information.

The next subsection describes the system ontology used for the experiments. Then, an example scenario is provided in the subsequent subsection that illustrates the application of the framework with a realistic scenario.

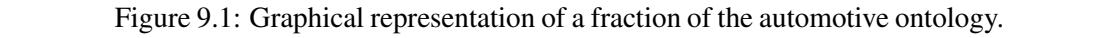
9.1.1 System Ontology

The system ontology describes the physical construction of a car, including mechanical connections and physical relations between system properties. Figure 9.1 provides a graphical representation of a part of the ontology used for the experiments. It mainly models the relations between the steering angle at the steering wheel and the four wheels of the car. In order to reduce the number of elements and to improve the clarity of the representation, in some cases input and output relations to transfer concepts have been combined. This means, that the corresponding transfer function can be inverted, such that the property can either act as an input or an output. In the RDF description of the ontology, these relations are decomposed into two (or more) transfer concepts with appropriate transfer functions. The concepts marked green represent properties for the subsequent example scenario, which are provided by a service or known system constants. The red concept denotes the property for which a service composition has to be found.

Beside the concepts and relations present in the figure, the automotive ontology also describes parts of the power train, like the engine, gearboxes, the clutch and the rotational dependencies between them, as well as the undercarriage, including the steering, dampers and springs. The following models are used to build the ontology:

- *Terminology*: introduction of the basic concept types and allowed relations,
- *Ptypes*: models the generic property and transfer types (e.g., translation between radius and perimeter),
- *Car*: models the high level concepts of a car,
- *Undercarriage*: models the concepts related to the undercarriage as part of the car,
- *Powertrain*: models the concepts related to the power train as part of the car.

In order to be able to observe the influence of the size of the ontology on the performance of the algorithms, the experiments have been repeated with three ontologies of different size, which are referred to as *full*, *medium* and *small* ontology. Table 9.1 summarizes the number of concepts in each of these ontologies after instantiations by the ontology preprocessor. Obviously, as the three ontologies are composed of different numbers of concepts, they cover different parts of the system, and hence, encompass different system requirements. Anyway, the purpose of



these ontologies is to highlight the applicability of the proposed knowledge-based reconfiguration framework, instead of representing knowledge bases that can already be applied for real automotive systems.

Table 9.1: Number of concepts and relations in automotive ontology.

Concept or Relation	full	medium	small
Structure Concepts	51	39	26
Property Concepts	98	75	47
Transfer Concepts	141	101	47
Property Type Concepts	10	10	10
Transfer Type Concepts	6	6	6
<i>isA</i> Relations	26	12	4
<i>partOf</i> Relations	32	30	21
<i>input</i> Relations to Transfer Concepts	313	257	115
<i>output</i> Relations from Transfer Concepts	141	101	47
<i>input</i> Relations to Transfer Type Concepts	8	8	8
<i>output</i> Relations from Transfer Type Concepts	6	6	6

9.1.2 Example Scenario

The simplified scenario presented in this section is intended to provide an example for the problems that can be solved by the reconfiguration framework and to demonstrate how the algorithms arrive at a valid service composition. For this purpose it is supposed that a car equipped with the dynamic reconfiguration framework offers the functionality of an adaptive light, which turns the headlights into the heading direction of the vehicle, based on the measurements of a sensor that reads the *steering angle* at the *steering wheel*. Now it is assumed that this sensor ceases to work, but the functionality of the adaptive light should be preserved if a valid service composition can be found. Otherwise, if no such solution exists, a safe state has to be entered by, for instance, positioning the lights straight forward.

After a fault detection mechanism observes the fault of the sensor, the missing property concepts are determined and it turns out that the *steering angle* α (the red property concept in Figure 9.1) is needed for the adaptive light. The composition search algorithm is triggered, which finds that the *steering angle* is transformed (by the *steering gear*) to a certain angle β at the *front wheels*. Since this property concept is not provided by a service, the search operation continues and the property *curve radius* r is reached. The transformation of the *curve radius* to the *wheel angle* requires the property *wheel base* a – denoting the distance between the front and the rear wheels – to be known. As this is a constant in the system it is marked as provided, which is indicated by the green color in the figure. Essentially this means, that if the *curve radius* is known, also the *steering angle* can be calculated. The ontology in the figure provides two possibilities to deduce the *curve radius*. Either by combining the *speed* v of the vehicle and its *lateral acceleration* v'_q , or by using the *angular speed* ω of the *left* and *right rear wheels*

together with the system constant *track width* b – denoting the distance between the left and the right wheels. In the example, all of these properties are provided by services, and hence, the composition search algorithm has to decide for one of both choices. Here it opts for using the *angular speeds* of the wheels, such that the composition tree of Figure 9.2 is obtained.

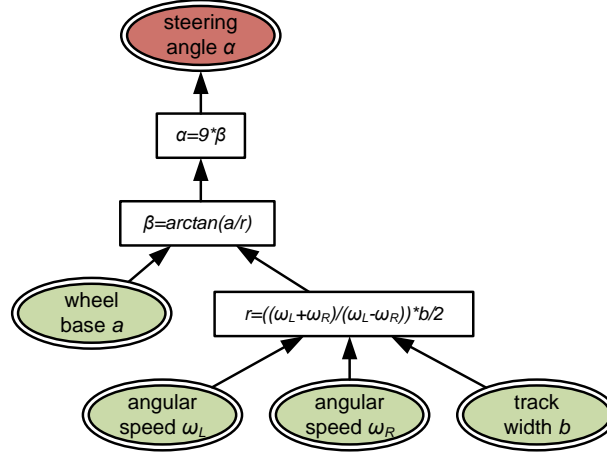


Figure 9.2: Composition tree of example scenario.

The resulting transfer service has to periodically read the *angular speeds* of the wheels from the providing services, as well as the two system constants *track width* and *wheel base*. Then it is able to provide the missing *steering angle* to the adaptive light application.

9.2 Runtime of Composition Search

The runtime of the composition search is crucial for the maximum delay until the system recovers after a fault by means of reconfiguration, or until a predefined safe state is entered in case no valid composition can be found. Therefore, experiments were conducted to observe the algorithm's runtime behaviour w.r.t. the size of different ontologies. Within these experiments, the fault of a service, which is providing a property concept, is simulated. Thereby the impact of the ontology size – measured as the number of property concepts visited – as well as the required search time are observed, until the algorithm returns with either a valid service composition or the information that no such solution exists.

During the operation of the system, the number of provided properties (as represented by the entries in the SOM) varies due to service failures, changing modes of operation or system evolution with new services. Also the runtime experiments require that there exists a set of provided properties, as otherwise no solution can be found. Furthermore, the number of provided property concepts influences the runtime of the search operation, because the algorithm immediately stops exploring a subgraph, if one of the input properties to a transfer concept is not provided. Consequently, the rest of the inputs and the related parts of the ontology might not be visited anymore. For this reason, a low number of provided property concepts will lead to a faster termination of the algorithm (but with an increased probability of a negative result).

A similar behaviour – although with a high chance for a valid service composition – occurs with a high number of provided properties, because most of the inputs to the transfer concepts are directly provided and the rest of the graph does not have to be explored. To be able to observe the influence of the provided properties on the runtime, the number of provided properties is systematically varied during the experiments, by selecting properties from a set of providable concepts.

The experiments are based on the assumption that $N + 1$ provided property concepts exist, whereas for one of them the providing service ceases to work. Therefore, a combination of the remaining N provided properties has to be found, such that the value of the missing property concept can be calculated. Since N influences the runtime, for each test N distinct properties are selected from a pool of 14 property concepts, which are assumed to be directly providable, and corresponding SOM entries are created. Thereby, the number of properties is varied between $N = 0 \dots 14$. Due to the fact that the runtime is not only affected by N , but also by the combination of provided properties, every possible combination of N properties (i.e., $2^{14} = 16,384$) needs to be evaluated. Obviously, the number of possible combinations changes for different values of N , as for instance, $N = 0$ allows only one choice, while $N = 7$ enables $\binom{14}{7} = 3,432$ combinations.

The purpose of the experiments is to find the maximum execution time of the algorithm for a particular N . For each value of N at least one selected missing property concept in the ontology leads to the longest search time. To identify this property concept, all properties in the ontology have to be examined. Performing all tests required for the full ontology results in an effort of more than 1.6 million (i.e., $16,384$ combinations \times 98 property concepts) executions of the service composition algorithm.

On the one hand, the number of property concepts, which have to be visited until the composition search is finished, is measured. On the other hand, to get an estimate of the actual time needed for composition search, the execution time of the algorithm is measured based on a Raspberry Pi (Model B). This small computer is equipped with an ARM11 CPU operated at 700 MHz and 512 MB SDRAM, which is running the Raspbian Linux distribution.

9.3 Probability of Finding Service Compositions

The second performance measure to be investigated quantifies the ability of finding a valid service composition for property concepts. It is of interest because it signifies the probability that the value of a property can be calculated from a set of provided property concepts in the ontology, which also represents the degree of implicitly and explicitly redundant information in the system. For a certain configuration of provided properties the probability is determined by dividing the number of those property concepts in the ontology, which can be derived from the provided properties, by the total number of property concepts in the ontology. This probability is influenced by the distribution of the provided property concepts in the ontology (i.e., the average semantic distance between these property concepts), even if the number of provided properties is constant. Therefore, during design time it is useful to perform an automated evaluation to find the distribution of a fixed number of provided properties in the system, for which the probability of finding a service composition is maximized. Alternatively, the lowest number of provided

properties can be found, when a fixed probability is given. For example, a car manufacturer wants to reduce the number of sensors in a car such that a certain probability of finding a composition is given, and thus, ensuring safety and availability with the lowest possible hardware effort. Because not every property concept can be provided by a sensor, the system engineer has to mark all property concepts in the ontology for which a sensor exists. Then the automated evaluation creates all combinations of sensors and calculates the corresponding probability to find a service composition. This procedure can also be performed for dedicated properties only, in order to ensure that a certain number of service compositions exist for a given property concept (e.g., a property concept used in a safety relevant application), such that the required level of dependability is reached.

In the experiments the upper bound for the probability of finding a service composition with the given ontologies is determined for different numbers of provided property concepts. This upper bound is similar to the probability of finding a service composition when the given number of sensors is distributed optimally. Similarly to the runtime experiments, the individual testruns are conducted with distinct numbers of provided property concepts selected from a pool of 14 providable concepts. Among the combinations of N provided properties, the highest probability value (i.e., the upper bound) is selected.

Note: The probability of finding compositions is not a quality measure for the algorithm, but for the combination of the system ontology and a particular set of provided properties.

9.4 Simulation of Propagation of Uncertainty

To demonstrate the propagation of uncertainty with a non-trivial realistic use case, a concrete service composition from the presented automotive example ontology is used. The use case is non-trivial, since it does not only include linear functions for transfer concepts, but it also demonstrates the usage of non-linear functions. Furthermore, two composition trees are combined by weighted averaging, which improves the overall uncertainty of the service composition.

In the next section, the used service composition is presented in detail. Thereafter, the assumptions about the allowed input values and their uncertainties are specified, which are based on technically feasible parameters of a realistic vehicle. The methodology for finding the sensitivity indices for the transfer functions of the use case with MATLAB is explained in Section 9.4.3. This section also describes the implementation of the calculation of the worst-case uncertainty using the equations in Section 7.2, and how the worst-case uncertainty is compared to the actually achievable uncertainty.

9.4.1 Service Composition Overview

The automotive use case to demonstrate the propagation of uncertainty in a composition tree describes the combination of inputs from four different sensors to derive the steering angle applied at the steering wheel of a vehicle. This composition tree is useful in a vehicle that uses a mechanical link between the steering wheel and the front wheels (i.e., no steer-by-wire), when the sensor measuring the steering angle fails, like described in the example scenario above.

Figure 9.3 provides an overview of the service composition use case. It shows four transfer functions (i.e., the rectangles) with the corresponding equations, and an ellipse denoting the combination of two sub-trees by weighted averaging. The numbers beside the arrays near to the arrows (e.g., : 10) indicate the number of uncertainty intervals used for this path during the use case simulation. In order to achieve a more accurate representation of non-linear transfer functions, the numbers are higher between such functions.

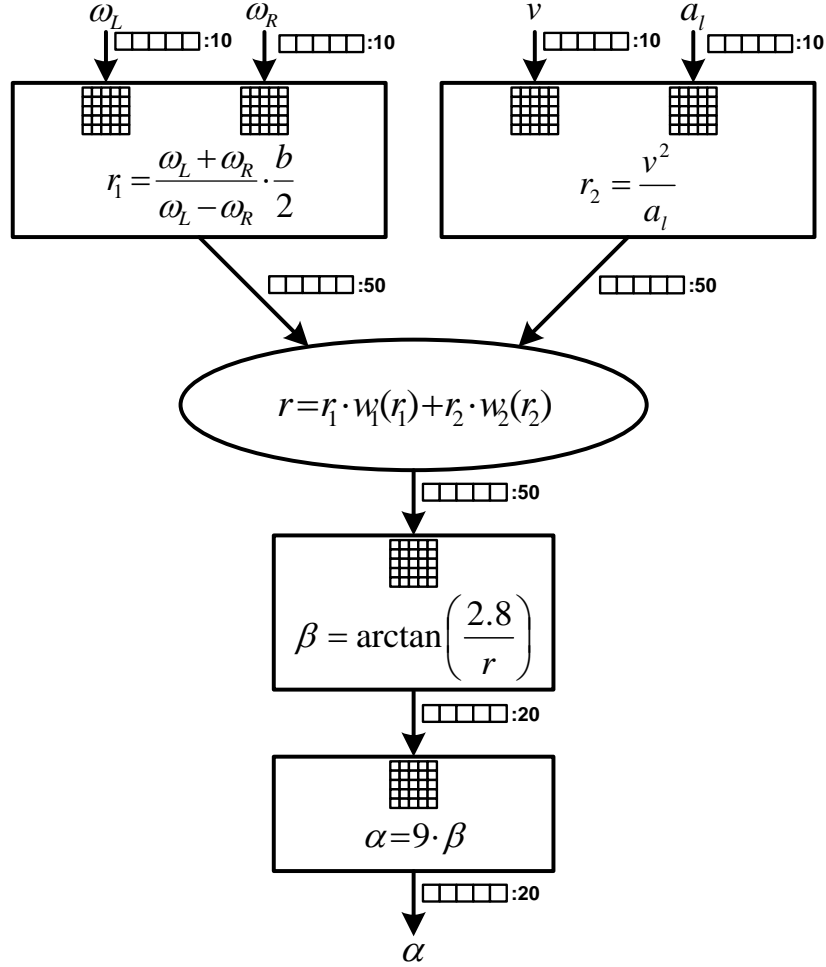


Figure 9.3: Overview of service composition use case for propagation of uncertainty.

The example uses the rotational sensors of the left and right wheels of the vehicle to obtain the angular velocity of the left ω_L and the right ω_R wheels. From these measurements the curve radius r_1 of the vehicle can be calculated as shown in Equation 9.1, where b denotes the constant track width of the vehicle.

$$r_1 = \frac{\omega_L + \omega_R}{\omega_L - \omega_R} * \frac{b}{2} \quad (9.1)$$

Additionally, velocity measurements v – which can, for instance, originate from a GPS receiver – and the lateral acceleration a_l are combined to also derive the curve radius r_2 . Equation 9.2 presents the corresponding formula to perform this calculation. Ideally (i.e., with perfect sensors and no uncertainties in the transfer functions), r_1 and r_2 would come to the same result.

$$r_2 = \frac{v^2}{a_l} \quad (9.2)$$

Both equations are multivariate (i.e., they depend on multiple variables) and non-linear, and hence, the sensitivity matrices cannot be obtained by simply filling the main diagonal with the slope (i.e., the first order partial derivative) of the transfer function. These functions have to be analyzed over their whole range of possible input values to determine the coefficients in the matrices (see Section 7.2.2).

The output of both functions is fused using the weighted averaging method, described in Section 7.3, to get the combined curve radius value r . In the corresponding equation (Equation 9.3), the weights for r_1 and r_2 depend on the values of r_1 and r_2 themselves, which is indicated by the two weight functions $w_1(r_1)$ and $w_2(r_2)$. The reason for this is that the weights are defined for certain intervals of the value range of the curve radius, and thus, the weight functions select the appropriate weights for the current interval.

$$r = r_1 * w_1(r_1) + r_2 * w_2(r_2) \quad (9.3)$$

Since the curve radius of the vehicle depends on the current angle of the front wheels, this value is transformed by the univariate non-linear arc tangent function according to Equation 9.4. It provides the current angle of the front wheels β . The constant a in the equation denotes the wheelbase (i.e., the distance between front and rear wheels) of the vehicle.

$$\beta = \arctan\left(\frac{a}{r}\right) \quad (9.4)$$

Usually in a car a gear box is mounted between the steering wheel and the wheels, which mechanically transforms the steering angle into an angle of the front wheels, such that the steering wheel can be turned almost 360 degrees to the left and the right, while the wheels only turn a few degrees. This behaviour is modelled in a transfer concept by a simple linear transformation that multiplies the angle of the wheels β with a constant gain factor g to calculate the steering angle α , as shown in Equation 9.5.

$$\alpha = g * \beta \quad (9.5)$$

9.4.2 Parameter Assumptions

The parameters for the use case are chosen such that they meet assumptions about a realistic car. Especially those parameters that are used in non-linear equations might strongly influence the sensitivity indices, and thus, the worst-case bound for the uncertainty. Therefore, it is important

to carefully select the parameters and avoid unrealistic sets of input values to the transfer functions. For instance, in normal driving conditions, the left and right wheels of a non-holonomic vehicle will rotate in the same direction. Hence, all cases where the rotational speeds of the wheels have opposite directions do not have to be considered in the analysis of sensitivity indices. Restricting such parameters results in more realistic sensitivity indices and reduces the time needed to numerically analyze transfer functions.

In general, the parameters are based on the following assumptions about the vehicle:

- The vehicle cannot exceed a speed of $100 \text{ m/s} \hat{=} 360 \text{ km/h}$.
- The rolling circumference (i.e., perimeter) of the wheels is 2000 mm – which applies, for instance, for *225/45 ZR 18 Y* wheels.
- The minimum turning radius is 5 m .
- The left and right wheels have to rotate in the same direction.
- The gear transmission ratio of the gear box between steering wheel and the front wheels is $9 : 1$, which means that a full revolution of the steering wheel results in a 40° angle of the front wheels. Hence, the gain factor $g = 9$.
- Wheelbase $a = 2.8 \text{ m}$
- Track width $b = 1.5 \text{ m}$
- Since it is obviously not possible to determine the steering angle of the vehicle – with the presented composition tree – when the vehicle is not in motion, a minimum absolute velocity of 4 m/s is required.

Generally, the analysis of sensitivity indices of a transfer function has to exceed the technically feasible range of input values, especially for non-linear functions. This is due to the fact, that in case of an input value that is perfectly aligned with the analyzed value range, the uncertain deviation could already be outside of the analyzed range. As a consequence, the real output uncertainty of the transfer function could be higher than the calculated worst-case bound. Hence, the overlap of the analysis should at least cover the expected uncertainty of the transfer function. This has also been applied for the minimum velocity value, where the analysis has been extended to an absolute minimum of 3 m/s .

Based on these assumptions, the parameters in Table 9.2 have been calculated and rounded to values that overestimate the assumptions. During the determination of the sensitivity indices only sets of input values are considered, if the corresponding parameters lie within the bounds specified in the table.

As pointed out in Section 8.3, it is not possible to numerically evaluate positive or negative infinity of inputs to transfer functions. In the use case, only the input to Equation 9.4 (i.e., the curve radius) has an allowed range including infinity. Hence, the maximum value of the curve radius $|r|$ was chosen to be 500 km . The maximum error that might result from this selection is

Table 9.2: Parameters for the simulation of the use case.

Parameter	min Value	max Value	Unity
ω_L	-350	350	rad/s
ω_R	-350	350	rad/s
$ v $	3	100	m/s
a_l	-15	15	m/s^2
$ r $	5	∞ (approx. 500k)	m
β	-30	30	$^\circ$
α	-270	270	$^\circ$

1m deviation on a driving distance of 1km, when the maximum value is used in stead of infinity. This error is assumed to be covered by the model uncertainty θ of the transfer function.

The input at the leafs of the composition tree consists of sensors that also suffer from uncertainty. Table 9.3 summarizes the assumed standard deviations σ of the sensors in the use case.

Table 9.3: Assumed sensor uncertainties for use case.

Sensor	Symbol	Standard Deviation
left wheel	ω_L	0.5 rad/s
right wheel	ω_R	0.5 rad/s
velocity	v	1.0 m/s
lateral acceleration	a_l	0.2 m/s^2

Those uncertainties which are introduced where it is assumed that the transfer functions of the use case are not a perfect model of the reality, are presented in Table 9.4.

Table 9.4: Model uncertainties for use case.

Transfer Function	Symbol	Standard Deviation
Equation 9.1	θ_{r1}	0.25 m
Equation 9.2	θ_{r2}	0.25 m
Equation 9.4	θ_β	0.01 rad
Equation 9.5	θ_α	0.0 rad

9.4.3 Implementation in MATLAB

The algorithms needed for the simulations are implemented in MATLAB. In Section 8.3 the determination of sensitivity indices, as well as the weights for weighted averaging, have been discussed. Here use case specific implementations regarding the calculation of the worst-case uncertainty, and the comparison of the obtained uncertainty bounds with the true uncertainty of the composition tree, are discussed.

Determination of Worst-case Uncertainty Bounds

The worst-case uncertainty of a composition tree is calculated beginning at those transfer functions, which are directly connected to sensors (i.e., the transfer functions corresponding to Equations 9.1 and 9.2). For both equations, which calculate the curve radius, the worst-case uncertainty is calculated as described in Section 7.2.4, where the uncertainties of the sensors (see Table 9.3) are used as input uncertainties. Thereafter, the combined uncertainty for both equations (i.e., subtrees) is calculated with Equation 7.12, where the previously calculated worst-case uncertainties act as the input to the weighted averaging algorithm. The output uncertainty of the fusion is propagated through the transfer function of Equation 9.4, and subsequently, to Equation 9.5. As the resulting worst-case uncertainty is the variance of the composition tree, its square root is taken to get the standard deviation, which has the same physical dimension as the function output.

Evaluation of Uncertainty Bounds

To evaluate the calculated worst-case uncertainty bounds of the sensitivity intervals, these worst-case uncertainties are compared to the real uncertainties obtained by applying the transfer functions of the composition tree for the whole input space and the specified sensor uncertainties as input. This evaluation demonstrates that the procedures for calculating the worst-case uncertainty provide a reasonable bound for the composition tree of the use case.

For each set of input values, the correct output value is calculated first, and afterwards the Monte Carlo method [45], as described in the first supplement to the GUM, is applied to determine the uncertainty for this particular set of input values. As several combinations of input values result in the same output of the steering angle, only the maximum uncertainty among all these combinations is used. The corresponding pseudo code for this evaluation of the uncertainty bounds is presented in Algorithm 14.

The algorithm starts by initializing the array oi , which divides the possible output space into n equally sized output intervals. Then, for each possible combination of input values iv the uncertainty is calculated. This is done by systematically selecting a new valid vector of input values (line 3). Afterwards, the optimal output value for that input vector is calculated, to obtain the actual steering angle for that input vector (i.e., without any assumed disturbances of sensor inputs or model uncertainties). In order to do that, it has to be checked if the input vector is technically feasible and complies with the parameter requirements of the use case (e.g., the vehicle speed obtained by the wheel rotations has to correspond – within a certain range – with the speed input value). In case one requirement is violated, the next input vector is selected. Otherwise, the equations for the curve radius are applied (lines 7 and 8) to obtain the curve radius estimations of both composition sub-trees $r1$ and $r2$, respectively. Since, at this step no deviations are assumed, both sub-trees have to derive the same value for the curve radius, as otherwise at least one of the sensors would deliver a wrong value – which is in contrast to the assumption. But, as floating point operations in computer systems do not provide perfect results, and also due to discretization of input values, a small deviation ϵ ($0.1m$ in the experiments) of the values $r1$ and $r2$ is allowed. If this condition is not satisfied, the input vector is technically not reasonable for the use case. The combined curve radius r is calculated by weighting the individual estimations

Algorithm 14 Pseudo code for evaluation of uncertainty bounds

```

1: function EVALUATEUNCERTAINTYBOUNDS( $n, S$ )
   Input: Number of output intervals  $n$ ;
           Number of samples  $S$  drawn for each function input vector
   Output: Array of true worst-case uncertainties

2:    $oi \leftarrow initializeOutputIntervals(n)$ 
3:   while  $iv \leftarrow getInputVector()$  do
4:     if  $checkInputVector(iv) == false$  then
5:       continue
6:     end if
7:      $r1 \leftarrow ((iv.\omega l + iv.\omega r)/(iv.\omega l - iv.\omega r)) * 0.75$ 
8:      $r2 \leftarrow iv.v^2/iv.a$ 
9:     if  $abs(r1 - r2) > \epsilon$  then
10:      continue
11:    end if
12:     $r \leftarrow r1 * weight\_w(r1) + r2 * weight\_v(r2)$ 
13:    if  $checkRadius(r) == false$  then
14:      continue
15:    end if
16:     $\alpha \leftarrow arctan(2.8/r) * 9$ 
17:    for all  $i \in 1 \dots S$  do
18:       $deviations \leftarrow randDeviations()$ 
19:       $sv[i] \leftarrow executeCompTree(iv, deviations)$ 
20:    end for
21:     $\sigma \leftarrow calculateUncertainty(sv, \alpha)$ 
22:     $idx\_out \leftarrow getOutputIndex(oi, \alpha)$ 
23:    if  $oi[idx\_out] < \sigma$  then
24:       $oi[idx\_out] \leftarrow \sigma$ 
25:    end if
26:  end while
27:  return  $oi$ 
28: end function

```

$r1$ and $r2$. In the pseudo code (line 12), $weight_w(r1)$ and $weight_v(r2)$ denote the individual weights of both variables, which depend on the input interval for weighted averaging (see Section 7.3). If also the combined radius satisfies the requirements about the minimum curve radius, then the actual steering angle α is calculated (line 16).

Now the Monte Carlo method is applied to calculate the uncertainty for this particular input vector. Therefore, the complete composition tree is executed repeatedly (i.e., S times) with the input vector and randomly drawn deviation – with gaussian distributions – for sensor inputs and model uncertainties (lines 17 - 20). After all samples have been stored in the vector sv , the uncertainty is calculated using Equation 9.6.

$$\sigma^2 = \frac{1}{S-1} \sum_{i=1}^S (sv_i - \alpha)^2 \quad (9.6)$$

In the simulation, the Monte Carlo method was conducted with $S = 1000$ samples. Note, that the mean value of the S samples might slightly deviate from the steering angle α , due to the non-linear behaviour of the composition tree. As by assumption α is the correct value for the current input vector, the uncertainty is calculated based on α instead of the mean value of all samples. This leads to a small overestimation of the uncertainty, which is smaller than $|\epsilon|$, that follows the equations:

$$\begin{aligned} \epsilon &= \alpha - \text{mean}(sv) \\ \epsilon^2 &= \sigma_\alpha^2 - \sigma_{\text{mean}(sv)}^2 \end{aligned}$$

Finally, after the uncertainty has been calculated, the output interval is determined, which is necessary due to the fact that several different input vectors result in the same steering angle output value, imperfect handling of floating point numbers in computer systems, and the resulting discretization of the output range. For the determined output interval, it is checked if the current uncertainty exceeds the previously stored value. And if this is the case, the old uncertainty is replaced by the current one. At the end, the output interval array oi stores the maximum uncertainties that have been observed for all output intervals.

CHAPTER 10

Results

This chapter presents and discusses the outcome of the experimental evaluation and the simulation, which lead to the following three results about:

1. the runtime of composition search,
2. the probability of finding a valid service composition in the given ontologies, and
3. the calculation of an uncertainty bound.

The first two results have been obtained from experiments with a the depth first search implementation of the composition search on an embedded computer system that performs the search within the given ontologies. As the propagation of uncertainty in a service composition is independent from the composition search algorithm and the computer system, the corresponding results were obtained from MATLAB simulations. A detailed description of the experiments and the simulation can be found in Chapter 9. The last section of this chapter is dedicated to a discussion of the ability of the presented reconfiguration framework to fulfill the system requirements introduced in Section 3.1.

10.1 Experimental Results and Interpretation

In this section the results of the experiments performed on the Raspberry Pi computer are shown and interpreted first. Afterwards, the outcome of the MATLAB simulations are provided. Therefore, one specific service composition was selected to evaluate the propagation of uncertainty in a service composition that includes non-linear transfer functions.

10.1.1 Results of Experiments for Determination of Runtime and Probability to find Service Composition

The execution time as well as the probability to find a valid service composition were determined for different numbers of provided property concepts, which corresponds to the number of

SOM entries (cf. Section 9.2 and 9.3). Within each of the following figures, the results of the experiments are shown for the different ontology sizes (i.e., *full*, *medium* and *small* – as defined in Table 9.1).

In Figure 10.1 measurements about the number of property concepts visited during composition search (including multiple visits of the same concept), are presented as an indicator for the runtime complexity. Among all testruns with the same number of SOM entries, the maximum number of visited property concepts are plotted. If no properties are provided (i.e., the number of SOM entries is zero), only a few transfer and property concepts are visited. For instance, in case of the full ontology, property concepts were visited 48 times (including multiple visits of the same property concept), while the maximum number of visits is 313 (cf. number of input relations to transfer concepts in Table 9.1). This is due to the fact that the required input properties of the transfer concepts in the proximity of the missing property cannot be provided and the algorithm stops to further explore that path in the ontology graph. As the number of SOM entries increases, also the probability that all inputs to transfer concepts can be provided rises. Consequently, the search paths become longer and more concepts are visited (e.g., the maximum number of visits in the full ontology increases from 120 to 150 when the number of provided property concepts is raised from 3 to 4). When the number of SOM entries reaches a certain value (e.g., 6 for the medium ontology), the number of visited property concepts stagnates and finally starts to decrease again. This is due to the fact, that the inputs of more transfer concepts, which are closer to the missing property, are directly provided, and therefore, the search algorithm does not have to identify service compositions for those inputs. Valid service compositions are found with lower semantic depth. Furthermore, the figure shows a noticeable drop in the number of visited properties between 13 and 14 SOM entries. This is caused by the amount of combinations which are possible with the given SOM entries, and thus, the number of performed testruns. In case of 14 entries, only a single testrun is possible, since all 14 providable concepts have to be selected from the pool. In contrast, when 13 providable concepts have to be selected, 14 different configurations are possible, where in each configuration one providable property is not used. If this unused property has a central location in the ontology graph (i.e., most of the search paths have to visit this concept), the maximum runtime increases significantly.

A similar behaviour is observed in the execution time measurements, as depicted in Figure 10.2. The figure shows the maximum time measured in μs , until a valid service composition is found or until a negative response is returned. These measurements also include a temporal overhead for the process management of the operating system, which results in the more irregular shape compared to the previous figure.

Figure 10.3 provides the results about the upper bound of the probability to find a valid service composition for one missing property concept. This means, that for each number of SOM entries a combination of provided properties was looked for, based on which the percentage of successful composition search operations is reported. For the full ontology, the highest percentage of 73.47% was reached with 13 (and more) SOM entries, while for the other two ontologies the maximum was already observed with 11 provided properties. Therefore, with the given setting, where a pool of 14 directly providable properties exists, not all of them have to be active to achieve the highest probability of finding a service composition. Obviously, the probability has to increase monotonously up to 100%, and thus, better results can be achieved by adding more

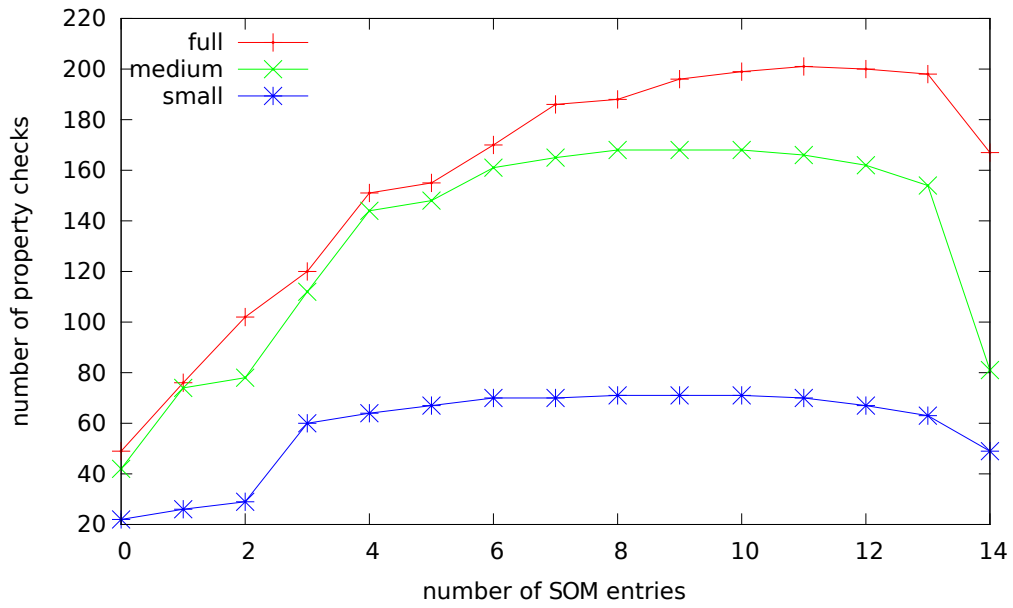


Figure 10.1: Maximum number of checks for property provision over the number of entries in the SOM.

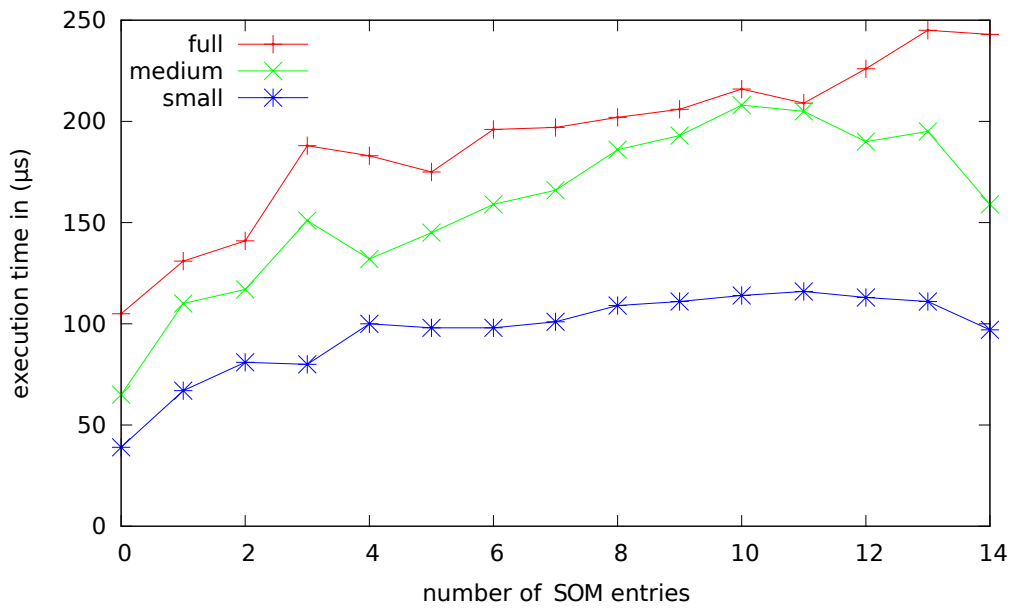


Figure 10.2: Maximum execution time on Raspberry Pi over the number of entries in the SOM.

properties to the pool of providable properties.

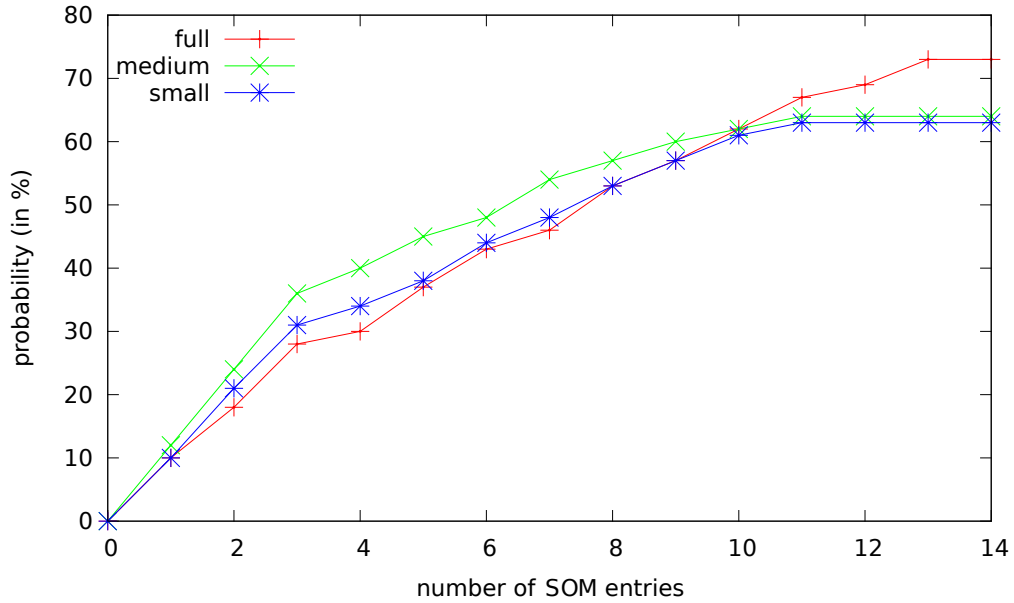


Figure 10.3: Upper bound of probability to successfully find a service composition for a property concept over the number of entries in the SOM.

It has to be noted that the results about runtime and probability of a service composition highly depend on the size and granularity of the system ontology. However, a correlation of the total number of input relations to transfer concepts and the maximum number of properties visited (and the maximum execution time, respectively) as discussed in Section 6.5, can be observed in Figures 10.1 and 10.2. In the presented algorithm (cf. Algorithm 3) a property concept cannot be visited more often than input relations to transfer concepts exist. When comparing the numbers given in Table 9.1 with the maximum number of property checks in Figure 10.1, the expected behaviour is confirmed. In order to actually observe the worst-case runtime behaviour for a given system ontology, which is the case if all input relations to transfer concepts have been examined, the ontology as well as the provided properties have to be perfectly aligned. At no transfer concept the algorithm might track back without having examined all input properties, which essentially requires that for all transfer concepts each input is providable (i.e., directly or by deduction). On the other hand, the algorithm must not omit exploring any sub-graph SG of the ontology because all inputs of the transfer concepts, which connect SG to the rest of the ontology, are directly provided. Since these conditions are usually not found for most ontologies of real applications, runtime measurements are not supposed to exhibit values that are linearly increasing with the number of input relations of transfer concepts. However, the upper bound for the runtime increases linearly with these input relations.

The observed probabilities to find a service composition encourage the application of the presented dynamic reconfiguration approach for CPSs, especially to implement the *never-give-up* strategy for cases where the fault assumption is violated. Furthermore, during the design process of the CPS the evaluation of the probability to find service compositions can be used

to compute and increase the reliability of the system with dynamic reconfiguration by ensuring 100% probability for property concepts that are required for system services. Thus the composition search mechanism can be used to optimize explicitly and implicitly redundant information in the system.

10.1.2 Results about Propagation of Uncertainty

The results of the simulation of calculating the worst-case uncertainty for the use case, as well as the uncertainty values obtained by executing the transfer functions of the composition tree are compared to show that the presented framework is capable of providing a reasonable estimation of the uncertainty of a composition tree. Additionally, to demonstrate the benefit of weighted averaging, the use case has been simulated in two different versions. The *short* simulation does not include weighted averaging, and it only uses the path with velocity v and lateral acceleration a_l as input. For the *full* simulation, the complete use case (as shown in Subsection 9.4.1) is examined.

Figure 10.4 presents the results of the short simulation of the use case. It can be seen in the figure, that the calculated worst-case uncertainty bound provides an upper limit for the actual uncertainty obtained by executing the composition tree. The same applies for the results of the simulation of the full use case, which is depicted in Figure 10.5. But in contrast to the short simulation without averaging, the worst-case bounds as well as the actual uncertainty have been significantly improved in the full simulation.

In Figure 10.6 the weights for weighted averaging are shown for each interval of the curve radius. The lower bars denote the weights for the sub-tree using the velocity and the lateral acceleration, while the upper bars represent the sub-tree using the wheel sensors. As the weights strongly depend on the input uncertainties of the sensors, the sub-tree of the wheel sensors has in most cases lower weights, since in the example a higher uncertainty is assumed for wheel sensors compared to the other types of sensors. It has to be noted that the intervals of the curve radius are not evenly distributed, but much smaller for a smaller curve radius – i.e., 0.2m interval at 5m curve radius – and large for a high curve radius – i.e., ∞ for a radius higher than 25,000m. Furthermore, the weights are symmetric for negative curve radiuses.

10.2 Requirement Fulfillment

Theoretical analyses of the models and algorithms, as well as the above results of the experimental evaluations reveal that the proposed dynamic reconfiguration framework is capable to fulfill the requirements defined in Section 3.1. In the following these requirements are revisited in order to discuss their satisfaction in more detail.

10.2.1 R1: High Dependability at Low Cost

The results of the experiments for determining the probability to find a valid service composition show that the composition search algorithm effectively identifies implicit redundancy in the system. Obviously, the probability to find a service composition for a given property concept strongly depends on the completeness of the system ontology and the number of available

10. RESULTS

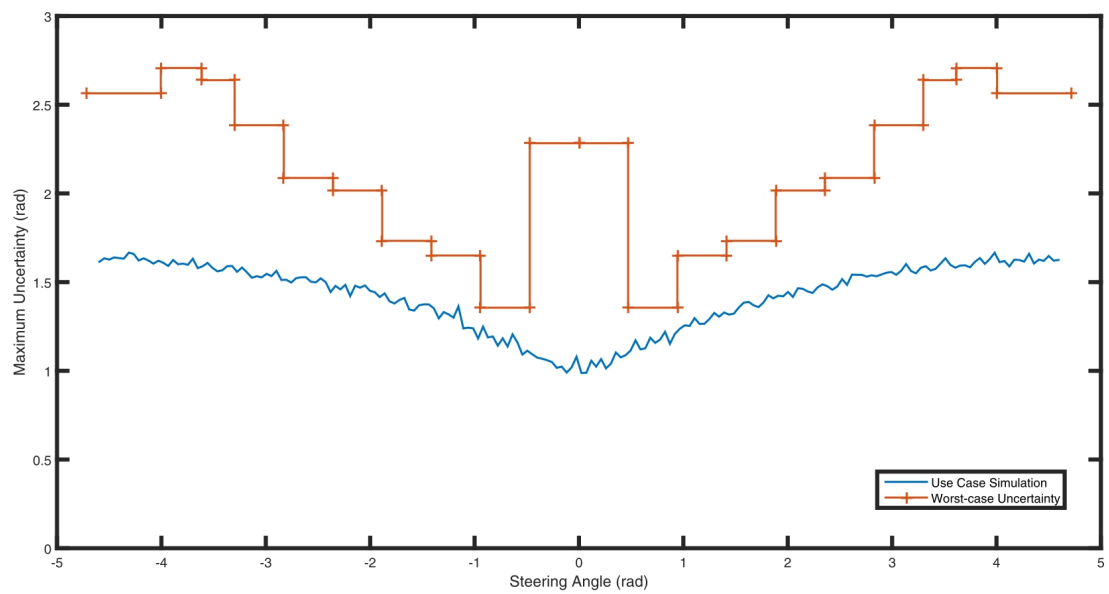


Figure 10.4: Comparison of worst-case uncertainty and simulation of short use case.

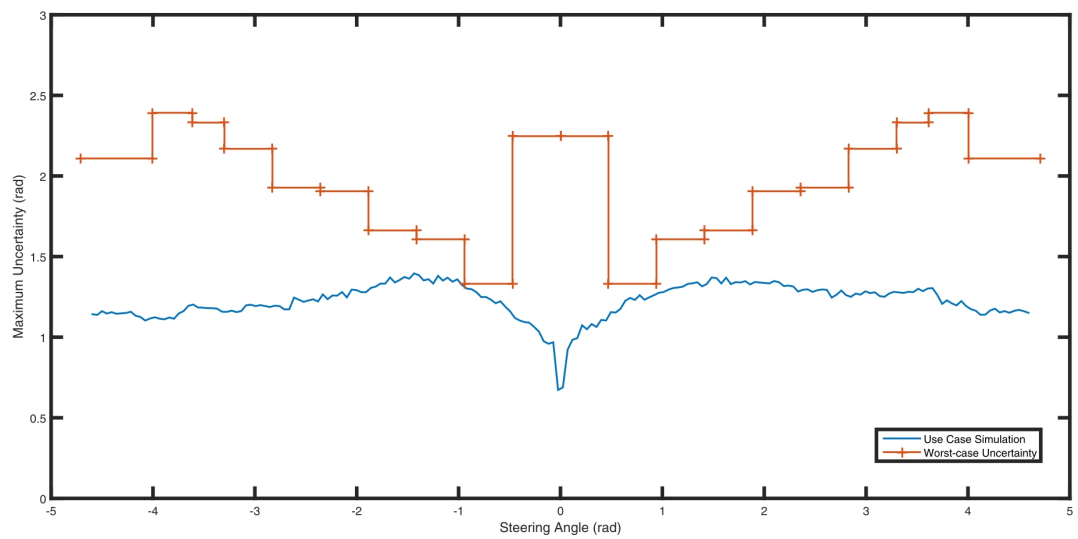
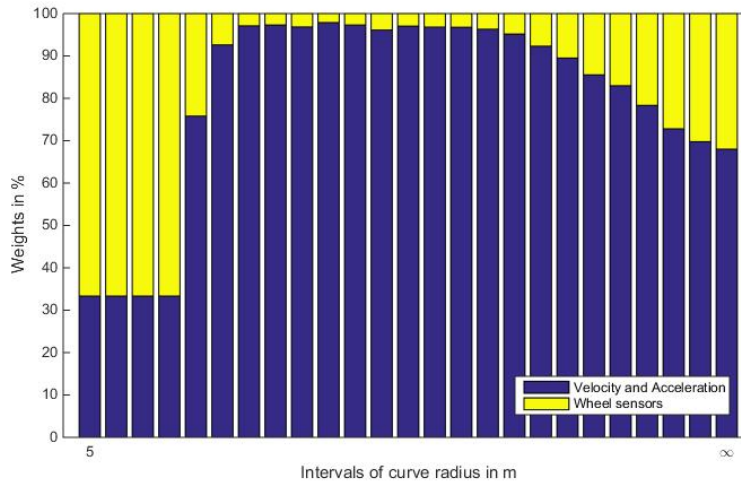


Figure 10.5: Comparison of worst-case uncertainty and simulation of full use case.



of a system, guaranteed temporal bounds for reconfiguration can hardly be given at design time. In contrast, the worst-case runtime of the framework presented within this work only depends on the system model ontology, which usually does not change unless physical modifications of the CPS are conducted. Therefore, a WCET analysis for a particular ontology can be performed at design time, allowing to pre-validate that the system is able to meet all deadlines.

10.2.4 R4: Semantic Correctness of Configurations

Configurations in the system are semantically correct, if for all service compositions the semantics of the itoms requested by services matches the semantics of the corresponding provided itoms. This is particularly true for dynamically created service compositions. Approaches that simply try to connect available services with syntactically matching interfaces might fail to ensure semantic correctness. For instance, connecting a component that outputs a temperature in the range of $-20^{\circ}C$ to $150^{\circ}C$ with a component that requires such temperature as input. Without a semantic check the oil temperature of a car can be confused with the temperature of the cooling water. The proposed dynamic reconfiguration framework avoids this problem by explicitly modelling semantic relations in the system ontology and mapping service interface specifications to the property concepts in the ontology graph. Hence, the composition search algorithm is only able to combine components when their interface specification is semantically compatible.

10.2.5 R5: Accuracy of Information

The proposed methodology to determine the uncertainty of information of service compositions can be used to ensure that service compositions are only employed to provide input itoms for other services, if their worst-case uncertainty is within the specified bounds. As shown by the experimental results of the simulation of uncertainty propagation in service compositions, the methodology overestimates the true uncertainty, which is a prerequisite to guarantee that the service composition provides the required quality of information.

Conclusion and Future Work

This chapter concludes the thesis with a summary of the major contributions and achieved results. Furthermore, an outlook on possible future research topics is provided.

11.1 Conclusion

Within this thesis a framework for dynamic reconfiguration of embedded real-time systems has been presented, which uses a knowledge base with a system model for making reconfiguration decisions. It can be applied for CPSs to increase their dependability by autonomous reconfiguration of component interactions. Thus, in case of component failures system services can be recovered by substituting the failed component by a group of other available components that provide a semantically equivalent service compared to the original component. Upon the removal of components the system is able to preserve the services affected by the removal. After new components are introduced, the system can automatically enable services which could not be provided before due to missing input information. Furthermore, during system design engineers can use the framework to evaluate the degree of redundancy in the system and to minimize the number of explicitly redundant components.

The main contributions of this thesis to obtain a dynamically reconfigurable CPS, which go beyond the state-of-the-art, are:

- the definition of an appropriate component-based system architecture with dedicated building blocks for reconfiguration,
- specification of a description language for a knowledge base with a system model and ontology,
- provision of time-bounded algorithms for service orchestration, and
- a methodology to calculate the worst-case value uncertainty of information provided by service compositions.

The system architecture consists of distributed nodes, where one node hosts a centralized component responsible for the configuration of application components. In order to establish fault-tolerance and avoid a single point of failure, this component can be replicated. For the purpose of reconfiguration it incorporates a system ontology acting as a knowledge base for service composition search. On the remaining nodes components are located, which offer their services to other components. The overall system services are provided by applications that consist of interacting groups of components. In the system ontology, the logical structure and properties of the system and its environment, as well as the relations between these properties, are modelled. Beside the static system ontology, dynamic changes in the service availability are modelled in the knowledge base. It contains a mapping of required input properties and provided output properties of services to the properties modelled in the system ontology.

When dynamic reconfiguration is required (e.g., due to a component failure), the system ontology is searched for service compositions that provide information with the requested semantics. This search operation is performed as a graph search in the system ontology, starting from the property that is requested. A tree-shaped sub-graph, which consists of properties and relations between these properties, is determined, that is rooted at the requested property and all properties at the leafs are mapped to an available service. This tree structure is used to automatically create a transfer service that reads the values of all input properties and outputs the value of the requested property.

In order to determine whether the accuracy of the value provided by a service composition is sufficient for the requesting application, a methodology is described within this thesis that calculates the worst-case uncertainty of the provided values. This is achieved by offline analysis of the influence – called sensitivity indices – of transfer functions, which represent the relations between properties, on the uncertainty of the output value of the function. After a service composition is proposed by the search algorithm, an uncertainty algorithm uses the uncertainty of all input properties to the composition and propagates them through the transfer functions by multiplication with the sensitivity indices. If the resulting worst-case bound is below the required value, the service composition can be enabled.

An automotive use case has been described that is used to conduct experiments with the implemented algorithms. For the experiments of service composition search an example ontology was built and a set of example services, which provide certain properties, were added to the service-to-ontology mapping. The objective of the experiments was to complement the theoretical analysis with information about the runtime of the algorithm with measurements. To demonstrate the effectiveness of the proposed calculation of the worst-case uncertainty for a non-trivial service composition of the automotive use case, a MATLAB simulation was performed.

The results of the experiments and simulations, as well as theoretical analyses underlined the applicability of the proposed dynamic reconfiguration approach for CPSs. The framework is capable to fulfill the requirements identified in the thesis for embedded real-time systems, which is not the case for existing approaches found in related literature. Accordingly, the framework allows to minimize the amount of explicit redundancy in the system in order to achieve a defined level of dependability. With dynamic reconfiguration it is possible to react to unforeseen events, and thus, to provide a never-give-up strategy. As the runtime of composition search mainly de-

depends on the static system ontology, a WCET analysis can be performed at design time, such that it can be ensured that reconfiguration completes within a bounded time. By the construction of the system ontology and the composition search algorithm the semantic correctness of component interactions is guaranteed. Finally, calculating the worst-case value uncertainty of service compositions allows to reject those compositions, which do not satisfy the requirements on the value accuracy.

11.2 Future Work

While this thesis already provides results on the applicability of knowledge-based dynamic reconfiguration for CPSs, further challenges remain for future work.

- **Initial state of state-based transfer functions:** For the semantics of a service composition it is irrelevant whether it contains state-based transfer functions (e.g., integrals), because the semantics is only defined by the interconnection of concepts, which does not necessarily require the transfer function to be known. However, the correctness of the output provided by the composition depends on the initial value of the state variables. In a system within which a faulty component is exchanged by a similar component, the problem of the initial state can be solved by providing the state variables at the component interface. Upon the exchange of a component, the last known state – possibly with additional roll-forward state estimation [78] – is used as the initial state for the new component. This is not always possible for service compositions, since a composition might include state variables that have not been explicitly processed in the system before. Additional research is required to investigate, how the initial state can be obtained from provided properties.
- **Conditional search for service compositions:** Some system services might only be required when certain conditions hold. For example, an adaptive curve light is only permitted with a driving speed below 50km/h . Likewise, the passenger air bag does not need to engage in case no passenger is present. On the other hand, several relations between properties are only useful for service compositions in certain modes of operation. For instance, when the clutch is open, the rotational speeds at the gearbox and the wheels are not correlated with the rotations at the engine, but a relation exists if the clutch is closed. When certain conditions about modes or property value ranges are provided, these can be evaluated by the composition search algorithm. As such a composition does not have to be valid in all other modes, the possibility of finding a valid configuration can be increased, or different solutions for distinct modes can be proposed. The main question is whether this can be done without obtaining execution times that are infeasible for reconfiguration of CPSs.
- **Machine learning of relations:** Currently the system ontology is assumed to be created by system engineers during the design time of a CPS. Therefore only properties and relations are included, which are known by these engineers. However, in many cases relations exist which have not yet been analyzed or which might vary between different implementations of the CPS. Machine learning techniques can be applied to automatically identify relations

between properties while the system is operational. The possibilities for machine learning reach from learning of parameters of relations already defined between properties, to fully autonomous recognition of properties and their relations. For instance, an engineer indicates a possible relation between properties and the learning mechanism calculates the corresponding transfer function.

- ***Uncertainty in hybrid service compositions:*** In general, a service composition can include properties represented by continuous values (e.g., temperature of oil), as well as discrete values (e.g., clutch state signal [open/closed]). Usually, discrete values are used in transfer functions to distinguish between different modes, within which the transfer behaviour might differ significantly. For instance, in the first gear of a car the transmission rate of the rotational speed from the engine to the wheels is lower than in the second gear, while in the reverse gear the rotation even changes its direction.

The current methodology for uncertainty propagation considers discrete values to have no uncertainty. Similarly, the output uncertainties of transfer functions, which use discrete properties as inputs, are not affected by a discrete property. Instead the worst-case uncertainty among all possible modes has to be assumed, leading to a pessimistic valuation of uncertainty. While this assumptions are valid for many discrete properties appearing in a CPS, they are often obtained from a continuous value that is discretized (e.g., by using thresholds). Consequently also the discrete property inherits some uncertainty of the continuous properties. This leads to the question, how the uncertainty of discrete properties additionally influences the output uncertainty of a service composition, when considering that also mode switches become uncertain?

- ***Evaluate statistical independence of information flows in service composition:*** In this thesis it was assumed that information flows within service compositions are statistically independent. This reduces the effort to calculate the worst-case uncertainty of a composition, as otherwise also covariances of dependent inputs would have to be considered. Statistical independence implies that each input of the service composition appears in exactly one transfer function. But the composition search algorithm can also return solutions within which this condition is violated. This might compromise the calculation of the worst-case uncertainty. Further research has to be made to either ensure that no service composition is used with statistically dependent paths, with the disadvantage of reducing the number of possible solutions. Alternatively, the uncertainty propagation methodology is extended to include covariances of inputs and transfer functions.

Bibliography

- [1] R. Adler, D. Schneider, and M. Trapp. Engineering Dynamic Adaptation for Achieving Cost-Efficient Resilience in Software-Intensive Embedded Systems. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 21–30, March 2010.
- [2] Rasmus Adler, Ina Schaefer, Mario Trapp, and Arnd Poetzsch-Heffter. Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems. *ACM Trans. Embed. Comput. Syst.*, 10(2):20:1–20:39, January 2011.
- [3] A. Allard and N. Fisher. Sensitivity analysis in metrology: study and comparison on different indices for measurement uncertainty. In F. Pavese, M. Bär, A. B. Forbes, L. M. Linares, C. Perruchet, and N. F. Zhang, editors, *Advanced Mathematical and Computational Tools in Metrology and Testing: VIII*, pages 1–6. World Scientific Publishing Company, 2009.
- [4] AMADEOS Consortium. D2.2 – AMADEOS Conceptual Model. Technical report, 2015.
- [5] A. Avizienis. Fault-tolerant computing: An overview. *Computer*, 4.
- [6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan 2004.
- [7] J. Balasubramanian, A. Gokhale, A. Dubey, F. Wolf, Chenyang Lu, C. Gill, and D.C. Schmidt. Middleware for resource-aware deployment and configuration of fault-tolerant real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 69–78, April 2010.
- [8] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. *CoRR*, abs/1409.0413, 2014.
- [9] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *Configurable Distributed Systems, 1998. Proceedings. Fourth International Conference on*, pages 35–42, May 1998.

BIBLIOGRAPHY

- [10] E. Bini and G.C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *Computers, IEEE Transactions on*, 53(11):1462–1473, Nov 2004.
- [11] C. Bolchini, A. Miele, and M.D. Santambrogio. Tmr and partial dynamic reconfiguration to mitigate seu faults in fpgas. In *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT '07. 22nd IEEE International Symposium on*, pages 87–95, Sept 2007.
- [12] M. Bozzano and A. Villaflorita. *Design and Safety Assessment of Critical Systems*. CRC Press, 2010.
- [13] R.W. Butler, J.L. Caldwell, and B.L. Di Vito. Design strategy for a formally verified reliable computing platform. In *Computer Assurance, 1991. COMPASS '91, Systems Integrity, Software Safety and Process Security. Proceedings of the Sixth Annual Conference on*, pages 125–133, Jun 1991.
- [14] G. C. Buttazzo. Periodic task scheduling. In *Hard Real-Time Computing Systems*, volume 24 of *Real-Time Systems Series*, pages 79–118. Springer US, 2011.
- [15] Mark Campanelli, Raghu Kacker, and Rüdiger Kessel. Variance gradients and uncertainty budgets for nonlinear measurement functions with independent inputs. *Measurement Science and Technology*, 24(2):025002, 2013.
- [16] Bo Chen, ChrisN. Potts, and GerhardJ. Woeginger. A Review of Machine Scheduling: Complexity, Algorithms and Approximability. In Ding-Zhu Du and PanosM. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 1493–1641. Springer US, 1999.
- [17] M. Cirinei, E. Bini, G. Lipari, and A. Ferrari. A flexible scheme for scheduling fault-tolerant real-time tasks on multiprocessors. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.
- [18] Flavin Cristian. Understanding Fault-tolerant Distributed Systems. *Commun. ACM*, 34(2):56–78, February 1991.
- [19] J. Cubo, C. Canal, and E. Pimentel. Context-Aware Service Discovery and Adaptation Based on Semantic Matchmaking. In *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on*, pages 554–561, May 2010.
- [20] Mohammad I. Daoud and Nawwaf Kharm. A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 68(4):399 – 409, 2008.
- [21] Bureau d’Enquêtes et d’Analyses pour la sécurité de l’aviation civile (BEA). Final report on the accident on 1st June 2009 to the Airbus A330-203 registered F-GZCP operated by Air France flight AF 447 Rio de Janeiro – Paris, 5 July 2012. On-line: <http://www.bea.aero/docspa/2009/f-cp090601.en/pdf/f-cp090601.en.pdf> (accessed: 2015-08-21).

-
- [22] L. D’Orazio, F. Visintainer, and M. Darin. Sensor networks on the car: State of the art and future challenges. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.
 - [23] Jr. E. G. Coffman, M. R. Garey, and D. S. Johnson. An Application of Bin-Packing to Multiprocessor Scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.
 - [24] T. Erl. *Service-Oriented Architecture: Concepts, Technology and Design*. Prentice Hall, 5th edition, 2006.
 - [25] I. Estévez-Ayres, L. Almeida, M. Garcia-Valls, and P. Basanta-Val. An architecture to support dynamic service composition in distributed real-time systems. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC ’07. 10th IEEE International Symposium on*, pages 249–256, May 2007.
 - [26] I. Estévez-Ayres, P. Basanta-Val, M. Garcia-Valls, J.A. Fisteus, and L. Almeida. Qos-aware real-time composition algorithms for service-based applications. *Industrial Informatics, IEEE Transactions on*, 5(3):278–288, Aug 2009.
 - [27] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.
 - [28] European Technology Platform on Smart System Integration. European Roadmap: Smart Systems for Automated Driving. Technical report, 2015.
 - [29] Jérôme Euzenat, Pavel Shvaiko, et al. *Ontology matching*, volume 18. Springer, 2007.
 - [30] Lei Feng, DeJiu Chen, and M. Torngrén. Self configuration of dependent tasks for dynamically reconfigurable automotive embedded systems. In *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, pages 3737–3742, Dec 2008.
 - [31] A Ferrero, M. Prioli, and S. Salicone. Processing Dependent Systematic Contributions to Measurement Uncertainty. *Instrumentation and Measurement, IEEE Transactions on*, 62(4):720–731, April 2013.
 - [32] W.J. Fleming. New automotive sensors – a review. *Sensors Journal, IEEE*, 8(11):1900–1921, Nov 2008.
 - [33] L. Gantel, S. Layouni, M. Benkhelifa, F. Verdier, and S. Chauvet. Multiprocessor task migration implementation in a reconfigurable platform. In *Reconfigurable Computing and FPGAs, 2009. ReConFig ’09. International Conference on*, pages 362–367, Dec 2009.
 - [34] J.L. Gersting. *Mathematical Structures for Computer Science*. W. H. Freeman, 2007.
 - [35] R. Goebel, R.G. Sanfelice, and A. Teel. Hybrid dynamical systems. *Control Systems, IEEE*, 29(2):28–93, April 2009.
 - [36] S. Goss, S. Aron, J.L. Deneubourg, and J.M. Pasteels. Self-organized shortcuts in the Argentine ant. *Naturwissenschaften*, 76(12):579–581, 1989.

BIBLIOGRAPHY

- [37] T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International Journal of Human-Computer Studies*, 43(5-6):907 – 928, 1995.
- [38] H. Guo. *Automotive Informatics and Communicative Systems: Principles in Vehicular Networks and Data Exchange: Principles in Vehicular Networks and Data Exchange*. Premier reference source. Information Science Reference, 2009.
- [39] O. Höftberger and R. Obermaisser. Ontology-based Runtime Reconfiguration of Distributed Embedded Real-Time Systems. In *Proc. of the 16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2013)*, June 2013.
- [40] O. Höftberger and R. Obermaisser. Runtime Evaluation of Ontology-based Reconfiguration of Distributed Embedded Real-Time Systems. In *Proc. of the 12th IEEE International Conference on Industrial Informatics (INDIN 2014)*, pages 544 – 550, July 2014.
- [41] R. Isermann. Supervision, fault-detection and fault-diagnosis methods – An introduction . *Control Engineering Practice*, 5(5):639 – 652, 1997.
- [42] Rolf Isermann. Model-based fault-detection and diagnosis – status and applications. *Annual Reviews in Control*, 29(1):71 – 85.
- [43] International Standardization Organization (ISO). ISO 26262-1:2011(en) Road vehicles – Functional safety – Part 1: Vocabulary, 2011.
- [44] Joint Committee for Guides in Metrology. JCGM 100: Evaluation of Measurement Data - Guide to the Expression of Uncertainty in Measurement. Technical report, JCGM, 2008.
- [45] Joint Committee for Guides in Metrology. JCGM 101: Evaluation of Measurement Data - Supplement 1 to the “Guide to the Expression of Uncertainty in Measurement”– Propagation of Distributions Using a Monte Carlo Method. Technical report, JCGM, 2008.
- [46] M. Kafil and I. Ahmad. Optimal task assignment in heterogeneous distributed computing systems. *Concurrency, IEEE*, 6(3):42–50, Jul 1998.
- [47] R. E. Kalman. A New Approach to Linear Filtering and Prediction Problems. *ASME Journal of Basic Engineering*, 1960.
- [48] Soo Dong Kim and Soo Ho Chang. Soar: An extended model-based reasoning for diagnosing faults in service-oriented architecture. In *Services - I, 2009 World Conference on*, pages 54–61, July 2009.
- [49] J.S. Kinnebrew, W.R. Otte, N. Shankaran, G. Biswas, and D.C. Schmidt. Intelligent resource management and dynamic adaptation in a distributed real-time and embedded sensor web system. In *Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC '09. IEEE International Symposium on*, pages 135–142, March 2009.

-
- [50] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1st edition, 1997.
 - [51] H. Kopetz. Fault containment and error detection in the time-triggered architecture. In *Autonomous Decentralized Systems, 2003. ISADS 2003. The Sixth International Symposium on*, pages 139–146, April 2003.
 - [52] H. Kopetz. On the Fault Hypothesis for a Safety-Critical Real-Time System. In Manfred Broy, IngolfH. Krüger, and Michael Meisinger, editors, *Automotive Software – Connected Services in Mobile Networks*, volume 4147 of *Lecture Notes in Computer Science*, pages 31–42. Springer Berlin Heidelberg, 2006.
 - [53] H. Kopetz. Pulsed Data Streams. In Bernd Kleinjohann, Lisa Kleinjohann, RicardoJ. Machado, CarlosE. Pereira, and P.S. Thiagarajan, editors, *From Model-Driven Design to Resource Management for Distributed Embedded Systems*, volume 225 of *IFIP International Federation for Information Processing*, pages 105–114. Springer US, 2006.
 - [54] H. Kopetz. The Complexity Challenge in Embedded System Design. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 3 – 12, May 2008.
 - [55] H. Kopetz. The Rationale for Time-Triggered Ethernet. In *Real-Time Systems Symposium, 2008*, pages 3–11, Nov 2008.
 - [56] H. Kopetz. *Real-Time Systems – Design Principles for Distributed Embedded Systems*. Springer, 2nd edition, 2011.
 - [57] H. Kopetz. A Conceptual Model for the Information Transfer in Systems-of-Systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*, pages 17–24, June 2014.
 - [58] H. Kopetz, C. El Salloum, B. Huber, R. Obermaisser, and C. Paukovits. Composability in the time-triggered system-on-chip architecture. In *SOC Conference, 2008 IEEE International*, pages 87–90, Sept 2008.
 - [59] H. Kopetz, B. Frömel, and O. Höftberger. Direct versus Stigmergic Information Flow in Systems-of-Systems. In *Proceedings of the 10th IEEE International Conference on System of Systems Engineering (SoSE 2015)*, pages 36–41, 2015.
 - [60] H. Kopetz and G. Grunsteidl. TTP-a protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, Jan 1994.
 - [61] H. Kopetz and N. Suri. Compositional design of RT systems: a conceptual basis for specification of linking interfaces. In *Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on*, pages 51–60, may 2003.
 - [62] Hermann Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, Jan 2003.

- [63] K. Kotecha and A. Shah. Adaptive scheduling algorithm for real-time operating system. In *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 2109–2112, June 2008.
- [64] H. H. Ku. Notes on the use of propagation of error formulas. *Journal of Research of the National Bureau of Standards. Section C: Engineering and Instrumentation*, 70C:263–273, Oct. 1966.
- [65] J.H. Lala and R.E. Harper. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82(1):25–40, Jan 1994.
- [66] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.
- [67] J.-C. Laprie. Dependable Computing: Concepts, Limits, Challenges. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS)*, June 1995.
- [68] J.C. Laprie. Dependability: Basic Concepts and Terminology. In J.C. Laprie, editor, *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*, pages 3–245. Springer Vienna, 1992.
- [69] J.L.M. Lastra and I.M. Delamer. Semantic web services in factory automation: fundamental insights and research roadmap. *Industrial Informatics, IEEE Transactions on*, 2(1):1–11, Feb 2006.
- [70] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. *Handbooks in operations research and management science*, 4:445–522, 1993.
- [71] Bach Thanh Le, Rose Dieng-Kuntz, and Fabien Gandon. On ontology matching problems. *ICEIS (4)*, pages 236–243, 2004.
- [72] E.A. Lee. Cyber physical systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369, May 2008.
- [73] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems – A Cyber-Physical Systems Approach*. <http://LeeSeshia.org>, 2nd edition, 2015.
- [74] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, and Sebastian Thrun. Anytime search in dynamic graphs. *Artificial Intelligence*, 172(14):1613 – 1643, 2008.
- [75] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, January 1973.
- [76] M. Loskyll, J. Schlick, S. Hodek, L. Ollinger, T. Gerber, and B. Pirvu. Semantic service discovery and orchestration for manufacturing processes. In *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–8, Sept 2011.

- [77] F. Manola, E. Miller, D. Backett, and I. Herman. RDF Primer – Turtle version, Jan. 2014.
- [78] V. Mikolášek and Hermann Kopetz. Roll-forward recovery with state estimation. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on*, pages 179–186, March 2011.
- [79] G. A. Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review*.
- [80] I. Moir and A. Seabridge. *Aircraft Systems: Mechanical, Electrical and Avionics Subsystems Integration*. 2011.
- [81] F. Moo-Mena and K. Drira. Reconfiguration of web services architectures: A model-based approach. In *Computers and Communications, 2007. ISCC 2007. 12th IEEE Symposium on*, pages 357–362, July 2007.
- [82] F. Moo-Mena, J. Garcilazo-Ortiz, L. Basto-Diaz, F. Curi-Quintal, and F. Alonzo-Canul. Defining a self-healing qos-based infrastructure for web services applications. In *Computational Science and Engineering Workshops, 2008. CSEWORKSHOPS '08. 11th IEEE International Conference on*, pages 215–220, July 2008.
- [83] R. Neches, R. E. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. R. Swartout. Enabling Technology for Knowledge Sharing. *AI Magazine*, 12(3):36–56, 1991.
- [84] Mark Nicholson. Health monitoring for reconfigurable integrated control systems. In Felix Redmill and Tom Anderson, editors, *Constituents of Modern System-safety Thinking*, pages 149–162. Springer London, 2005.
- [85] R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz. The time-triggered System-on-a-Chip architecture. In *Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on*, pages 1941–1947, June 2008.
- [86] R. Obermaisser and P. Peti. A Fault Hypothesis for Integrated Architectures. In *Intelligent Solutions in Embedded Systems, 2006 International Workshop on*, pages 1–18, June 2006.
- [87] Dong-Ik Oh and T.P. Bakker. Utilization Bounds for N-Processor Rate Monotone Scheduling with Static Processor Assignment. *Real-Time Systems*, 15(2):183–192, 1998.
- [88] B. Osterloh, H. Michalik, S.A. Habinc, and B. Fiethe. Dynamic partial reconfiguration in space applications. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 336–343, July 2009.
- [89] I. Pandis, J. Soldatos, A. Paar, J. Reuter, M. Carras, and L. Polymenakos. An ontology-based framework for dynamic resource management in ubiquitous computing environments. In *Embedded Software and Systems, 2005. Second International Conference on*, Dec 2005.

BIBLIOGRAPHY

- [90] Massimo Paolucci, Takahiro Kawamura, TerryR. Payne, and Katia Sycara. Semantic matching of web services capabilities. In Ian Horrocks and James Hendler, editors, *The Semantic Web — ISWC 2002*, volume 2342 of *Lecture Notes in Computer Science*, pages 333–347. Springer Berlin Heidelberg, 2002.
- [91] P. Pedreiras, P. Gai, L. Almeida, and G.C. Buttazzo. Ftt-ethernet: a flexible real-time communication protocol that supports dynamic qos management on ethernet-based systems. *Industrial Informatics, IEEE Transactions on*, 1(3):162–172, Aug 2005.
- [92] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008.
- [93] J. Pitman. *Probability*. Springer Texts in Statistics. Springer-Verlag, 1993.
- [94] David Powell. Failure Mode Assumptions and Assumption Coverage. In Brian Randell, Jean-Claude Laprie, Hermann Kopetz, and Bev Littlewood, editors, *Predictably Dependable Computing Systems*, ESPRIT Basic Research Series, pages 123–140. Springer Berlin Heidelberg, 1995.
- [95] K. Ramamritham and J.A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67, Jan 1994.
- [96] A. Rasche and A. Polze. Configuration and dynamic reconfiguration of component-based applications with Microsoft .NET. In *Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on*, pages 164–171, May 2003.
- [97] A. Rasche and A. Polze. Dynamic reconfiguration of component-based real-time software. In *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*, pages 347–354, Feb 2005.
- [98] J.B. Rawlings and D.Q. Mayne. *Model Predictive Control: Theory and Design*. Nob Hill Pub., 2009.
- [99] Javier Resano, Juan Antonio Clemente, Carlos Gonzalez, Daniel Mozos, and Francky Catthoor. Efficiently scheduling runtime reconfigurations. *ACM Trans. Des. Autom. Electron. Syst.*, 13(4):1–12, 2008.
- [100] Andrei Rădulescu and Arjan J. C. van Gemund. On the Complexity of List Scheduling Algorithms for Distributed-memory Systems. In *Proceedings of the 13th International Conference on Supercomputing, ICS '99*, pages 68–75. ACM, 1999.
- [101] A. Saltelli, M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli, M. Saisana, and S. Tarantola. *Global Sensitivity Analysis: The Primer*. Wiley, 2008.
- [102] Larry D. Schröder, David L. Sjoquist, and Paula E. Stephan. *Understanding regression analysis*. Quantitative Applications in the Social Sciences. Sage Publications, 1986.

- [103] P. Shvaiko and J. Euzenat. Ontology Matching: State of the Art and Future Challenges. *Knowledge and Data Engineering, IEEE Transactions on*, 25(1):158–176, Jan 2013.
- [104] M. Soika. A sensor failure detection framework for autonomous mobile robots. In *Intelligent Robots and Systems, 1997. IROS '97., Proceedings of the 1997 IEEE/RSJ International Conference on*, volume 3, pages 1735–1740, Sep 1997.
- [105] John Soldatos, Ippokratis Pandis, Kostas Stamatis, Lazaros Polymenakos, and James L. Crowley. Agent based middleware infrastructure for autonomous context-aware ubiquitous computing services. *Computer Communications*, 30(3):577 – 591, 2007. Special Issue: Emerging Middleware for Next Generation Networks.
- [106] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for Hard-Real-Time systems. *Real-Time Systems*, 1(1):27–60, 1989.
- [107] Christoph Stiller, Fernando Puente León, and Marco Kruse. Information fusion for automotive applications – an overview. *Information Fusion*, 12(4):244 – 252, 2011. Special Issue on Information Fusion for Cognitive Automobiles.
- [108] Robert Tarjan. Depth-first search and linear graph algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 114–121, Oct 1971.
- [109] Sebastian Thrun. Toward robotic cars. *Communications of the ACM*, 53(4):99–106, April 2010.
- [110] Mario Trapp, Rasmus Adler, Marc Förster, and Janosch Junger. Runtime adaptation in safety-critical automotive systems. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering, SE'07*, pages 308–315, Anaheim, CA, USA, 2007. ACTA Press.
- [111] Paolo Traverso and Marco Pistore. Automated composition of semantic web services into executable processes. In SheilaA. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *The Semantic Web – ISWC 2004*, volume 3298 of *Lecture Notes in Computer Science*, pages 380–394. Springer Berlin Heidelberg, 2004.
- [112] W.T. Tsai, Qian Huang, Jingjing Xu, Yinong Chen, and R. Paul. Ontology-based Dynamic Process Collaboration in Service-Oriented Architecture. In *Service-Oriented Computing and Applications, 2007. SOCA '07. IEEE International Conference on*, pages 39–46, June 2007.
- [113] P. Ulbrich, F. Franzmann, F. Scheler, and W. Schroder-Preikschat. Design by uncertainty: Towards the use of measurement uncertainty in real-time systems. In *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*, pages 275–278, June 2012.
- [114] J. Valappil and C. Georgakis. Systematic estimation of state noise statistics for extended Kalman filters. *AIChE Journal*, pages 292–308, Feb. 2000.

BIBLIOGRAPHY

- [115] Venkat Venkatasubramanian, Raghunathan Rengaswamy, Kewen Yin, and Surya N. Kavuri. A review of process fault detection and diagnosis: Part I: Quantitative model-based methods. *Computers & Chemical Engineering*, 27(3):293 – 311, 2003.
- [116] P. Veríssimo and L. Rodrigues. *Distributed Systems for System Architects*. Advances in Distributed Computing and Middleware. Springer US, 2012.
- [117] Y. Wand and R. Weber. Research Commentary: Information Systems and Conceptual Modeling – A Research Agenda. *Information Systems Research*, 13(4):363 – 376, 2002.
- [118] Hai Wang and Zengzhi Li. A Semantic Matchmaking Method of Web Services Based on SHOIN+ (D)*. In *Services Computing, 2006. APSCC '06. IEEE Asia-Pacific Conference on*, pages 26–33, Dec 2006.
- [119] Shen Shen Wu and David Sweeting. Heuristic algorithms for task assignment and scheduling in a processor network. *Parallel Computing*, 20(1):1–14, 1994.
- [120] Xianrong Zheng and Yuhong Yan. An efficient syntactic web service composition algorithm based on the planning graph model. In *Web Services, 2008. ICWS '08. IEEE International Conference on*, pages 691–699, Sept 2008.
- [121] S. Zilberstein. Using Anytime Algorithms in Intelligent Systems. *AI Magazine*, 17(3):73–83, 1996.

Acronyms

- ADC** analog-to-digital converter. 131
- BEA** Bureau d'Enquêtes et d'Analyses pour la sécurité de l'aviation civile. 3
- CCM** Component Configuration Manager. 36–39, 55, 57, 88, 109, 132
- CEC** Component Execution Control. 38, 39, 42
- CEM** Component Execution Manager. 36, 37, 45, 88
- CPS** cyber-physical system. 7, 9, 11, 12, 14, 16–18, 21–24, 32, 33, 35, 47, 48, 50, 55, 61, 73, 74, 87, 89, 93, 150, 153–158
- CSCP** Component Schedule & Communication Planning. 40–42, 45
- CSR** Central Service Registry. 37–43, 47, 49, 66, 68, 79, 80, 116, 132
- ECU** electronic control unit. 2, 3, 24, 133
- EMI** electromagnetic interference. 14, 15
- FCR** fault containment region. 14–16
- FMEA** Failure Mode and Effect Analysis. 16
- FTA** Fault Tree Analysis. 16
- FTTI** fault tolerant time interval. 3, 13, 22, 80
- GUM** Guide to the Expression of Uncertainty in Measurement. 18, 94, 96–98, 103
- HAZOP** Hazard and Operability. 16
- IIF** input interface. 84
- IP** intellectual property. 2

- LP** local peripheral. 36, 37, 43
- LSM** Local Service Monitor. 37, 38
- LSR** Local Service Registry. 37, 39, 40, 42, 43
- MTBF** mean time between failure. 12
- MTTF** mean time to failure. 12
- MTTR** mean time to repair. 12
- OFAT** One-factor-at-a-time. 98, 99
- OIF** output interface. 84
- OWL** Web Ontology Language. 30
- PDF** probability density function. 94
- QoS** Quality of Service. 24, 26, 27
- RCM** Resource Configuration Manager. 39–42, 132
- RDF** Resource Description Framework. 110–112, 117, 120, 134
- RM** Resource Monitor. 40–43
- ROS** Robot Operating System. 129, 131, 132
- RUI** Relied Upon Interface. 9, 11
- RUMI** Relied Upon Message Interface. 9, 10, 42
- RUPI** Relied Upon Physical Interface. 9, 10, 43
- SOE** Service Orchestration Engine. 37, 39–45, 47, 51, 65, 132
- SOM** service-to-ontology mapping. 40, 43, 44, 56, 65, 68, 137, 138, 148
- TMR** triple modular redundancy. 14
- URI** Uniform Resource Identifier. 110, 111
- WCET** worst-case execution time. 22, 77, 88, 89, 91, 154, 157

Index

A	
abstraction	17
accuracy	18
anytime algorithm	80
assertion	14
assumption coverage	14
attribute	7
availability	12
B	
babbling idiot	15
breadth first search	75
building block	35
C	
category	17
Central Service Registry	39
channel	
cyber	9
stigmergic	10
child concept	54
component	9, 42
Component Configuration Manager	36, 38
Component Execution Control	38
Component Execution Manager	36, 37
Component Schedule & Communication Plan- ning	41
component-based design	9
composition search	12, 68
composition tree	67
concept	17, 47, 48
property concept	49
property type concept	51
structure concept	48
transfer concept	50
transfer type concept	51
conceptual landscape	17
conceptual modeling	17
confidentiality	12
configuration	11
control loop	8
controlled object	8
CPS	7
cyber-physical system	7
D	
deadline	8
firm	8
hard	8
soft	8
dependability	12
depth first search	74
distribution	
Gaussian	94
normal	94
diversity	16
E	
embedded system	7
equivalence search	69
error	13
detection	13
random	19
systematic	19
estimation	18
event-triggered	8
experiment	133
explanation	18

F

failure	13
arbitrary	15
byzantine	15
crash	14
fail-stop	14
intermittent	15
masquerading	15
mode	14
omission	14
permanent	15
persistence	15
rate	12, 15
timing	15
transient	15
fault	13
development	13
forecasting	13
hypothesis	14
masking	14
operational	13
prevention	13
removal	13
tolerance	13
tolerant time interval	13

G

guided search	76
---------------------	----

I

information	18
inheritance	57
instance	47
integrity	13
itom	18, 56

K

knowledge	17
base	17

L

local peripheral	36, 42
Local Service Monitor	37
Local Service Registry	37

M

maintainability	12
mean	94
mean time between failures	12
mean time to failure	12
mean time to repair	12
measurand	18
measurement	18
function	18, 97
noise	19, 95
uncertainty	95
metrology	18
MTBF	12
MTTF	12
MTTR	12

N

never-give-up strategy	16
node	8, 35

O

object	110
ontology	17
description language	110
graph	57
matching	61
optimization	59
preprocessor	41, 57, 117

P

parent concept	54
partitioning	17
predicate	110
probability	94
process noise	96
process uncertainty	96
property	8
dynamic property	49
static property	49
publish-subscribe	129
publisher	129
pulsed data stream	87

R

random variable	94
-----------------------	----

real-time image	23
real-time system	7
distributed	8
reconfiguration	11
dynamic	11
evolutionary	11
programmed	11
recovery interval	15
redundancy	13
explicit	13
implicit	14
relation	47, 52
<i>has-property</i> relation	53
<i>input</i> relation	52
<i>is-part-of</i> relation	54
<i>output</i> relation	53
isA relation	54
pType relation	55
reliability	12
relied upon	
interface	9
message interface	9
physical interface	9
repair rate	12
replication	13
requirement	21
Resource Configuration Manager	40
Resource Monitor	40
Robot Operating System	129
RUI	9
RUMI	9
RUPI	9

S

safety	12
schedule	35
scheduling	87
segmentation	17
semantic	18
depth	73
distance	72
service description	37
width	73
sensitivity	

analysis	98
index	98
interval	101
matrix	103
service	9
-to-ontology mapping	55
composition	12, 68
failure	9
orchestration	12, 65
substitution	65
Service Orchestration Engine	41
simulation	133
slightly-off-specification	15
standard deviation	94
state	7
observable	8
partially observable	8
unobservable	8
subject	110
subscriber	129
system	
behavior	7
configuration	11
ontology	47
recovery	13
safety-critical	8

T

time-triggered	8
communication	35
topic	129
transfer	
behaviour	83
function	51
service	65, 83

U

uncertainty	18, 93
use case	133

V

value accuracy	93
variance	94
voter	14

INDEX

W

weighted averaging 106

CURRICULUM VITAE

Oliver HÖFTBERGER

PERSONAL INFORMATION

DATE OF BIRTH: October 24, 1982
ADDRESS: Hildebrandgasse 30/1/2, 1180 Vienna, Austria
PHONE: +43 (1) 58801 - 18229
EMAIL: oliver.hoeftberger@tuwien.ac.at

EDUCATION

SINCE 03/2010 **Doctoral Studies**, *Vienna University of Technology*, Austria.

02/2007 – 02/2010 **Master Studies: Computer Science**, *Vienna University of Technology*, Austria, Degree: Dipl.-Ing. (with distinction).

02/2007 – 07/2007 Exchange Semester at **University of Alicante**, Spain.

09/2003 – 01/2007 **Bachelor Studies: Computer Science**, *Vienna University of Technology*, Austria, Degree: BSc.

01/2003 – 09/2003 Military service, Hörsching, Austria.

09/1997 – 06/2002 **Technical High School**, *Höhere Technische Bundeslehranstalt Leonding*: EDV & Organisation, Austria, Matura with distinction.

WORK EXPERIENCE

SINCE 03/2010	Research assistant at <i>Vienna University of Technology, Institute of Computer Engineering</i> , Austria. Participation in research projects: universAAL – UNIVERSal open platform and reference Specification for Ambient Assisted Living. ACROSS – ARTEMIS CROSS-Domain Architecture. AMADEOS – Architecture for Multi-criticality Agile Dependable Evolutionary Open System-of-Systems.
------------------	---

SINCE 01/2013	Teaching assistant at <i>Vienna University of Technology, Institute of Computer Engineering</i> , Austria.
07/2011 – 02/2012	TTTech Computertechnik AG , Vienna, Austria. Self-employed project consultant.
05/2005 – 10/2009	Mühlegger GmbH , Traunkirchen, Austria. Part time job; safety inspection of measurement and control systems of nuclear power plants in Germany.
08/2004 – 09/2004	Elektro-Wasner GmbH&Co , Haag/Hausruck, Austria. Electric installation.
09/2002 – 12/2002	Elektro-Wasner GmbH&Co , Haag/Hausruck, Austria. Electric installation.
12/2000 – 07/2001	HAN Dataport CAD-Systeme GmbH , Linz, Austria. Part-time job; 3D Software development.

LANGUAGES

ENGLISH: Fluent
 GERMAN: Mothertongue
 SPANISH: Basic Knowledge