Philipp A. Baer

Platform-Independent Development of Robot Communication Software

kassel
university
press

This work has been accepted by the faculty of Electrical Engineering and Computer Science of the University of Kassel as a thesis for acquiring the academic degree of Doktor der Ingenieurwissenschaften (Dr.-Ing.).

Defense day: 04th December 2008

Bibliographic information published by Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie;
detailed bibliographic data is available in the Internet at http://dnb.d-nb.de.

Zugl.: Kassel, Univ., Diss. 2008
ISBN print: 978-3-89958-644-2
ISBN online: 978-3-89958-645-9
URN: urn:nbn:de:0002-6453

Printed by: Unidruckerei, University of Kassel
Printed in Germany

Für **Mutti** und **Mama**.

Mutti, du bleibst unvergessen.
* 29. September 1923 † 14. Oktober 2008

Contents

List of Figures

List of Tables

Abstract

Robotic appliances have gained significant influence in various fields of computer science research during the last fifteen years. Although industrial applications have thereby been in the focus for a long time, new technologies and the availability of suitable hardware particularly boosted the development of service robotics. Most recently, mobility and cooperation turned out as new challenges where in particular biologically inspired systems and swarm robotics are involved.

The degree of software distribution increases with a growing number of cooperating, mobile, and spatially distant robotic systems. This directly involves challenges for robot software architects: Mastering heterogeneity issues and finding suitable interaction schemes that are reconcilable with the capabilities of the communication media. We thereby have to consider that communication may fail or behave in an unexpected manner. The adopted software architecture has to address these challenges and facilitate collaboration by providing supportive functionality.

In this thesis, we address the platform-independent software development of communication infrastructures for mobile robotic systems. The Spica development framework supports the developer's needs by realising a platform-independent development methodology that rests upon the model orientation paradigm. It captures important entities and characteristics of the application domain in abstract, reusable models. Spica furthermore hides conflicting details from the underlying platform and cares about viable solutions self-dependently. An extensible transformation approach interprets and transforms these models. It finally creates a platform-specific, optimised realisation for a specific programming language. This involves two sorts of implementations: Data structures and module stubs. The latter ones are responsible for establishing the communication infrastructure. For this purpose, they employ a distributed resource discovery engine. It triggers channel establishment and cares about dynamic reconfiguration of the communication infrastructure. In order to facilitate the development of distributed architectures, Spica provides a decentralised monitoring facility for logging and monitoring in a highly dynamic, distributed environment.

We evaluate the viability of the Spica development framework in two different ways: Its integration into an elaborate software architecture for autonomous mobile robots is first introduced in detail. Based thereon, we present the composition and integration of two existing software components. The second part of the evaluation is more of analytic nature: We review several non-functional aspects of the Spica development framework such as applicability and efficiency. In the conclusions we finally revisit the advantages and weaknesses of the development approach.

Zusammenfassung

Anwendungen der Robotik haben in den letzten fünfzehn Jahren immer mehr an Einfluss im Bereich der Informatik gewonnen. Auch wenn dabei die industriellen Anwendungen lange im Fokus standen, wurde der Bereich der Servicerobotik durch neue Technologien und die zunehmende Verbreitung von entsprechenden Hardwarekomponenten gefördert. Vor allem Mobilität und Kooperation stellten in der jüngsten Vergangenheit neue Herausforderungen dar. Biologisch inspirierte, aber auch auf Schwarmverhalten basierende Systeme finden hierbei interessante Anwendungsgebiete.

Der Grad an Vernetzung zwischen Robotern und die damit einhergehende erhöhte Verteilung von Software Ressourcen verstärkt sich mit der Zahl an kooperierenden, mobilen und räumlich getrennten Robotersystemen. Die Herausforderungen für die Entwicklung von Robotersoftware sind davon direkt betroffen. Wichtig ist dabei, Lösungen für Heterogenitätsprobleme zu finden und Interaktionsmuster zu entwerfen, die mit den Fähigkeiten des Kommunikationsmediums verträglich sind. Da die Kommunikation unzuverlässig ist, oder sich unvorhergesehen verhalten kann, ist dies besonders bedeutsam. Die eingesetzte Softwarearchitektur muss daher einerseits auf die Herausforderungen zugeschnitten sein und andererseits die Zusammenarbeit zwischen Robotern unterstützen können.

Diese Arbeit behandelt die plattformunabhängige Entwicklung von Software für Kommunikationsinfrastrukturen mobiler Roboter. Das Spica Entwicklungssystem unterstützt dabei den Entwickler, indem es eine plattformunabhängige Entwicklungsmethodik bereitstellt, die auf dem Paradigma der modellbasierten Softwareentwicklung beruht. Mit Hilfe von abstrakten, wieder verwendbaren Modellen werden dabei wichtige Elemente und Charakteristiken des Anwendungsgebiets erfasst. Weiterhin kümmert sich Spica selbstständig um spezifische oder widersprüchliche Details der zugrunde liegenden Plattform, ohne dass der Entwickler eingreifen muss. Mittels eines flexiblen, erweiterbaren Ansatzes zur Modelltransformation werden Modelle interpretiert und in plattformspezifischen, optimierten Quelltext für eine spezifische Programmiersprache umgesetzt. Zwei Arten von Implementierungen werden dabei erzeugt: Datenstrukturen und Module, die sich um den Aufbau und die Verwaltung der Kommunikationsinfrastruktur kümmern. Ein verteilter Ansatz zur *Resource Discovery* überwacht dabei das Netzwerk, um Ressourcen automatisch entdecken zu können. Ändert sich die Verfügbarkeit einer Ressource, so informiert der Dienst seine Klienten, die dann ihre Konfiguration entsprechend ändern können. Spica enthält weiterhin einen verteilten Überwachungs- und Protokollierungsdienst, um die Entwicklung von verteilten Architekturen zu unterstützen. Dieser ist auf hochdynamische, verteilte Umgebungen ausgelegt.

Die Anwendbarkeit des Spica Entwicklungssystems wird auf zweierlei Arten untersucht. Zuerst wird die Integration in eine Softwarearchitektur für autonome, mobile Roboter sowie zwei weitere Komponenten vorgestellt. Der zweite Teil der Evaluation ist mehr analytischer Natur. Dabei werden verschiedene nichtfunktionale Aspekte wie die Handhabbarkeit und die Effizienz des Spica Entwicklungssystems untersucht. Abschließend werden die Vor- und Nachteile des Entwicklungsansatzes nochmals aufgegriffen.

Acknowledgements

I am deeply indebted to many people for supporting and encouraging me during my research work. The following lines gratefully acknowledge most of them. All the ones I missed unfortunately have to live with it. That's life.

My doctoral adviser **Kurt Geihs** supported my studies and the joint attempt together with my colleagues to establish robotics research in our group. **Gerhard K. Kraetzschmar** interested me in robotics more than seven years ago. I cannot assess whether this was a good or bad move. I will rather leave it to the judgement of others. In this context, I would like to mention **Martin Riedmiller** who nominated me for the RoboCup Middle Size League chair during the RoboCup German Open Championships 2008. A thank you also goes to my only recently appointed committee members **Alfred Zündorf** and **Klaus David**.

First of all, I would like to thank **Ingrid Gimple** who supported me during the last years and the exciting stages of my doctorate. My colleagues were a source of inspiration during the last about three and a half years; at least every now and then. **Roland Reichle**, **Michael Wagner**, and **Hendrik Skubch** are my co-workers in our robotic soccer team, a bunch of really bright minds. We would not have done so well in the past tournaments without them. Even more important, they keep up the motivation in the team with fresh ideas and pizza, sometimes also with beer. I would furthermore like to thank **Hans Utz** from whom I got insights into the current research at NASA Ames. **Michael Zapf** and **Thomas Weise** called my attention to areas other than robotics. This is especially important in order to not get stuck in the same ideas over and over again. Even though our research areas did not overlap that much, I do not want to leave out my colleagues **Steffen Bleul**, **Mohammad U. Khan**, and **Diana Comes** who were always up for a talk. I finally want to thank my former colleague **Jochen Fromm** for many inspiring discussions. Many thanks to our administrative stuff for giving me a reality check.

A really big thanks goes to all our students in the Carpe Noctem team. You all did a really great job in the last years! More personal, my thanks goes to **Christof Hoppe**, **Daniel Saur**, **Jens Wollenhaupt**, **Kai Baumgart**, **Stefan Triller**, **Till Amma**, **Florian Seute**, **Andreas Witsch**, **Andreas Scharf**, **Martin Segatz**, and **Stephan Opfer**. It was really fascinating and inspiring to work with you! Keep up this way!

In this last paragraph, I am not going to talk about the people in the last paragraph of a long list. The herein contained acknowledgements are rather second to none. My most grateful thanks go to two very special persons: my grandma **Annemarie Baer** who unexpectedly died in October 2008 and my mother **Ursula Baer**. Unlike me, you two never stopped believing in me and supported me so much during the last months and years. I really would not have made it without you!

Part I

Introduction

1 Introduction

"The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them."

— Sir William Lawrence Bragg
In Beyond Reductionism: New Perspectives in the Life Sciences, 1968

With a growing number of ready-made robotic systems, affordable hardware, and steadily increasing computation power, mobile robots gain popularity not only in research. The formation of new application domains and a general trend towards automation promotes application in the mass market. Some typical application domains of robotics are:

- Industrial automation and replacement of manpower
- *Search and Rescue* (SAR) as well as military application
- Service robotics in commercial and domestic environments
- Social, medical, and elderly care robotics

Even though some of these application domains are subject to controversial discussions, each one promotes and supports robotic research in its very own manner: Industrial automation requires dependable robots, military application stresses adaptivity to previously unknown scenarios and environments. Service robots furthermore promote specialisation and social robots focus on interaction and the *Human-Machine Interface* (HMI). Robotic research basically targets subsets of these domains. Besides aspects of mobility, also autonomy and intelligent behaviour are in the focus of research.

Robots gained even more importance with the spread of wireless communication technology. Even though network communication is only a small step towards natural interaction schemes, it boosts research in cooperation[1] and even collaboration between autonomous and mobile robots. This approach at least allows simulating the behaviour of humans or animals that follow specific goals cooperatively. Communication is here the link between individual participants that leads to the desired solution. It further facilitates the adaptation of complex behaviour in dynamically changing environments.

The hardware of a robot consists of a broad range of components such as different types of sensors or actuators. This is especially true for *Autonomous Mobile Robots* (AMR) that need to operate independently. Besides, it is common practise to use hardware tailored to a specific application domain. This, however, yields problems for software developers that need to deal

[1]We consider cooperation a mutual characteristic as defined in [17, Section 4.2].

with an increased degree of heterogeneity. Hence, a *Robotic Control Architecture* (RCA) – the control software that runs a robot – is mostly tailored to a specific hardware layout and the application domain of the respective robot. Each hardware component furthermore exhibits a specific behaviour in data acquisition and delivery. For multi-robot scenarios, control and monitoring capabilities have to be taken into account as well, which again influence heterogeneity considerations.

Whenever hardware is involved in heterogeneity considerations, it has implications on the software as well. This is because software typically interfaces hardware, so these two aspects are tightly coupled. The most important influencing factor on the overall complexity of a robot architecture must therefore not be disregarded: Different programming languages target different application domains and are thus not equally well suited for all kinds of applications. This is why in most cases an RCA adopts several languages; a reduction to the least common denominator is typically no option. However, if several languages are used, maybe in combination with different hardware platforms, heterogeneity issues get even more prevalent and have to be taken seriously. It is furthermore common practise to adopt third-party libraries or middleware platforms. These solutions have severe influence on the applicability and portability of an RCA.

A closer look at existing RCA solutions reveals several starting points for a development strategy that explicitly targets platform-independence: Middleware platforms are among the most well known approaches. They have shown their applicability for such a development process in numerous projects [134, 129, 51, 128, 45, 24]. However, some more solutions for robotic software development have been proposed in the last years, each of which emphasises other goals. Chapter 3 gives an overview of the most important representatives. A complete solution, suitable for many different platforms and applications is nevertheless still not available.

A middleware platform that supports different hardware and software platforms, and one that is in addition available for different programming languages most likely constitutes a quite complex solution. This, in turn, might render it inappropriate for a broad range of applications, possibly because of the lack of intuitive handling. It is mostly due to the increasing number of heterogeneity issues that asks for a new development approach for RCAs in general. As pointed out above, the design of the software as well as the selection of hardware components depends on the application domain of a robot. Which kind of sensory input is required? What actuators are indispensable for interaction with the environment? How is the system configuration? What is the task of the robot?

With Spica, we propose a new development approach that aims at providing an integrated yet platform-independent development framework based on abstract modelling capabilities. Compared to other approaches it explicitly addresses multi-robot scenarios and deals with their specific interaction characteristics. It finally builds a foundation for high-level development of robotic behaviour with no specific platform requirements.

1.1 Motivation

Based on the considerations discussed above, we will outline the motivation for a development approach covering communication infrastructures for AMRs. It must take into account the heterogeneity of involved systems and be applicable in a platform-independent fashion.

An RCA will presumably need to remain tailored to a specific robot, its hardware layout, or the task it has to perform in order to exploit the platform's full potential. A suitable development framework thereby supports development by providing measures for integrating platform-specific components and mastering heterogeneity not only in the local case. Below, we discuss three basic classes of characteristics a development framework has to address in this context.

1.1.1 Software Structure

The software structure is important characteristic of an RCA. It may be implemented monolithically where communication between functional entities is *in-process* and thus most efficient as no process boundary must be crossed. It nevertheless has the clear drawback that it is almost impossible to integrate existing solutions implemented for other platforms.

Modular software development puts things right by bridging the gap between efficiency and flexibility. In this context, we understand modularity as the ability of replacing components before or during runtime. This implies that functional entities of a software architecture must be identified and made accessible through appropriate interfaces. *In-process* communication is still possible with this approach but *out-of-process* communication – i.e. based on *Inter-Process Communication* (IPC) mechanisms – may be required for software components implemented as self-contained processes, realised in different languages, or executed remotely as independent entities.

Most RCAs clearly require the additional flexibility of a modular software architecture because of third party dependencies. This further allows distributing an RCA across a network by providing independent, self-contained modules that encapsulate specific functionality.

1.1.2 Development Methodology

Apart from the software structure, a robot has several further characteristics that need to be addressed appropriately. It is typically composed of several hardware components, each of which exhibits specific characteristics for access and control. An RCA might thus have to adopt existing driver implementations or other software components. A suitable development methodology must provide measures that facilitate integration of existing components and help to master heterogeneity issues in this context. This not only promotes reusability but also allows using the best-suited programming language and platform for a solution, simplifying the creation of specialised RCAs.

The interaction within a group of robots introduces another source of heterogeneity: The different robots may exhibit proprietary hardware and software platforms that complicate interaction and cooperation. A suitable development methodology assists developers in several dimensions: It must be applicable for different target platforms, master heterogeneity in hardware and software, and facilitate interaction between robots. The development methodology has to follow an abstract and platform-independent development approach in order to address these requirements. Distributed operation as required for interaction and cooperation further implies that each participating robot is able to deal with network unreliability.

1.1.3 Communication

Compared to human or animal behaviour where communication may be based on sound, gestures, or more indirect measures like pheromone traces [21], for example, AMRs typically fall back to network communication that is something completely different. It is, however, the best possible solution to date as the other approaches are still subject to active research and not generally applicable for AMRs yet. We assume AMRs to use wireless communication. Even though wire-based communication is possible as well [18], mobility and group operation of AMRs are better supported with wireless media.

AMRs depend on efficient, low latency, and non-obstructive communication. However, due to the nature of wireless media, these requirements cannot be guaranteed: Wireless media are more vulnerable to error and media access is more expensive compared to their wired counterparts. Depending on the current physical characteristics, available bandwidth is furthermore limited.

In order to address these weaknesses, we identified suitable communication characteristics and resource-efficient transmission schemes that contribute to a better utilisation of available networking resources. The communication behaviour of AMRs is especially important in this case: Robots typically emit many small messages on a regular basis. Due to the costly media access and physical limitations, it is possible that the data flow gets disrupted. Furthermore, involved routers may have problems with high load while oversized messages may cause congestion because of excessive bandwidth consumption. As the frequency of message transmission typically cannot be decreased at will, a balance between message size and transmission frequency is required for coexistence with other network participants.

Wireless data communication further involves the discussion of security and secure data communication. Cryptographic techniques commonly assure authenticity and confidentiality. Adopted for group communication and resource constrained systems, specialised techniques are required that involve only minimal overhead in both space and time.

1.1.4 Configuration and Monitoring

Supportive technology is typically also advised when regarding the configuration of a distributed system. This includes not only the interaction traits with external software components, but covers the needs of the local host as well. The configuration expense of the communication setup thereby easily grows exponentially, depending on the number of participants. Taking into account the adaptations required during operation adds further overhead. This is why support for self-configuration and self-adaptation is profitable. A suitable approach should consume only little resources and processing time. It must furthermore be able to cope with network unreliability, in particular with fluctuations in the availability of local and remote components.

The same is true when regarding development support for distributed systems. Here, typically runtime information has to be acquired in order to draw conclusions from the observed behaviour of a component or a system. Such monitoring and distributed logging support must not be vulnerable to networking issues and operate in a resource-efficient manner as well. It should not impose a major impact on the communication behaviour and hence the operation of a robot.

1.2 Problem Analysis

Section 1.1 above introduced several characteristics of robots, in particular for a subclass that covers mobility. We will now refine the understanding by establishing challenges that further describe the needs of AMRs. Requirements are derived that build the foundation for the decisions reach in the course of the subsequent chapters. We will roughly divide the considerations in three independent parts, namely the *Development Methodology* (Section 1.2.1), *Communication Infrastructure* (Section 1.2.2), and *Resource Discovery* (Section 1.2.3).

1.2.1 Development Methodology

The discussion on a development methodology has already pointed at heterogeneity issues that range from hardware to software and cover other domains as well. They directly influence the development methodology as suitable measures are required that cope with heterogeneity in an abstract and platform-independent manner. We decided in favour of *Model-Driven Software Development* (MDSD) [121, 124] where modelling captures the development process at an abstract level. The most important properties in such a development methodology are appropriate models and an adequate modelling granularity. Otherwise, the application domain may not be covered or represented well enough.

A model has to be transformed into a concrete implementation that represents a part of the communication infrastructure for a specific platform. Here, several target platforms have to be supported in order to address heterogeneity in distributed systems. The challenges listed below represent some fundamental characteristics of the MDSD approach but also include the considerations of the software structure discussed in Section 1.1.1.

Abstract Modelling A *Domain-Specific Language* (DSL) captures the application domain in an abstract fashion, often realising a non-procedural modelling approach. MDSD builds on a suitable DSL as an interface for developers that hides irrelevant or platform-specific details. The respective model then promotes platform-independence because of abstraction from a concrete underlying platform.

Model Transformation A model transformation process must be able to address several target platforms. It is furthermore important that the modelling language as well as the model transformation process are customisable and extensible. The ability to add new languages and respective transformation processes must be retained.

Applicability The generated source code must be tailored to a particular target platform regarding integration and application. It further must be resource-efficient in both, space and time dimensions.

Modularity The generated source code must exhibit a modular structure and promote modular software development. Integration into existing solutions has to be accomplished in a modular fashion as well. This means that the exchangeability of modules has to be preserved.

These challenges cover the development methodology and the software structure of an RCA. The involved communication behaviour, however, exhibits very specific characteristics that need to be addressed appropriately. We therefore focus on the challenges and requirements the Spica development framework has to cope with.

1.2.2 Communication Infrastructure

Information exchange between robots is in most cases accomplished by distributing messages. The reason for this is obvious: Information that is worth being distributed is typically derived from sensor data periodically, possibly with the help of further knowledge. Once enough information is available, a new message is prepared and sent. Message orientation has some more advantages that are introduced in the course of this thesis.

Especially groups of AMRs depend on the information exchanged by their fellows: Compared to humans or animals that do not only perceive sound but also observe congeners and other elements of their surroundings, most robots either lack respective sensory systems or do not have enough resources available for processing. The deficiency of sensors compared to their natural examples is another handicap robots must be able to cope with. Sensor data provided by other robots do not solve the problem but compensate for shortcomings in this case. More information captured at different positions in the environment makes it possible to derive knowledge in a more stable fashion.

The communication infrastructure is not only responsible for delivering messages but also serves control and monitoring purposes. However, the specific characteristics of wireless communication media impose difficulties for this approach as outlined in Section 1.1.3. Besides a well-managed infrastructure, the following challenges have to be addressed in order to use the communication media appropriately:

Message Orientation Message-oriented communication is recommended for periodical data transmissions or packets consisting of only a specific number of bytes. An unreliable transport protocol such as UDP [110] is furthermore capable of handling data loss and network unavailability appropriately. This is especially important in unreliable, wireless communication networks. Data processing must thereby be robust against data loss in order to be able to cope with lost messages.

Service Guarantees It must be possible to transmit data streams such as a sequence of camera images, for example, from one robot to arbitrarily many recipients. UDP thereby handles delivery where a message fragmentation approach cares about splitting and reassembling of messages. Appropriate group communication schemes help in reducing communication overhead. Reliable data transmission is required for remote control messages, for example. They have to be delivered reliably, so UDP alone does not suffice.

Compression Network communication bandwidth is limited. A message compression facility should be provided in order to reduce the size of network messages. This constitutes a trade-off between computation expense and reduction of bandwidth usage.

Communication Security Information exchanged within a group communication scenario must be authentic and confidential. Suitable and resource-efficient authentication and encryption schemes must be provided for this purpose.

1.2.3 Resource Discovery

Further questions arise when regarding the configuration of a communication infrastructure. In most cases, many different communication channels have to be maintained for a group of mobile robots. However, not only the initial configuration of communication channels is important, a dynamic adaptation of the channel configuration is rather crucial in order to address dynamics in the communication infrastructure. Participants may enter or leave a communication or the network configuration may change unexpectedly. Components may further be added, replaced, or removed during runtime.

A dynamic resource discovery approach addresses these issues, for example. Communication channels are thereby established based on offered and requested resources dynamically. The following challenges are linked with such an approach. A concrete realisation must address the associated requirements in order to be applicable in an AMR communication scenario.

Fault-Tolerance A resource discovery approach must be able to tolerate network errors, partitioning, and breakdown. Participants may furthermore join or leave a group unattended at any time. This implies that the system must be able to reconfigure itself and dependent components dynamically according to the environmental state.

Distributed Operation Highly dynamic environments require a resource discovery system to operate in a completely decentralised fashion. The respective knowledge on available resources must thereby be stored redundantly in order to prevent loss of information.

Applicability A resource discovery approach must operate in a resource-efficient manner with respect to bandwidth consumption and required processing time. Access to global information shared by all participants must be possible from any platform.

The problem analysis revealed several requirements a development approach has to face. They were derived from challenges that belong to the three characteristics introduced in Section 1.1. Below, we outline a solution approach that addresses these requirements.

1.3 Solution Approach

Spica, our approach for the development of mobile robot communication infrastructures is based on the MDSD methodology. Model orientation addresses software development for heterogeneous distributed systems. Heterogeneity in this context refers to the incompatibilities between hardware and software platforms. Abstraction from the platform-specific peculiarities and fundamental characteristics allows developers to focus on the implementation. The underlying model transformation functionality cares about the realisation automatically. The completion of a concrete implementation for specific target platform typically involves one or more intermediate processing steps. Section 2.2 presents a formal description of the MDSD methodology.

Communication infrastructures for mobile robots are a specialised application domain. This is why we created the *Spica Modelling Language* (SpicaML), a *Domain-Specific Modelling Language* (DSML) that explicitly addresses the needs of AMRs. It also targets the requirements of distributed modular software systems in general. Among other things, this includes measures for group communication as a basis for collaboration. The semantics of data transmission and data management are tuned towards message-oriented communication behaviour as proposed in the motivation above.

SpicaML provides specification means for data structures and data flow. It supports three different types of data structures: *Headers*, *messages*, and *containers*. All data structures are composed of fields that either have a primitive or complex data type. SpicaML containers thereby represent the complex types. Messages are intended for communication only where each one is assigned exactly one header carrying required control information. Data structures may be arranged hierarchically by nesting containers. The modelling language further supports single inheritance and semantic annotations for all types of data structures.

A semantic annotation consists of a *concept* and a *representation* [113]. The concept specifies a type whereas the representation determines the interpretation of the corresponding value. This information is used in two different ways: If an annotation is assigned to a header field, it is marked for special treatment during model transformation. This is required for fields which have a special meaning such as the message id or a timestamp, for example. The second application relates to message filtering and automatic data conversion [7]. Section 5.5 shortly introduces this approach. It is nevertheless not considered any further.

Regarding data flow, the modelling language employs three main modelling elements: *Modules*, *messages*, and *message buffers*. Modules represent stubs for arbitrary components that allow them to join the communication infrastructure. Messages are the data structures introduced before. Message buffers synchronise message transmission or reception. This is required because of the asynchronous communication behaviour.

Communication channels between two or more modules are not modelled explicitly. Instead, a matching is carried out that brings together offered and requested message types. Geminga, the dynamic resource discovery engine we created for Spica, then handles the negotiation of communication channels.

Once the model specification is finished, a concrete implementation of the communication infrastructure is generated. Spica provides the model transformation tool Aastra that accomplishes this task. It first transforms the abstract model into an intermediate representation, completes it, and verifies its consistency. Afterwards, a template-based code generation approach transforms it into concrete source code.

The generated communication infrastructure promotes the idea of a modular RCA that facilitates adaptivity and code reusability. We combine an MDSD-based approach with characteristics of middleware architectures and concepts of a *Service-Oriented Architecture* (SOA) [41] that mainly serves as a representative for modular software development in this context. We therefore introduce a new approach for the development of robot software that goes beyond basic communication needs. Instead of providing a middleware architecture implemented in a specific programming language, we propose a platform-independent development approach that enables developers to address arbitrary target platforms.

We have moved frequently used functionality for code templates to the utility library Castor that is also part of the Spica development framework. It covers basic solutions for configuration and file management, and cares about system-related tasks in general. Further fields of application are communication tasks such as sockets, network addresses, or message encoding, for instance. The security-related part that covers authentication and encryption mostly consists of filter routines used for message processing. Castor finally provides the implementation for Geminga and the *Spica Decentralised Monitoring Facility* (SpicaDMF), a distributed monitoring framework we created exclusively for Spica.

The most fundamental parts of Castor such as networking, monitoring, and Geminga are provided for every supported platform. The other functionality is not mandatory but provided to ease code template creation. Starting with Chapter 4, we introduce the technology created for the Spica development framework, namely SpicaML, Aastra, Geminga, SpicaDMF, and Castor.

1.4 Major Results

The major scientific contributions of this thesis are the identification of requirements for communication among mobile robots and the specification of a development approach with a DSML that captures the most important characteristics of the application domain. Along with the specification of a distributed resource discovery, a complete development solution is provided for application in groups of AMRs. We further provide a prototypical implementation of the whole development approach that successfully proved its applicability in several applications.[2]

Development Framework With Spica [8, 7, 9] we provide a generic development framework for robot communication infrastructures that explicitly targets AMRs. The desired communication infrastructure is specified using SpicaML, an abstract DSML that captures the important characteristics of the application domain. A model transformation process then generates concrete source code for a specific platform from this model automatically. Our Aastra model transformation tool thereby cares about the interpretation and transformation of the model, incorporating a template-based code generation approach.

Self-Configuration The Geminga [10] resource discovery approach takes care of the configuration of the communication infrastructure. In case of changes in the environment or the system configuration, it initiates a reconfiguration of the communication infrastructure. It is designed explicitly for application in ad hoc networks. Because of its generic design, it is well suited for a broad range of applications.

Distributed Monitoring The SpicaDMF distributed monitoring facility assists in the development and the monitoring of mobile robots by providing transparent logging functionality. It is designed explicitly for application in ad hoc networks and tuned towards resource efficiency.

Secure Communication Security measures embedded into the Spica communication infrastructure address communication security. They deal with authenticity and encryption on a per message basis, explicitly addressing message orientation and unreliability. Only symmetric ciphers are supported as they provide the best possible efficiency by maintaining an adequate level of security.

Geminga and SpicaDMF are part of the Castor utility library. A team of autonomous soccer-playing robots uses the Spica development framework as a basis for the entire communication infrastructure. It thereby covers communication within a single robot and between the robots of the team, providing the basis for cooperation. The contribution to the state-of-the-art comprises the new development approach for AMRs and the automatic configuration of communication infrastructures in unreliable ad hoc networks.

[2]The Carpe Noctem robotic soccer team of Kassel University successfully attended the RoboCup World Championships 2006 as well as the RoboCup German Open 2007 and 2008.

1.5 Overview

The remainder of this work is organised in three parts: Part I discusses the foundations, Part II covers the main contribution, and Part III presents an extensive evaluation.

Chapter 2 introduces the foundations of Spica by covering robots, modelling, service orientation, networking, and security. Chapter 3 points to related work and discusses the differences to our approach. The second part that covers the main contribution starts with an introduction to the Spica development framework in Chapter 4. It outlines the design decisions and introduces the specification of SpicaML in Chapter 5 as well as the transformation process in Chapter 6. Chapter 7 finally outlines SpicaDMF and Chapter 8 the Geminga resource discovery. The last part starts with an experimental evaluation of Spica in Chapter 9, examining its applicability with the help of three examples: First, the integration of Spica with the Carpe Noctem RCA is discussed, followed by the adoption of Miro [129, 128] and the simulation engine Gazebo [77]. Chapter 10 evaluates the Spica approach qualitatively with respect to applicability and resource utilisation. The thesis concludes and points to future work in Chapter 11.

2 Foundations

"If you have built castles in the air, your work need not be lost; that is where they should be. Now put the foundations under them."

— Henry David Thoreau
Walden; or, Life in the Woods, 1854

This chapter reviews the basic principles underlying the Spica development approach. Starting with an introduction to autonomous robots in Section 2.1, we give the reader a basic idea of the application domain, considering the characteristics of the architecture and software. In the course of this discussion, we will establish the fundamental understanding of the characteristics and problems involved with mobile robots.

Based thereupon, we shift the focus towards a possible development approach in Section 2.2. The decision in favour of MDSD is found by the fact that abstract modelling capabilities explicitly support platform-independent development. A dedicated transformation then maps model instances onto concrete source code for several platforms. Regarding an architectural layout that is applicable for robots and facilitates integration of existing approaches, we rely on SOA principles as introduced in Section 2.3.

Section 2.4 introduces networking as a fundamental functionality for communication and cooperation. It outlines specific issues of the communication media used by mobile robots and motivates the design decisions followed for the Spica development approach. Communication security, which is directly connected to the aforementioned networking functionality, is introduced in terms of confidentiality and authenticity. Section 2.5 outlines realisations for each measure.

2.1 Autonomous Mobile Robots

Robots are available in various different shapes, equipped with different sensors and actuators, and typically based on different hardware and software platforms. In this section, we give an overview of the characteristics and an introduction to design principles of AMRs. We start with the robot hardware architecture that builds the foundation for the considerations of the software architecture design and behaviour representation following thereafter. Heterogeneity issues introduced by differences in hardware and software will be of most interest here. By finally addressing communication, we shift the focus to the most important technique required for cooperation and team coordination. A short outline of the requirements will explain why a communication infrastructure is required that is tailored to the needs of AMRs. For now, we start with the basics.

There is neither a general definition of what a robot is, nor for what tasks it is responsible. The Encyclopædia Britannica defines a *robot* as follows:

Definition 1 (Robot) *Any automatically operated machine that replaces human effort, though it may not resemble human beings in appearance or perform functions in a human like manner.*[1]

This is a very wide-ranging and generic definition but captures almost any important aspect. In this context, however, we are mostly interested in AMRs, i.e. a kind of specialised robot subclass thereof. The Oxford Dictionaries define the term *autonomy*[2] as "self-government" or "freedom of action", the term *mobile*[2] as the ability "[...] to move or be moved freely or easily". Based on the common agreement on what an AMR is and which capabilities and characteristics it typically has, we propose the following definition:

Definition 2 (Autonomous Mobile Robot) *A robot that is able to move freely and make decisions based on its experience that, in turn, typically relies on sensory input. Conclusions represent knowledge derived from experiences.*

A cooperative autonomous mobile robot is further able to interact with other robots and use the provided additional information for its own or the group's benefit.

2.1.1 Hardware Architecture

Mainly the task and the field of application influence the hardware architecture of a robot. In this context, AMRs typically have to be able to navigate in dynamically changing environments without being controlled by a third party. Suitable sensors help the robots to perceive their surroundings and the respective physical properties. Based on the measured data, conclusions may be drawn regarding a robot's current situation. The result is then applied to actuator devices that interact with and influence the environment.

2.1.1.1 Sensory Hardware

Sensor devices are antenna for robots to gather information about their environment. The Oxford Dictionaries define the term *sensor* as follows:

Definition 3 (Sensor) *A device which detects or measures a physical property.*[2]

In a more comprehensive view, physical measures include chemical and electrochemical measures as well. Sensors are the interfaces of a robot to its environment and thus resemble the characteristics of typical sensors used by humans or animals. Sensory skills in nature

[1]In Encyclopædia Britannica, `http://www.britannica.com/` (accessed 2008-07-12).
[2]In Compact Oxford English Dictionary, `http://www.askoxford.com/` (accessed 2008-07-12).

evolve during the development of individuals. They are trained in the interpretation of signals generated by sensors, continuously adapted by a learning process. A sensor device used by a robot partly follows similar characteristics: It must be configured and calibrated in order to be applicable for a given task. Configuration here covers the operational frequency, the resolution of measured data, or the mode used for data delivery, for example. Calibration is required to concentrate the sensitivity of a sensor to a specific characteristic of the environment and to reduce the influence of other sources of interference. Due to changes in the environment, however, the quality of a static calibration may degrade over time. A camera, for instance, that has been calibrated for a given colour temperature[3] may perform much worse in detecting coloured objects if the surrounding colour temperature changes. The human brain automatically adapts its perception appropriately in this case. Even though techniques are available in computer vision processing to achieve a similar functionality, a dynamic adaptation process is required for most sensors to keep track of environmental changes. This lack of adaptivity in robotic systems represents one major weakness compared to their natural examples. Hence, it is not only important to correctly configure sensor devices but rather handle and interpret acquired data appropriately according to the environmental situation.

Any type of sensor only estimates the current state of a measured physical property. As measurements are always subject to error due to the underlying physical laws, the data gathered through a sensor device must be assumed inaccurate. The adoption of an error model and a suitable handling and interpretation of the data are thus indispensable. A typical approach is to average over a number of measurements in order to lower the statistical noise in the data. This, however, implies some delay in measure data.

The mode of delivery for new measurement data of a sensor device falls in two classes: Either, the receiver of the data is signalled asynchronously (*push*) once new data are available or new data must be explicitly requested from the sensor device (*pull*). It mostly depends on the type of sensor and the characteristics of the measured property which delivery approach is used. The robot must finally be able to gather the data for further processing in both cases.

2.1.1.2 Processing Hardware

Processing sensory input is the most important capability of autonomous robots. This implies that a robot owns a processing unit that is able to read measurements from sensors, process and interpret them in a suitable fashion, and turn the derived knowledge into specific actions. The processing units applied here range from microcontrollers to laptop computers or even full-fledged server systems, depending on the application domain and required processing power. Laptop computers are commonly used in AMR research as they already fulfil the need for mobile application by exhibiting an own battery, for example.

Retrieving data from sensor hardware requires any of these processing units to provide suitable interfaces. Some are provided by default, some are supplementary equipment. Serial interfaces are the most widespread interface type used in any robotic appliance. Examples are EIA-232 [39, 19] and I^{2}C [108], the latter of which is typically used in microcontroller-based solutions. The *Controller Area Network Bus* (CAN-Bus) [42] is an asynchronous serial bus initially introduced for automotive appliances. Due to its real-time capabilities is it

[3]The colour temperature of a light source is determined by comparing its chromaticity with that of an ideal black-body radiator. It is measured in Kelvin (K).

popular in AMR as well. The *Universal Serial Bus* (USB) [31] replaces the EIA-232 interface and some more of this family, providing higher speed and hot plugging capabilities. It is used mostly because of its widespread availability, even though it does not provide any real-time guarantee for data transmission. *FireWire* (IEEE 1394) [71, 72, 62, 67] bridges this gap: The isochronous[4] real-time data transfer is especially advantageous for vision processing where reliability and low latencies are most important.

The aforementioned interface technologies are well supported by current operating systems or covered by library support, facilitating easy integration. If more specialised of proprietary interface techniques are used, specific driver components may be required. Regardless of the approach, data processing routines must be able to cope with the respective data delivery characteristics of the sensors.

2.1.1.3 Actuator Hardware

The processing unit of a robot connects sensors and actuators in basically the same way. The main difference is the direction of the data flow: Sensors provide measurement data while actuators accept commands that control their behaviour and thus the interaction with their environment. Merriam-Webster defines the term *actuator* as follows:

Definition 4 (Actuator) *A mechanical device for moving or controlling something.*[5]

Actuators and sensors are typically connected to the main processing unit of a robot using similar interfaces. An actuator needs to be configured and calibrated, as it is a matter of accuracy whether it is effective or not. Most actuators further integrate sensors that provide feedback of the actuator's behaviour and current state. Regarding the measurement errors of sensors, actuators exhibit a similar phenomenon: The execution of an action is not instantly realisable because of physical limitations. This involves latencies in the current flow or inertia of affected objects. There is furthermore no guarantee for an accurate result.

Theories of cybernetics deal with this deficiency. Advanced control approaches are available for motion devices to achieve reliable results. For fine-grained control, some devices require commands to be sent periodically and with a high frequency. Other devices exhibit different behaviour: The state of a *pan unit* or a kicking device of a soccer-playing robot, for example, may only need to be updated in reaction to specific events. The update procedure, however, has to adopt a control function itself again.

The respective control software must be capable of different interfacing and control approaches. The type of software architecture is further an important criterion for the scalability of the approach when dealing with several sensor and actuator devices. Below, we introduce the principal structure and most important parts of a robot software architecture.

2.1.2 Robot Software

An RCA complements the capability of the hardware in terms of data processing and decision-making. We will concentrate on the most important components of an RCA. This includes

[4]The term *isochronous* means *equal or uniform in time*.
[5]In Merriam-Webster Online Dictionary, `http://www.merriam-webster.com/` (accessed 2008-07-12).

the facilities that process sensor data, carry out decision handling, and finally control the actuator hardware. Driver components that interface sensors and actuators are out of scope of this review and are thus left out.

2.1.2.1 World Modelling

A *world model* is a common approach for bringing together data from different sources. It maintains them in a consistent fashion and thereby realises two functions: Accepting the data and transforming them into natively supported representations, i.e. caring about the transformation of physical units. Afterwards, data fusion may be triggered in order to generate more valuable data from raw measurements. This step delays processing until all required data has arrived. A typical example for considering past measurements is the estimation or tracking of a moving objects by a *Kalman filter* [76], for instance. A world model may further take care of error reduction in measurement data by applying appropriate filtering mechanisms.

A world model is, however, not only limited to the sensor information provided by the own robot. Integrating knowledge from other robots may establish a more comprehensive view on the world. Robots share their sensor information and further knowledge with their colleagues for this purpose and so resemble kinds of providers for remote sensor data. This data is then either integrated into the standard world model or another, independent instance that is often referred to as the *shared* or *global world model*. It is exclusively responsible for the data of the other robots.

In summary, a world model represents the state of a robot's environment at a discrete point in time – possibly including the robot's own state as well. Depending on the implementation, the world model is either updated continuously whenever new data arrive or in discrete intervals. Resembling a knowledge base for subsequent processing steps, the world model is the most fundamental and important component of an RCA.

2.1.2.2 Decision Making

After integrating the measurements into the world model and computing some more elaborate data from them, the decision process of a robot makes use of this information. The main difficulty here is to express the task and the desired goal of a robot so that it is readable and understandable by a machine. Many different approaches have been proposed to solve this problem. One of the most widely adopted solutions in current robotic research is the *behaviour-based* approach [23]. It is a typical bottom-up strategy dividing the overall behaviour of a robot into several basic actions called *behaviour*s. These behaviours are then combined in such a way that a more complex behaviour for the robot is created. This process may then be repeated until the desired, more complex activity pattern is realised. Again, several combination techniques are available. One example is the hierarchical state machine-based approach [131].

Behaviours are typically kept simple and minimal requiring only limited execution time. They do not use indeterministic loops or any other type of statement that may lead to indeterministic termination behaviour. A basic processing sequence of a behaviour thus involves the following steps:

- **Fetch**. First, a behaviour fetches the required information from a world model.
- **Process**. It then processes this data according to its requirements for the final decision.
- **Act**. Depending on the outcome of the processing, a command is sent to an actuator such as the motion of the robot. It is also possible that a behaviour decides to change the strategy and thus signals to change the active behaviour set.

A discretised emulation of the robot behaviour is realised by this approach. The execution strategy and frequency of affect the control achieved granularity that, in turn, influences the reactivity of the robot.

2.1.2.3 Control

Multiple behaviours are typically executed concurrently. Hence, it is possible that conflicting commands are sent to an actuator resulting in undesired behaviour. Arbitration schemes help in resolving such a conflict. Merriam-Webster defines the term *arbiter* as follows:

Definition 5 (Arbiter) *A person or agency whose judgement or opinion is considered authoritative.*[5]

As this definition mostly falls into the area of Sociology and we are only seldom concerned with persons or agencies, we consider arbiters algorithms or software components in this case. They thus resolve conflicting situations based on a predefined criterion. Some commonly used arbiter types are:

- **Priority Arbitration**. Filters on a given command priority.
- **Content Arbitration**. Filters the command's content. This may be used to allow only commands from a given behaviour or with specific value ranges to be passed through.
- **Window Arbitration**. This type of arbiter may be used for motion commands, for example. It thereby maintains a collection (window) of possible accelerations, velocities, and directions, maybe even more, and filters commands according to these allowed values.

2.1.2.4 Summary

We have discussed some basic characteristics of RCAs and have shown examples of existing solutions. Most of the presented tasks have very specific requirements. World modelling, for example, needs to handle large amounts of data efficiently. Decision-making, in turn, would benefit from a language that explicitly addresses logic programming such as Prolog [29], for example. Control mechanisms finally depend on high reactivity and efficiency. This implies that an RCA presumably consists of various components implemented in mutually incompatible programming languages and based on different implementation concepts.

However, one question remains: How to connect different architectures and how to integrate existing components? This is where the Spica development approach comes into play: It facilitates the integration of different architectures and components into a consistent RCA. World modelling and decision-making are typically contained in a tightly bound functional entity because of the huge amount of communication between these components. Modules for sensors or actuators, however, may have been implemented in separate components or even need to be implemented differently. In order to facilitate interaction between theses components and to provide a more convenient development experience, assistance is certainly desired.

We will now have a look at the communication requirements between robots, mainly taking into account the exchange of sensor information. This is in particular important for cooperation and coordination between robots and their RCAs. Predominant use of wireless communication media imposes very specific requirements in this case.

2.1.3 Mobile Robot Communication

Communication is typically the basis for cooperation between individual agents or several AMRs. This is mostly a matter of information transfer from a sender to a receiver using some medium. The definition of the term *Cooperation* thereby reads as follows:

Definition 6 (Cooperation) *To associate with another or others for mutual benefit.*[5]

Bergmüller et al. [17] present a similar definition where the authors further distinguish between *cooperation* and *cooperative behaviour*. The latter one relates to an act that benefits another with the hope for revenue. We will follow this differentiation throughout this thesis.

Another form of cooperation typically found in nature is *stigmergy* [21], a sort of spontaneous, indirect coordination between agents. A typical example in this context is swarm intelligence of ants that coordinate themselves through pheromone traces. This behaviour applied to autonomous robots is presented and evaluated by Kube and Bonabeau [78]. Bee colonies, schools of fish, or flocks of birds exhibit similar characteristics with different indirect interaction schemes. Spica nevertheless focuses on network communication, a sub-category of communication where a computer network is the transfer medium of choice. It is still predominant in groups of AMRs, so we will restrict our considerations to this precise technique for the rest of this thesis.

Several forms of cooperation are possible within a group of AMRs: The simplest form is working *side by side* where individuals try not to interfere with each other. In a more complex understanding, interactions between individuals are involved. *Coordination*, i.e. *"the harmonious functioning of parts for effective results"*,[5] allows creating a more structured group behaviour that may facilitate the creation of more complex scenarios. *Collaboration* can finally be seen as the strongest form where all participants work together very closely.

Within the context of this thesis, we restrict considerations to basic cooperation between agents. It may be realised differently, depending on the application scenario. We therefore distinguish the three different types of cooperation scenarios as outlined below, each of which targets a specific application domain:

- **Hierarchical**. A hierarchical cooperation structure may be well suited for the domain of rescue robots where a more strict organisational structure is required. Each group thereby selects a leader for coordinating the group. If a group is split into smaller subgroups, a leader is required for each of the subgroups again.

- **Democratic**. In a democratic cooperation structure, a decision is required to be accepted by a given percentage of the group members. It therefore involves voting a procedure.
- **Independent**. An independent cooperation structure finally represents the weakest form of cooperation. Group members simply share all important information and make decisions based on their locally available information.

It is possible to combine these approaches if required. The first two clearly involve closer interactions: The hierarchical approach needs for the election of a suitable group leader whereas the democratic approach depends on voting. They are both vulnerable to communication failures. The independent cooperation approach is also vulnerable but the consequences are not as severe: Even in case of complete network breakdown, each robot remains functional without external knowledge as well.

The independent approach as the weakest form of cooperation resembles the characteristics we demanded for the shared world model approach introduced in Section 2.1.2.1: Each robot shares its local information with the other robots, resembling a virtual sensor data provider. Data are sent periodically with a given frequency for this purpose. To keep the communication overhead low, the transmission frequency is limited to a reasonable level.[6] It is also possible to adapt the transmission frequency and the amount of communicated data according to the environmental situation.

The most important characteristic of AMR communication has not yet been addressed: Communication builds on wireless media that is known to be unreliable under certain conditions. This is due to physical constraints that are further discussed in Section 2.4. In order to address the unreliability, network communication overhead must be kept low and data should be sent in an unreliable fashion as well. An unreliable transport protocol may put things right: Even though data loss is possible, it contributes to the applicability in unreliable communication media by doing without any further protocol interactions or retransmissions. Robots typically appear and disappear in a spontaneous manner, hence reliable or connection-oriented communication schemes are incongruous. As all robots must be able to cope with unavailability of other group members anyway, this should not impose further issues.

2.2 Model-Driven Software Development

Spica employs a model-driven development approach to achieve platform-independence. An abstract model is thereby transformed into concrete implementations, taking the characteristics of the target platform into account. A platform in this context refers to a hardware platform, an operating system, a middleware architecture, or a programming language.

Model transformation processes are well established especially in the domain of software development ever since. For example, a programming language resembles a more or less abstract model that is transformed into a concrete implementation for a specific platform with the help of a compiler. Intermediate processing steps involve completion, specialisation, and optimisation of the model. The MDSD methodology [135] is one generalised approach towards such a development strategy. It realises a top-down approach with clear layers

[6]The transmission frequency depends on the application domain and networking infrastructure.

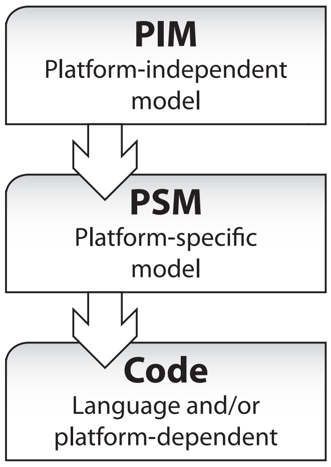

PIM
Platform-independent model
PSM
Platform-specific model
Code
Language and/or platform-dependent

Figure 2.1: The MDSD transformation hierarchy: An abstract, platform-independent model is first transformed into an intermediate, platform-specific model, and finally into concrete source code

that keep the different transformation tasks separated. The list below outlines the tasks with respect to their responsibilities, Figure 2.1 visualises the different stages of an MDSD transformation graphically.

- **Abstract**. The topmost layer in the MDSD methodology holds the platform-independent model, an abstract description of the implementation that has to be generated. MDSD does not prescribe a representation for the model but leaves it to the developer to decide what kind of model and which representation are best suite for the given task. XML or UML-based approaches are feasible but text-oriented representations are also commonly used.
- **Intermediate**. The second layer involves the first transformation step: The transition from the abstract into a platform-specific model. A first step parses and interprets the abstract model. An intermediate transformation then completes, specialises, and optimises the model. This intermediate representation is not yet tailored to the target platform but to the transformation in the code generation layer. It is thus called platform-specific with regard to code generation.
- **Concrete**. The third layer deals with the transformation into platform-specific code. The target is typically no platform-dependent binary representation but an implementation in a programming language for a specific platform. This renders MDSD a kind of meta development approach located above the programming language layer.

The MDSD is a refinement of *Computer Aided Software Engineering* (CASE) and forms a superset of *the Model-Driven Architecture* (MDA) development methodology. MDA [85] is a software development methodology initiated by the *Object Management Group* (OMG), promoting strict model view separation. The main motivation of MDA is interoperability and portability of software. Interoperability is thereby guaranteed by the OMG standardisation process, portability is provided by an appropriate abstract modelling language and suitable model transformation tools. MDA divides the development process into four different stages. They are represented as modelling abstraction layers, each of which increases the specialisation – or, in turn, decreases abstraction – of the model until platform-specific code is generated. Hence, it also realises a top-down development approach.

2.2.1 Model-Driven Architecture

The listing below shows the modelling layers. Most MDA-based development approaches dispense with the first layer by starting directly with the PIM.

- **CIM**. The *Computation-Independent Model* (CIM) is a colloquial model specification.
- **PIM**. The *Platform-Independent Model* (PIM) aims at business processes.
- **PSM**. The *Platform-Specific Model* (PSM) aims at the architecture and services. It is derived from the PIM through a model-to-model transformation.
- **Code**. The final mapping to concrete source code completes the transformation. The transition from the PSM to the Code layer involves a model to code transformation.

The separation of models – more specifically, the separation of concerns – is a substantial extension to the UML standard [100, 101]. The CIM and PIM layers aim at language and system independence whereas PSM and Code layers are unique to specific platforms. Just like the MDSD methodology, MDA does not define the type or structure of a modelling language, so implementers are free to choose the best-suited alternative. MDA-based approaches, however, adopt UML-based specification languages in most cases.

MDA typically conducts two model transformation steps: First, a model-to-model transformation is carried out between the PIM and PSM layers. The final code mapping from the PSM to the Code layers then employs a model to code transformation by means of code template approaches, for instance. Intermediate transformation steps are possible. The *Meta Object Facility* (MOF) [99] and the related *MOF Query View Transformations* (MOF QVT) [102] build the foundation for the description of modelling language meta data and the model-to-model transformation. The OMG standard MOF describes specialised meta data architectures. It consists of four layers that classify data as follows:

- **Layer M0** contains concrete data.
- **Layer M1** represents the user model. It may contain physical or logical data as well as object, data, or process models. These models define the data specified in layer M0.
- **Layer M2** introduces meta models that define in which way models are structured. Here, UML is used for modelling.
- **Layer M3** finally introduces the meta models describing the models on layer M2. MOF is located in this layer.

The MOF QVT specifies a programming language for model-to-model transformations. MDA further encourages the use UML profiles[7] for creating domain-specific modelling languages. MDSD is a pragmatic, application-centric approach. MDA is a standard promoted by the OMG which relies on integrated modelling approaches, resembling a specialisation of MDSD.

2.2.2 Advantages and Drawbacks

The MDSD methodology does not prescribe the procedure of transforming a model into platform-dependent code. Template-based code generation is the most common and most

[7]The concept of UML profiles is an extensions of the UML 2 meta model [100, 101].

flexible one in terms of addressable target languages. It nevertheless involves controversial discussions: Opponents argue that template-based code generation is error-prone and complex. This is because templates contain source code in a generalised form, complicating implementation as well as maintenance. In most cases, algorithms furthermore cannot be separated from the actual code and thus have to be implemented and kept in sync for each target platform. In-depth knowledge of the target platforms is required and typically embedded directly into the code templates. Changes to the model imply modifying the code generation templates. The same is true when adding new features or fixing bugs.

These reasons often limit the extensibility of models and the applicability in large-scale applications. Without a clear separation, code generation templates and the generated code will get unmaintainable. Currently available code generation approaches further lack formal verification capabilities: There is no common agreement on how to verify the completeness or correctness of the code template [124]. Formalisation of the code generation process with support for verification is nevertheless strongly advised.

An issue with less impact on the feasibility but major importance with respect to the applicability is the decision for a modelling language: In order to be independent from the underlying platform, an abstract model is required. It hereby is neither possible nor always desirable to represent the entire application domain. Instead, the expressiveness of the model should be limited to the most important characteristics and functionality. This decision plays an important role for the applicability of the modelling language and the entire modelling approach.

Regarding the supporter's point of view, advantages of MDSD clearly countervail against the weaknesses introduced above, given some restrictions are acceptable. Modelling, for example, is a well-established technique in the domain of computer science: As outlined earlier, compilers for programming language represent one of the first model-based transformation approaches with programming languages being the models that abstract from the underlying platform. An MDSD-based development approach provides further abstraction and increased cross-platform applicability. This fact and problem-specific modelling elements clearly involve simplifications for users. Modelling and model transformation are especially well suited for reoccurring and coherent implementation tasks. This is where a restriction is required in order to mitigate some of the drawbacks outlined above: A transformation template should be written once, requiring little or no feature additions or extensions. This reduces costly template modification.

MDSD furthermore promotes a separation of technical and domain-specific knowledge, resulting in less redundancy. This must not be mixed up with the separation of algorithms and code in the transformation template that is still an issue, though. It is possible to validate and check the abstract model, allowing for domain-specific but platform-independent analysis and assessment of the model quality and correctness.

An MDSD-based development process thus exhibits a simplification for users in many aspects and overcomes weaknesses of manual cross-platform development. We will now discuss a promising structuring approach for software architectures as a possible target layout aimed at by the Spica development approach.

2.3 Service-Oriented Architecture

The *Service-Oriented Architecture* (SOA) paradigm [41] proposes a management approach for *loosely coupled* services that address requirements for business processes. Rather being a management concept than a concrete system architecture, SOA is a descriptive extension to the modular design approach. The term “loose coupling” is prevalent in the SOA domain but there is no general definition of its actual meaning. We will thus rely on the following definition for the remainder of this thesis:

Definition 7 (Loose Coupling) *Loose coupling describes a characteristic of system interoperation. Two or more systems are loosely coupled if mutual assumptions are reduced to a minimum, thus lowering the risk that a change in one system will force a change in another system.*

Services in a SOA-based architecture provide access to resources in an abstract, sometimes platform-independent manner. SOA does not prescribe a specific technique, language, or development methodology, even the communication approach used for service interaction may be chosen freely. Architectures that realise a SOA approach may rather be implemented as required or based on techniques suited best for the respective task as long as loose coupling is retained. Interoperation with other services is reduced to interaction through interfaces, the concrete functional realisation of which is not important anymore. This relates to the reduction of assumptions demanded in Definition 7 above.

Different services may offer the same functionality but use a different implementation to achieve it. They may further be located on different hosts in a distributed manner. This promotes exchangeability and distribution of services. Below, we introduce the basic definitions and principles of SOA. We do not cover descriptive languages for service interface specification, semantics, or *Service Level Agreements* (SLA), for example. We rather restrict ourselves to introduce the basic concepts and requirements that are useful for the Spica approach.

2.3.1 The Essence of a SOA

As a SOA targets loose coupling of business and computational resources, common agreements have to be found which formalise the interaction between services. A common misunderstanding is that a SOA directly relates to a concrete technology such as a Web Services architecture, for example. A SOA is rather a technology-independent approach for modular distributed computing. In the absence of a common general SOA definition, the remainder of this work relies on the following description of *The Open Group* that captures some characteristics of SOAs in general:

> “*Service-Oriented Architecture (SOA) is an architectural style that supports service orientation. Service orientation is a way of thinking in terms of services and service-based development and the outcomes of services.*”

A SOA service is a self-contained logical representative of an activity with a specific outcome. The composition of several services to a more powerful one allows system architects to create the functionality of a system in a bottom-up fashion. Reusability is hereby a matter of complexity and genericity: Highly specialised services are presumably reused in rare cases only whereas services that are more generic are typically applicable for a broader range of applications. Interoperability in a platform-independent manner further promotes the reusability of a service. Discovery and dynamic binding of services in a SOA is especially attractive for distributed software architectures where parts are located in self-contained entities anyway.

Modularity in Spica and robot software in general clearly argues for the adoption of a SOA-like software structure. This is because a modular software architecture but also groups of AMRs are inherently distributed and require a consistent approach for module composition and interaction.

2.3.2 Advantages and Drawbacks

The basic idea of a SOA traces back to component or module-oriented software development where functionality is maintained in self-contained, independent entities. SOA transfers this to a more formal approach for modular distributed computing. Because of no commitment to a concrete implementation or communication technique, incompatibilities between different SOA-based solutions are possible. This is nevertheless all right because of the increased flexibility and the far-reaching applicability of this approach. It furthermore renders SOA-like structures especially attractive for ex post integration into existing software systems.

The absence of a common agreement on a general definition is certainly a weakness of the SOA approach. This not only hinders interoperability but also cuts an architecture's value. We therefore committed ourselves to a definition that is generic enough to be used as a foundation for Spica. Interpreting SOA this way furthermore renders it well suited for a broad range of applications. Other weaknesses are reported mostly in conjunction with Web Service technology. Because there is no direct link between these two techniques, this criticism does not apply to SOAs in general.

2.4 Networking

This section reviews fundamental networking techniques required for communication in groups of autonomous mobile robots. The scenario basically resembles characteristics of *Mobile Ad Hoc Networks* (MANET) [32, 13] because participants may appear and disappear in an indeterministic fashion, resulting in a very dynamic network structure. This and the fact that robots are mobile systems imply that only wireless communication media are applicable.

Below, we shortly introduce two types of networking technology. We thereby restrict ourselves to the standard family IEEE 802.11 and the standard IEEE 802.15.1 (Bluetooth) [20]. All communication media in this context make use of the *Industrial, Scientific, and Medical* (ISM) radio band, located at 2.45GHz and 5.8GHz (centre frequencies). Other radio networking technologies such as IEEE 802.15.4 (ZigBee) [64, 69] additionally adopt the 868MHz (no ISM radio band) and 915MHz radio bands.

2.4.1 IEEE 802.11 Family

The IEEE 802.11 family [68] covers *Wireless Local Area Network* (WLAN) standards located in the 2.4GHz and 5GHz ISM radio bands. At the time of writing, the most popular representatives are the IEEE 802.11a [68, 66], IEEE 802.11b [68], and IEEE 802.11g [68, 65] standards. The most current proposed networking standard that directly follows IEEE 802.11g is IEEE 802.11n [68, 70]. It provides increased throughput but is still downward compatible to IEEE 802.11b/g. IEEE 802.11n is no approved standard yet.

All protocols of the IEEE 802.11 family are required to use the *Carrier Sense Multiple Access/Collision Avoidance* (CSMA/CA) [30] media access protocol as a common basis. Additionally, a positive acknowledge scheme [68, pp. 257–310] is implemented as follows: Once a receiver has checked the CRC checksum of an incoming packet, it instantly sends an acknowledgement back to the sender, informing it of successful reception. The sender will retransmit the packet until it has received the acknowledgement. The acknowledgement scheme is disabled for packets directed at several receivers, i.e. packets sent via broadcast or multicast.

- **802.11a**. An IEEE 802.11a network has a data rate of 54Mbit/s (PHY) with a maximal throughput of about 23Mbit/s (gross). The data rate may be phased down to 6Mbit/s in case of increased noise. IEEE 802.11a operates in the 5GHz radio band. It is well established in the RoboCup community.
- **802.11b**. An IEEE 802.11b network has a data rate of 11Mbit/s (PHY) with a maximal throughput of about 4.3Mbit/s (gross). The data rate may be phased down to 1Mbit/s in case of increased noise. It operates in the 2.4GHz radio band. It is well established in the RoboCup community.
- **802.11g**. The IEEE 802.11g wireless network standard is the downward compatible successor of IEEE 802.11b. It has a data rate of 54Mbit/s (PHY) with a maximal throughput of about 19Mbit/s (gross). The data rate may be phased down to 1Mbit/s in case of increased noise as well.
- **802.11n**. The IEEE 802.11n wireless network standard is the downward compatible successor of IEEE 802.11b/g. It has a data rate of 288Mbit/s[8] (PHY) with a maximal throughput of about 100Mbit/s (gross). All the data rates of the predecessor standards are supported as well. IEEE 802.11n furthermore supports MIMO (Multiple Input Multiple Output) and is thus capable of combining several data channels. Four channels may yield a data rate of up to 600Mbit/s (PHY).

In the past, IEEE 802.11b was more widely applied. IEEE 802.11g mostly replaced it; more specialised appliances typically use IEEE 802.11a. Due to its less widespread application and less occupied frequencies, IEEE 802.11a provides better performance in most cases.

Two principle modes of operations are possible with IEEE 802.11: *Ad hoc* and *managed* mode. Systems in ad hoc mode talk with each other directly. In managed mode, one or more *Access Points* (AP) control communication. Even though managed communication clearly

[8]Some sources say 248Mbit/s but other state 288Mbit/s. It actually depends on the modulation, the length of the guard intervals, and other factors. IEEE 802.11n is furthermore capable of doubling the channel width from 20MHz to 40MHz, practically doubling the number of subcarriers from 52 in earlier IEEE 802.11 standards to 108. This yields a total channel throughput of about 150Mbit/s.

involves more communication overhead, it typically provides better performance for large amounts of participants because of negotiated data transmissions.

All standards share the weaknesses of wireless communication media: It is nearly impossible to guarantee transmission reliability. This is mostly due to physical constraints (reflection, cancellation, interference, diffraction, and scattering) which cannot be eliminated.

Below, we continue our review of communication media with a near field communication standard that exhibits less signal propagation than IEEE 802.11.

2.4.2 IEEE 802.15.1

The IEEE 802.15 family covers *Wireless Personal Area Networks* (WPAN) standards. Just like WLAN, most standards of the family adopt the ISM bands. The most popular representative is IEEE 802.15.1 [63], also known as *Bluetooth v1.2*. Newer versions such as *Bluetooth v2.1+EDR* (Enhanced Data Rate) [20] provide an increased bandwidth of up to 2.1Mbit/s and add further enhancements. Obviously, both are still a great deal slower than WLAN, though.

Bluetooth exhibits further differences compared to the IEEE 802.11 family: It adopts a *frequency hopping* approach, where the available frequency spectrum is divided into several, quite small channels. Each channel is used only for a given amount of data before switching to another one. The sequence in which channels are switched is called hopping sequence. This makes Bluetooth more robust against jamming. *Adaptive Frequency-Hopping* (AFH) [63, 54] introduced with Bluetooth v1.2 improves resistance to interference and coexistence with other wireless devices. Signal dispersion of a WPAN is furthermore limited to a typical range of at most 10m, in most cases far below.

Communication channels in a WPAN are established on demand spontaneously. Robotic appliances currently prefer IEEE 802.11 mostly because of its increased range and bandwidth compared to near field communication approaches such as WPAN.

We will now have a look at communication models and discuss which are suited well for application in wireless communication networks.

2.4.3 Communication Models and Protocols

The physical constraints as outlined above typically render wireless communication media unreliable. This manifests in temporary breakdown, increased packet loss, or high transmission latencies. In order to address these issues and to make robots more robust against network failure, communication should be regarded optional where possible. In case of emergency or for remote control of a robot, however, reliable transmission is required even though the network may be temporarily unavailable.

These conflicting requirements render the decision for a suitable communication model difficult. There exist no appropriate measures that are able cope with network unreliability and guarantee reliable transmission and freshness of data. The most suitable transport protocol should therefore be selected for each task. Below, we introduce some communication models and emphasise their characteristics with respect to network unreliability.

2.4.3.1 Unicast Communication

Unicast realises *a point-to-point* (1:1) communication with exactly two involved systems. It is the most widely adopted communication model that typically provides either reliable or unreliable transmission characteristics. Unicast channels may further be operated *connection-oriented* or *connection-less*.

Connection-oriented unicast communication is typically used in conjunction with reliable data transmission. TCP [112], for example, takes care of data delivery, message order retention, and retransmission. However, reliable transportation is not suited for all applications equally well, as transmission may stall in case of network failures. This results in increased delays.

Connection-less unicast communication, in turn, typically realises unreliable data transmission. UDP [110] is an example of a connection-less, unreliable transport protocol. It is well suited for mass data transmission where low transmission latencies are important but not all data are required to reach their destination.

2.4.3.2 Broadcast Communication

A broadcast communication model realises *point-to-multipoint* (1:n) communication where one system talks to several other systems in a local network segment. Broadcast is particularly efficient when addressing several receivers in parallel. It thereby reduces the communication overhead from $\mathcal{O}(n)$ to $\mathcal{O}(1)$ as only one message has to be sent. It thus exploits the broadcast transmission capabilities of the communication media if present.

Data transmission is typically unreliable and message-oriented. Communication channels operate in connection-less mode. Broadcast communication is well suited for distributing periodical data among several receivers. An example from the domain of mobile robots is the periodical distribution of sensor data.

2.4.3.3 Multicast Communication

Just like broadcast, the multicast communication model realises *point-to-multipoint* (1:n) communication. It provides the additional capability of limiting the number of receivers by subscribing to a specific multicast address.[9] It thereby enables participants to form virtual subgroups within a network. Receivers that are not involved in a communication are so not forced to receive the data. This characteristic renders it especially well suited for mobile robots where participating robots may have to be assigned to a specific group.

Multicast is also often adopted for infrastructure maintenance in computer networks where multiple, but not all systems may have to be addressed. It is furthermore possible to route traffic beyond the local network segment. Transmission characteristics are the same as for broadcast.

[9]Multicast addresses are allocated from the networks `224.0.0.0/4` (IPv4) or `ff00::/8` (IPv6).

2.5 Security

Spica incorporates functionality for confidential communication as well as message authentication and modification detection. We will therefore introduce some fundamentals in cryptography in this section. Authentication is required to identify authorised and unauthorised participants that take part in the conversation. Encryption is required to prevent unauthorised eavesdropping and amendment.

We only discuss techniques that are efficient and applicable for mobile robots. This implies that processing costs have to be kept low. We assume that communicated data must be kept confidential for only a limited time span. With group communication in mind, we further assume that a *Pre-Shared Key* (PSK) is used in order to dispense with a respective distribution protocol. Key exchange in groups would require the adoption of a *Group Key Agreement Protocol* such as *Group Diffie-Hellman* [126] which is quite expensive and depends on asymmetric authentication.

2.5.1 Authentication

Authentication is typically linked to asymmetric cryptosystems such as RSA [117, 116] or ElGamal [40]. We nevertheless prefer authentication schemes that are less computationally expensive: A *Message Authentication Code* (MAC) is a symmetric approach for authenticating arbitrarily long portions of data. In combination with a secret key, it generates a so-called authentication token that uniquely identifies the message, protecting both, *data integrity* and the *authenticity*. In contrast to asymmetric authentication approaches, only one key is involved for signing and verification. This implies that every participant in a group may generate valid signatures. Hence, MACs are not applicable for *individual trust relationship* and *non-repudiation* guarantees.

A MAC is a *Cryptographic Hash Function* (CHF) with additional security requirements to resist *existential forgery*[10] under *chosen-plaintext attacks.*[11] The *Keyed-hash MAC* (HMAC) [15, 91] algorithm is a type of MAC which is calculated by combining an iterative CHF with a secret key. The MD5 [115] or SHA-1 [90] hash functions are typically used in this context. We will not consider *Modification Detection Codes* (MDC) here as MACs already address message integrity. Have a look at Menezes et al. [84, Chapter 9] for more information on message authentication and hashing in general.

2.5.2 Encryption

Encryption is typically adopted to preserve the confidentiality of a conversation. In cryptography, ciphers are algorithms that perform encryption and decryption of data. A cipher takes at least two input arguments – an *unencrypted plaintext* and a *key* – and returns a *ciphertext* representing the plaintext encrypted with the key. Modern cryptography distinguishes two cipher types: *Symmetric* and *asymmetric ciphers*. Symmetric ciphers use trivially related but often identical cryptographic keys for both decryption and encryption. They must be kept secret in order to preserve the security guarantees. Asymmetric ciphers require two keys,

[10]Any message m and a valid MAC h for m where m has not been authenticated before.

[11]An attacker is able to choose arbitrary plaintexts to be encrypted and obtain the corresponding ciphertexts.

a public key that is used for encryption and a private key for decryption that must be kept secret again. They cannot be derived from each other in a trivial fashion. Have a look at Menezes et al. [84, Chapter 8] for more information on asymmetric cryptography.

In Spica, we rely on symmetric ciphers because of their less expensive operation and better applicability for streams of data. Symmetric ciphers can be divided into two classes again, namely *Block ciphers* and *stream ciphers*.

2.5.2.1 Symmetric Block Ciphers

A block cipher is a symmetric cipher that operates on groups of bits with a predefined length. Each such group is called block. The keys control the encryption and decryption operations, the result of which is typically a block of the same length. A well-known representative of the family of block ciphers is the *Advanced Encryption Standard* (AES) [89]. Have a look at Menezes et al. [84, Chapter 7] for a more detailed introduction to the fundamentals of block ciphers.

A block cipher alone does not provide appropriate encryption capabilities for streams of potentially endless data. This is because of the non-randomised encryption of most ciphers. Hence, encrypting one and the same block twice will result in the same ciphertexts. A *Block Cipher Mode* (*Cipher Mode* or BCM) may be used for turning a block cipher into a cipher capable of encrypting and decrypting streams of data.

2.5.2.2 Block Cipher Modes

A Cipher Mode facilitates symmetric encryption and decryption of arbitrarily long input. This is achieved by consecutive execution of a block cipher with some key and typically an *Initialisation Vector* (IV).[12] The plaintext is divided into blocks of the same length, whereby the last one may need to be padded. Each plaintext block is then processed accordingly. Earlier cipher modes only guaranteed either confidentiality or integrity; the more recent ones guarantee both at the same time. Modes of the latter case are referred to as *Authenticated Encryption with Associated Data* (AEAD).

One condition must be met by a cipher mode to be applicable for unreliable communication media: It must not take the output of previous processing steps into account (*feedback*), as this would imply that decryption is sensitive to data loss. Table 2.1 outlines some cipher modes along with their fundamental characteristics. The NIST approved a subset of them for use with AES.

CTR is well suited for the given application domain. A MAC as outlined above in Section 2.5.1 must be used in this case in order to compensate for the absence of the data integrity guarantee. The AEAD modes CCM and GCM both have the specific requirement that Nonce/Key[13] pairs must not be reused. This would destroy the security guarantees. As a result, it is extremely difficult to use CCM or GCM securely with statically configured keys. OCB is especially interesting as it involves only little overhead, approximately as much as CBC.

[12]An IV is a random block of data used for initialising data encryption; it does not have to be kept secret.

[13]A Nonce for AEAD modes is typically the IV combined with a salt (random bits).

Mode Name	AEAD	No Feedback	NIST Approved	Comments
Electronic Codebook (ECB)	✗	✓	✓	Insecure
Cipher-Block Chaining (CBC)	✗	✗	✓	
Output Feedback (OFB)	✗	✗	✓	
Cipher Feedback (CFB)	✗	✗	✓	
Counter (CTR)	✗	✓	✓	
Counter with CBC Mode (CCM)	✓	✓	✓	[139]
EAX	✓	✓	✗	[16]
Galois/Counter Mode (GCM)	✓	✓	✓	[83]
Offset Codebook (OCB)	✓	✓	✗	[118], Fast

Table 2.1: A selection of Cipher Modes with their integrity as well as feedback characteristics

2.5.2.3 Stream Ciphers

Stream ciphers exhibit basically the same capabilities as block ciphers that operate in a Cipher Mode: Both are capable of encrypting and decrypting messages of arbitrary length. A Stream Cipher generates a pseudo-random key stream that is applied to a plaintext, similar to a *One-Time Pad* (OTP).[14] The period of the pseudo-random key stream must be long enough to prevent security weaknesses.

Receivers have to synchronise themselves to the key stream of the sender if ordinary stream encryption is used, i.e. the same restrictions apply as for the Block Cipher Modes. In case of *self-synchronising* stream ciphers, the key stream is derived from the last N ciphertext messages. Receivers thus only have to wait for N ciphertext messages for resynchronisation. Have a look at Menezes at al. [84, Chapter 6] for more information on stream ciphers.

[14] A One-Time Pad applies a key to a plaintext where both must have the same length. This cipher provides perfect security if the key is truly random, kept secret, and never reused.

3 Related Work

"If I have seen further than others, it is by standing upon the shoulders of giants."

— Sir Isaac Newton
In a letter to Robert Hooke, 1676

The Spica development approach we propose in this work builds on lessons learnt in robot software development and related work in this area. Hence, we will first review different approaches that bear resemblance to Spica. Existing solutions can thereby be roughly classified into middleware and development approaches. We will present and discuss examples of both classes, starting with a review of the classical middleware approach. In the course of this chapter, we will furthermore justify the decision for our distributed resource discovery approach Geminga. Section 3.3 introduces related work in this area, mainly discussing service discovery in computer networks. It covers some well-known approaches that relate to the techniques we adopted.

3.1 Robot Middleware

Research on robot software architectures mostly focused on middleware frameworks in the past. The Spica development framework shifts the focus right to the development methodology with a more abstract view on the application domain. It is our intention to make the development process and the implementation more platform-independent, allowing the developer to focus on the actual functionality rather than bothering with characteristics of the platform.

This section introduces some existing solutions in the field of robotic middleware frameworks. We will point out the addressed problems as well as the strengths and weaknesses of the approaches related to the problem analysis given in Section 1.2. None of the presented approaches addresses security concerns explicitly.

3.1.1 Miro

Miro (**Mi**ddleware for **Ro**bots) [129, 128] is a middleware framework for robotic software written in C++ that follows a strictly layered software design. Its implementation is mainly based on *ACE* (Adaptive Communication Environment) [119] and *TAO* (The ACE ORB) [120], a realisation of the *Common Object Request Broker Architecture* (CORBA) [98].

The development started at the University of Ulm and is now continued at the NASA Ames Research Center where Miro became a central part of the SORA framework as introduced in Section 3.1.3. Miro is licensed under the LGPL open source license. Several layers abstract from the underlying robot hardware and software platforms. The *Device Layer* provides hardware abstraction and takes care of interaction with the operating system. The *Communication Layer* offers mostly CORBA-based services that are required for information exchange and cooperation between robots. The *Service Layer* abstracts from sensor and actuator hardware by decoupling the device interfaces from the driver implementations. Miro additionally provides several high-level toolkits.

Communication within a robot and beyond the local scope is based on remote method invocation and the *CORBA Notification Service* [97] that provides asynchronous event distribution channels – typically referred to as *Notification Channels*. In order to guarantee fault-tolerance and scalability, Miro offers *NotifyMulticast* (NMC) for efficient, multicast-based data distribution. The NotifyMulticast module attaches directly to the Notification Channel as consumer and supplier. For delivery, it takes events off the Notification Channel and wraps them into a NotifyMulticast-specific message structure. Reception works just the other way round. Implementations are available for C++ and Java. NotifyMulticast is the predecessor of the group communication capability in Spica.

Addressed Problems Miro provides communication facilities for local and remote operation. Thanks to NotifyMulticast, it furthermore addresses unreliability and efficiency for group communication. Auto configuration is not supported. Miro instead relies on the *CORBA Naming Service* that maintains a database for name-address mappings, delivering location transparency.

Miro follows a middleware-oriented rather than a modelling-oriented approach. Its architecture abstracts from the underlying hardware and software platforms, there is no dedicated platform-independent model as such. Considering high-level control, Miro provides extensible frameworks that target behaviour control [131] and vision processing [132].

Major Contributions Miro is capable of resource-efficient, asynchronous event distribution thanks to NotifyMulticast. Its multicast-based design furthermore facilitates application in unreliable communication media. If furthermore abstracts from the underlying hardware and software characteristics by adding software layers. Miro provides interface definitions for common component, service, and device types found in robotic platforms; heterogeneity issues are handled by CORBA. Elaborate frameworks finally address abstraction on the highest level.

Weaknesses Data distribution via NotifyMulticast is resource-efficient but may require a large amount of communication bandwidth. This is because notification events are wrapped into *CORBA Any* objects. If serialised, the type information of the contained data are typically encoded explicitly. However, it is possible to dispense with this additional information.

The complex, highly abstracted software architecture of Miro furthermore results in a quite steep learning curve. Due to the comprehensive underlying middleware solution, it is furthermore quite resource demanding. Miro does not explicitly address automatic configurability. It merely utilises the location transparency provided by the CORBA Naming Service instead.

3.1.2 CLARAty

CLARAty (**C**oupled **L**ayer **A**rchitecture for **R**obotic **Au**tonom**y**) [134, 92] is a software framework for robot control systems with a focus on reusability and integration of existing algorithms and components. Developed at NASA JPL in cooperation with NASA Ames Research Center, the Carnegie Mellon University, and the University of Minnesota, CLARAty resembles a two layered, generic object-oriented framework implemented in C++. It is released under the *CLARAty TSPA License*.[1]

The software architecture decomposes into a *Decision Layer* and a *Functional Layer*. The Decision Layer contains declarative activity and executive definitions as well as realisations of functional requirements. The Functional Layer is an interface to the system hardware and respective capabilities.

The CLARAty framework focuses on software modularity and composition of functional primitives rather than communication and collaboration. It provides a proprietary communication scheme with publish/subscribe characteristics between the two layers but no general communication facility. The ACE library helps addressing heterogeneity in the local scope.

Addressed Problems The focus of CLARAty is clearly on single robot development and not on collaborative operation. It exclusively targets the NASA rovers and hence aims at their architectural design. CLARAty provides abstraction for the hardware and some high-level features. A more fundamental abstraction is almost impossible because of the two-layer approach.

Major Contributions The major contribution of CLARAty lies in its comprehensive collection of algorithm implementations. Some more facilities may be of use for other robotic platforms as well. The two-layer approach clearly has the potential for higher efficiency because of less abstraction. The drawbacks, however, are obvious: Developers are required to deal with low-level issues more often.

Weaknesses With respect to the application in autonomous mobile robots, the most conspicuous weakness of CLARAty is the absence of an integrated communication approach. The publish/subscribe scheme is implemented on top of a connection-oriented protocol which renders it less suitable for communication between robots. Consequently, it does not explicitly address multi-robot scenarios. Automatic configuration is not supported in any form. CLARAty addresses heterogeneity only in the local scope and mostly for NASA rovers and research platforms.

The weak separation of structure and functionality in the functional layer is clearly efficient but presumably harder to maintain than a more strictly structured approach. CLARAty mainly relies on object orientation for abstraction and generalisation. It furthermore disqualifies for commercial use because of its restrictive license.

[1]Proprietary license of the *California Institute of Technology* (Caltech) covering *Technology and Software Publicly Available*. It explicitly prohibits commercial use.

3.1.3 SORA

The *Service-Oriented Robotic Architecture* (SORA) [45] is currently being developed at NASA Ames Research Center. It merges the fundamental technologies of Miro and CLARAty to build a services-oriented architecture for robot software development. Just like Miro, SORA adopts ACE and TAO that contribute support for remote method invocation and event distribution based on the CORBA Notification Service. NotifyMulticast was recently[2] integrated as well. SORA adopts some components of CLARAty's functional layer in order to facilitate the integration with NASA rovers and provide further valuable functionality. The ACE service framework [119] facilitates service orientation.

The Miro-based parts of SORA are licensed under the LGPL open source license. However, because SORA as such is not yet publicly available, it is considered closed source.

Addressed Problems SORA addresses the same problems as Miro but aims at modularity and the needs of NASA's robotic architecture. It thus provides viable measures for resource-efficient group communication. CLARAty contributes fundamental algorithms for robot control.

Just like Miro, SORA follows a middleware-oriented rather than a model-oriented approach. The focus is mainly shifted towards a more modular architecture design and the integration of external components.

Major Contributions SORA inherits its major functionality from Miro, so it basically shares its contributions as well: NotifyMulticast delivers a resource-efficient and failure-tolerant group communication scheme. Abstraction layers decouple high-level functionality from fundamental hardware or software peculiarities. The underlying middleware frameworks take care of heterogeneity issues. Thanks to the service-oriented structure, integration of existing components may be accomplished quite conveniently. This is especially important for the functionality borrowed from CLARAty.

Weaknesses Weaknesses resemble these of Miro and CLARAty as well: If not handled appropriately, event data distribution with NotifyMulticast may result in excessive network utilisation. SORA furthermore inherits the complex architecture from Miro, including the underlying middleware architectures.

3.1.4 Player/Stage

Player/Stage [51, 133] is a software framework for RCAs. Player provides a network interface to a variety of robot and sensor hardware based on a simple file I/O API. Its client/server-based architecture allows robot control programs to be written in almost any programming language and for any platform, as long as a networking functionality is available. Platform abstraction is thereby quite weak. Player supports multiple concurrent client connections to devices, creating new possibilities for distributed and collaborative

[2]During a field test of the NASA K10 rovers in 2008, NotifyMulticast served as a communication channel for the data transmission from the testing area to Houston.

sensing and control. Stage is a two-dimensional simulation component used in conjunction with Player. Gazebo, another component of the Player/Stage project is a three-dimensional simulation environment. Stage and other components typically share information via TCP or UDP. Communication between Player and Gazebo, in contrast, relies on shared memory interactions.

Addressed Problems Player is a network server for robotic systems and therein contained hardware components. Network connectivity through the BSD socket interface achieves basic platform independence, so clients may access Player in a portable fashion using the TCP or UDP transport protocols. The file I/O interface realises only a simple form of hardware abstraction but qualifies for cross-platform applicability as well, mostly because of its simplicity. Communication in unreliable communication media may further be realised with the UDP transport protocol.

Major Contributions Player merely realises a network server that makes hardware components of a robot accessible by remote clients. For the sake of simplicity, it thereby reduces its high-level functionality to the possible minimum: It builds on raw IP-based transport protocols and a very basic interface towards provided services. This not only reduces the need for elaborate middleware functionality but also enhances Player's genericness, realising a less intrusive programming model and thus facilitating integration with existing architectures.

Weaknesses Player primarily aims at single robot development and therefore dispenses with group communication and elaborate collaboration facilities. An optional driver for multicast support was proposed[3] that nevertheless seems not to be maintained anymore. Player is therefore difficult to use in highly dynamic multi-robot scenarios because is lacks a viable communication scheme. A UDP transport is the only viable measure that addresses unreliable communication media. Player, adopting a Zeroconf-based solution, natively supports service discovery. The development of the underlying library has nevertheless been suspended, rendering this support ineffective.

Compared to the other robotic software architectures, Player provides the weakest form of abstraction. Hardware is accessed through a file I/O-like API. No more elaborate approaches or software abstraction layers are provided, mostly for sake of simplicity. High-level functionality is incomplete and relies on third-party support.

3.1.5 Orocos and Orca

The Orocos (**O**pen **Ro**bot **Co**ntrol **S**oftware) [24, 25] project provides a general-purpose, real-time-oriented, modular framework for robot and machine control. It comprises four libraries written in C++, each of which targets a specific application domain: *The Real-Time Toolkit* (RTT) targets the development of highly configurable and interactive component-based real-time control applications. The *Kinematics and Dynamics Library* (KDL) targets modelling and computation of kinematic chains. The *Bayesian Filtering Library* (BFL) [47,

[3]Referring to a conversation on the respective Player mailing list, started 2006-04-25.

Chapter 7] provides a framework for inference in dynamic Bayesian networks including Kalman or Particle Filters [76]. The *Orocos Component Library* (OCL) is based on RTT, KDL, and BFL. It contains software and driver components to build robot control architectures.

An *Orocos template* is a description of components that work well together and the interfaces of which are compatible. Applications build on the *Orocos Control Component* that can be interfaced through properties, events, methods, commands, and data flow ports. Events resemble broadcast semantics where commands are unicast by design. A data flow port represents a transport mechanism to communicate data either buffered or unbuffered between components. For remotely accessible components, Orocos optionally integrates CORBA.

Orca is an open-source framework for developing component-based robotic systems, forked from an earlier version of Orocos. Its communication infrastructure relies on the *Internet Communication Engine* (Ice) [60, 61], a high-performance remote procedure call framework similar to CORBA. With Orca it is possible to plug together existing or newly developed components and create quite complex robotic applications with only little effort.

Both projects are licensed under the GPL and LGPL open source licenses. KDL and OCL of Orocos are licensed under the LGPL whereas RTT and BFL force the GPL with the exception that any program may link the library. Orca is dual licensed.

Addressed Problems Orocos and Orca abstract from the underlying platform by providing ready-to-use components that ease the development of robotic software platforms. Coverage thereby includes hardware, software, and even high-level functionality. They partly follow the Player approach but adopt existing middleware platforms for component interaction: Orocos relies on CORBA whereas Orca integrates Ice, two similar but fundamentally different distributed middleware approaches. The naming service of the respective middleware framework finally delivers location transparency but no automatic configuration.

Orca includes a viable solution for ad hoc and group communication. Orocos provides a template mechanism with which applications can be created from several components in an abstract fashion.

Major Contributions The most important characteristic of Orca – at least with respect to AMRs – is the IceStorm-based event distribution facility. It is capable of multicast-based data transmission and thus explicitly addresses the needs of the application domain. Both projects provide a huge collection of ready-to-use components, facilitating the development of robotic software. CORBA and Ice finally care about the more prevalent heterogeneity issues and provide the basic functionality for component interaction.

Weaknesses Orocos and Orca focus more on functionality than on communication. Established middleware frameworks cover the latter one. Orocos adopts CORBA and does therefore not provide applicable ad hoc and group communication capabilities. Naming services only provide location transparency but not explicit automatic configuration facility.

	Communication			Dynamic	Abstraction		
	Middleware	Group	Ad Hoc	Config	HW	SW	Ctrl
MIRO	CORBA	NMC	NMC	—[5]	✓	✓	✓
CLARAty	—[1]	✗	✗	✗	✓[6]	—[7]	✓[8]
SORA	CORBA	NMC	NMC	—[5]	✓	✓	✓
Player/Stage	—[1]	Multicast[3]	UDP	Zeroconf[4]	✓	✗	—[2]
Orocos	CORBA[2]	✗	✗	—[5]	✓	✓	✓
Orca	Ice	IceStorm	IceStorm	—[5]	✓	✓	✓

1 = Proprietary 2 = Optional 3 = Patch 4 = Obsolete library 5 = Name-address mapping service
6 = Drivers 7 = Two layer architecture 8 = Execution, Planning, Communication

Table 3.1: Properties of robotic programming frameworks

3.1.6 Summary

Table 3.1 summarises the most important properties of the aforementioned robotic software frameworks. It particularly emphasises their communication and abstraction capabilities. Miro, SORA, Orocos, and Orca provide quite satisfying abstraction characteristics, covering the hardware (HW), software (SW), and event high-level control (Ctrl). Only Player/Stage restricts itself mainly to hardware abstraction. This is not necessarily a drawback but inherently depending on the application. CLARAty is clearly an exception as it primarily targets the NASA rovers.

With regard to the communication capabilities, only Miro, SORA, and Orca provide practicable solutions for groups of robots. It is especially important to mention that only some architectures support group communication schemes such as multicast. As SORA builds on Miro, it inherits all its valuable features such as NotifyMulticast for group communication. The Ice middleware adopted by Orca has built-in support for UDP or multicast communication for its event distribution system, rendering techniques such as NotifyMulticast unnecessary.

Conceptual support for automatic dynamic configuration is only available in Player/Stage. However, the obsolete realisation will possibly break compilation.

3.2 Development Frameworks

Opposed to software frameworks that facilitate development of RCAs, development frameworks target the development process and therefore establish a new layer of abstraction above. Several development frameworks for RCAs exist but address quite different application domains: While some are applicable in a platform-independent manner, others are bound to a specific platform but provide more specific development support. This section discusses solutions that address all of the aforementioned characteristics. We thereby restrict ourselves to only the most well known approaches.

3.2.1 MARIE

MARIE (**M**obile and **A**utonomous **R**obotics **I**ntegration **E**nvironment) [34, 33] is a software tool and middleware framework for building robotic software systems using a component-based approach. It targets the development of new and the integration of existing software components, and thus facilitates the creation of software systems out of multiple heterogeneous software elements. MARIE explicitly addresses distributed, networked systems and different platforms. Its main building block is the *Mediator Interoperability Layer* (MIL), a realisation of the *Mediator Design Pattern* [48] that offers a common interaction language for components in the system. The MIL is composed of four components: *Application Adapters* (AA) interface applications with the MIL. AAs exist for a variety of established robotic middleware and development frameworks. *Communication Adapters* (CA) represent communication link logic for components with incompatible communication mechanisms and protocols. *Application* and *Communication Managers* (AM; CM) are system level components that instantiate and manage components locally or remotely. MARIE itself is written in C++ for UNIX environments, incorporating the ACE library.

The MARIE middleware for the development process decomposes into *Core*, *Component*, and *Application* layers. The Core layer is responsible for communication, data handling, distributed computing, and low-level operating system functions. The Component layer specifies and implements useful frameworks to add components and to support domain-specific concepts. Useful tools to build and manage applications based on available components are contained in the Application layer.

Addressed Problems MARIE explicitly addresses communication issues as outlined in Section 1.2. It therefore provides the MIL for message conversion and distribution. Unsupported applications depend on custom adapters Communication channels between components rely on either shared memory or network-based communication based on TCP or UDP. MARIE explicitly supports group communication via UDP broadcast. Security concerns and automatic configuration, however, are not addressed. By adopting the graphical data flow modelling tool FlowDesigner [33], MARIE facilitates the interconnection of components.

Major Contributions MARIE particularly contributes to the development of robotic software in heterogeneous environment. It enables developers bringing together heterogeneous software modules in an abstract fashion. By supporting established software solutions for robots, MARIE promotes reuse of existing solutions and facilitates rapid development. UDP and broadcasting furthermore allow operation in ad hoc networks and group communication. With the help of FlowDesigner, MARIE supports visual configuration and programming of robotic appliances.

Weaknesses Even though MARIE allows broadcasting data, a more elaborate approach such as multicast would be valuable. There is furthermore no automatic configuration facility available. This results in the need for a static communication setup, implying an increased configuration effort. Even though a static configuration has clear advantages such as no erroneous reconfiguration, it definitely is not suited well for all the requirements of dynamic scenarios, though.

3.2.2 Microsoft Robotics Developer Studio

The *Microsoft Robotics Developer Studio* (MSRDS) [74] is a development environment for robotic systems. It provides configuration tools tailored to the development of robot software applications. The Visual Programming Language (VPL) supports graphical programming. The MSRDS promotes modular service-oriented software development for robots, allowing users to interact with robots through Web- or Windows-based user interfaces. A simulation engine with realistic three-dimensional models provides simulation capabilities for robotic applications.

The MSRDS adopts an asynchronous communication model based on Web Service technology. The *Concurrency and Coordination Runtime* (CCR) library thereby facilitates application development. It promotes threaded application development and takes the burden from developers having to care about asynchrony. Service orientation is supported by the *Decentralised Software Services* (DSS) library, primarily promoting modularisation and code reuse. DSS services communicate over the SOAP-based *DSS Protocol* (DSSP) that itself relies on the *Representational State Transfer* (REST) [43, 44] model. SOAP is the *Simple Object Access Protocol* (SOAP) [142]. The programming model can be applied to a variety of robot hardware platforms and third parties may supply additional functionality. The Microsoft .NET *Common Language Runtime* (CLR) [22, 114] supports several different programming languages and even different target platforms.

Addressed Problems The MSRDS facilitates rapid prototyping and visual programming for robotic applications in a service-oriented fashion, promoting modularisation and code reuse. It furthermore explicitly supports the development of behaviour and input processing, but also targets the specification of robot communication. Service discovery and automatic configuration are based on UPnP [58], realised as an independent service rather than a programming concept. VPL facilitates modelling of robotic interactions and control applications. It simplifies software development for novice users but does not cover the entire functionality. The CLR guarantees for cross platform applicability.

Major Contributions The VPL simplifies programming to some extend. Even more important, it raises development to a more abstract level by increasing platform independence at the same time. Service orientation thereby facilitates modularisation and code reuse. Being the basis for the MSRDS, the Microsoft .NET framework brings a broad coverage of programming languages and platforms. Heterogeneity issues are so handled in a clean way, similar to the middleware approach of robotic software frameworks.

Weaknesses The MSRDS is mostly a prototyping and evaluation platform rather than a generic development environment for robotic software. This is because the MSRDS lacks the ability to guarantee a certain level of service qualities for robotic software components, such as high efficiency or low response time. Applications furthermore have to be able to interact with the CLR or provide appropriate communication facilities for the adopted Web Service interfaces. The MSRDS does not explicitly address highly dynamic and unreliable environments, because of the large overhead involved in the Web Service-oriented architecture and the lack of suitable communication schemes for group communication.

3.2.3 CoSMIC

CoSMIC (**Co**mponent **S**ynthesis using **M**odel-**I**ntegrated **C**omputing) [52, 11] is a generative tool-chain for distributed real-time and embedded applications development following the MDA paradigm. It comprises a collection of domain-specific modelling languages and generative tools that support developers in creating, configuring, and validating distributed component-based real-time systems. The focus is on QoS constraint modelling. Applications created and configured with the help of CoSMIC are based on the *Component Integrated ACE ORB* (CIAO) [136] and the *Quality Objects framework* (QuO) [145]. CIAO as such is written in C++ and based on ACE and TAO [119, 120].

COSMIC addresses generative techniques for component middleware configuration metadata and container policies, synthesising aspectised application component logic, and end-to-end QoS assurance. Other languages than C++ are not directly addressed but components compatible with CORBA or the *CORBA Component Model* (CCM) [95] may be integrated naively.

Addressed Problems The CoSMIC tool chain primarily aims at provisioning large-scale distributed real-time and embedded systems. It is a model-driven approach for generating applications out of basic components. The CORBA middleware thereby cares about the interaction facilities. CoSMIC does not explicitly address security considerations and automatic configuration; the CORBA Naming Service merely delivers location transparency. Developers are provided with tools for modelling and analysing application functionality along with the respective QoS requirements. The CoSMIC development approach adds further analysis and verification capabilities. Configuration and composition metadata are generated automatically for the underlying CCM middleware. QoS characteristics are one of the most important modelling aspects in CoSMIC: Code generation directly depends thereon in order to provide efficient implementations.

Major Contributions CoSMIC is a powerful MDA-based modelling and code generation approach that targets embedded real-time applications. It takes the burden from developers to create code for resource-constrained systems in a quality-aware fashion. Analysis of models is further an important characteristic because correctness of applications is important in the embedded domain, even though it is hardly possible to prove.

Weaknesses The complex development approach and the comprehensive underlying middleware platform render CoSMIC inappropriate for the development of AMRs. This is because CoSMIC aims at the development of large-scale, quality aware applications. AMRs, in contrast, exhibit rather small-scale software architectures and have to handle unreliability and uncertainty in an appropriate manner. The provided networking capabilities of CoSMIC furthermore depend on CORBA that is not well suited for ad hoc networking between mobile robots. Another issue is the steep learning curve, so application development will presumably be advantageous for large-scale systems only.

	Communication			Dynamic	
	Middleware	**Group**	**Ad Hoc**	**Config**	**Modelling Domain**
MARIE	MIL	✓[1]	✓[1]	✗	Interaction
MSRDS	.NET/SOAP	✗	✗	✓[2]	Interaction, Processing
CoSMIC	CORBA	✗	✗	✗	Large-scale configuration

1 = UDP Broadcast 2 = UPnP

Table 3.2: Properties of robotic development frameworks

3.2.4 Summary

The three presented development solutions realise quite different approaches. Any single one, however, deals with the interaction between components. MARIE and MSRDS explicitly aim at the composition of independent components. CoSMIC, in turn, provides modelling means for the characteristics of the communication links between components, i.e. it cares about QoS constraints and service guarantees. It thereby addresses large-scale systems with service quality demands. This functionality clearly targets a quite different application domain. MARIE provides a graphical data flow modelling tool. MSRDS addresses several programming languages and supports visual programming with the VPL.

None of the solutions explicitly addresses cooperation and group communication between independent robots. MARIE allows using UDP broadcast but does not provide further support. The MSRDS and CoSMIC do not even provide a fault-tolerant transport protocol. Only the MSRDS addresses dynamic configuration by providing a UPnP-based discovery service. Quite strong differences in the application domains obviously had a severe influence on the focus of the development approach. Table 3.2 summarises the main differences of the three development frameworks.

3.3 Resource and Service Discovery

Service discovery in computer networks is a broad domain. Many different approaches target discovery, integration, and configuration of services or hosts. Robust service discovery schemes for unreliable or ad hoc networks are, however, still subject to active research. Below, we discuss some well-established approaches for service discovery and auto configuration in computer networks and evaluate them according to our requirements. We will show that only few of these approaches really meet our requirements.

3.3.1 UDDI

Universal Description, Discovery and Integration (UDDI) [94] is an open industry initiative for a platform-independent, XML-based service registry where access is realised through SOAP. It provides three, actually even four different types of registries: *White Pages* contain descriptions on individual business units including, for example, their name, address, and contact data. *Yellow Pages* provide industrial categorisations based on standard taxonomies. This facilitates classification of services according to their functionality. *Green Pages* provide human-readable technical details for services. The *Service Type Registry* represents the machine-readable equivalent.

The internal structure of a UDDI directory follows the definitions of the respective UDDI XML schema [94]. It is made up of several data structures that roughly exhibit the classification schema: The `businessEntity` structure characterises the service providers, i.e. the businesses that provide a service. The `businessService` structure defines the classification of a service while the `bindingTemplates` structure gives detailed specifications. A `bindingTemplates` structure references a so-called `tModel`, a representative of unique concepts or constructs and the most central part of a UDDI directory: It further describes the characteristic of a service. By defining an optional `publisherAssertion` structure, relations to other businesses may be established.

Entries in the UDDI directly relate to WSDL (*Web Service Description Language*) [143] instances. WSDL is a meta language that describes the functionality, data types, and protocols of a Web Service. It thereby acts as a platform-independent interface description of publicly accessible methods. The WSDL `service` structure relates to the UDDI `businessService` structure, a WSDL `port` to a UDDI `bindingModel`. The WSDL `service` structure finally relates to a respective `tModel` in the UDDI.

The UDDI therefore resembles a WSDL registry used as a service directory for automated service composition in a SOA [138]. It exhibits a standardised interface through which services may be published, located, and maintained. As a typical representative of centralised directory-based service discovery approaches, UDDI disqualifies for adoption in mobile ad hoc networks.

Addressed Problems The UDDI is an attempt to address the lack of a standardised directory service for the Web Service architecture. It facilitates the discovery of businesses and Web Services along with their contact information or communication protocols. UDDI focuses on the maintenance of interface descriptions that relate to WSDL definitions but furthermore adds information on their respective providers. It is therefore well suited for traditional remote method invocation. Thanks to the genericness of the `tModel` structure, other concepts may be represented as well.

Major Contributions Interface descriptions of Web Service are typically represented in WSDL. The UDDI provides a service directory that allows publishing, locating, and maintaining Web Services in a standardised, consistent fashion. Its internal data structure thereby exhibits the central elements of a WSDL description, simplifying the mapping between these two representations. The design primarily targets large-scale deployment over the Internet but also small- or medium-scale deployment within a local network.

Weaknesses A centralised discovery approach is not appropriate for the given application domain. This is because of the fact that network connectivity in a mobile ad hoc network is not guaranteed to be stable and may exhibit reduced bandwidth guarantees. The adopted SOAP protocol is no appropriate solution in this case: It typically relies on the connection-oriented TCP protocol that does not tolerate data loss with respect to real-time behaviour. The quite large amount of overhead furthermore influences the reactivity and network utilisation. UDDI promotes *active service discovery* [12] which behaves more vulnerable to network unavailability than *passive service discovery*.[4]

3.3.2 Zero-Configuration Networking

Zeroconf [125] comprises a set of technologies facilitating automatic network configuration. The automatic link-local address assignment for IPv4 (IPv4LL) [28] cares about the IPv4 [111] network configuration. Zeroconf furthermore proposes techniques that facilitate service configuration and discovery. *Multicast-enabled DNS* (mDNS) [27] thereby simplifies name resolution in the local network. It dispenses with a central DNS (*Domain Name System*) [86, 87] server by requiring each device to maintain a private DNS server instance. A name is resolved by simply multicasting a name resolution query. Service discovery with Zeroconf also dispenses with a central registry. It instead relies on the same private DNS servers and embeds respective service description records. Zeroconf calls this *DNS-based Service Discovery* (DNS-SD) [26]. The tree-layer architecture of a Zeroconf installation therefore comprises these three components, namely IPv4LL, mDNS, and DNS-SD. Because Zeroconf restricts itself to low-level configuration of the networking infrastructure and service discovery, it provides viable measures against misuse of bandwidth and for application in wireless computer networks. It is thus even applicable for mobile ad hoc networks. Well-known implementations are *Bonjour*[5] by Apple Inc. and the open source project *Avahi*.[6]

Addressed Problems Zeroconf first cares about automatic network configuration. It thereby assigns unique link-local network addresses to devices (IPv4LL) and cares about the fundamental name resolution facility (mDNS). An *active service discovery* further allows locating services in a local network (DNS-SD). Zeroconf explicitly addresses misuse of communication bandwidth and application in unreliable wireless networks. The Zeroconf protocol stack aims at making manual configuration dispensable for a local network. Security is not considered explicitly but may be realised through DNSSEC [2, 3, 4].

Major Contributions Zeroconf cares about network configuration and name resolution automatically. No dedicated servers such as DHCP (*Dynamic Host Configuration Protocol*) [38] or DNS are required anymore for the local case. Zeroconf thereby relies on well-established techniques and simple protocol operation. As the service discovery relies on the same infrastructure, no additional technology is required. The Zeroconf approach is resource efficient and even applicable for ad hoc networking to some extend.

Weaknesses Service discovery depends on active discovery queries that may impose increased communication expense: A passive announcement approach would dispense with query messages, exhibiting only a unidirectional data flow from the service to the consumers. Provided services are characterised through DNS TXT records. Even though this is a very generic and viable solution, it may involve increased communication overhead and processing expense as values are represented as strings. Hence, a recipient has to care about

[4]Passive service discovery involves periodical announcements that are distributed by the respective service.
[5]Apple Bonjour: `http://developer.apple.com/bonjour/` (accessed 2008-08-23).
[6]Avahi Zeroconf: `http://www.avahi.org/` (accessed 2008-08-23).

their interpretation self-dependently. Despite the few drawbacks, Zeroconf would also be well suited for application in the proposed Spica communication infrastructure.

3.3.3 UPnP Service Discovery

Universal Plug and Play (UPnP) is a collection of protocols that facilitates the configuration of networked devices. Among other services, UPnP adopts zero-configuration networking and service discovery. Automatic network configuration is thereby based on an early draft of the IPv4LL [28] proposal and hence resembles the functionality of Zeroconf. The *Simple Service Discovery Protocol* (SSDP) [53] – an expired IETF Internet draft – builds the basis for service discovery. SSDP announcements rely on HTTP NOTIFY requests delivered via UDP multicast. Once a device has acquired an IP address, it emits an announcement. UPnP also allows searching for specific devices in the network actively. This renders SSDP a *hybrid service discovery* scheme consisting of *active* as well as *passive service discovery* elements. The announcement message only contains very sparse information on the service. It instead provides a reference to a more detailed description expressed in XML and typically vendor-specific. Further high-level protocols are provided on top of this basic functionality.

The implementation of zero-configuration networking is the same for Zeroconf and UPnP but service discovery differs fundamentally. UPnP furthermore provides application protocols for specific devices. The services these two approaches provide are considered orthogonal and may so be used in parallel: Zeroconf provides basic network configuration facilities while UPnP cares about device-specific access.

Addressed Problems UPnP does not address a specific need of computer networking. It rather addresses many different aspects of automatic network configuration and service discovery: Zero-configuration networking, service discovery, and device access using vendor-specific. Hence, it addresses three important aspects where Zeroconf covers only two. Device access is by far the most important aspect of UPnP as zero-configuration networking resembles IPv4LL and service discovery has some severe weaknesses.

Major Contributions The most important characteristic of UPnP is its device access layer. It allows users to access and configure devices that are equipped with a suitable UPnP protocol implementation. Many router or firewall devices may furthermore announce themselves as Gateway Devices, allowing local UPnP controllers to retrieve or modify their configuration.

Weaknesses The SSDP protocol has some deficiencies especially with respect to network utilisation. In the problem statement of Goland et al. [53, Section 6.1] the authors note that "A mechanism is needed to ensure that SSDP does not cause such a high level of traffic [...]". Even though UPnP adopts the UDP transport protocol, it is not applicable in mobile ad hoc networking. This is partly because of the complex protocol operation, partly because of the number of required interactions: A detailed service description must be retrieved explicitly from the device. Furthermore, UPnP does not implement any authenticity measures.

3.3.4 Service Location Protocol

The *Service Location Protocol* (SLP) [57, 55, 56] is an IETF proposed standard of a service discovery protocol for local area networks. Devices that join the network use multicast for initial service discovery. An SLP infrastructure defines several actors: *User Agents* (UA) are devices that actively search for services; *Service Agents* (SA) are devices that announce services. *Directory Agents* (DA) are special caching instances for service offers that improve the scalability of SLP and help to reduce the traffic. They are considered optional. If present, they emit a heartbeat signal so that UAs and SAs get aware of their presence. Services are characterised by an URL and a set of attributes, i.e. key-value pairs.

The existence of a DA influences the behaviour of the SLP fundamentally: If no DA is present, UAs and SAs communicate directly with each other. In this case, multicast is used for delivering queries. If, however, a DA is available, UAs and SAs are forced to use it. SAs therefore have to register all their services with the DA. UAs, in turn, may only query the DA for required services. Given a query is too long for a UDP packet, it may be sent via TCP. If answers are too long, a UDP reply indicates that the answer follows via TCP. Multicast queries are typically delivered several times in order to overcome the unreliability of UDP.

Addressed Problems The SLP performs tasks similar to Zeroconf service discovery and SSDP of UPnP but is the only protocol that has reached the IETF standard status. A SLP installation may be operated in one of two modes, namely with or without a DA. Hence, unlike the other approaches it explicitly addresses scalability. UAs can search for services in a multitude ways using the LDAP (*Lightweight Directory Access Protocol*) search filters [123]. This renders SLP an *active service discovery* approach. Message authenticity based on asymmetric ciphers is supported but only rarely used.

Major Contributions The SLP is an Internet standard for service discovery in local area networks. It facilitates distributed discovery without the need of a central registry. DA caching instances may be added optionally. They resemble sorts of central repositories. UDP-based message transmission and redundant query delivery further consider message loss.

Weaknesses The active service discovery may impose increased communication expense as well. This is because of the fact that each query is intended to trigger one or more replies. Standard SLP furthermore requires UAs to retrieve the attributes of a result separately. This makes SLP vulnerable to network error. An extension of the protocol solves this issue [55].

3.3.5 Field Theoretic Service Discovery for Ad Hoc Networks

Lenders et al. [81, 109] present a service discovery approach modelled after electrostatic fields. It is lean, robust, and completely decentralised, resembling a form of content-based publish/subscribe system where processes can subscribe to messages containing information on specific subjects. The system explicitly targets mobile ad hoc networks without routing capabilities. It therefore introduces its own routing functionality.

Services are announced periodically. A predefined characteristic thereby determines the charge of a service. The available network capacity is one possibility in this case. Each service instance furthermore requires a unique identifier. Here, the MAC address or a time values may be used to achieve uniqueness. As messages are flooded, furthermore a well-determined TTL (*Time to Live*) value is required. Moreover, adjacent nodes in a network periodically exchange their potential values and thereby indicate their presence at the same time.

Clients may then search for a service. Queries are either forwarded by neighbours based on the potential values or on proximity, i.e. the lowest distance of a service in terms of hops to the destination. The reply contains the service address and optional information that further characterises a service.

Addressed Problems The field theoretic service discovery approach aims at wireless ad hoc networks without a communication infrastructure. It therefore does not assume routing protocols to be available. Service discovery relies on a hybrid scheme: Announcements establish the routeing infrastructure based on which service queries and replies are delivered. Node mobility is handled by timeout mechanisms and capabilities of the underlying communication media. The protocol explicitly addresses network unreliability and resource limitation.

Major Contributions Service discovery follows the physical example of potential fields. Hence, it builds on a well-founded theoretical basis. It exhibits a simple and clear design that presumably can be adapted for further applications such as routing protocols as well. There are furthermore no specific demands on the network infrastructure thanks to the independence from the underlying transport protocol. Network partitioning and network error in general are covered by timeout-based heuristics and capabilities of the underlying communication media.

Weaknesses Our proposed resource discovery approach Geminga exhibits similar characteristics but relies on the routing capabilities provided by the underlying network. It can do with fewer messages and less protocol interactions as it is a purely *passive service discovery* approach. Announcements and queries have to be flooded in the field theoretic service discovery approach. This may imply increased network load and a limiting factor for scalability. Countermeasures proposed by the authors include caching and aggregation of messages. The approach can nevertheless be regarded as an alternative to a plain multicast announcement approach and be used as a basis for application of Geminga in non IP-based mobile ad hoc networks.

3.3.6 Summary

Most of the approaches we have discussed are not applicable in unreliable communication media or ad hoc networks. Active service discovery is the predominant discovery method. This scheme involves at least two messages: A query sent by the client and zero or more responses from the respective services. This is clearly more vulnerable to error than a passive scheme where only one message has to be delivered by the client. Table 3.3 summarises the evaluation results.

	Protocol	Discovery	Ad Hoc	Comment
UDDI	SOAP	Active	✗	Directory
Zeroconf	UDP[1]	Active	✓	Discovery, Configuration
SSDP/UPnP	UDP	Hybrid	—[2]	Discovery, Configuration
SLP	UDP/TCP	Active	—[3]	Discovery
Potential Field	—	Hybrid	✓	Routing, service discovery

1 = DNS   2 = Periodical announcements   3 = Deals with message loss

Table 3.3: Properties of service discovery approaches for computer networks

The UDDI is merely a service directory but no service discovery approach. It furthermore depends on SOAP and thus a connection-oriented transport protocol. As a centralised approach, the UDDI disqualifies for application in Spica. Zeroconf is mostly used for automatic network configuration. However, it contains a very promising service discovery approach with a lean and simple design. Because of its unique characteristics, Zeroconf is well suited even for resource-constrained devices. The SSDP is part of the UPnP protocol suite and serves similar purposes. Its service discovery nevertheless exhibits several weaknesses and the complexity reduces applicability even more. The SLP is a well-designed service discovery protocol that considers scalability issues. Even though it deals with message loss, plain SLP is not suited well for application in groups of AMRs. The potential field approach facilitates service discovery for mobile ad hoc networks. It implements a hybrid discovery approach where services emit announcements and clients have to query for services. Announcements are basically used for the routing of client queries. The approach shares many characteristics with Geminga but involves additional overhead for the own routing approach.

Part II

Spica

4 The Spica Development Framework

"Order and simplification are the first steps toward the mastery of a subject – the actual enemy is the unknown."

— Thomas Mann
The Magic Mountain, 1924

So far, no standard strategy or platform is available for the development of autonomous mobile robots. Neither a general agreement on how to develop software for robotic systems nor on a software architecture as such has been achieved, although a number of different approaches were proposed in the past. This is not so much because these proposals lack any important features. The most important reason is that existing development and middleware frameworks build on specific standards for the underlying platform, the programming language, or the component interfaces. For AMRs, however, such a restriction does not seem to be appropriate, as their development is a task with many facets and interdisciplinary requirements. Robots make use of hardware sensors and actuators along with appropriate control components, and typically require some form of information fusion, world modelling as well as *artificially intelligent* planning strategies. Many of these aspects are often addressed through independently developed components, and for each of them a different programming language and underlying implementation framework may be most appropriate. For example, low-level hardware drivers are often implemented in C, a planning framework in Lisp, while Java may be adopted for user interfaces and visualisation tools. Encompassing all the different needs in a standardised development and middleware framework is a nearly impossible task. Thus, proprietary development frameworks and infrastructures are provided that reflect the particular importance of components and functions. In conclusion, heterogeneity issues are often observable even within a single robotic system. They are of course more predominant in heterogeneous groups that accomplish complex tasks.

In the following chapters, we introduce the fundamental parts of the Spica development framework for mobile robots. With Spica, the developer captures component interaction and data management in reusable abstract models. A programming language or platform is thereby not prescribed for the implementation of a specific functional component. It may be rather chosen freely according to the respective requirements of the development task. The resulting set of heterogeneous components is considered as a set of *resource providers* and *consumers*, each of which integrates into the robotic software system in order to fulfil a desired functionality. Spica facilitates the integration of services by supporting the development of an integration infrastructure following the MDSD paradigm. Platform-independent modelling and the generation of stub modules thereby connect components with the communication infrastructure, simplifying interoperation. In order to boost interoperability even further and to deal with heterogeneity issues arising from cooperation of independently developed robots, an ontology may serve as a common modelling basis.

Spica not only comprises modelling and code generation approaches but also a library containing commonly used functionality that assists in code template creation. In this chapter, we introduce the development methodology, followed by the design decisions for the Spica development approach. Chapter 5 outlines the *Spica Modelling Language* (SpicaML), Chapter 6 the associated model transformation process. The latter one is responsible for reading and interpreting a model and finally generating a suitable implementation for the desired target platform. Chapter 7 presents the *Spica Decentralised Monitoring Facility* (SpicaDMF) for Spica-based communication infrastructures. To ease configuration of complex communication infrastructures, we furthermore introduce the distributed resource discovery approach Geminga thereafter in Chapter 8. It is tailored to the needs of mobile robots and a Spica-based communication infrastructure in particular.

4.1 Development Methodology

Spica allows users to concentrate on the specification of the desired functionality rather than bothering with implementation requirements of a specific target platform. It has been designed with the intention to exhibit characteristics of well-known programming paradigms. This is why Spica provides the abstract modelling language introduced in Chapter 5. It resembles a programming language but exhibits highly domain-specific statements and modelling elements. A user is thereby given the power to express a desired scenario in a platform-independent fashion. As Spica provides development support for communication infrastructures of autonomous mobile robots, this modelling support has been intentionally restricted to exactly this domain. Hence, a user only needs to model data structures and paths for data flow, all the rest is covered by Spica. The model may nevertheless be extended to support a broader domain if required.

Once a model is completely specified, it has to be translated into source code for a concrete platform. This transformation is carried out by the model transformation tool Aastra as introduced in Chapter 6. It parses, verifies, and transforms the model before a template engine creates the platform-dependent code.

After these two tasks are finished, a new, ready-to-use implementation of the modelled scenario is available in form of a library. The source code does not have to be modified or adapted; it can rather be used out of the box and integrated into the desired application. By realising an event-oriented architecture with message-oriented communication, a Spica-based infrastructure provides suitable means for sending and receiving messages. Message reception here relies on callback methods where incoming message processing is serialised. This prevents race conditions on behalf of the Spica-based infrastructure.

Once all required functionality has been integrated, an instance of the Spica resource discovery service Geminga must be running. It is responsible for the negotiation of the data flow within the communication infrastructure. Details are introduced in Chapter 8. The required components of the application may now be started. They configure themselves automatically without the need of a manual setup.

An overview of the Spica development framework is shown in Figure 4.1. It covers the modelling as well as the model transformation and code generation steps. All the components

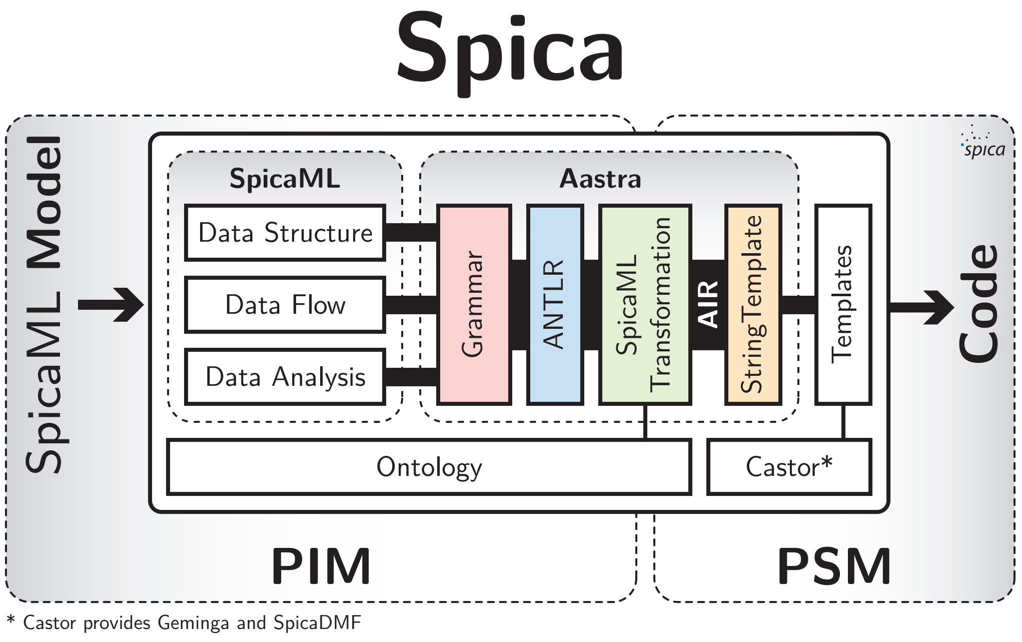


Figure 4.1: The Spica development framework mapped onto the MDSD classification. It visualises the abstract concept in SpicaML and the model transformation process carried out by Aastra.

that contribute to the Spica development framework are presented according to their assigned task. Components that have not been mentioned yet will be introduced in the course of this chapter.

4.2 Spica Design Decisions

As Spica realises an MDSD approach, it provides an abstract, domain-specific modelling language and associated model transformation tools. Keeping in mind the requirements worked out in Section 1.2, we decided to go with a custom DSL in favour of existing UML profiles or XML applications. We justify this decision as follows.

Domain-Specific Language UML as such provides broad modelling support and a large number of specialised profiles. It also allows to specify custom profiles tailored to a given application domain. However, a graphical representation exhibits weaknesses when dealing with large and complex scenarios where the visualisation may get confusing or unclear. Most important, it depends on graphical modelling tools that may not always be available.

A textual representation is indispensable if models must be editable without a dedicated editor. The most common output generated from a UML model is some sort of XML application such as XMI (*XML Metadata Interchange*). However, if generated automatically by an application, XML is typically difficult to understand and complex to edit. It furthermore obeys a limited syntax that is not always adequate. It can definitely show its strengths when storing data in a structures fashion.

Domain-specific languages, in turn, combine advantages of both aforementioned approaches. They exclusively aim at a concrete application domain, allowing them to address specific needs and provide a compact syntax. Depending on the level of abstraction, specialised concepts and statements may be introduced that simplify modelling and reduce specification effort. DSLs further retain the possibility attach a graphical representation – a specific UML profile, for instance.

Model Orientation Once appropriate specification means are available, it is important to choose a suitable development approach that makes use of provided capabilities. The MDSD approach arose in the last years and strongly influenced software development. Modelling shows its strengths in particular for platform-independent development. In addition, it addresses complex scenarios that need to be handled in a more abstract fashion. Not only computer science but also other disciplines make extensive use of it as well. MDSD takes the generic concept of modelling over to high-level development of software systems.

MDSD is one possible solution for heterogeneous software development. A middleware platform is another. Abstraction layers here typically enforce the software structure. Ready-to-use implementations addressing the most urgent problems of the application domain thereby support development. This approach is much closer to the platform, addressing platform-specific issues more precisely.

Middleware A drawback of middleware architectures is the fact that they must be implemented for every involved programming language. Communication middleware solutions such as Ice [60], for example, provide implementations or bindings for a huge range of languages. With CORBA [98], however, it is more difficult to find viable implementation of certain services for multiple languages. This might imply that specific third-party components have to be integrated additionally. An RCA middleware has to address these shortcomings if it aims at cross-platform applicability. There is furthermore no generic model that specifies the layout and the respective functionality of a middleware. Even though UML and assorted design approaches deliver modelling capabilities, they nevertheless do not address the realisation of the functional characteristics for different platforms. This is why it still depends on the developer and its adherence to the specification if the middleware functionality is implemented as desired.

The Virtual Machine (VM) approach follows yet another development strategy. Java or the Microsoft .NET framework are well-known examples that implement a VM. This approach shifts the platform dependency issue to the virtual machine and the supplier that needs to deal with the platform-dependent parts. A VM provides a consistent platform layout and cares about platform-specific issues. The downside of this approach is lowered execution efficiency and the need for a VM on every target platform. It thus renders this approach unsuitable for certain domains where reactivity and efficiency are most important.

As a combination of MDSD and middleware approaches seems to provide promising results, we decided in favour of this combination. The next chapter introduces the modelling language as a basis for the Spica development framework.

5 Spica Modelling Language

"The problems of language here are really serious. We wish to speak in some way about the structure of the atoms. But we cannot speak about atoms in ordinary language."

— Werner Karl Heisenberg
Physics and Philosophy: The Revolution in Modern Science, 1958

In order to create a language that covers the most important characteristics of communication infrastructures for mobile robots, an appropriate set of modelling elements must be provided. The primary goal for designing such a language should therefore be a convenient syntax with clear semantics. A developer must be able to overlook the development task in even complex scenarios. Hence, an appropriate level of abstraction is needed so that all relevant characteristics can be represented by a very compact notation. These requirements are somewhat conflicting but highlight the need for a convenient textual specification by reducing language complexity to a minimum.

Below, we outline the modelling elements considered most important in this context. A language that implements these elements is presumably applicable only for the very specific application domain but for no other. The elements have been chosen based on the communication behaviour of mobile robots as outlined in Section 1.1.3 and lessons learnt from several years of mobile robot development.

- **Data Structures**. Data structures are a fundamental element to overcome heterogeneity issues. Similar to IDL approaches and ASN.1 [73], Spica must be able to provide platform-independent structure modelling and code generation capabilities.
- **Modules**. Modular architectures consist of a collection of self-contained modules or services that interact with each other. A fundamental modelling element has thus to be provided for modelling modules that are the starting point for data transmission and data management.
- **Data Flow**. Interaction between modules further depends on communication channels. The modelling language must cover the specification of data flow and appropriate solutions for data exchange. Elaborate protocol interactions should be avoided if possible, keeping in mind the unreliability of the network media. It is possible, however, that some situations depend on specific service guarantees. This is why it must be possible to express the demand for message fragmentation and transmission reliability, for example.

- **Data Management**. An important building block for message-oriented communication architectures such as promoted by the Spica development framework is buffering of incoming and outgoing data. Message buffers typically facilitate synchronisation of asynchronous communication. Spica has to provide message buffers that are customisable for different interaction schemes and application domains.

Network messages introduce yet another aspect that needs to be addressed: How to encode data and which information must be included? Binary and textual encoding approaches represent two different solutions as answers to the first question. Regarding the second question, again two alternatives are available, namely *including* or *excluding* type information.

Textual encoding is possible but introduces much more overhead than binary encoding. Data type identifiers encoded along with the values of the data structure fields introduce even more overhead. In case all participating parties are aware of the data structure layout, this information may safely be omitted. Another issue introduced with binary encoding is the integer value *endianness*, i.e. the respective the byte order for numerical values. Again, two approaches are possible here: The *Sender Makes it Right* (SMR) approach forces the sender to convert all values to a predefined endianness. For *Receiver Makes it Right* (RMR), in turn, the receiver is responsible for data conversion. The latter approach is less computationally expensive as data conversion is not required in the majority of cases.

We will now introduce the modelling capabilities of SpicaML for network messages and data structures. Afterwards, we cover the data flow between independent modules. SpicaML further addresses data processing: Specialised language elements for implementing message filters for processing or conversion of messages are provided. This language support is provided for future feature extension and is thus only outlined.

5.1 SpicaML Syntax

The SpicaML syntax builds on the concepts used by established programming languages. Each statement in a SpicaML model is terminated by a semicolon and may so span several lines. *Block statements* are framed by curly braces and therefore self-terminated, dispensing with the need a dedicated termination symbol. Space characters – that is, spaces and tabulators – are ignored except in strings that are enclosed in quotes. In contrast to imperative languages, SpicaML is not sensitive to the relative position of statement. The layout of a SpicaML specification consists of a set of *Block Type Definitions* (BTD) that are composed of a *block identifier*, a *block name*, and optionally a set of *annotations*. Curly braces frame the *body* of a block.

We will employ ANTLR v3 [103] grammar definitions for all syntax definitions, starting with the generic form of a BTD below. The structure of the annotation symbol `annotation` is introduced later.

```
block : type identifier annotation? '{' body '}' ;
```

The `type` symbol assigns each block into a specific class: Data structures may be classified as `'container'`, `'header'`, `'message'`, or `'enum'` whereas all modules are of type `'module'`. The block type `'filter'` is used by the conceptual filter definitions. We will introduce the meaning of these types later on.

A *Block Type Instance* (BTI) describes a concrete instance of a block with a specific type. BTIs are the most fundamental modelling element in SpicaML as they provide the high-level view on the scenario that is modelled. Each block comprises type-specific modelling elements. SpicaML does not support the concept of scopes, as blocks may not be nested.

The identifier of a block is specified by the `identifier` symbol. It is a textual representative that uniquely references a BTI. Textual identifiers always start with a letter or an underscore, followed by any combination of letters and digits like function or variable names in C:

```
ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
```

5.1.1 Annotations

Annotations in SpicaML build the foundation for *semantic annotation* and *generic parameterisation*. As a matter of principle, data communicated in a network are typed but typically do not have a meaning attached. This is true for all kinds of interactions, ranging from text streams to RPC calls. Annotations in Spica allow the developer to provide semantic knowledge for specific modelling elements. This knowledge may be used for automatic data interpretation and conversion. Influencing the semantic meaning is nevertheless only one application of annotations in SpicaML. The much more important function of annotations is generic parameterisation. Using a simple key value interface, the configuration of every language element in SpicaML may be modified using annotations. With this approach, we borrow a bit of XML's genericness regarding attribute assignment for language elements.

Every BTI accepts annotations that allow redefining its semantic meaning or changing certain parameters. Square brackets thereby frame an annotation consisting of a set of key value pairs. Semicolons delimit several pairs. An equal sign delimits keys and values. A key may hold several values delimited by colons. Empty values are permissible as well. The grammar below visualises the generic form of an annotation:

```
annotation : '[' kv ( ';' kv )* ']' ;
```

The structure of a key value pair follows the grammar rule given below. The definition of the `key` symbol is equal to `ID` and `value` may additionally contain strings or numbers:

```
kv : key ( '=' value ( ',' value )* )? ;
```

Being a rather generic concept, annotations are used throughout SpicaML for parameterisation. Annotations in particular define or redefine semantics of statements where applicable. Data structures and their content are an example in this case. Semantic annotations are used to embed additional information taken on by the subsequent processing steps. Section 5.3 introduces further details in this topic.

5.1.2 Semantics

As already outlined in the section above, a semantic identity may be assigned to specific elements of SpicaML. The semantic model of SpicaML follows the guideline given in Reichle et al. [113]. It provides two types of classes, namely *concepts* and *representations*. A concept defines the conceptual appearance of an element while a representation defines its concrete representation or realisation. Thus, several representations may be defined for one concept. An ontology hereby represents knowledge. The semantic model therefore extends the basic notion of a data type – this includes primitive types but also hierarchical type structures defined in programming languages, for example – to a more generic form. It serves classification and inference purposes. The experimental data analysis language of SpicaML outlined in Section 5.5 depends on this information.

In order to reference a semantic concept or representation, SpicaML employs *Uniform Resource Names* (URN) as persistent and location-independent resource identifiers. The URN namespace for SpicaML is defined as `urn:spica` but it may be abbreviated by a number sign (`#`) in SpicaML. A semantic annotation pair consisting of the keys `concept` and `rep` is assigned to every BTI automatically. The key `concept` references the concept in the ontology and `rep` its representation. This is done automatically because the definition of a BTI implicitly establishes a new concept and a new representation.

The keys `refconcept` and `refrep` may be used to reference to a concept or a representation in the ontology. The `concept` and `ref` keys are thereby not overwritten but still act as unique identifiers. The `refconcept` and `refrep` are rather used as additional identifiers. This technique is only required for data structures and thus introduced in Section 5.3.

It has to be noted here that SpicaML does not depend on an ontology. If no ontology is available, the semantic specification is used as a unique key. Further interpretation and processing of data representations is then obviously not possible.

SpicaML defines a structure for automatically generated semantic concept and representation identifiers. A concept here obeys the structure given by `concept_urn` shown below. The `type` symbol is replaced by the name of the SpicaML statement type and `identifier` by the respective identifier of the statement.

```
concept_urn : 'urn:spica:' type ':concept:' identifier ;
```

The structure of a representation URN is the same except for another namespace component:

```
rep_urn : 'urn:spica:' type ':rep:' identifier ;
```

A message BTI with the identifier `Example`, for instance, would therefore translate to the concept `urn:spica:message:concept:Example` and the representation `urn:spica:message:rep:Example`.

Example Let us illustrate the use of the semantic model in SpicaML. We will assume a simple two-dimensional point structure (`Point`) that contains two variables `x` and `y` of type double. This point structure may represent virtually any physical object in a plane. The following listing visualises the example using a generic BTI syntax as introduced above.

```
struct Point {
  double x;
  double y;
}
```

Let us now assume that two robots want to share the positions of a ball and an obstacle, both represented as two-dimensional coordinates. Each robot must either be aware of the structure layout and the meaning of the fields or be able to derive the respective knowledge from the structure somehow. In the first case, each data structure has a specific type. Both robots have to be aware of either their layout or the meaning of their fields.

Given the two platforms differ fundamentally and no robot is aware of its opponent, a mere data type does not provide the required knowledge. In this case, they both need to find another way to establish an understanding of the data and how to process them. The semantic description of SpicaML may be consulted for this purpose. It is possible to assign an annotation to the structure itself:

```
struct Point [refconcept=urn:spica:Point2D,refrep=urn:spica:Opponent] {
```

It is also possible to characterise the structure fields further in more detail:

```
  double x [refconcept=urn:spica:XCoord,refrep=urn:spica:mm];
```

If both robots build on the same ontology, they are able to derive the meaning of the structure and its fields. They are furthermore able to carry out conversions on the data, given the semantic description is accurate enough. Section 5.5 outlines the respective language support in SpicaML that is currently under development.

5.1.3 Name Variants

Identifiers of BTIs have to be unique in SpicaML. In order to support the definition of multiple BTIs with the same name but different representations, SpicaML introduces the concept of *Name Variants* (NV). Each NV actually defines a logical namespace in which the respective BTI is defined. It is also possible to create one and the same BTI with different NVs. This technique is a conceptual extension of SpicaML and does not have direct influence on robot communication. The logical separation of names nevertheless facilitates cooperation between robots that rely on Spica but realise different SpicaML models that need to be merged.

An NV is created by attaching one or more arbitrary textual *Variant Identifiers* (VI) to the BTI, framed by angle brackets. Colons delimit multiple VIs. The syntax is given by:

```
nv : identifier '<' ID ( ',' ID )* '>' ;
```

The symbol `identifier` here represents the identifier of the BTI; `ID` refers to the textual VI. A default VI has to be specified at code generation time. This resembles the definition of a default namespace. All the BTIs that are contained in the given namespace may therefore be accessed directly without specifying the NV.

An NV influences the internal appearance of a BTI identifier. It is generated as follows: The plain BTI identifier is used if its VI equals the default VI specified at code generation time. Otherwise, the respective VI is simply attached to the end of the plain BTI identifier. With this approach, different models may be combined and maintained in parallel without influencing each other.

The adoption of NVs also influences the concept and representation identifiers. Their general structure is adapted for use with NVs as shown below. The main difference is the URN component for the VI. It spans a new namespace in order to isolate different models logically. Hence, the concept obeys the following structure.

```
concept_urn : 'urn:spica:' vi ':' type ':concept:' identifier ;
```

The structure of a representation URN then reads as follows:

```
rep_urn : 'urn:spica:' vi ':' type ':rep:' identifier ;
```

Below, we introduce the details of SpicaML. Starting with the general operations for a SpicaML specification, we describe the concepts for creating data structures and specifying data flow.

5.2 Basic Concepts of SpicaML

SpicaML offers some global statements that allow developers to customise a model specification. We will introduce and discuss these statements below before the fundamental modelling capabilities are introduces in Section 5.3 and Section 5.4.

5.2.1 URN prefix substitutes

To simplify writing down URNs for semantic annotations, an arbitrary number of *prefix substitutes* may be defined using the `urnpfx` statement. It defines an arbitrary string constant to be used as a substitute for a valid URN prefix in the model. These substitutes are mapped back to the URN prefix during model processing. The syntax of the statement obeys the following structure:

```
prefix : 'urnpfx' abbreviation urn_prefix ';' ;
```

The `abbreviation` symbol represents the shortcut by which the `urn_prefix` symbol is represented. This statement allows overwriting the default prefix `#`.

5.2.2 Namespace

Class or function names of generated source code may conflict with names of other classes of functions in the target system. A namespace may therefore be defined for each SpicaML model that prevents such a naming clash. The namespace value will be considered during code generation only. For C++ and C# it may be directly mapped onto the respective namespace construct. In Java, a package definition has to be created. In functional programming languages such as C where no such measure is available, the SpicaML namespace value may influence the procedure or function names. Namespace values are structured values consisting of strings delimited by dots. The SpicaML namespace statement obeys the following structure:

```
namespace : 'namespace' ( component ( '.' component )* ) ';' ;
```

The `component` symbol here relates to the definition of `ID`. If no namespace or an empty one is defined, no namespace will be considered during code generation.

5.2.3 Identifier Generation

Spica depends on unique numeric identifiers for data structures and modules. The Aastra model transformation tool generates them automatically from their semantic identifiers. It therefore requires an appropriate *Hash Function* that handles the mapping from the semantic concept and representation strings onto a unique numeric identifier. It must be unique at least within the model. The term *appropriate* reflects the need for a *collision resistant* scheme that produces 32bit integers.

SpicaML provides an optional configuration flag that allows developers to change the hash algorithm used by Aastra for the current model. Each statement thereby obeys the syntax below. The `ID` symbol represents the name of the hash algorithms.

```
hash : 'hash' ID ';' ;
```

Algorithms Hashing algorithms for this purpose do not have to exhibit any special properties except providing good uniformity, i.e. a low collision potential. Review and evaluation of current hashing approaches [59] have shown that several non-cryptographic hash functions are suited well for this purpose. We finally decided to implement two approaches, namely *Jenkins' hash function* [75] (`'Jenkins96'`) and the *Modified FNV* [93] (`'FNV'`). Especially Jenkins's algorithm performs well in speed and uniformity. Further hashing algorithms may be added, though.

Internal Operation In order to generate unique identifiers from a semantic annotation, the concept and representation values are simply concatenated with an ampersand in between. Converted into an ASCII-encoded array, the bytes are passed to the hash function. The resulting 32bit integer value represents the identifier. More formally it reads:

$$id = hash\left(concept \parallel representation\right)$$

where $\parallel$ represents the concatenation with an inside ampersand.

As uniqueness of identifiers is mandatory, it must be guaranteed that no collisions occur. This is simple if all generated identifiers are available: Every identifier is compared to all the previously generated ones. The generation process is cancelled once a collision occurs. In this case, another hash function should be considered for integration.

This approach is feasible because the model is used by every other system and the identifiers are unique to this specific model. Identifiers are furthermore generated in a deterministic fashion, so there is no stochastic randomness involved.

5.3 Specification of Messages

Modelling of data structures in a platform-independent fashion is addresses by many different approaches. Since it is required for everything related to network communication, coverage is broad and well understood. First, we will outline the most important approaches related to data structure specification. Afterwards, the SpicaML specification approach is presented.

The *Abstract Syntax Notation One* (ASN.1) [73] is a widely accepted language for specifying the layout of data structures. It provides a set of formal rules, independent of machine-specific characteristics or encoding techniques. Encoding and decoding is covered by so-called *encoding rules* that provide serialisation and deserialisation functionality for structured data. ASN.1 is commonly used for the specification of network protocol messages, but also for other purposes where data structuring is required. It provides a basic set of primitive data types and measures for specifying structured data types.

Interface Definition Languages (IDL) are used in distributed middleware architectures where remotely accessible interfaces have to be specified in a consistent fashion. CORBA [98] uses an IDL [96] for the portable description of remote object interfaces, for instance. Besides providing specification means for modelling interfaces, an IDL typically allows specifying data structures as well. Structure definitions are similar to ASN.1 but address characteristics of the application domain and exhibit the IDL language layout.

ASN.1 and IDL are quite different approaches but exhibit similarities in the specification of data structures. SpicaML combines features of both and adds some more extensions. In order to ease specification and to retain future extensibility, SpicaML maintains a clean language syntax. It exclusively covers the required capabilities for a very specific application domain.

The *Protocol Header Definition Language* (PHDL) [6, Section 7.4.2] is another specification language for data structures. It bears resemblance to ASN.1 and IDL in some aspects as it defines an abstract and platform-independent specification language as well. PHDL is a by-product of the *Adaptive Secure Communication Infrastructure* (ASCi) [6, Chapter 7] framework and was the starting point for the development of SpicaML.

Java or other current object-oriented programming languages provide built-in support for object serialisation. They address the persistent storage of language-level objects. More specifically, mostly object member variables are considered. It is thereby not possible to serialise or share functionality, except for plain source code or languages that execute in a runtime environment. This is why not objects but object data are serialised. The language handles primitive types automatically in most cases. Serialisation of complex types, however, remains a manual task. Languages that do not provide serialisation support may adopt a serialisation library.

Even though all approaches aim at data structure serialisation, they are mostly incompatible. This means that an agreement must be reached concerning the programming language or the serialisation toolkit. All approaches have in common that they basically process data structures, constructs supported by almost all languages. Object serialisation so cares about data serialisation but provides appropriate object-oriented frontends for developers.

Spica follows exactly this approach. SpicaML provides platform-independent specification means for data structures. As usual for interface descriptions, no functionality may be specified. Aastra cares about the transformation from the abstract specification to concrete implementation. Language-level objects – or data structures for imperative programming languages, for example – then provide a convenient interface for the developer. As Spica aims at message orientation, no such concept as a remote procedure call is supported. Instead, messages are exchanged between participants – represented as objects or data structures and implemented in different languages.

5.3.1 Structure Characteristics

The specification capabilities for data structures in SpicaML rely on annotated BTIs containing variable declarations. The four different BTI types offered by SpicaML represent the most important aspects of network messages: *Containers* define ordinary data structures that may be serialised. *Headers* represent management information required for message handling. *Messages* are the elements that are exchanged between modules. They consist of a message header and one or more containers attached as payload. A similar concept to headers is wrapping messages into *envelopes*. Here, not only a header but also a footer is provided for each message. As SpicaML does not yet support this scheme, it is left for future extension. The fourth type of block specification is an *enumeration* where a collection of named constants of a given type with specific values may be defined.

Within each data structure BTI, zero or more variables are declared. The declaration itself consists of a type and a variable name. SpicaML thereby follows the principles of imperative programming languages. Each line is terminated by a semicolon. Variables are strongly typed with one exception: Fields in a header structure typically require some more advanced data structures that have to be generated automatically during code generation. For this reason, we provide the generic type `auto` that instructs the code generator to inspect the variable's semantic concept. If no concept is provided, the model is considered invalid and the code generation is aborted. Variables in general do not exhibit any semantic properties by default except their type. A semantic meaning is assigned using the `refconcept` and `refrep` keys.

The relative position of a variable within the data structure BTI defines its position in the resulting implementation and finally in the encoded data. This knowledge is used by participating systems for encoding and decoding SpicaML-based data structures. SpicaML defines a set of primitive variable types for this purpose. They are shown in Table 5.1 including a verbose description of their representation. It has to be noted here that the encoding of SpicaML-based data structures depends on an encoding rule and is not bound to the type as such. The encoding rule provided with Spica is outlined below; others may be used as well.

Annotations of a data structure BTI not only specify the structure's semantic meaning. With their help, it rather is possible to configure the comparison behaviour of the data structure. The annotation key `compare` is used for this purpose. It allows specifying the order in which fields of the data structure are compared to fields in another structure of the same type. The compare key annotation obeys the structure shown below. The order of the fields listed defines the order in which comparison is executed.

```
compare : 'compare' '=' field ( ',' field )* ;
```

Primitive	Description
auto	Type is derived from the semantic concept
bool	Boolean value (1bit)
uint8/16/32/64	Unsigned integer (8bit, 16bit, 32bit, or 64bit)
int8/16/32/64	Signed integer (8bit, 16bit, 32bit, or 64bit)
float	Single precision floating point number (32bit)
double	Double precision floating point number (64bit)
string	Length-encoded string with arbitrary length
address	Arbitrary network address

Table 5.1: List of SpicaML primitive types and according value spaces

5.3.2 Identification of Data Structures

Handling and processing of messages depends on additional information. This is why some supplementary fields are provided by the topmost header structure automatically: When trying to decode a message, the first three bytes of the received data are compared to the *SpicaML magic number*.[1] If the comparison failed, message decoding is aborted. Otherwise an 8bit Boolean field is extracted which indicates the endianness of the encoded data. It is required for the RMR approach where the receiver may have to adjust the values' endianness. The 32bit numeric type identifier that uniquely identifies the message is finally extracted. This value enables the decoding procedure to commence with the remaining message fields. The baseline for the management overhead of each message is thus 8B. The management structure is outlined in Figure 5.1 (a).

Containers are assigned management information as well, because they may be serialised and deserialised independently. A magic number is omitted for containers. The very first field is an 8bit Boolean value that indicates the endianness of the encoded data. The second field is a 32bit numeric type identifier. The third field finally indicates whether the container is initialised or not. If not, it is assigned a `null` value. Here again an 8bit Boolean field is used. This results in a baseline for the management overhead required for containers of 6B. The structure is shown in Figure 5.1 (b).

5.3.3 Basic Language Syntax

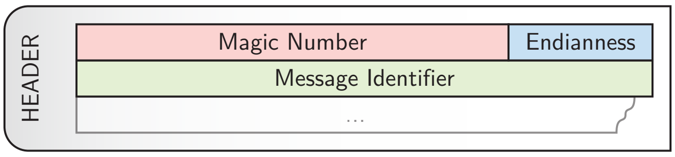
The general structure of a BTI is almost the same for all other data structures: It is composed of the type (`type`), the identifier (`ident`), an optional inheritance specification (`':' parent`), optional annotations (`annotation`), and the body (`body`) of the data structure definition:

```
struct : type ident ( ':' parent )? annotation? '{' body '}' ;
```

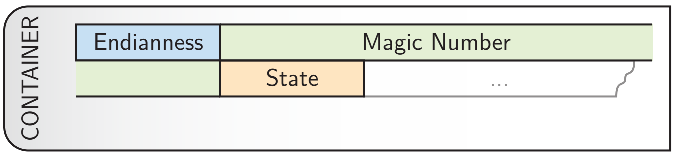
The `type` symbol references the valid types for data structures: `'container'`, `'header'`, or `'message'`. The `parent` symbol refers to identifiers of other data structure. A data structure body contains field definitions. It is arranged as follows:

```
body : ( type name ( '=' value )? annotation? ';' )* ;
```

[1] SpicaML-based messages have to use the 24bit value `0xCA0408`.



(a) Header management fields



(b) Container management fields

Figure 5.1: Management overhead for SpicaML-based messages and containers. A header contains a magic number, whereas a container defines a field that indicates its initialisation state.

Messages are handled differently from headers or containers: Only complex data types are allowed in the body of a message, so the grammar rule must not permit primitive type names (`type`) in this case only. Chapter A presents the complete grammar rules for SpicaML data structures. The specification of the `name` symbol again relates to `ID`. A default argument (`value`) for the field is optional.

The definition of enumerations is very similar to that of the other blocks, but the inheritance and annotation specifications as well as the body differ. An enumeration inherits the type of its values from a primitive type. The model allows specifying any available primitive type. Due to interoperability issues with a couple of programming languages, support is currently limited to only integer values. The BTI structure for an enumeration is shown below:

```
enum : 'enum' ident ':' primitive '{' enum_body '}' ;
```

The body of an enumeration lists name value pairs delimited by semicolons. The identifiers (`ident`) may be chosen arbitrarily according to the definition of `ID`. Their values (`value`) must correspond to the type given in the inheritance specification.

```
enum_body : ( ident '=' value ';' )+ ;
```

5.3.4 Encoding Rule Interface

Data structures defined in SpicaML do not provide integrated encoding capabilities. Instead, the *SpicaML Encoding Rule Interface* (SERI) as shown in Figure 5.2 proposes appropriate means for realising encoding and decoding in a modular fashion. The placeholder `<type>` is a template argument that covers all available primitive types. The interface `Serialise` is responsible for serialisation whereas `Deserialise` is responsible for deserialisation. `SERI` combines these interfaces and adds methods required by both of them. We have decided to split the responsibilities in order to realise a separation of duties.

SERI follows a *per-type encoding* approach, without being aware of the global structure to be processed. It is nevertheless possible that a SERI realisation keeps track of the encoded or

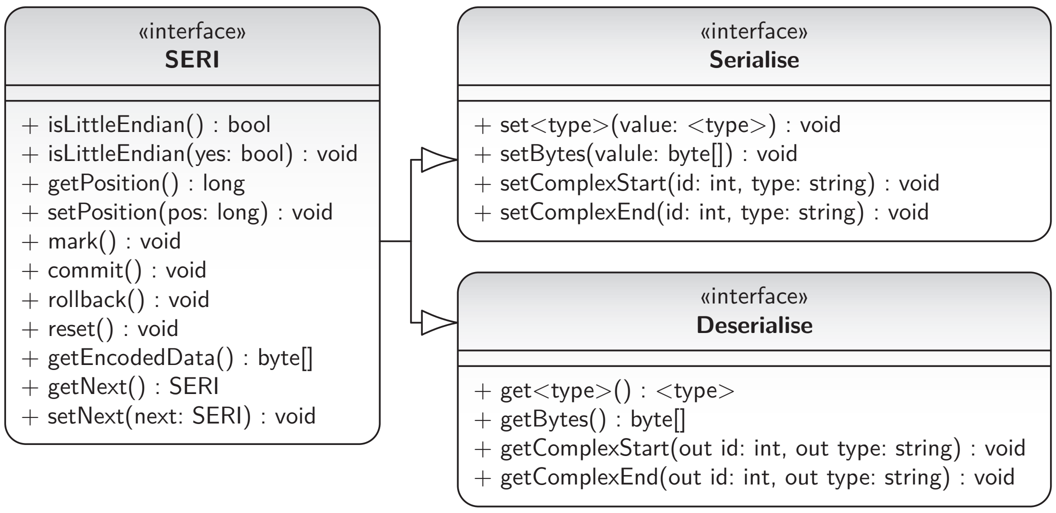

«interface»
SERI
+ isLittleEndian() : bool
+ isLittleEndian(yes: bool) : void
+ getPosition() : long
+ setPosition(pos: long) : void
+ mark() : void
+ commit() : void
+ rollback() : void
+ reset() : void
+ getEncodedData() : byte[]
+ getNext() : SERI
+ setNext(next: SERI) : void
«interface»
Serialise
+ set<type>(value: <type>) : void
+ setBytes(valule: byte[]) : void
+ setComplexStart(id: int, type: string) : void
+ setComplexEnd(id: int, type: string) : void
«interface»
Deserialise
+ get<type>() : <type>
+ getBytes() : byte[]
+ getComplexStart(out id: int, out type: string) : void
+ getComplexEnd(out id: int, out type: string) : void

Figure 5.2: The structure of the SpicaML Encoding Rule Interface

decoded entities, maintaining a history this way. Look-ahead is nevertheless not possible. It would require a more comprehensive approach.

Even though the SERI approach has only limited capabilities regarding intelligent or dynamic encoding and decoding of data structures, it is especially efficient in processing because no dynamic decisions are taking place during runtime. The generic interface allows integrating almost any encoding scheme that targets encoding and decoding of fields with a predefined position within the data structure. This implies, however, that it is not possible to reliably interpret and decode a self-descriptive encoding such as XML, if the content does not obey any fixed order. For the majority of network messages this is not important, so it is not considered being a fundamental drawback for this application.

A data structure is encoded sequentially from the management data down to the last field. SERI provides two methods for each primitive type, one for serialising (`set<type>`) and the other for deserialising (`get<type>`). Arrays and embedded data structures require a more elaborate concept: Start and end markers of the encoded data as well as the type are required for the serialisation process (`setComplexStart` and `setComplexEnd`). Deserialisation needs the type information mostly for consistency and integrity checks (`getComplexStart` and `getComplexEnd`). The fields contained in the embedded data structure are encoded as usual. SERI furthermore allows embedding and extracting unformatted data (`setBytes` and `getBytes`).

The SERI resembles a data stream where data are read or written sequentially. Forward and backward seeking is supported (`setPosition`). A transactions concept simplifies data handling by supporting atomic commit and rollback operations for a series of changes (`mark`, `commit`, `rollback`, and `reset`). It is further possible to divide serialised data into smaller segments. Several instances of a SERI realisation are thereby chained together, maintaining the order of the data (`setNext` and `getNext`). This provides a very efficient approach for realising fragmentation and defragmentation of network messages. Section 9.1.3 presents a concrete SERI implementation.

5.4 Specification of Data Flow

Modelling the data flow between modules or other self-contained entities is addressed by several existing approaches. The most influential solution is the *Data Flow Diagram* (DFD) [49] specification, a graphical representation of the data flow through an information system. The language consists of only four modelling entities: *Processes* identify functional entities, *external entities* are data sources or sinks, *data stores* provide storage for data coming from processes, and *data flows* identify the flow of data between processes, external entities, and data stores. At least one process must be involved in a data flow.

The concept and the general layout of the DFD model have influenced the design of SpicaML and other flow specification languages. *FlowDesigner* [33] is a data flow-oriented development environment that follows a similar approach as DFD, targeting mostly rapid prototyping. Development support is provided through libraries (*toolboxes*) of reusable modules (*nodes*). Nodes are processing entities that accept input data and return processed data to the next node, thus resembling processes in a DFD. FlowDesigner provides a user interface with modelling functionality to create connections between nodes. In some aspects it is similar to *MATLAB/Simulink*[2] and *LabView*,[3] but is hardly a clone of either.

While DFD and FlowDesigner both provide modelling capabilities for data flow between self-contained entities, the configuration of data channels cannot be adapted dynamically during runtime. Spica supports automatic configuration and dynamic reorganisation for communication infrastructure with no additional expense.

The MSRDS introduces data flow modelling between service entities in a way very similar to Spica. Here, services are composed to applications that are more complex by providing message-oriented communication capabilities. Data flow modelling as such is based on programming concepts instead of using a dedicated modelling language. Heterogeneity is addressed by the underlying Microsoft .NET framework that provides a common computing environment for a multitude of programming languages.

Concept and layout of the MSRDS are almost identical to Spica. SpicaML, however, explicitly targets heterogeneity issues between arbitrary platforms without depending on a homogeneous abstraction layer. It thus facilitates the integration of existing solutions even if realised with completely different software concepts. There are furthermore differences in how modules or services are connected and how data are exchanged.

Below, we introduce the data buffering and the data flow-related language syntax of SpicaML. The data flow modelling is designed in such a way that it seamlessly integrates with the message specification and allows developers to create a complete data flow model with strict typing.

5.4.1 Data Management Containers

A *Data Management Container* (DMC) is a highly configurable, typed data buffer. Not only the behaviour of a DMC but also the behaviour of the underlying data structure is configurable. Three basic structures are supported – *Ringbuffer*, *Queue*, and *Priority Queue* – but more may

[2]`http://www.mathworks.com/products/simulink/` (accessed 2008-08-23).
[3]`http://www.ni.com/labview/` (accessed 2008-08-23).

be added with low effort. In order to be applicable within SpicaML, each data structure has to support the following configuration options that further characterise its data handling:

- **Structure Insertion Policy (SIP)**. Each data structure has a default strategy how data insertion is handled. It is nevertheless possible to change this default behaviour in favour of another. Valid insertion policies of a DMC are *FIFO* (`fifo`) and *LIFO* (`lifo`). All provided structures default to FIFO. Because this semantic is not directly applicable for priority queues, FIFO relates to *largest-in-first-out* and LIFO to *least-in-first-out* in this case.
- **Structure Duplicate Policy (SDP)**. Duplicate handling in structures depends on the application domain. SpicaML provides three different approaches for this purpose: *Allow duplicates* (`allow`), *deny duplicates* (`deny`), and *overwrite duplicates* (`overwrite`). When allowing duplicates, a new item is inserted into the list with no further checks. If duplicates are denied, a new item is discarded if the structure already contains an item that is equal to the new one. If duplicates should be overwritten, the available item is first removed from the list before the new one is inserted. The default mode for all structures is to allow duplicates.
- **Structure Retrieval Policy (SRP)**. Another issue arises with the retrieval of entries from the list. Two possible actions are supported by SpicaML: *Remove* (`remove`) and *keep* (`keep`). In case of remove, a retrieval of an item triggers its removal from the structure. Otherwise, the item is left in the structure. The default policy for all lists is to remove on retrieval.
- **Size**. The size defines the maximal capacity of the data structure. The size of a ringbuffer is fixed. For queues, the size annotation defines the maximal size to which the structure may grow. If no size is defined, all data structures are assumed to provide space for 10 elements by default.

Especially data handling for a Priority Queue depends on the comparison of data structures as introduced in Section 5.3.1.

It is, however, not only important how to react on insertion or removal of items. It is furthermore important how synchronisation is accomplished and in which way clients are informed of new entries. Several options are provided that allow parameterise the DMC accordingly.

- **Synchronisation Policy (SYNP)**. Messages arrive asynchronously in a Spica communication infrastructure. It is thereby especially important to synchronise them to the internal processing frequency of the respective recipient. Three possible configurations are provided by SpicaML for this purpose: *No synchronisation* (`none`), *soft synchronisation* (`soft`), *hard synchronisation* (`hard`). In case of no synchronisation, incoming messages are signalled directly with no delay. Soft synchronisation requires the interval between two successive messages to be equal to or greater than a given time value. Messages that violate this condition are discarded. For hard synchronisation, a dedicated timer is started that signals the most current message at discrete intervals. The default behaviour of DMCs is to do without synchronisation.
- **Synchronisation Interval (SYNI)**. For soft and hard synchronisation, a synchronisation interval must be specified. This interval is required for soft synchronisation to determine which messages have to be discarded and which are signalled. For hard

synchronisation, this value is used to initialise the timer. An arbitrary time value may be specified here.

- **Time-To-Live (TTL)**. Each DMC internally records insertion timestamps. The time-to-live parameter specifies the validity period of an entry. The outdated entries are purged from the structure upon the next retrieval operation. An arbitrary time value may be specified here as well.
- **Invocation Policy (INVP)**. The invocation policy defines how new data are signalled. It distinguishes between polling and pushing from which three different modes are derived: *No invocation* (`none`), *direct invocation* (`direct`), and *threaded invocation* (`thread`). In case of no invocation, the DMC is configured to operate in polling mode. Clients are not informed of the arrival of new messages; they rather have to poll independently. In direct invocation mode, each new message is signalled synchronously, depending on the synchronisation interval. The DMC thus blocks until the handler routine of the client returns. This mode is suited for handlers that are not reentrant. Threaded invocation takes a thread from a thread pool for signalling new messages. Handlers must be reentrant for this invocation type. The default mode for a DMC is direct invocation if a respective callback handler is registered.

An ordinary DMC further stores the remote identification, that is, the address and hostname of the sender along with the message. The remote identification, however, is not taken into account for the message order within a DMC. SpicaML therefore supports so-called *DMC Arrays* that realise dynamic lists of DMCs. Each DMC in a list is then responsible for only one dedicated sender; messages from different senders are so kept separated.

5.4.2 Basic Language Syntax

Data flow modelling introduces new modelling entities for SpicaML. The most fundamental building block in a data flow model is the concept of a module. Modules relate to processes in a DFD model. They offer or request messages and act as platform-independent placeholders for functional entities, either existing or newly developed components that provide a specific service. A SpicaML-based module specification follows the specification of a BTI construct as shown below:

```
module : 'module' ident annotation? '{' module_body '}' ;
```

Capabilities of a module are configured through annotations (`annotation`). Only two annotation types are supported so far that allow defining the monitoring capabilities of a module. Both annotations are supposed to be flags only; they thus do not require values.

- **monitor**. A module flagged with the `monitor` annotation is capable of receiving and interpreting monitoring information issued by other modules. By implementing a subscription mechanism, it may declare its interest in specific sources or types of information. Hence, it obtains a filtered view on the available monitoring data.
- **monitorable**. The `monitorable` annotation indicates whether a module may provide monitoring information. It is thereby provided with measures that facilitate data distribution under a certain identifier.

Both flags only trigger the integration of monitoring logic into the generated Spica stub. Monitoring itself builds on a subscription mechanism on the receiver's side. Hence, monitors are able to receive monitoring information selectively. The monitoring capability in Spica is supported by a framework that aims at resource-efficiency with respect to communication bandwidth and processing power consumption. It is introduced in Chapter 7.

The body (`module_body`) of a module specification defines which messages it is about to handle and how the respective communication channels have to be established. It further specifies which data management strategy is used.

```
module_body : ( offer | request | define )* ;
```

The `offer` and `request` symbols represent the statements for offering and requesting message types. They define the message type and the communication channel mode to use. The `offer` symbol is defined as follows:

```
offer : 'offer' 'message' ident annotation '->' peers scheme dmc+ ';' ;
```

A message offer is introduced with the keywords `'offer'` and `'message'`, followed by the message type identifier (`ident`) and an annotation (`annotation`). This annotation only accepts a flag that defines whether the offer is enabled (`enabled`) or disabled (`disabled`). If disabled, it must be enabled explicitly at a later point in time. It is disabled by default.

Following the annotation, a right arrow (`'->'`) introduces the peer specification (`peer`). It defines to which peers the message may be sent. If no peers are defined, all modules are allowed. Next, the communication scheme (`scheme`) and one or more DMCs (`dmc`) are defined. A semicolon finally closes this statement.

The statement for requesting a message type is quite similar. It only exhibits some minor structural differences that reflect the changed semantics:

```
request : 'request' 'message' ident annotation '<-' peers scheme dmc+ ';' ;
```

A request is introduced by the two keywords `'request'` and `'message'`. Almost any element in the statement remains unchanged except the arrow for the peer specification: It is introduced by a left arrow (`'<-'`), indicating from which modules the message may be received. There is another difference in the capabilities of the DMC: A receiver is capable of disassembling messages. Specific containers may thereby be extracted and insert into dedicated DMCs or DMC Arrays. This way, the interesting contents may be extracted without even touching a message.

Besides the specification of message offers and requests, SpicaML supports the definition of stand-alone DMCs. They may be exported or used internally for further processing. The respective symbol is defined as follows:

```
define : 'define' dmc ';' ;
```

We will now introduce the `peer`, `scheme`, and `dmc` symbols as used in the symbol definitions for message offers and requests.

DMC SpicaML builds on the DMC concept that provides a quite flexible alternative to ordinary data buffers. One or more DMCs or DMC Arrays are typically defined for each message type or as stand-alone instances. The definition below shows how this is done:

```
dmc : 'dmc' structure_type ( '[]' )? ident annotation ;
```

A DMC specification is introduced with the keyword `dmc` followed by the buffer type. Its symbol `structure_type` represents one of the three alternatives `'ringbuffer'`, `'queue'`, or `'priority_queue'`. A DMC Array is defined by appending an array identifier to the data structure, consisting of an opening square bracket directly followed by a closing square bracket. The `ident` symbol defines the DMC name according to the definition of `ID`.

Annotations are used for configuring the container. Additionally, a privacy flag is supported (`noexport`). It indicates whether a DMC is accessible only within the stub or if it is exported to the stub interface.

Transmission Scheme Three different data transmission schemes are supported by SpicaML, each of which has its very own application domain: *One-to-one* (`oto`), *one-to-many streaming* (`otm/stream`), *one-to-many publish/subscribe* (`otm/pubsub`). Section 8.1 introduces the detailed characteristics of these schemes. The `scheme` symbol required for offering and requesting messages is defined as follows:

```
scheme : 'scheme' scheme_type ;
```

The `scheme_type` symbol relates to one of the three alternatives `oto`, `otm/stream`, or `otm/pubsub` that are supported by the Geminga resource discovery. The respective communication channels are established dynamically based on the availability of resources.

Peers The `peer` symbol defines to which other modules an offered message may be delivered or, respectively, from which a requested message may be accepted. The peer specification is defined as shown below:

```
peers : ident annotation? ( ',' ident annotation? )* ;
```

It consists of a collection of module names (`ident`) delimited by colons. An associated optional annotation (`annotation`) configures the scope of the resource discovery: A module may be located only on the local host (`local`), it must not be located on the local host (`!local`), or no restriction is imposed. In the latter case, no annotation is required.

5.5 Specification of Data Filtering

Support for filtering and processing of incoming as well as outgoing messages is still under active research in Spica. In the following, we will present its current state and outline the most promising features.

The *Data Analysis Definition Language* (DADL) [7] is an interface to data filtering and analysis interface capabilities in SpicaML. Because it is currently work in progress and the functionality is still evolving, we will only outline its basic characteristics. DADL integrates directly with the data flow model of SpicaML and depends on the provided message specification capabilities. Facilitating the development of a collaboration infrastructure in terms of cooperative world modelling clearly requires going beyond the mere specification of a data flows: The interpretation of exchanged data and calculations carried out on them are fundamental for the interaction not only with the environment but with other robots as well. This may also include humans or animals, though. Taking into account the environment robots operate in, the diversity of hardware and software, and the multitude of different implementations of behaviour, it turns out that elaborate techniques have to be provided for platform-independent modelling of cooperation.

For the realisation of *cooperative world modelling*, sensor data has to be collected from different locations and combined into a consistent worldview. This mostly requires to take the impreciseness of observations into account: Probabilistic state estimation approaches such as, for instance, *Bayesian Filtering* [5, 46, 127], derivatives like *Kalman Filtering* [76], or *Particle Filtering* [82, 1] are commonly applied here. The *Dempster-Shafer Theory* [36, 122], *Bayesian Inference* [107], or *Fuzzy Sets* [144], in turn, provide approaches that deal with uncertainty. The implementation of these measures is typically non-trivial and time-consuming. We will therefore identify frequently used patterns and provide appropriate *predefined filter* components for DADL.

In order to provide the user with a consistent and familiar interface, we decided to build on a well-known language syntax: *MATLAB* is widely accepted in research and often adopted for rapid prototyping, especially in the area of computer vision but also in robotics. It is a numerical computing environment and programming language, designed to efficiently deal with matrices and operations on them. For arrays or lists, it provides a very compact syntax and elaborate indexing methods. DADL therefore adopts a MATLAB-like syntax that facilitates the handling as well as processing of data and provides fundamental support for data fusion.

The main modelling element in DADL is a filter BTI. Its structure depends on whether a *Spica Predefined Filter Pattern* (SpicaPFP) or a *Spica Custom Filter Pattern* (SpicaCFP) is required. A SpicaPFP relies on existing implementations of filtering approaches such as provided by *Carmen* [88], for instance. A SpicaCFP represents a custom algorithm that is specified in the MATLAB-like DADL syntax. DMCs represent the fundamental storage primitive for all kinds of filters. We have adopted concepts like implicit typing, control statements, basic arithmetic and logical operations, definition of matrices and vectors, and matrix operations from the MATLAB programming language, to mention only some. The filtering interface is subject to ongoing research as well. Chapter A presents the SpicaML grammar with DADL support.

5.6 Structure of a SpicaML Model

The fundamental elements of a SpicaML model have been introduced above. The overall structure, however, has not been defined yet. It is kept simplistic and only consists of the aforementioned language elements, including the global statements as defined in Section 5.2, the structure definitions from Section 5.3, and the module definitions from Section 5.4. They all integrate well into a consistent language layout shown below:

```
model : ( prefix )* namespace? hash? ( struct | enum | module )+ ;
```

A model at first consists of zero or more URN prefix definitions (`prefix`). The default URN prefix `#` is defined implicitly. A namespace (`namespace`) is not required but at most one is allowed. The same is true for the hash algorithm (`hash`). Aastra nevertheless falls back to Jenkins96 [75] as outlined in Section 5.2.3 if none was specified. Even though there is no practical reason for a static order like this, it is retained so all models exhibit roughly the same structure.

No ordering is defined for the data structure (`struct`, `enum`) and data flow (`module`) specifications. At least one element must be defined. Modules depend on messages and they, in turn, depend on headers. A minimal model therefore defines at least one header or a container. It is thus possible to define only data structures or messages but dispense with module definitions. Chapter A lists the SpicaML grammar including all the presented elements as well as a grammar for DADL.

6 Model Transformation

“Die Mathematiker sind eine Art Franzosen: redet man zu ihnen, so übersetzen sie es in ihre Sprache, und dann ist es alsobald ganz etwas anders.”

— Johann Wolfgang von Goethe
Maximen und Reflexionen 1279 in Artemis-Goethe, 1962

We have only discussed the syntax of SpicaML and the capabilities of the monitoring implementation so far. The next step in the Spica development approach is the interpretation and transformation of the models. The Spica model transformation tool Aastra carries out model transformation, dividing the transformation process into three main tasks: *Model parsing*, *model interpretation and completion*, and *code generation*. Figure 6.1 outlines its general workflow. Here, SpicaML is a concrete realisation of an *Abstract Architecture Specification* (AAS), a class of modelling languages Aastra is capable of processing. The *Abstract Intermediate Representation* (AIR) is Aastra’s internal representation of the SpicaML model.

Model parsing is handed over to a parser engine generated by ANTLR v3 [103], a mature and highly renowned parser generator targeting multiple platforms. It introduces a new parsing strategy called $LL(*)$ [103, pp. 262–291]. Compared to traditional $LL(k)$ parsing with a finite look-ahead of k, $LL(*)$ provides dynamic look-ahead and therefore enhances predictive capabilities of the LL decisions. ANTLR v3 furthermore unifies the notions of lexing, parsing, and tree parsing. A lexical grammar is required for recognising tokens. With ANTLR v3, lexers are capable of context-free grammars rather than only regular expressions. Parsers and tree parsers are identical except that a tree parser can modify a tree self-dependently. This simplifies the creation of grammars by depending only on a single, coherent specification.

For parsing SpicaML, we decided in favour of the tree parser that generates an *Abstract Syntax Tree* (AST). The second processing step of Aastra handles the AST representation of the SpicaML model. Aastra is thereby responsible for interpreting, checking, and completing the model. These tasks are carried out all at once. Chapter A in the appendix lists the SpicaML grammar for the sake of completeness.

In the following, we give a detailed insight into the integrated model checking and transformation process. It covers the processing of a SpicaML parse tree, extraction of specific language elements, and the creation of the AIR. Afterwards, we shift the focus to the code generation approach taken by Aastra.

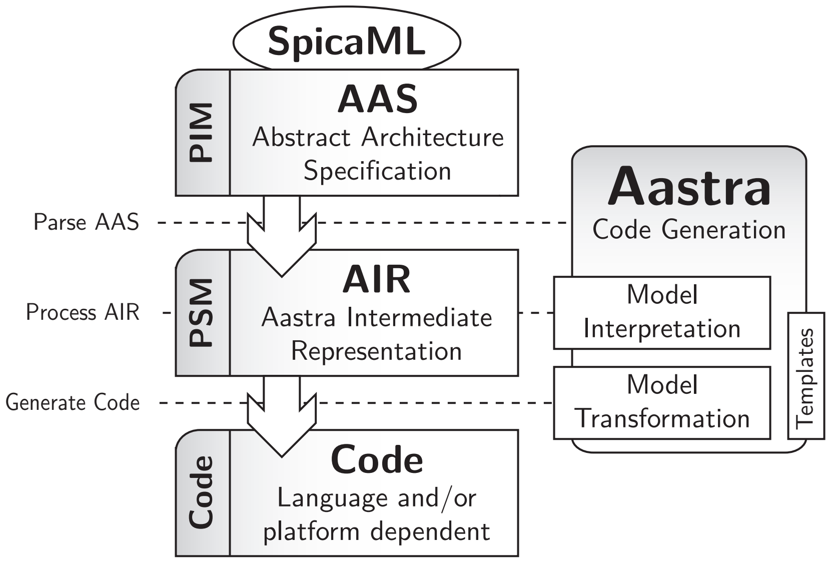


Figure 6.1: The workflow of the Aastra model transformation tool. The responsibilities of Aastra are clearly concentrated on the model transformations and the intermediate representation.

6.1 Model Conversion and Representation

The ANTLR parser returns a reference to an AST tree, an in-memory representation of a SpicaML model. Aastra transforms this tree into a representation that is better suited for the model transformation and code generation processes. For this purpose, we introduce the AIR, another type of tree structure that facilitates dynamic modification and extension of tree nodes. Each node thereby maintains several fields: A *name*, a *root node* reference, a *parent node* reference, a list of *children*, and finally a list of *properties*.

A property in this context constitutes a key-value pair that represents supplementary information for a node. Not only the annotations defined in a SpicaML model but several other characteristics are mapped onto properties as well. Some more properties are generated during the model transformation process, in the course of model completion and specialisation. The properties assigned to an AIR tree node thereby represent its configuration. Code templates make use of these properties during code generation later on.

Specific language-specific features in C#, namely C# Properties and C# Indexers [114] provide a convenient solution for information retrieval and modification at an AIR node. C# Properties thereby facilitate unfiltered access to the lists of an AIR node. Because C# Indexers accept a single argument, they allow implementing filtered access. We use a string in this case for filtering on the element's name. An AIR node maintains two C# Properties and two C# Indexers for the child and property lists: The first of each returns only a single value (`Child`, `Property`), the second returns a list of elements (`Children`, `Properties`).

There is another reason why we decided in favour of the aforementioned approach: The StringTemplate engine we adopt for our code generation approach explicitly supports C# Properties and C# Indexers. It is therefore possible to access the lists of an AIR node directly from within a code template without any further expense. Section 6.6 discusses the details.

Each independent modelling element of SpicaML is then processed on its own. The Aastra model transformation is therefore characterised by six consecutive steps, namely the extraction of *prefix*, *namespace*, *hash*, *enumeration*, *structure*, and *module* definitions. Below, we discuss each of them in detail.

6.2 Extraction of the Hash, Prefixes, and Namespaces

The `'hash'` statement in SpicaML configures the hash algorithm that generates unique identifiers for data structures. As outlined in Section 5.2.3, Spica by default provides two implementations, namely Jenkins96 and FNV. Jenkins96 is the default hash algorithm if none was defined in the model.

Once extracted from the AST, Aastra tries to locate and instantiate the respective implementation. It thereby searches supplementary libraries and its own address space for suitable class definitions. Applicable hash algorithms have to implement a generic interface, exhibiting a single method for generating a 32bit hash value for arbitrary input. If no implementation was found, the model transformation is aborted.

Prefixes are defined using the `'urnpfx'` keyword. All available definitions are read from the AST and inserted into a hash map data structure maintained by Aastra. These prefixes are not required in the AIR but during model transformation. If the default prefix `#` was not redefined in the model, a new entry is created that maps to `urn:spica`.

The namespace definition is finally read from the AST where it is represented as a sequence of string components. After being converted into a string list, it is added to the root node of the AIR as a property named `Namespace`. The code templates require it during code generation.

6.3 Extraction of Enumerations

Enumerations constitute the simplest form of BTI in a SpicaML model. Each one consists of name, an optional NV (Name Variant) as defined in Section 5.1, and a number of element definitions.

Aastra first reads all the respective definitions from the AST. If no NV is defined for the enumeration, a single node named `Enum` is created in the AIR. Otherwise, one such node is created for each VI (Variant Identifier). The enumeration identifiers are generated according to the name expansion rules outlined in Section 5.1. The plain name is used only if the respective VI equals the default VI or no NV was defined. The following properties are then assigned to the newly created nodes:

- **Identifier**. A string that specifies the name of the enumeration. It is either the plain or the expanded type identifier.
- **PrimitiveType**. A string that specifies the name of the primitive type the enumeration is derived from. Table 5.1 lists the type names that are allowed here. Code generation cares about the final mapping to platform-dependent types.
- **Concept**. A URN string that specifies the semantic concept of the enumeration. If the `concept` annotation is missing in the SpicaML model, a new property is generated automatically from the structure identifier and optionally the respective VI. Its value obeys the following structure where `prefix` typically expands to `'urn:spica'`:

```
enum_concept : prefix ':' ( vi ':' )? 'enum:concept:' ident ;
```

- **Rep**. A URN string that specifies the semantic representation of the enumeration. If the `rep` annotation is missing in the SpicaML model, a new property is generated automatically from the structure identifier and optionally the respective VI. Its value obeys the following structure where `prefix` typically expands to `'urn:spica'` again:

```
enum_rep : prefix ':' ( vi ':' )? 'enum:rep:' ident ;
```

- **Default**. A string that specifies the default enumeration element. This is typically the first element defined in the enumeration model.

Now, all element of the enumeration are extracted from the AST. For each element a new child node called `Item` is generated and attached to the enumeration node in the AIR. Each child is assigned two properties: `Identifier` specifies the name of the enumeration element and `Value` defines its constant value. Currently, enumerations only support integer values. This restriction stems from the inconsistent handling of enumerations in different target languages. Integer elements are supported in most cases but other types would have meant further challenges in code generation.

6.4 Extraction of Data Structures

The extraction and interpretation of the remaining structure definitions of a SpicaML model, i.e. headers, containers, and messages, involves substantially more processing effort than it was required for enumerations. In order to simplify the description of the conversion procedure and because all structures are handled roughly in the same way, we will present a more high-level view of the processing.

The transformation process first locates all structure definitions within the AST. It thereby preserves the following order: Headers, containers, and finally messages. One definition is then processed at a time: If a structure inherits from another structure, the whole inheritance hierarchy is traversed until either the topmost structure was reached or some structure in between was processed already. This implies a violation of the processing order. The order was prescribed in this way because it minimises the depth of the hierarchy traversal. The transformation is aborted if the procedure was unable to locate a structure. This recursive approach guarantees that all structures and all inherited structures are processed and made available in the AIR. Cyclic inheritance is furthermore detected and prevented efficiently.

When processing a structure, Aastra extracts the type identifier and expands it considering all available VIs. The procedure is the same as that for enumerations. Each type identifier is then inserted into a list of already processed structures. If it is listed already, the transformation is aborted in order to prevent name clashes. For every type identifier, a new node is created at the root node of the AIR, representing the new structure. This node is assigned the name of the structure type, i.e. either `Header`, `Container`, or `Message`. The following properties are assigned to the node, further characterising its structure:

- **Identifier**. A string specifying the name of the structure. Here, the plain or expanded identifier is used.

- **Inherit**. A reference to the AIR node of the parent. The property is omitted if no inheritance relationship is defined.
- **HierarchyTop**. A reference to the AIR node that is located at the top of the current inheritance hierarchy. This may be a self-reference if the current structure does not inherit from any other structure.
- **Concept**. A URN string that specifies the concept of the current structure. If the `concept` annotation is missing in the model, a new property is generated automatically from the variant, the structure type, and the structure identifier. It obeys the structure introduced in Section 5.1.
- **Refconcept**. A URN string that specifies a concept referenced by the current structure. If the `refconcept` annotation is missing in the model, this property is omitted.
- **Rep**. A URN string that specifies the representation of the current structure. If the `rep` annotation is missing in the model, a new property is generated automatically from the variant, the structure type, and the structure identifier. It obeys the structure introduced in Section 5.1.
- **Refrep**. A URN string that specifies a representation referenced by the current structure. If the `refrep` annotation is missing in the model, this property is omitted.
- **Hash**. A 32bit integer which must be unique for structures within the current SpicaML model. It is generated from the concept and representation of the structure, concatenated by an ampersand as introduced in Section 5.2.3. The value is inserted into a list of all generated hash values. If a value clash is detected, the transformation process is aborted. A notice is printed out which advises the use of another hashing algorithm.

At this point, the structure is represented in the AIR but without any fields yet. They have to be extracted from the AST and convert into the AIR appropriately.

6.4.1 Message Capabilities

SpicaML provides developers with the ability to mark fields of a header structure so that they are handled in a special way during the transformation process. The respective fields therefore need to have assigned a concept that emphasises their differences from other fields in the structure. In doing so, the type of a field is mapped onto a more specific and maybe more elaborate data structure. However, it may also imply that encoding or decoding procedures modify portions of the structure during runtime. It mainly depends on the semantics of the respective concept up to which degree this influences the behaviour.

Currently supported examples of capabilities that involve modifications of a structure's content include *authentication*, *compression*, *encryption*, and *fragmentation*. All these aspects depend on additional data structures for control information. Other capabilities that are more content-oriented provide single fields of data that are handled in a special way. Examples are *message id*, *timestamp*, and *hostname*.

If a representation is assigned to a field, it determines the concrete implementation for the desired concept. Table 6.1 shows a comprehensive list of capabilities with their concept and available representations. The URN prefix for all concepts is assumed to be

Capability	Concepts	Representations
message id	`msg:msgid`	—
timestamp	`msg:timestamp`	—
hostname	`msg:hostname`	—
authentication	`crypto:auth`	`crypto:hmacmd5`
encryption	`crypto:encrypt`	`crypto:aes{128..256}`
compression	`msg:compress`	`msg:deflate, msg:gzip`
fragmentation	`msg:fragment`	—

Table 6.1: Supported message capabilities with their concept and available representations

`urn:spica:message:concept`, for all representations it is `urn:spica:message:rep`. The message id capability facilitates duplicate detection. Below, we shortly introduce the more elaborate capabilities.

Authentication The authentication capability involves insertion of two fields at the position where the authentication field is defined in a message header. The first field (`authEnabled`) is a Boolean field which indicates whether the capability is enabled or disabled. In the latter case, no further fields are added. The second field (`authSig`) reserves space for a message signature, the size of which depends on the digest length of the authentication scheme. The data structure added by the authentication capability has the following structure:

```
bool authEnabled;
uint8[] authSig;
```

Encryption The encryption capability behaves quite similar: A field marked with the encryption capability is expanded to a set of three fields, the first of which (`encEnabled`) specifies whether the capability is enabled or disabled. The main difference between authentication and encryption is the fact that authentication just inserts a new field and leaves the other data in the structure untouched. The content of the signature field nevertheless depends on the other field's values.

For encryption, all fields are removed starting from the position of the encryption capability field. The data structure required by the encryption capability provides storage for control information (`encPosition`) and encrypted data (`encData`). The encryption capability field thus expands to the following fields:

```
bool encEnabled;
uint32 encPosition;
uint8[] encData;
```

Only symmetric stream encryption algorithms may be used for this purpose. This includes block ciphers in a Cipher Mode or stream ciphers. In order to preserve the fault tolerance requirement, ciphers must further be capable of seeking forward in the key stream. This implies that the encryption must not depend on past messages. Given this property is held, the current position within the key stream (`encPosition`) allows receivers to decrypt the structure again.

Compression The compression capability behaves just like encryption except that it dispenses with a key stream position. Compression must always start from the beginning. A field holding the compression capability expands to the fields shown below. Again, one field (`compEnabled`) indicates whether the capability is enabled or disabled. Another provides storage for the compressed data (`compData`).

```
bool compEnabled;
uint8[] compData;
```

Fragmentation Data fragmentation is different from the previously introduced capability types. If present, it is enabled automatically once the length of an encoded message (N) exceeds a previously defined threshold ($thresh$). A typical threshold is around 1000B. This choice takes into account the *Maximum Transmission Unit* (MTU) – typically 1500B for Ethernet – and the fact that network messages should not be too small or too large for the given scenario (Section 1.1).

The data structure required for the fragmentation capability is a bit more complex than for the other capabilities: It translates to a Boolean variable (`fragEnabled`) and four 16bit integer values that characterise a fragment (`fragId`, `fragLen`, `fragNumber`, and `fragTotal`). The fragment header thus needs 9B of extra space.

Given the fragment capability field is located at position n in the encoded data stream A of size N, $n < N$. The first fragment contains the original message header up to the fragment capability field ($A_0, \dots, A_{n-1}$), the fragment header (9B), and $(thresh - n - 9)$B of payload ($A_{n+9}, \dots, A_{thresh}$). The algorithm divides the remaining $(N - (thresh - n - 9))$B into $\frac{N-(thresh-n-9)}{1000-9}$ pieces.

The layout of the fragment header is shown below.

```
bool fragEnabled;
uint16 fragId;
uint16 fragLen;
uint16 fragNumber;
uint16 fragTotal;
```

A message was not fragmented if `fragEnabled` is set to false. In this case, the remaining fields are omitted. Otherwise, one or more parts of the message will arrive separately as single fragments. A fragment header prefixes every fragment except the first one as it carries the original message header. Each fragment is assigned a unique identifier (`fragId`) which is required to locate fragments that belong together. In order to be able to conduct consistency checks, the length of the current fragment is provided as well. The current fragment number (`fragNumber`) and the total number of fragments (`fragTotal`) is required for reassembly. The total number of fragments computes as shown above.

As capabilities depend on the structure and the internal state of structure encoding, the respective functionality is realised together with the encoding and decoding procedures of the messages.

6.4.2 Structure Fields

So far, we have not discussed how to convert the field definitions of a structure from the AST to the AIR. Below, we introduce the conversion as well as the properties and cross-references required for data structure fields. We first have to determine which fields to create for a data structure in the AIR. A structure definition in the AST only contains the fields defined in the SpicaML model but no fields of the inherited structures. In order to ease the creation of code templates, the respective model in the AIR will need to have all fields defined, including the inherited ones.

For every header and message structure, all header fields are copied from the parent structure to a child node named `InheritedHeader`. If the parent already defines such a node, it is copied first. A container does not have any header fields that would need to be copied.

For every message and container structure, furthermore all the ordinary fields have to be copied from the parent structure into a child node named `Inherited`, given the parent is another message or container. If the parent already defines such a node, it is copied first as well. Headers are not considered in this case as they do not have ordinary fields.

With this approach, every structure is aware of its parent, the respective fields, and its newly defined fields. After copying all the fields from the parent structure, the new fields need to be added as defined in the AST. They are extracted and processed individually in the order of appearance. For each field, a new node named `Field` is created in the AIR and attached directly to the structure node as a child. First, the type name is determined. Two different types are defined so far:

- **Primitive Type**. A primitive type is provided by SpicaML and taken as is. Its textual representation is added to the AIR node as a property named `PrimitiveType`. Table 5.1 lists all supported primitive types.
- **Complex Type**. A complex type is a container type defined in SpicaML. The type name is looked up in a list of locally defined container types. In case no type was found, the transformation is aborted. If the type was found but has not yet been processed, processing is triggered automatically. If the lookup succeeded, a reference to the respective type description in the AIR is added under the property name `CompositeType`.

Each type may further be defined as an array. SpicaML supports two array types, namely *fixed* and *dynamic* ones. The general array marker thereby obeys the following structure: `'[' ( int ( ',' int )* )? ']'`.

The dimensions of a fixed array always have to be specified explicitly. For multi-dimensional arrays, the size of each dimension must be declared individually. Dynamic arrays are only capable of a single dimension. Their array marker thus reduces to `'[]'`.

The array type including the dimensions is extracted from the AST if a type is marked as an array. The following additional properties are then defined for the AIR node:

- **DynamicArray**. A Boolean property that defines a field to be a one-dimensional dynamic array.
- **FixedArray**. An array property defines a field to be a fixed array. The sizes of the dimensions are contained in the property value.

The name of the field, its default value, and the associated annotations are extracted next. Only the semantic annotations are supported: A reference to the concept (`refconcept`) and a reference to the representation (`refrep`). The following properties are therefore added to the field nodes:

- **Identifier**. A string that specifies the name of the field. As no name variants are applicable here, the name is taken directly from the model.
- **Default**. A property that specifies the default value for the field.
- **Refconcept**. A URN string that specifies the concept referenced by the field. If the `refconcept` annotation is missing in the model, this property is omitted.
- **Refrep**. A URN string that specifies the representation referenced by the field. If the `refrep` annotation is missing in the model, this property is omitted.

In order to support the code generation process, further properties are added to the structure which point to interesting entities, such as locally defined fields or other structures in the AIR. Properties that reference local fields are typically generated for the *message identifier*, *timestamp*, and *hostname* fields, for example. The name of the property obeys the following format:

```
prop_name : field_name 'Field' ;
```

where `field_name` is replaced by the variable name of the field as defined in the SpicaML model. Only fields with a capability assigned are applicable for this service. See Section 6.4.1 for details.

At this point, almost every part of the AST structure definition has been transferred to the AIR. The only thing left for conversion is the annotation that defines the comparison process. The annotation key `compare` accepts a list of arguments delimited by commas. The arguments are interpreted as field names in the current structure, including derived fields that have been copied over. Each one is then mapped onto an independent `Compare` property that references the respective field.

The algorithm used for localising a field in a structure is kept very simple. First, the inherited header fields are searched. If no field was found, the list of inherited fields from messages or containers is inspected. Finally, if the name was still not found, the names of all locally defined fields are compared to the desired field name. The transformation process is aborted if the algorithm was unable to locate the field. Otherwise a new property `Compare` is created for the respective field. It is not required to trigger a recursive search since complex types define comparison operations on their own.

6.5 Extraction of Modules

Just like the messages, module definitions are converted from the AST into a corresponding representation in the AIR. As modules build the foundation for data flow modelling, the model transformation relies on the previously extracted data structure definitions.

Every module declaration consists of a *name*, a set of *annotations*, several *offered* and *requested message types* along with the respective data management facilities, and optionally a collection of *additional DMCs*. Data filtering definitions are typically located in the module specification body as well. Due to the unavailability of filtering in the current version of Spica, details on the filter conversions are omitted here but shown in the SpicaML grammar in Chapter A of the appendix.

For each module definition in the AST, a new child node called `Module` is created at the root node of the AIR. If variants have been defined, the module names are expanded according to the rules introduced in Section 5.1. An independent child node is created for each expanded name again, comprising at least the following three properties:

- **Identifier**. A string that specifies the module's name. If variants have been defined, the respective expanded module name is used.
- **Concept**. A URN string that specifies the concept of the current module. If the `concept` annotation is missing in the model, a new one is generated automatically from the variant name, the module type, and the module name. It obeys the structure introduced in Section 5.1.
- **Rep**. A URN string that specifies the representation of the current module. If the `rep` annotation is missing in the model, a new one is generated automatically from the variant name, the module type, and the module name. It obeys the structure introduced in Section 5.1.

Depending on the annotations specified in the SpicaML model, some more properties are generated. They mostly relate to the monitoring capabilities of the module. Section 6.5.4 introduces details on these annotations and the monitoring capabilities.

6.5.1 Offering Message Types

For offering a message type, SpicaML provides a modelling primitive that comprises the message type identifier, the names of the destination modules, the transmission scheme, and a list of DMCs. Section 5.4.2 covers its formal structure. An example of a message type offering is shown below:

```
offer message WodlModelData<cn> [enabled]
    -> Base [local]
    scheme otm/pubsub
    dmc ringbuffer worldModelData [size=1];
```

In this example, the message `WorldModelData` with the name variant `cn` is offered for a local `Base` module through an `otm/pubsub` channel. The DMC used for inserting of new messages is a ringbuffer of size 1. This offer is furthermore registered automatically during module initialisation because of the annotation `enabled` that is provided in the statement's annotation list. We will now convert the AST structure of this definition into its respective AIR representation.

A new offer node (`Offer`) is generated in the AIR as a child of the module node (`Module`). A message node (`Message`) is then created as a child below the offer node. This is repeated for every such message of the module. The offer nodes are then assigned the properties listed below.

- **Ref**. A property that references a message description node in the AIR. The example above references the `WorldModelData<cn>` message.
- **RefDMC**. A property that references a DMC description node for the current message. DMC description nodes are introduced in Section 6.5.3. Several such references may be required for a single message offer.
- **Module**. A property that references a module description node. The module referenced by this property may receive the message type regardless of its location. The property **ModuleLocal** is used instead if the module must be located on the local host. **ModuleNotLocal** is used if the module must not be located on the local host. Several such references may be required for a single message offer.
- **LocalOnly**. A Boolean flag that specifies whether only local receiver modules have been configured for this message or not. It is required for the selection of the correct addressing scheme during code generation.
- **Scheme**. A string that specifies the name of the channel transmission scheme. Valid choices are `oto`, `otm/stream`, and `otm/pubsub`. For each scheme, an additional Boolean property is created which indicates that a specific transmission scheme was selected. The respective properties are: **SchemeOTO**, **SchemeOTMstream**, and **SchemeOTMpubsub**.
- **IsEnabled**. A Boolean value that specifies whether the offer is enabled by default or not. If it is set to false, the offer must be activated explicitly.

The message node acts as a representative for the message type and the configuration of the respective communication channel. It contains only properties that reference other nodes or characterise its configuration and behaviour. This very specific structure allows templates to access the relevant information directly.

6.5.2 Requesting Message Types

The SpicaML statement for a message type request is almost identical to a message offer and so is its layout. The example shown below visualises this circumstance.

```
request message WorldModelData<cn> [enabled]
    <- Vision [local], Simulator
    scheme otm/pubsub
    dmc ringbuffer wm [size=1; ttl=5s; export]
    dmc ringbuffer wmBall [type=BallInfo<cn>; size=1; ttl=5s; export]
    dmc ringbuffer wmDistanceScan [type=DistanceScanInfo<cn>; size=1;
        ttl=5s; export];
```

Even though this example is very limited, it reveals the similarity to message offers and emphasises the differences. A new request node (`Request`) is created as a child of the module node (`Module`). The properties and children assigned to this node are the same as for the message offer introduced in Section 6.5.1. The example nevertheless exhibits an important difference between these two models: Some DMCs are bound to a specific container type using the `type` annotation. Is this the case, Aastra tries to resolve these references during model transformation. If a reference fails to resolve, the model transformation is aborted.

A reference is resolved by analysing the contents of every field with a complex type in the current message. If the type of a field is compatible with the requested type, the resolution process succeeded and is terminated. Otherwise, the same procedure is applied to the types of all these fields recursively until a resolution is found or no more complex types are available. Compatibility in this context refers to compatibility in the sense of type compatibility in object-oriented programming.

If the reference resolution succeeded, the discovered location and the path towards this location must be made available for the code generation process. A specific data structure named `CallPath` is introduced for this purpose. It represents either a message or the path towards a field of a container that, in turn, is part of the message in question. Code generation now requires every DMC to have exactly one `CallPath` instance available. Only one instance is thereby created per field. As polymorphism allows different type references to resolve to the same field, it is common that several DMCs share the same `CallPath` instance. A message node, in turn, must be able to maintain several `CallPath` instances that potentially address several fields. For this purpose, each `CallPath` instance is assigned to a message node as the value of the **ReceiveCallPath** property.

The `CallPath` data structure maintains a list of AIR nodes that resemble the actual path towards a field. The first element in the list references the message type. The second element, if available, references a field within this message that must have a complex data type. The third element, if available, references a field within the data type of the field referenced by the second element. This is continued until the final field is reached. The associated DMCs are contained in another list.

6.5.3 Data Management Containers

A DMC is modelled as an independent DMC node (`DMC`) which is located beneath the respective offer (`Offer`) or request node (`Request`). It is assigned several properties that further describe its configuration. The properties typically include the name of the DMC and its configuration options. A flag is furthermore added that defines the visibility of the DMC. The comprehensive list of properties below outlines the configuration of a DMC node in the AIR.

- **Identifier**. A string that specifies the name of the DMC. In the example shown in Section 6.5.1, the identifier would translate to `worldModelData`, in Section 6.5.2 the respective identifiers would read `wm`, `wmBall`, and `wmDistanceScan`.
- **ElementType**. A property that references the AIR node of the data structure managed by the current DMC. For message offers, only messages are allowed here. For message requests, however, containers are also applicable.
- **ElementDerived**. A Boolean property that specifies whether the DMC also handles descendants of the element type. If this property is not set, only direct instances of the respective structure are allowed. This property is created automatically during model transformation. In the example shown below, **ElementDerived** would be set to true for the DMC `qYA` in module `ModuleA`:

```
container Y;
container YA : Y { int x; }
container YB : Y { double y; }
message X { Y body; }
module ModuleA {
    request message X <- ModuleB scheme oto dmc queue qYA [type=YA];
}
module ModuleB { ... }
```

The type of the field `body` in the message `X` has two descendants, `YA` and `YB`. The message request statement requests the message `X` and defines a DMC that manages only the derived container `XA`. The property **ElementType** will reference the container node of the `XA` container in the AIR but the body field in the message provides only the parent of `XA`, `X`.

- **Ref**. A property that references the message that defines the managed element type. In the example above, it would reference the message node for the message `X` in the AIR.
- **Struct**. A string that specifies the underlying data structure of a DMC. Allowed values are `ringbuffer`, `queue`, and `priorityQueue` where the latter value is derived from the `'priority_queue'` argument in SpicaML.
- **Size**. An integer value that specifies the size of the underlying data structure.
- **Export**. A Boolean value that specifies whether the DMC is exported or hidden for internal use only. In the latter case, the `private` annotation flag was defined.

The configuration values of the underlying data structure are adopted according to the specification given in Section 5.4.1. Properties are string values unless otherwise noted. **SIP** defines the insertion policy, **SDP** the duplicate policy, and **SRP** the retrieval policy for the underlying data structure. **SYNP** and **SYNI** define the synchronisation policy and interval. **INVP** finally specifies the invocation policy upon insertion of new elements. The **TTL** property indicates the validity interval for the managed elements. The data type for this property is integer.

If a DMC was created automatically during model transformation, its node depends on some more properties:

- **IsInternal**. A Boolean value that specifies whether the current DMC was created automatically and should be kept private or not. In the latter case, the property is omitted.
- **InternalId**. An integer value that specifies an internal identifier for the DMC. This simple counter is increased for every internal DMC, preventing naming clashes.

6.5.4 Monitoring Capabilities

SpicaDMF is a mature and lightweight monitoring solution for distributed, self-contained modules in a Spica communication infrastructure. Chapter 7 introduces SpicaDMF in detail. Monitoring capabilities are activated by adding specific annotations to a module definition in the SpicaML model. It is thereby possible for a module to become a *monitoring producer* (`monitorable`), a *monitoring consumer* (`monitor`), or both. If at least one of the two annotations is specified, the Boolean property **MonitoringEnabled** is assigned to the respective module node in the AIR.

In the next step, a message node is created (`Message`). For a producer, it is attached to the offer node (`Offer`), for a consumer to the request node (`Request`). All the properties of an ordinary message node are assigned to the node, except the **Ref** property. It is replaced by **RefDirect** that specifies the message's type name as a string (`'Beacon'`). The Boolean property **IsInternal** indicates that the message definition was generated automatically. The property **InternalId** assigns a unique identifier. These two properties are borrowed from DMCs as shown in Section 6.5.3.

If a module is configured to be a monitoring producer, the following additional properties are generated and attached to the respective module node automatically. Two DMC nodes are furthermore created and configured according to the Regular DMC and Express DMC specifications listed in Table 7.1. The properties **IsInternal** and **InternalId** are assigned to both nodes as demanded in Section 6.5.3.

- **IsMonitorable**. A Boolean flag that specifies whether a module is configured to be a monitoring supplier or not.
- **MonitorableRegularMessage**. A property that references the automatically generated node for the `Beacon` message.
- **MonitorableRegularDmc**. A property that references the automatically generated Regular DMC, configured according to Table 7.1.
- **MonitorableExpressMessage**. Another property that references the automatically generated node for the `Beacon` message.
- **MonitorableExpressDmc**. A property that references the automatically generated Express DMC, configured according to Table 7.1.

A monitoring consumer exhibits a similar configuration. However, it only depends on a Consumer DMC that is configured according to the specifications listed in Table 7.1. The following properties are assigned to the module node:

- **IsMonitor**. A Boolean flag that specifies whether the module is configured to be a monitoring consumer or not.
- **MonitoringMessage**. A property that references the automatically generated node for the `Beacon` message.
- **MonitoringDmc**. A property that references the automatically generated Consumer DMC, configured according to Table 7.1.

A typical AIR tree contains a collection of structures and modules augmented with supplementary information. These models build the foundation for the creation of data structure and a communication infrastructure.

There are no explicit cross-references in the AIR, except for references to the tree root. Cross-references to other nodes within the tree are possible but modelled through properties. This leads to a logical overlay structure that is independent from the underlying AIR tree. An example for such an overlay structure is the inheritance hierarchy of data structures.

The AIR was designed with the intention to be passed directly to a StringTemplate instance. Its structure facilitates access to information in a lean and straightforward manner. Below, we introduce the code generation process that makes use of the AIR tree.

6.6 Code Generation

Spica follows and promotes *generative programming*, a style of computer programming that uses abstract programming facilities and conversion tools to generate source code automatically. It typically comprises code generation techniques such as code templates. The MDSD approach introduced in Section 2.2 represents a generic form of generative software development.

The transition from an abstract model such as the AIR to concrete source code is thereby the weakest link in every generative approach. Namely, knowledge covering the abstract model and the target platform is required in order to create transformation and code generation templates. This process is still a manual one that may hardly be automated. In Section 6.7.3, an approach for automated code generation is proposed. However, it must be emphasised that manual control will always be required to some extend.

The code generation approach we follow with the current version of Spica employs a template engine that depends on manually created code templates for every supported target platform. We will introduce the template engine next, followed by the code generation approach of Spica.

6.6.1 StringTemplate Template Engine

A template engine combines one or more templates with a data model in order to produce one or more output documents. Templates are a wide spread approach for all kinds of transformation processes and so many different solutions are available. Spica adopts StringTemplate [105, 104, 106], a template engine powered by a domain-specific, functional language for generating structured text from arbitrary data structures. It is realised in Java but with ports for other languages such as C#. The distinguishing characteristic of StringTemplate is that unlike other engines, it strictly enforces *model-view separation* [105]. It was created for dynamic web page generation at first but quickly proved its general applicability.

One important feature of StringTemplate that mainly caused its adoption in Spica is its ability to use arbitrary input arguments (*attributes*) for templates. An attribute is not limited to a primitive type. It is rather allowed to reference a complex data structure, the elements of which (*properties*) are made accessible in the templates as required.[1] Attributes are loosely typed, i.e. their actual representation and internal structure are determined during runtime. The example below outlines the support for structured input data. *Attributes* are framed by angle brackets, *attribute properties* are delimited by a dot.

```
Your name: <person.name>
Your email: <person.email>
```

By knowing object-oriented languages, the syntax implies that there is some object named `person` that acts as a template attribute. The two attribute properties `name` and `email` relate to getter methods defined on this object. A template like this is valid because

[1]StringTemplate applies reflection in order to get to know about the layout of a data structure. Access is restricted to getter methods but the C# port also supports C# Properties and C# Indexers. For more information, please have a look at `http://www.antlr.org/wiki/display/ST/Expressions` (accessed 2008-08-23).

StringTemplate applies reflection in order to determine the mapping for attribute properties onto the attribute's structure. StringTemplate will thus look for getters on a passed object, for example, and render or even further interpret their return value. Given the C# port of StringTemplate is used in the example above, two C# Properties or an Indexer may be employed for this template to work.

StringTemplate treats return values of attribute properties as attributes again, allowing creation of template constructs as shown below:

```
Your Street Name: <person.street.name>
Your Street Number: <person.street.number>
```

Support is furthermore available for indirect specification of property names as well as the definition of maps. If an attribute contains multiple values, StringTemplate supports foreach-style iteration over these values. In the example below we assume that the attribute `numbers` contains the integers $1 \dots 10$:

```
All numbers 1: <numbers>
All numbers 2: <numbers; separator=",">
```

An application of the template will result in the following output:

```
All numbers 1: 12345678910
All numbers 2: 1,2,3,4,5,6,7,8,9,10
```

The default behaviour is to output any element of a list, one after the other without a delimiter. The `separator` option defines a delimiter which results in the output shown on the second line. Formatted output of a list is also possible using the following syntax:

```
All numbers: <numbers:{n |<n>, }>...
```

An application of the template will result in the following output:

```
All numbers: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
```

The argument directly after the opening curly brace and before the pipe symbol (`n`) defines the control variable. The part following the pipe symbol (`<n>, `) is an ordinary template specification. Two lists may be concatenated using square brackets: `[list_1, list_2]`.

Each template is typically defined in its own file. The file name thereby defines the name of the template. In this case, however, it is not possible to specify which input attributes are required for the template. A so-called *Template Group* facilitates the maintenance of several templates in a single file. A signature specifying its name and a formal list of attributes identifies each template in this case. Template Groups are further capable of single inheritance. Multiple inheritance is supported if inheriting from so-called *Template Interfaces*. The signature of a single template obeys the following structure:

```
st1 : name '(' ( attribute ( ',' attribute )* )? ')' '::=' '<<' tmpl '>>' ;
```

The symbol `name` is a string that specifies the name of the template. The `attribute` symbols are called placeholders for attributes and the `tmpl` symbol is finally replaced by the actual template code. A compact variant of the above structure is possible for short templates and one-liners:

```
st2 : name '(' ( attribute ( ',' attribute )* )? ')' '::=' '"' tmpl '"' ;
```

Such a template can be called from within another template. The listing below outlines this circumstance for the templates `draw` and `drawAt`:

```
Output: <draw()>
Output at: <drawAt(x=10,y=20)>
```

It has to be noted here that arguments of a template call should be qualified as shown in the `drawAt` template. A template may furthermore be called at an attribute directly using a special syntax. This implies that the attribute is passed as the only default attribute for the template:

```
Output: <names:draw()>
```

The default attribute of a template is typically named `it`. A template that is applicable for the template snippet shown above thus looks as follows:

```
draw(it) ::= << All names: <it; separator=", "> >>
```

Obviously, StringTemplate offers much more functionality and potential for customisation. StringTemplate intentionally omits support for specific functionality in order to retain the model-view separation and to keep templates simple. Typical imperative language features like variable definitions and assignments, loops, or arithmetic expressions are considered unnecessary or error prone. StringTemplate is nevertheless powerful and capable of generating context-free languages, even without these capabilities.

The aforementioned functionality of StringTemplate is required for the code generation templates in Spica. Below, we introduce the transformation process of Spica that involves the AIR and respective code template.

6.6.2 Spica Code Generation Approach

The starting point for code generation in Spica is a *Target Configuration* (TC) file that contains the generation rules for the target platform. It obeys the very simple structure shown below:

```
1 tcb : (
2     'Target' name '{'
3         'TreeHandling' '=' ( 'None' | 'Node:' node_name )
4         'FileSpec' '=' st_rule
5         'Templates' '=' template_file ( ',' template_file )*
6         'EntryPoint' '=' template_name
7     '}'
8     )+ ;
```

A TC file is composed of one or more *Target Configuration Blocks* (TCB). Aastra uses the name of a block (`name`) as an internal identifier that may be referenced within the TCB (line 2). Each block contains four options. Their meaning and assignable values are outlined below. The order of presentation is determined according to their mutual dependencies.

Attribute	Description
`name`	A string that specifies the name of the TCB.
`root`	A reference to the root node in the AIR.
`node`	A reference to the current node in the AIR.

Table 6.2: Template attributes applicable for TCB options

The `'TreeHandling'` option (line 3) defines whether the root node of the AIR is passed to the main template (`'None'`) or a node with a specific name (`'Node'`). In the latter case, Aastra tries to locate one or more nodes with the given name (`node_name`) in the AIR. Each node that matches the search criterion triggers a code generation process. Parallel execution is possible and explicitly recommended here.

All the remaining TCB options are able to make use of the AIR model, facilitating conditional creation of template names. For this purpose, Aastra interprets the given option values as templates by first piping them through a StringTemplate instance. Table 6.2 lists the available template attributes.

The `'Templates'` option (line 5) defines a list of Template Group files that are required for code generation. Commas delimit individual paths. As Aastra retains the order of the files, Template Groups that depend on each other have to be listed accordingly. Aastra relies on a template with a specific signature that is used as the starting point for code generation. The `'EntryPoint'` option (line 6) specifies the name of the Template Group that contains the **`aastra_start`**`(node)` template.

The `'FileSpec'` option (line 4) defines a destination path for generated code. In its most simple form, the value points to a concrete file. However, if several nodes are processed, results will be overwritten if only a single path is available. This is why typically a template is used which allows to regenerate the path for every processed node. Nonexistent directories are created automatically if required. The example below illustrates how to configure a target and specify the files to be generated.

```
Target Enums {
  TreeHandling = None
  FileSpec = <root.Property.Namespace: { x |<x>/}>Enums.cs
  Templates = templates/csharp/<name>.stg
  EntryPoint = <name>
}
Target Module {
  TreeHandling = Node:Module
  FileSpec = <root.Property.Namespace:{x|<x>/}>m/<node.Property.Identifier>.cs
  Templates = templates/csharp/<name>.stg
  EntryPoint = <name>
}
```

The `Enums` TCB cares about the SpicaML enumerations. It therefore passes the root node of the AIR to the template. All enumerations are then processed at once and stored into a single file. The path is created from the namespace given in the SpicaML model and the filename `Enums.cs`. This is sufficient as only a single node is processed.

Module processing is a bit different. The `Module` TCB is exclusively responsible for the SpicaML module definitions. This is why it is important to regenerate the destination path for every processed node. Otherwise, files might be overwritten unintentionally. In this context, the module identifier (`node.Property.Identifier`) is used for the filename. The namespace is again a part of the path but an additional directory is appended (`m/`) in order to keep the files separate. The file name is generated from the module name.

6.7 Code Generation Examples

Most code templates for Spica exhibit a unique layout with two separate templates: The main template is responsible for a specific model and defines the structure and the implementation of the program. Its filename typically depends on the respective model. A second template (`Shared.stg`) acts as a repository for maps and commonly used functionality, thus resembles a kind of functional library.

We do not present templates in detail because their internal structure heavily depends on the actual task. Instead, we will have a look at the code generation approach of Aastra. By considering two examples, we illustrate the different aspects of the transformation process. The in-memory structure of the AIR tree is thereby of particular interest. This is because it represents the data source for the template-based code generation process afterwards.

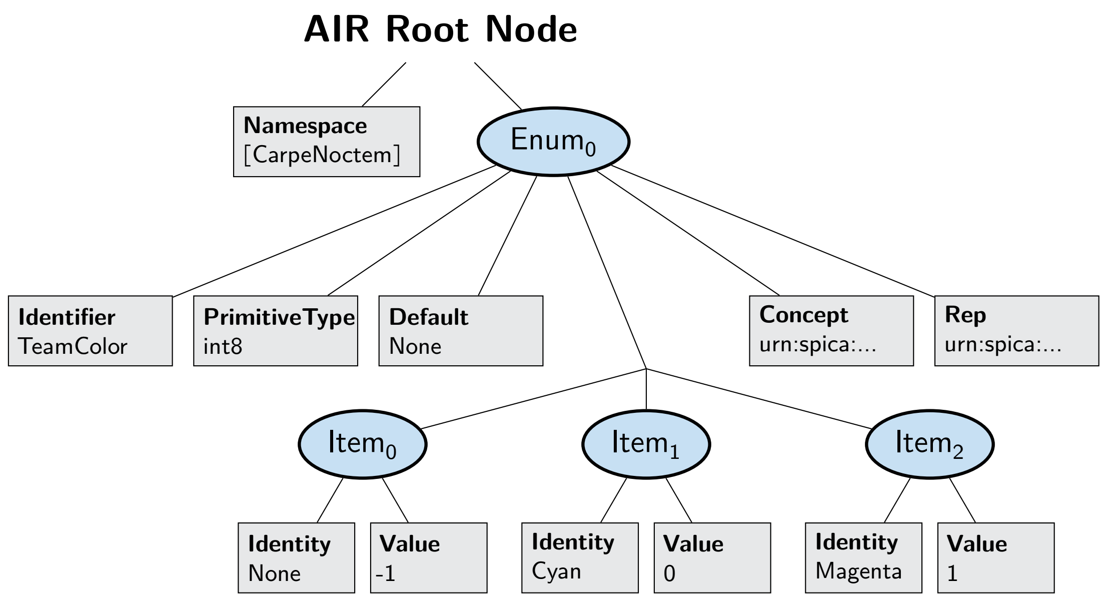
6.7.1 Enumerations

Let us assume a definition for an enumeration of team colours in SpicaML. It has to consider the team colours as defined for the RoboCup Middle Size League robots. The listing below illustrates a SpicaML model including a namespace definition:

```
namespace CarpeNoctem;

enum TeamColor : int8 {
    None = -1;
    Cyan = 0;
    Magenta = 1;
}
```

Aastra first parses and interprets the SpicaML model with the help of the ANTLR-based parser implementation. It then carries out the model transformation as introduced in Chapter 6. It results in an AIR representation of the enumeration model. It contains all the relevant element of the SpicaML model but exhibits a structure that is better suited for the code generation process. The graph below outlines the abstract layout:



Enum$_0$ represents an enumeration as described in Section 6.3. The nodes Item$_0$, Item$_1$, and Item$_2$ are the enumeration elements, each with the respective **Identity** and **Value** properties.

The listing below presents a suitable template definition for generating C#-based enumerations. The AIR root node is passed as the attribute `rootNode`, so the template has to deal with the whole tree instead of only a limb thereof.

```
 1  aastra_start(rootNode) ::= <<
 2  using System;
 3
 4  <if(rootNode.Property.Namespace)>
 5  namespace <rootNode:msg_get_namespace()> {<endif>
 6
 7  <rootNode.Children.Enum : {enum |
 8      public enum <enum.Property.Identifier> :
 9          <type_map.(enum.Property.PrimitiveType)>
10      { <enum.Children.Item : {item |
11          <item.Property.Identifier> = <item.Property.Value>, }>
12      }
13  }>
14  <if(rootNode.Property.Namespace)>} <endif>
15  >>
```

The template first tries to process the namespace as defined in the SpicaML model. It therefore checks whether the **Namespace** property (`Namespace`) is defined at the root node (lines 4 and 14). If no such property is available, the respective lines are omitted. The statement `rootNode.Property.Namespace` translates into a call to a C# Property and a respective C# Indexer at the AIR root node (`rootNode.Property["Namespace"]`). Lines 7, 8, 9, 10, 11, and 14 present similar statements.

Line 5 illustrates how to generate the namespace so that it conforms to the guidelines of the target language. The respective template code for `msg_get_namespace(it)` is quite compact and reads as follows:

```
msg_get_namespace(it) ::= << <it.Property.Namespace; separator="."> >>
```

The enumeration template iterates over all children of the root node and filters on the name `Enum` (line 7). It uses the formatted output, so the nested iteration (line 10) can insert the respective enumeration elements along with their values.

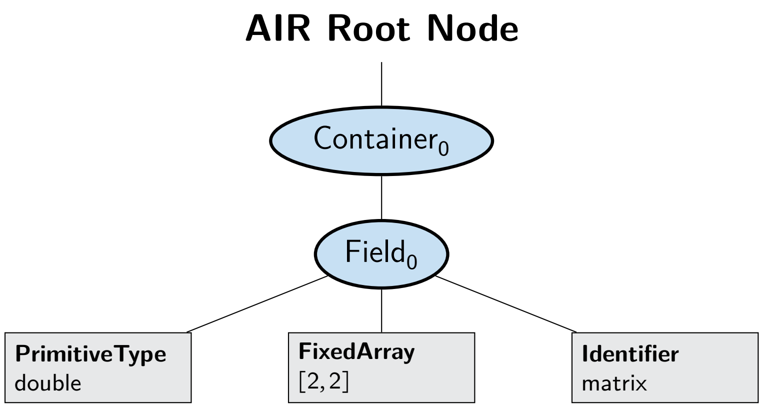
The names of the primitive types in the AIR have not been transformed yet; they are still the same as the ones in the SpicaML model. Because the code templates care about the mapping to platform-specific code, they also have to care about the mapping of primitive types to their platform-dependent counterparts. This is realised as a StringTemplate map. Map elements are either accessed directly (`<type_map.float>`) or indirectly using the indirection support provided by StringTemplate: `<type_map.(attr)>` where `attr` is an arbitrary attribute. Its output addresses a map entry. Line 9 illustrates this circumstance: The `PrimitiveType` property of the enumeration is mapped onto a target-specific primitive type.

6.7.2 Fixed Arrays

Another example illustrates the code generation for fixed size, multi-dimensional arrays. The field in the SpicaML snippet shown below represents a 2×2 matrix:

```
double[2,2] matrix;
```

We will assume that it was defined within an arbitrary container. This means that it translates to the following representation in the AIR:



The structure is basically the same as in the enumeration example above. Field_0 represents the first field specification of the node $\mathsf{Container}_0$. The properties simply reflect the important components of the SpicaML statement.

A code template that is able to deal with these data is shown below. It copies the contents of an array defined in the current data structure (`this`) to another array of the same size defined in another data structure (`input`).

The template has to be applied to a field node such as Field_0. Lines 3–5 generate the for-loops required for iterating over the array. For the two-dimensional array, this results in two for-loops, each counting from 0 to 2 excluding. The control variable `i` is provided by StringTemplate and incremented automatically in each iteration. Lines 8 and 10 illustrate a copy

```
 1  copy_fixed_array(it) ::= <<
 2
 3  <it.Property.FixedArray:{count|
 4  for (int i<i> = 0; i<i> \< <count>; i<i>++) \{
 5  }>
 6
 7  <if(it.Property.CompositeType)>
 8      input.<it:field_get_identifier()>[<it:varcount()>] =
 9      (<it:field_get_type_name()>)this.<it:field_get_identifier()>
10          [<it:varcount()>].Clone();
11  <else>
12      input.<it:field_get_identifier()>[<it:varcount()] =
13          this.<it:field_get_identifier()>[<it:varcount()>];
14  <endif>
15
16  <it.Property.FixedArray:{count|\}}>
17  >>
```

operation for a complex type using the `Clone` operation. The `field_get_identifier(it)` template returns the name of the field variable defined in the local data structure (lines 8, 10, 12, and 13):

```
field_get_identifier(it) ::= "__field_<it.Property.Identifier>"
```

In the same lines, the template `varcount(it)` generates a list of loop variables. They correspond to the variables defined by the for-loop (line 4):

```
varcount(it) ::= << <it.Property.FixedArray:{count|i<i>};separator=","> >>
```

The template referred to by `field_get_type_name(it)` returns the type name of the current field. As it is more complex, we will not discuss it here. Copying a primitive type is very similar to copying a complex one. However, a primitive field does not need to be cast or cloned.

These examples outlined the general approach used for code generation in Spica. They implicitly pointed to an issue of template-based code generation: Writing templates is a very expensive and tedious task, especially if statements that are more complex are involved. This is why other, more automated approaches are currently under active research.

Jens Wollenhaupt has developed a promising approach in the context of his Bachelor's thesis [140, 50] that addresses these issues. It facilitates template generation by shifting the focus to the semantic annotation of method calls in libraries. We will introduce it next.

6.7.3 Ontology-Supported Code Generation

A key objective of MDSD is to separate the overall design and architecture of applications from concrete realisations. The developer should be able to specify the application at an abstract level without bothering too much about the implementation details up-front. Separate modelling stages achieve this by guiding the developer from an abstract model down to concrete source code. Spica follows an approach with a transformation to platform-specific source code based on a template transformation. These templates have to be created manually.

MDSD shifts much of the complexity of software development from the application modeller to the transformation builder. It is thereby most important that model transformations work correctly and produce efficient output. In the context of a Bachelor's thesis, we have developed systematic support for the automation of model transformations.

Our approach utilises an ontology that captures application and domain-specific knowledge. It is used for the automatic transformation of an abstract model to several platform-specific models and finally to source code. Entities and concepts of the application domain as defined in the ontology are referenced in the model as well as in semantically annotated APIs of the target platform. This is where feedback from a developer and knowledge of the platform is required. This information facilitates automatic mapping of modelling elements to variables, objects, or interfaces of a target platform. We can thus specify generic transformation rules for platform-independent models that are adapted automatically to specific target platforms by using the ontology. When a new platform is introduced into a development team, only the appropriate semantic annotations to the objects and interfaces on the new platform have to be added.

The approach is not part of the Spica development framework yet. It nevertheless would replace parts of the template generation and transformation processes and thus simplify the integration of new target platforms. The main issue with this approach is that the underlying APIs of the target platforms need to be annotated with the semantic concepts of the ontology. This again represents a shift of development expense from template generation to the annotation of libraries. The main difference and simplification with this approach is, however, that annotations are located directly at the respective functionality and imply less specification overhead.

7 Monitoring Framework

"Where is the wisdom we have lost in knowledge?
Where is the knowledge we have lost in information?"

— Thomas Stearns Eliot
Choruses from The Rock, 1934

Modules in the SpicaML data flow model may be equipped with monitoring capabilities. Each module can thereby act as *producer* or *consumer* of monitoring information. It is also possible to assign both roles. Because of the possibly large number of participating modules, the monitoring approach should be conservative in network usage.

The Spica development framework provides the implementation for this additional functionality as part of the Castor utility library. Appropriate monitoring capabilities are automatically embedded into the generated stubs for the communication infrastructure. As monitoring is a quite far-reaching topic, we decided to restrict its functionality to the two main domains *logging* and *monitoring*. Logging enables producers to actively point to important events whereas monitoring is a functionality of consumers to interpret the information provided by producers.

The *Spica Decentralised Monitoring Facility* (SpicaDMF) implements the logging and monitoring functionality in a distributed fashion. The overall concept is simple and resembles the functionality of common logging frameworks: Producers only have to provide a *priority,* a predefined *importance level,* and the monitoring information. Consumers are then able to filter and roughly classify this information.

Logging information is typically generated irregularly in reaction to events. Monitoring information, in turn, is more of periodical nature. SpicaDMF must be able to detect and appropriately adapt itself to these emission behaviours. Network communication is furthermore known to be sensitive to failure as stated in Section 1.1.3 and ad hoc networking characteristics imply unexpected appearance and disappearance of participants. Hence, the network must be considered unreliable especially with respect to availability. This is why producers must not assume consumers to receive monitoring information. Redundant transmission is one possible solution to this issue.

The monitoring consumer must further be able to filter on incoming monitoring data. To keep the communication and implementation complexity low, a publish/subscribe scheme is not considered appropriate here. We will rather favour one-way transmissions over approaches that are more elaborate, mostly because of their simplicity.

Considering on these requirements, we created the SpicaDMF monitoring and logging facility that integrates well with the generated Spica communication infrastructure. Even though SpicaDMF is quite specialised and tailored to the application domain of AMRs, it is capable of delivering almost any type of information. We will now outline the data format used by SpicaDMF in Section 7.1 before introducing the communication characteristics in Section 7.2. A short discussion on the application interface closes this chapter in Section 7.3.

7.1 Message Format

Listing 7.1 shows a SpicaML model covering the SpicaDMF-related data structures. It first defines common importance levels – namely, `Ok`, `Info`, `Warning`, `Error` – in an enumeration of type `uint8` (line 1). The level `OOB` is used for out-of-band transmissions which either do not fit into the predefined scheme or represent fundamentally different information. One application for the `OOB` level is monitoring information on the state of resources, for example.

The body of the actual message structure is kept as simple as possible (lines 9–15). It contains the name of the sender (`name`), the priority (`priority`) and the importance level (`level`), followed by the monitoring information (`msg`) and a list of value containers (`data`). There is no prescription for the structure of a sender's name, so free text is allowed here. A Spica stub will use the semantic representation of the module for this purpose. The container list maintains complementary information and OOB monitoring data. The SpicaML model provides containers for the most common data types (lines 4–7).

```
 1 enum Level : int8 { Ok = 0; Info = 1; Warning = 2; Error = 3; OOB = 10; }
 2
 3 container BeaconData { string name; }
 4 container Floats : BeaconData { float[] data; }
 5 container Doubles : BeaconData { double[] data; }
 6 container Integers : BeaconData { int32[] data; }
 7 container Strings : BeaconData { string[] data; }
 8
 9 container BeaconBody [compare=level,priority,msg] {
10     string name;
11     uint16 priority;
12     Level level;
13     string msg;
14     BeaconData[] data;
15 }
16
17 message Beacon : Header [compare=body] { BeaconBody body; }
```

Listing 7.1: SpicaDMF message format for monitoring beacons

The most conspicuous element in the SpicaML model is the comparison annotation `compare`. It causes custom comparison operations to be defined for the message `Beacon` (line 17) and the container `BeaconBody` (line 9) in Listing 7.1. When comparing two `Beacon` messages, the request is directly handed-off to the `BeaconBody` container which itself consults three of its fields, in this case `level`, `priority`, and `msg`. As these fields all have primitive types, they will be compared in the given order.

	Regular DMC	Express DMC	Consumer DMC
TTL	1000ms	—	1500ms
SYNP	`soft`	`none`	
SYNI	330ms	—	
SIP	`fifo`		
SDP	`overwrite`		
SRP	`remove`		
INVP	`direct`		
Structure	`priority_queue`		

Table 7.1: Configuration of monitoring DMCs

DMCs with priority queue backends handle monitoring messages. The comparison setting is required to preselect the most important and most urgent message first. For SpicaDMF, the importance level is considered just before the priority value.

7.2 Communication Characteristics

In order to address the efficiency requirement worked out above, we specify the communication characteristics so that urgent announcements are possible but flooding is prevented. A monitoring producer offers the `Beacon` message as defined in Listing 7.1 and registers it with the Geminga resource discovery. The respective Spica stub creates two priority queue DMCs: The first is called *Regular DMC,* the second one *Express DMC*. A monitoring consumer maintains a so-called *Consumer DMC*. Table 7.1 presents the configurations of all DMCs, taking into account all relevant options.

The Regular DMC signals new beacons at 3Hz. This limits the number of sent messages and prevents flooding at the same time: Even if logging messages are generated at a high frequency, not more than three messages are sent per second. Inserted messages time out after 1s, the remaining ones are dropped. No limitations are configured for the Express DMC; it can rather emit messages at arbitrary rates and without any delay. The same is true for the Consumer DMC that nevertheless only received messages. Entries time out after 1.5s and are removed automatically in this case.

A combination of the Regular DMC and the Express DMC achieves a balance between immediate reaction and conservative bandwidth consumption. Whenever new monitoring information is generated which was not sent before, it is inserted into the Express DMC for immediate transmission. It is also inserted into the Regular DMC, so beacons are signalled regularly as long as the reason for the notification persists. Otherwise, elements of the Regular DMC time out after 1s and are removed immediately. Since the DMCs are priority queues, the most prioritised message is located on top of the queue. Once the topmost message was retrieved, it is removed from the queue. If a message has a lower priority, it is enqueued somewhere below the topmost element. In this case, the message is purged from the list as soon as it times out. For reoccurring events, the timestamp of the enqueued entry is constantly updated and kept fresh this way.

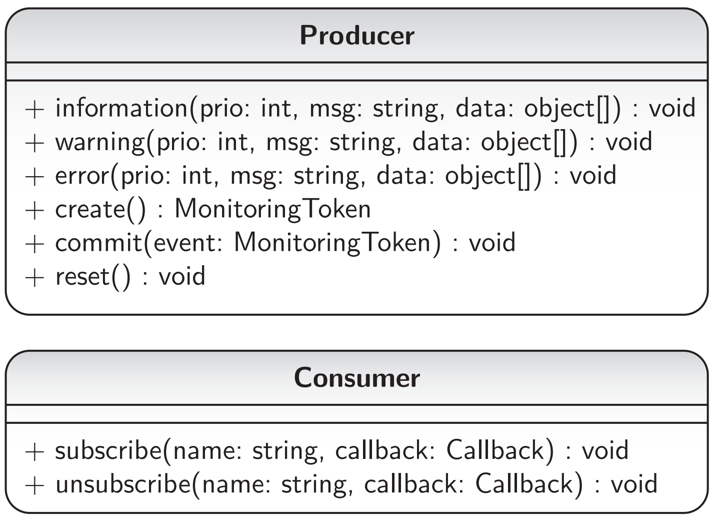

Producer
+ information(prio: int, msg: string, data: object[]) : void
+ warning(prio: int, msg: string, data: object[]) : void
+ error(prio: int, msg: string, data: object[]) : void
+ create() : MonitoringToken
+ commit(event: MonitoringToken) : void
+ reset() : void
Consumer
+ subscribe(name: string, callback: Callback) : void
+ unsubscribe(name: string, callback: Callback) : void

Figure 7.1: The SpicaDMF producer and consumer class interfaces.

With this approach, only the highest prioritised and therefore the most important message is sent. This is another measure for a conservative use of the communication media. Applicability has been verified during the RoboCup German Open championships in 2008.

7.3 Application Interface

The implementation of the producer offers several methods for signalling relevant events. It is shown in Figure 7.1 (`Producer`). The methods are classified according to the respective importance levels as, i.e. `information`, `warning`, and `error`. Every method call depends on a priority value and a verbose message describing the incidence. An array of arbitrary data may optionally be passed as well. The implementation takes care of delivery and determines which announcements have to be inserted into the Express DMC.

Two further methods are provided which allow generation of more complex monitoring beacons: The first generates a new instance of a `Beacon` message (`create`) and the second cares about delivery (`commit`). The message may so be initialised appropriately before transmission. The `MonitoringToken` type is a wrapper for a `Beacon` message.

The consumer implementation (`Consumer`) provides two methods: The first is used for subscribing to a producer (`subscribe`), the second is used for unsubscribing again (`unsubscribe`). Both methods accept a name and a callback. The latter one is triggered once new information is available that matches the given producer name. Its realisation depends on the implementation language. Delegates are used for C#. C requires function pointers while C++ may adopt a more elaborate technique such as object-oriented function object wrappers.[1] Java typically employs interfaces or classes with specific methods.

[1]Two solutions are possible here: A recent C++ standard library provides appropriate function wrapper primitives with TR1 [14]. The Boost C++ Libraries that build the foundation for the TR1 implementation offer `Boost.Function`, serving the same purposes: `http://www.boost.org/doc/html/function.html` (accessed 2008-08-30). SpicaDMF adopts the Boost alternative, so it also compiles against older standard libraries.

8 Resource Discovery and Auto-Configuration

"There is a theory which states that if ever anybody discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is another theory which states that this has already happened."

— Douglas Adams
The Hitchhiker's Guide to the Galaxy, 1979

Communication infrastructures of highly modular, distributed systems as targeted by Spica are often complex to configure and their management is furthermore typically expensive. This is especially important in the domain of mobile robots where the environment is dynamic and changes are frequent. In order to overcome these issues we propose Geminga,[1] a resource discovery approach for Spica-based communication infrastructures. It complements the capabilities of these infrastructures mostly by simplifying their configuration but also by providing discovery capabilities for resources and services in a distributed fashion.

Geminga is implemented as a service. An instance must be running on every system that wants to benefit from the provided functionality. Components may register their desired resource request or offer with the local Geminga service. Once a requested resource is available or an offered resource is requested by another system, the respective components are informed of this fact. Geminga does thereby not establish communication channels by itself, but triggers their creation between resource consumers and providers. Appropriate functionality that cares about the management of communication channels is provided for several platforms.

We will now discuss some requirement before introducing the architecture of Geminga. As it actually is a by-product of the Spica development framework, the provided service is strongly inspired by the requirements introduced in Section 1.2.

8.1 Mobile Robots and Discovery

Geminga [10] is designed for mobile robots that communicate via wireless networks. Such media are known to be vulnerable to failures. Mobile robots may furthermore get unavailable

[1]The name Geminga descends from the pulsar *Geminga* in the constellation Gemini which is closer to Earth than any other known pulsar. Pulsars are highly magnetised, rotating neutron stars which emit a beam of detectable electromagnetic radiation in the form of radio or x-ray waves. This fundamental characteristic bears resemblance to our resource discovery approach which periodically emits beacon messages.

or leave a group unattended at any time, even during controlled operation. As discussed in Section 2.4, this may happen because of physical effects or system failures. Hence, measures have to be taken to make dynamic discovery and negotiation processes robust.

Simplicity in design and protocol interaction is one possibility that contributes to the robustness of an interaction approach. We start with a discussion on desired functionality for Geminga.

- **Distributed Operation**. No central knowledge repository or discovery service may be employed because otherwise, discovery is vulnerable to network unavailability. The approach should furthermore minimise bandwidth consumption and therefore involve only sparse, if possible unidirectional protocol interactions.
- **Fault-Tolerance**. A resource discovery scheme for mobile ad hoc networks must be able to tolerate failures of the communication media as well as unavailability of other systems. It must furthermore be robust against unexpected malfunctioning.
- **Passive Operation**. Resource consumers should be able to listen to the communication in the network in a minimally invasive manner, i.e. without influencing resource providers in any way.
- **Scalability**. The resource discovery aims at small-scale infrastructures up to about 30 participants. This number is derived from the average size of a group of robots, including a limited number of additional systems.
- **Self-Management**. The resource discovery is primarily used for automatic configuration of a communication infrastructure. In this capacity, it must not need extensive configuration and be able to adapt itself to changing demands.
- **Semantic Matching**. The Spica development framework adopts semantic annotations for embedding meta information. A resource discovery approach must be kept generic and support matching of semantic resource types.

These considerations address the fault tolerance as well as the distributed operation requirements as defined in Section 1.2. Every participant must execute an instance of Geminga. This implicitly realises redundant storage of configuration data: Geminga maintains a global view on the world and thus a copy of the global configuration.

Spica supports developers in creating communication infrastructures for distributed modular software architectures. In order to ease setup and configuration and to facilitate reconfiguration in case of environmental changes, an observer seems to be beneficial in this case. A resource discovery complies with these requirements: It provides monitoring capabilities and further promotes dynamic adaptation. In the following, we will refer to components that interact with the Geminga service as *clients*, since they may act as resource provider, resource consumer, or both.

By addressing the requirements discussed in Section 1.2 and taking into account the related work analysis in Chapter 3, we decided in favour of a beacon-based resource discovery approach similar to Lenders et al. [81]. A Geminga service thereby announces the resource offers or requests of its clients on a regular basis. The other participants are so kept up-to-date. This announcement flooding clearly introduces an upper boundary for scalability that, however, is well above the requirements.

Each participant maintains a local data structure in which it maintains received beacons. Outdated entries are purged after a given amount of time automatically. Section 8.5 further discusses resource type matching based on the locally stored announcement information.

Geminga triggers creation or removal of communication channels at the respective clients directly. Such channels may operate in one of the three modes shown below:

- **One-to-one (oto)**. One-to-one channels are configured and established only once between two or more participants. No information exchange is required.
- **One-to-many/stream (otm/stream)**. Channels configured for streaming mode are set up and established passively. They resemble a media for streaming data with one or more recipients that only need to listen at specific addresses. Information exchange is required from providers to consumers only.
- **One-to-many/pubsub (otm/pubsub)**. Channels configured for publish/subscribe mode are established as soon as at least one subscriber is present. They are shut down after the last consumer has withdrawn its subscription. Bidirectional information exchange is required between the service peers.

8.2 Geminga Layout Decisions

Geminga is a core component of the Spica development framework. As its design follows a very generic approach, adoption for other platforms or applications is possible as well. A generic *identity* token identifies a resource type. It may be either a 32bit unsigned integer, a text string, or a semantic identifier consisting of a *concept* and a *representation* [113]. Thanks to the data structure definition in SpicaML, further identity tokens may be added with low effort. Numerical identifiers are typically derived from their semantic counterparts using a hash algorithm. Section 5.2.3 introduces this approach in detail.

Announcement messages are distributed among all Geminga instances, propagating the locally available resources as well as resource subscriptions. The discovery process, in turn, does not involve any network communication. It is based on the collected information from other participants and the information received from local clients only. As a result, the matching process becomes insensitive to networking issues.

The clients connect to their local Geminga service through which they publish or request resources. The communication between them relies on a protocol that addresses the specific needs such as event orientation.

8.3 Geminga Client Negotiation Protocol

The *Geminga Client Negotiation Protocol* (GCNP) is a lean, text-based protocol used for information exchange between clients and a local Geminga instance. TCP connections are used for this strictly local communication.

Clients register their resources with the local Geminga service. Once the configuration changes, a resource is available, or a resource is requested, Geminga provides feedback for the respective clients, triggering an appropriate action. It thereby only signals changes; it does not take any action on its own. The clients have to react on the feedback. However, in order to ease the integration with existing software packages, interface stubs implementing the GCNP are provided for some platforms (C++/Linux and C#). These stubs handle dynamic channel creation and removal as well as management tasks in a transparent manner.

8.3.1 Protocol Operation

The structure of a GCNP protocol statement obeys the grammar shown below. Just like in the previous structure definitions, we rely on the grammar syntax of ANTLR v3 [103]:

```
msg : id ' ' command ' ' subcommand? ' ' ( param ( ' ' param )* )? '\r\n' ;
```

The format is intentionally kept simple in order to facilitate the integration of clients that are not natively supported by the existing interface stubs.

Every statement is introduced by a 32bit identifier (`id`). An identifier that equals 0 implies that Geminga has generated the statement in order to provide management information for the client. Clients may use the remaining values for establishing a relationship between consecutive protocol statements. This might be, for example, a command issued by the client and an answer coming from Geminga. Clients may so keep track of asynchronous replies from Geminga.

Possible values for the `command` and `subcommand` symbols are introduced later. A `param` symbol is a key-value pair that is structured as follows:

```
param : key ( '=' value )? ;
```

Several parameters are delimited by spaces. The value part of a parameter may be omitted. Each statement is furthermore delimited by `CR+LF`.

8.3.2 Statement Types

GCNP uses four types of statements, namely *messages*, *commands*, *errors*, and *events*. We will now introduce their meaning and characteristics. Available commands and subcommands are shown where applicable.

Messages Messages are statements generated by Geminga in order to provide a client with relevant information. The `command` symbol in the definition of `msg` above specifies the type of the message whereas `subcommand` may be used for further specialisation.

A command value of `'msg'` introduces an informative message generated by Geminga. It has no direct effect but acts merely as state information.

A command value of `'info'` provides information on a specific topic. Subcommands further classify the available information. The client may query information on its own numeric identifier (`'provider'`), the internal numeric identifier of Geminga (`'geminga'`), or the commands (`'commands'`) available at the current Geminga instance. The respective values are provided as parameters.

The `'done'` command finally acts as an end-of-session marker. No subcommands or parameters are applicable here.

Commands Clients issue commands in order to trigger execution of operations at a Geminga service. No immediate responses are generated, except for errors. Instead, Geminga delivers responses asynchronously. The commands for resource publication and subscription are shown below, covering announcement, modification, and removal of resources.

The `'pub'` command allows resource providers to configure a resource offer. Subcommands trigger a creation (`'create'`), modification (`'modify'`), or removal (`'remove'`) of an offer. The resource itself is characterised by a set of parameters that include the *channel mode* and the *resource identity*, for example.

The `'sub'` command allows resource consumers to configure a resource subscription. The subcommands and parameters are identical to their counterparts in the `'pub'` command.

Events Subscription events inform the provider of changes in its registered publications. Publication events, in turn, inform the subscriber of changes in its registered subscriptions. Events build the foundation for adaptation in Geminga and thus resemble one of its most important features. The structure of an event is similar to that of the corresponding `'pub'` or `'sub'` commands, except that the subcommand names differ.

A subscription might trigger a publication event (`'pub'`) with the respective subcommand once a resource appears (`'add'`), disappears (`'del'`), or changes its configuration (`'modify'`). Parameters represent the configuration of the resource.

A publication might trigger to a subscription event (`'sub'`) with the respective subcommand. The subcommands and parameters have the same semantics as for publication, except that they relate to resource requests.

Errors In case of an error while executing a command, error messages are sent synchronously. The `command` symbol of the message `msg` is replaced by a triple-digit error code. The `subcommand` and `parameter` symbols are replaced by a descriptive string.

8.3.3 Application Example

Figure 8.1 visualises a typical scenario where Geminga negotiates a channel creation procedure. It involves a resource provider on the left-hand side and a resource consumer on the right-hand side. Both have a Geminga service running.

The provider registers a specific resource offer with Geminga (`'pub create'`). The resource's availability is advertised periodically (Δt) with the next Geminga announcement. At some later point in time, the consumer decides to subscribe to this specific resource (`'sub create'`). The request is registered and advertised in the next announcement. As Geminga is already aware of the fact that the resource is available, it informs its client by providing the relevant contact information. The consumer instantly triggers channel creation.

Once the provider has received the announcement, it triggers the establishment of the communication channel as well. At this point and without explicit synchronisation, both

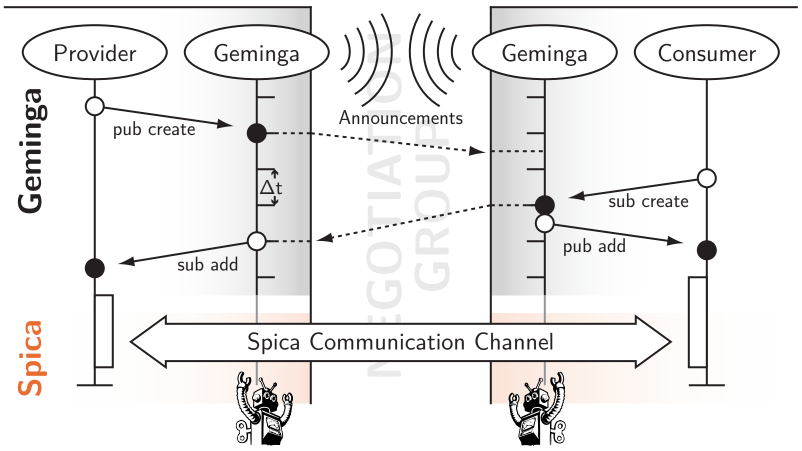


Figure 8.1: GCNP interaction and establishment of a Spica channel. A provider registers its offer with Geminga. It is informed once a consumer is available. The consumer is notified in the same manner once a provider is available.

parties share a communication channel. This is only possible with connection-less schemes. Their connection-oriented counterparts carry out synchronisation steps implicitly.

8.4 Announcement

Geminga follows the simple yet powerful approach of periodical resource announcement. It dispenses with elaborate protocol interaction schemes that are capable of handling network failures. Instead, we achieve reliability by providing redundancy in data transmission. Publications and subscriptions are transmitted periodically and combined in only one announcement message. Therefore, piggybacking effectively reduces the number of required messages.

We will now have a look at the operation of Geminga. In particular, the announcement approach is of most interest here. The respective announcement message format is the discussed in detail.

8.4.1 Geminga Identifier Generation

Before introducing the announcement scheme, we will outline the generation of unique identifiers for Geminga services. These identifiers are required because every host potentially has several valid network addresses that therefore disqualify as unique identifiers. This is why each Geminga service generates a unique numeric identifier using the procedure outlined below:

First, a random number is generated and broadcast to the other Geminga services. If another system already owns this number, the process is repeated until a unique identifier was found or an iteration limit has been exceeded. In the latter case, the Geminga service shuts down. Otherwise, a unique identifier was generated. It identifies this Geminga instance from now on uniquely within the local network.

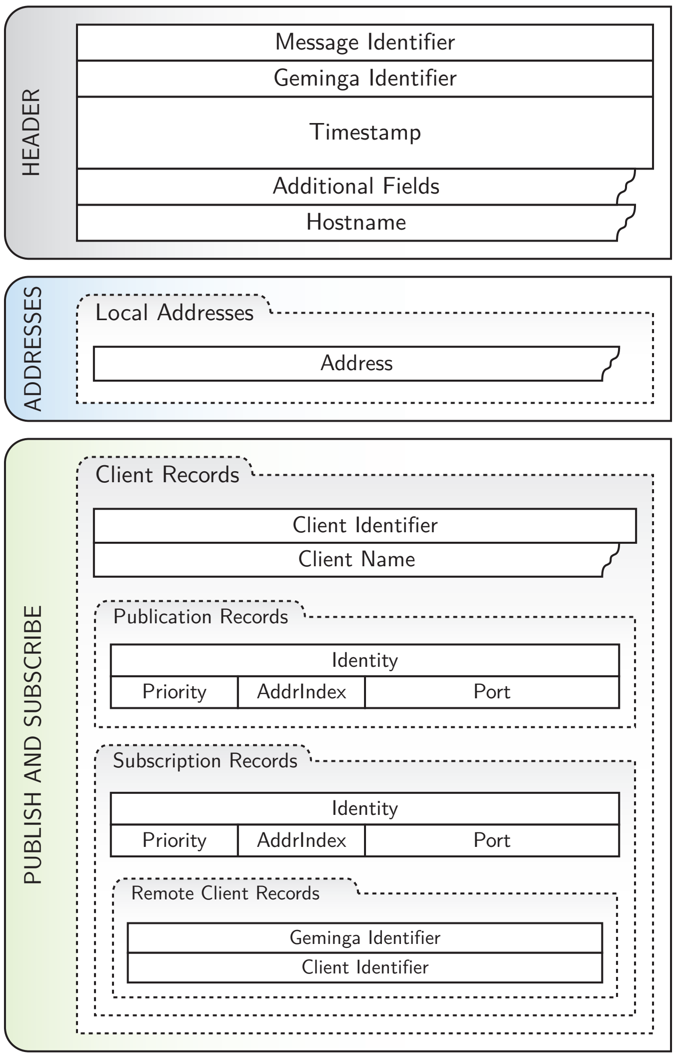


Figure 8.2: The Geminga announcement message consists of three main parts: A header for management information, a list of local addresses, and a list of offered and requested resources.

If an identifier clash is detected during runtime, i.e. after the respective identifiers have been created, all involved participants have to drop their respective identifiers and generate new ones, using the procedure outlined above. This will force all Geminga instances to cancel their resource type publications and subscriptions. They will nevertheless be renewed once a new identifier was found.

8.4.2 Resource Type Announcement

By default, Geminga announces resources every two seconds. It is possible to adapt the interval to the number of participating Geminga instances or the relative message loss automatically. The latter one is accomplished by monitoring the incoming traffic and interpreting its distribution.

Figure 8.2 shows the unified Geminga announcement message. It is responsible for resource requests and offers as mentioned earlier. The *message header block* contains management information required for processing. It will not be further discussed here as it only contains information that is required by the Spica communication infrastructure. The *address block* maintains a list of all local interface addresses. They were collected during start-up and are required for establishing communication channels. For each available interface, a unique announcement message is generated. It only contains addresses and resources that are valid for this specific interface. No announcements are generated for loopback devices.

The *publish and subscribe block* lists all the local clients with their publications and subscriptions. Each client is recorded with its name and a 32bit unsigned integer that identifies it uniquely within its local Geminga service. In combination with the Geminga identifier, a globally unique identifier for the client is furthermore established.

A *client description* contains two lists: The first one describes the *resource publications* with the channel mode, the resource identity, a priority, and the address index/port tuple. The priority field is required in the matching process for arbitration. The second list describes *resource subscriptions* along with the consumer identity, a priority, and address data. A subscription further contains a list of Geminga and client identifier tuples. This enables resource consumers to specify their favoured providers.

This message structure, along with a space efficient encoding scheme, results in reasonably sized announcement messages even if dozens of resources are involved. Section 10.3.2.1 evaluates the scalability of the Geminga announcement message. Thanks to SpicaML and the Aastra model transformation tool, a realisation of the message structure can be generated for a variety of target platforms. It is further possible to authenticate, encrypt, and compress the message.

8.5 Matching

A Geminga service maintains a repository for the most recent resource offers and requests (*resource description*) of remote as well as local clients. The entries of the repository are indexed in several ways. Each index thereby represents a search criterion that references a specific characteristic. We will restrict the examination of indices to only those that are essential for the matching process of Geminga. Some more indices may be used by an implementation for internal maintenance purposes.

Two indices consider all published and subscribed resources, taking into account four fields that uniquely identify a resource description: The *Geminga and client identifier tuple*, the *resource identity*, the *channel mode*, and the *address* of a resource. Address comparison considers address scope equivalence. Once a local client requests a certain resource, remote and local offers are tested for a match. If no match was found, Geminga just continues with its operation. In case of more than one result, an arbitration step may be carried out that takes into account the priority specification. However, this is only required by consumers to restrict the number of providers. The peer specification finally filters out the forbidden clients.

This procedure is executed whenever a new announcement is received, the local clients have changed their configuration, or a local timer elapsed because of communication problems. Changes are so detected and may be signalled as required.

GCNP events are triggered at a local client if a new resource match was found (`'add'`) or a currently subscribed resource changed its configuration (`'modify'`). Geminga furthermore monitors the last updates of resource descriptions. An appropriate index on the timestamp values of resource descriptions sorts out the oldest ones. Resource offers or requests not updated for a given amount of time are invalidated immediately and removed from the repository. The respective clients are signalled through the respective GCNP event (`'del'`). Explicit unsubscription is not supported.

Since resource providers and Geminga instances are not guaranteed to always sign off cleanly but rather may quit without notification, this automated pruning is a very important feature.

8.6 Summary

Geminga is a simple yet powerful resource discovery scheme that realises a beacon-based resource announcement approach. It explicitly dispenses with elaborate negotiation and notification protocols in order to be less sensitive to networking issues. This renders the Geminga especially useful for application in unreliable communication such as mobile ad hoc networks.

Geminga is realised as a service that must be executed by every participant. It implements a client-server architecture that allows local clients to use almost any programming language. The interaction between clients and the Geminga service relies on GCNP, a simple text-oriented protocol with support for asynchrony. This platform-independency renders it especially suitable for the application in a Spica-based communication infrastructure.

Part III

Evaluation

9 Application

"As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications."

— David Lorge Parnas
Evaluation of Safety-Critical Software, Communications of the ACM, 1990

The last chapters introduced and discussed the Spica development framework extensively. It provides modelling support for dynamically reorganising, message-oriented communication infrastructures by covering message and data flow modelling in a platform-independent fashion. Geminga handles reorganisation and adaptation during runtime. This chapter illustrates the applicability of Spica by three examples.

The *Carpe Noctem Software Architecture* (CaNoSA) is a distributed, modular software framework with ready-to-use application and driver components for robotic appliances. It explicitly addresses heterogeneity. This is required mostly due to its modular structure and the different programming languages that are involved (C++, C#, and Java). While being an RCA for the Carpe Noctem soccer robots, it also serves as a testbed for the Spica development framework. The entire communication infrastructure of CaNoSA thereby relies on the functionality provided by Spica. This includes not only the communication between robots but also in particular the communication between modules on the local host. Interoperation with other RCAs requires appropriate Spica adapters that interface the involved software architectures. This is illustrated on the basis of the following software framework.

Miro, as introduced in Section 3.1.1 is a comprehensive middleware solution that facilitates the development of robotic appliances and the integration of heterogeneous resources. Based on CORBA, it employs its own approach for building communication infrastructures. Miro as such is no dedicated RCA but a middleware framework on which further developments may be based on. The University of Ulm's robotic soccer team *The Ulm Sparrows* uses it. Ulm formed a mixed team together with the University of Kassel's Carpe Noctem robotic soccer team during the RoboCup World Championships in 2006, conducting the first interoperability testing for the new Spica development framework.

CaNoSA not only provides an RCA for controlling real robot hardware but also includes a simulation module that acts as a drop-in replacement for robotic hardware. Because of its modular design, the simulator can be replaced by any other simulation environment. As an example, we adopt Gazebo [77] for integration into CaNoSA. Spica thereby assists by providing an appropriate adaptor module. As a part of the Player/Stage project [51, 133], Gazebo complements Stage with more realistic simulation capabilities.

Figure 9.1: The Carpe Noctem robotic soccer team, Distributed Systems Group, Kassel University

Below, we first introduce CaNoSA and its most interesting characteristics regarding the Spica development framework. The considerations and examples regarding the integration of Miro and Gazebo follow thereafter.

9.1 Carpe Noctem Soccer Robots

CaNoSA, the RCA for the Carpe Noctem soccer robots is custom made and tailored to the needs of the hardware and the requirements determined for the software. Several RCA solutions are available that address multiple standard hardware platforms in a generic fashion [129, 128, 134, 51, 24, 45]. In case of a custom-made hardware platform, however, it is typically easier to implement a limited but in most aspects sufficient functionality than adapting an existing solution. A reason for that is the fact that very specific problems may occur during development of new hardware and software, so adaptations of existing solutions may complicate or delay the overall development process. This is why custom-made software platforms are often favoured for research and development and will presumably prevail even in the future. This is especially true for robotic appliances where hardware platforms are still evolving.

Figure 9.1 shows the Carpe Noctem team consisting of undergraduate and postgraduate students as well as six autonomous robots. We are most interested in robotics research that deals with cooperation and collaboration of multiple autonomous mobile robots. Interoperability and solutions to heterogeneity issues are one of the most prevalent challenges here. A communication infrastructure builds the foundation for cooperation and bridges the differences between systems. It must be well defined and tailored to the given application domain. The Spica communication infrastructure complies with these requirements and further integrates well into the CaNoSA structure. It directly addresses interconnection of software components, the location and availability of which is typically undetermined.

During the last years, software distribution and communication management have not been important topics in the robotic soccer community. Attention has mostly been given to artificial intelligence and vision processing approaches exclusively. Communication finally found its way into research with the event of team cooperation and collaborative problem solving. It is still getting even more important because cooperation depends on information exchange. Coordination that exclusively relies on vision approaches is not yet feasible. Naturally inspired interaction based on speech is currently no option as well.

The ability for communication and cooperation was available in CaNoSA ever since. Because of the completely modularised software architecture, functional entities are forced to communicate anyway. Certain components have furthermore been realised in a more problem-specific fashion, i.e. in a programming language that is better suited for the application domain, for example. Integration of these modules thus depends on a development approach that actively addresses heterogeneity issues. The Spica development framework not only targets the development of somewhat specialised software architectures that mostly depend on in-house developments. It furthermore facilitates integration of existing third-party solutions. Heterogeneity plays an important role in both cases and often hinders interoperability. Spica thus covers it in a transparent fashion.

We will now outline the design decisions for CaNoSA and the goals behind the Spica approach used for this purpose.

9.1.1 CaNoSA Architecture

The structure of CaNoSA is only roughly modelled after a formal specification. It is rather grown by time and demands. Several self-contained components thereby provide the basic RCA functionality. These are, for instance, a *Vision* component that controls a camera or a *Motion* component that is responsible for the robot movement. Each component thereby requests or offers a number of resources.

In order to build an RCA from a set of independently operated components, they have to be composed appropriately in a consistent fashion. This is where Geminga comes into play. CaNoSA thereby realises a SOA with Geminga acting as a kind of service broker.

CaNoSA's internal processing is inherently event-oriented. This is partly because of the sensors that exert direct influence on the data processing behaviour. The decision making process nevertheless contributes as well: It has to be discretised somehow, so that data supply and exchange will take place only at discrete points in time. We will therefore rely on a message-oriented communication behaviour between all involved components, even for components running on the same host. The application of platform-dependent IPC schemes such as *Shared Memory* or *UNIX Domain Sockets*, for example, has been rejected in favour of a common, homogeneous communication infrastructure mainly based on UDP. With TCP, a connection-oriented transport protocol is provided as well.

9.1.1.1 RCA Facilities

We will now have a look at the components provided by CaNoSA and their contribution to RCA capabilities. Figure 9.2 outlines the structure of CaNoSA schematically. Because of the dynamic and adaptive nature of the communication infrastructure, this figure does not

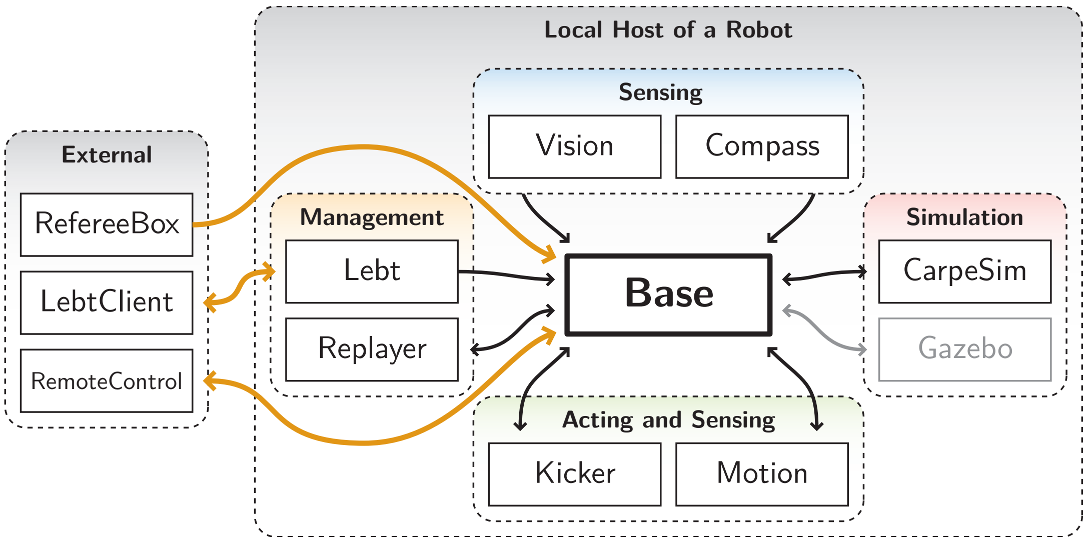


Figure 9.2: The CaNoSA structure. It comprises several self-contained modules, some of which serve similar purposes. Connections to external components are visualised by thicker orange arrows.

show all possible communication channels individually but abstracts by sketching data flow directions only.

The components of CaNoSA can roughly be classified into six categories:

- **Base**. The *Base* module is a sort of central control instance. It cannot be compared to the other component types, so it spans a class of its own.
- **Sensing**. The *Vision* and *Compass* components are typical sensor modules as they are responsible for collecting information regarding their environment. Here, data flows unidirectional towards the Base but normally not in the other direction.
- **Acting**. *Motion* and *Kicker* are mostly actuator components that accept commands from the *Base*. As most actuators depend on sensor information, they may be classified as sensor modules as well. Data flow is possible in both directions.
- **Simulation**. The *CarpeSim* module acts as a substitute for sensors and actuators. Gazebo can be optionally added as well for this purpose.
- **Management**. *Lebt* and *Replayer*[1] are supplementary management components, not directly involved in robot control but in controlling the RCA. Replayer is responsible for recording the entire communication and replaying it at a later point in time. This is why it potentially replaces all the other components.
- **External**. *RefereeBox, LebtClient, and RemoteControl* module are located outside a robot and directly interface the management components as well as the Base. Optionally, direct connections to other components are allowed as well.

We will now have a closer look at the classes and the associated components. It must be emphasised, however, that the Base is by far the most important module with regard to communication complexity.

[1]Replayer was developed by Florian Seute, a member of the Carpe Noctem robotic soccer team.

Base The *Base* is the main processing unit at which the most communication channels converge. It is responsible for behaviour control and world modelling, thus forming a quite complex module with monolithic characteristics. The decision for combining two fundamental facilities into a single module is because behaviours access world model data very frequently. Sharing the address space thus helps lowering data access latency and in turn optimises reactivity. Inter-process communication is furthermore expensive, regardless of which interaction scheme is used.

The *Behaviour Engine* (BE) implements hierarchical behaviour control facility similar but not equal to the BAP framework [131]. Self-contained, functionally limited entities of basic behavioural actions form the fundamental building blocks of most modern robotic behaviour modelling approaches. Such a building block is typically called *behaviour*, examples of which are: *Approaching the ball*, *kicking the ball*, or *driving to a given point*.

In the BE, one or more behaviours are executed concurrently within a so-called *Context*. Only one instance of a Context may be executed at a time. It thus represents the current capabilities of a robot. In order to model the behaviour of a robot over time, several Contexts are combined to a so-called *Policy*. A Policy is a realisation of a state machine in which the Contexts resemble states. Transitions are triggered by internal or external events. This basic functionality already allows modelling the behaviour of a robot. Additional hierarchy levels are advantageous for more complex processes as proposed by Utz et al. [131]. For this purpose, the BE introduces so-called *Global Policies* or *Globals* for short. They allow structuring the behaviour of a robot in a more abstract fashion: States in the Global Policy are represented by Policies that, in turn, maintain the Contexts as their states. Experience has shown[2] that a two-level policy is sufficient for modelling most scenarios for robotic soccer.

The world modelling capability as the second component in the *Base* follows two purposes: It stores and processes locally acquired environmental data and maintains data received from other robots. These two domains result in two different world model structures: The *WorldModel* is responsible for locally acquired data that are further processed, representing a snapshot of the robot's environment at a discrete point in time. Updates at the WorldModel occur whenever new sensor information is available. With a typical operational frequency of 50Hz for the motion and 33Hz for the remaining components, a lot more than 50 updates will occur each second.

The *SharedWorldModel* is responsible for the information provided by other robots. It mainly provides storage and data fusion capabilities. Every robot typically distributes a selection of its local state to all the other robots. The message used for this purpose is called `SharedWorldInfo` and defined in Listing 9.1 (lines 6–10). The `observation` list (line 7) in the same listing accepts SpicaML containers that are derived from `DataContainer`. The `signals` list (line 8) is used for signalling local events. `RobotSignals` is an enumeration of predefined signal constants.

A command added to the `commands` list (line 9) is used to trigger the execution of a function at a remote robot. It accepts arbitrary containers derived from `Commands`. Their types indicate which command to execute; the instance further provides the respective parameters. The distribution frequency is limited to only 10Hz in order to reduce the network load caused by

[2]The Carpe Noctem robotic soccer team successfully participated in three RoboCup tournaments between 2005 and 2008. We were thereby require to create quite complex policies for team coordination purposes. They still remained maintainable, even though we only had this limited two-layered policy approach available.

```
enum RobotSignals : int16 { Undefined = -1; GoalieIntercept = 0; ... }

container DataContainer; // Observations of a robot
container Commands; // Commands for other robots

message SharedWorldInfo : Header {
    DataContainer[] observations;
    RobotSignals[] signals;
    Commands[] commands;
}
```

Listing 9.1: The SharedWorldInfo message used for robot communication

robot communication. This mainly takes into account that the overall network load depends linearly on the number of participating robots.

The local state of a robot includes the robot's own position, a collection of observations, and optionally commands that have to be executed by specific receivers. By considering that all available robots are distributed over the playground, this additional sensor information may be used by the receiver to obtain a more detailed picture of the current situation.

It must be noted here that latencies affect the expressiveness of remote but also of locally generated sensory input. It depends on the latency of the respective sensors and on the latency of the communication media how fresh data is. Freshness directly relates to applicability in this case. A camera sensor, for example, will most likely involve a latency of about one frame, i.e. 33ms for a 30Hz rate. Processing and communication further add several milliseconds. Additional latencies imposed by network message retransmission, for example, should further be avoided. The *WorldModel* as well as the *SharedWorldModel* must thus be able to cope with increased latencies and unreliability in general. Outdated information must be either discarded or weighted according to their importance.

Vision The *Vision* module represents the most important source for sensor information for the Carpe Noctem robots. It interfaces the camera, captures and processes images, and delivers the extracted and interpreted information to the *Base*.

The main camera device of the Carpe Noctem robots is connected via IEEE 1394b [71, 62], delivering YUV422-encoded images. The *Vision* module has also built-in support for USB cameras. IEEE 1394 is nevertheless preferred because of its real-time capabilities as mentioned in Section 2.1.1.1.

After retrieving the image, it is passed to an image filter chain. It is first converted into grey and chrominance images where the chrominance image has a modified colour space. The grey image is required for extracting obstacles, line points for self localisation, and a distance profile. The ball is extracted from the chrominance image, as it is the only colour encoded object. The position of the robot is calculated based on the extracted line points using a particle filter.

The representative information of every extracted feature is assigned to a dedicated container modelled with SpicaML. All containers are then sent to the *Base* in a single message. This guarantees that the features arrive concurrently and expenses for transmission and reception are minimised.

Compass A football field is typically symmetric with mostly ambiguous landmarks through which a robot cannot determine its orientation. This is why soccer playing AMRs typically integrate an electronic compass. It does not provide the exact orientation but it is sufficient for distinguishing the two possible directions. The *Compass* module in CaNoSA provides an interface to a compass hardware module and constantly delivers measurements.

Motion and Kicker The *Motion* and *Kicker* components interface the respective actuator components of a robot, i.e. the motion and kicking devices. Both components exhibit a three-layered software design: On the topmost layer, they receive commands and prepare return messages. The middle layer converts the command values into a device driver-specific representation. The third layer represents the hardware driver. It directly interfaces the hardware and is structured in a modular fashion so that the driver implementation may be replaced. This guarantees that only the driver part accessing the hardware must be exchanged if new hardware is added.

The *Motion* currently supports two Fraunhofer motor controllers: The TMC200 and the VMC.[3] The *Kicker* module interfaces two custom-made kicking mechanisms. One Kicker is based on pneumatics; the other resembles characteristics of a rail gun.

Simulator The *Simulation* components act as a drop-in replacement for the *Vision*, *Compass*, *Motion*, and *Kicker* components. CarpeSim follows a very simplistic, two-dimensional approach with no concrete physics simulation. It is capable of simulating arbitrarily many playing fields with again arbitrarily many players. CaNoSA further provides a basic interface to the Gazebo simulation engine.

Replayer The messages exchanged between components and robots represent the entire environmental information of all robots. The *Replayer* module hooks into the communication channels, wiretaps all messages, and stores them to disk with appropriate time indices. At a later point, these messages may be replayed, simulating the recorded situation.

Lebt, LebtClient and RemoteControl The *Lebt* module is responsible for managing all the other components on a robot. It is thereby responsible for starting, stopping, and monitoring other processes. Lebt also monitors the system it is running on and provides clients with up-to-date information on the system status. *LebtClient* is a graphical interface for Lebt, visualising the data exchanged between the robots and the information provided by the respective Lebt components. The *RemoteControl* module provides an interface for game and process control. It is mainly used by LebtClient but resembles a generic interface that may be used by other applications as well.

RefereeBox The *RefereeBox* module accepts commands from the *RoboCup MSL refbox*,[4] a software program that is used by RoboCup referees to control the robots during a game. The RefereeBox module accepts the commands coming from the refbox and translates them into commands for the CaNoSA RCA.

[3]The TMC200 and the VMC motor controllers have been developed by *Fraunhofer Institute for Intelligent Analysis and Information Systems (IAIS)*, `http://www.iais.fraunhofer.de/` (accessed 2008-08-30).

[4]`http://sourceforge.net/projects/msl-refbox` (accessed 2008-08-30).

```
 1 message CompassMessage : Header { CompassInfo body; }
 2 container CompassInfo : DataContainer [compare=value] { int32 value; }
 3
 4 module CompassProvider [monitorable] {
 5   offer message CompassMessage [enabled] -> Vision, Base, CompassConsumer
 6     scheme otm/pubsub
 7     dmc ringbuffer out [size=1];
 8 }
 9 module CompassConsumer [monitor] {
10   request message CompassMessage [enabled] <- Compass
11     scheme otm/pubsub
12     dmc ringbuffer in [type=CompassInfo; size=1];
13 }
```

Listing 9.2: A SpicaML model for a compass value producer and consumer example.

9.1.2 Spica Integration

We will now show how to create a simple resource provider and consumer pair. The Compass component is thereby used as an example. A server component (`CompassServer`) will gather compass data from a compass device and send them to all interested consumers. For this purpose, we also create a dedicated client component (`CompassClient`).

Listing 9.2 presents the definitions of the message (line 1), its body (line 2), the module definition for the provider (lines 4–8), and the module definition for the consumer (line 9–13). It is important to mention here that the consumer takes advantage of the automatic container extraction capability as introduced in Section 5.4.2 (line 12). Both modules activate the monitoring capability.

Four class files will be created from the SpicaML model if aiming at an implementation in C#: `CompassMessage.cs`, `CompassInfo.cs`, `CompassProvider.cs`, and `CompassConsumer.cs`. The server (`CompassServer`) basically instantiates and activates the `CompassProvider` module stub. Apart from compass values, the server will furthermore provide logging information. The code snippet shown below exclusively considers the functionality required by the `CompassServer` for the integration of the `CompassProvider` stub.

```
public class CompassServer {
  protected CompassProvider comp = null;

  public CompassServer() {
    this.comp = CompassProvider.GetInstance();
    this.comp.Open();
  }
  protected void OnError(string msg) { this.comp.Mon.Error(1000, msg); }
  protected void OnNewData(int data) {
    CompassMessage cm = new CompassMessage();
    cm.Body.Value = data;
    this.comp.Out.Insert(cm);
  }
}
```

If supported by the target language, a SpicaML module definition is realised as a singleton [48]. Even though this pattern is often considered harmful,[5] it is well suited for the application in CaNoSA: Components typically exhibit a modular internal structure themselves, so a reference to the stub instance must be available in different places. A singleton accomplishes this almost transparently. Developers of existing modules will further appreciate such an approach, as it does not require fundamental modifications to the implementation.

The `CompassServer` component first instantiates (line 5) and then initialises (line 6) the `CompassProvider` stub. Once an error occurs, the `OnError` method is called which passes the error message down to the monitoring provider implementation, assigning a priority value of 1000 (line 8). If a new compass value is available, the `OnNewData` method creates a new message (line 10), assigns the respective value (line 11), and triggers delivery (line 12).

The consumer (`CompassClient`) exhibits a similar structure, yet it does not provide any information for other components. It rather consumes incoming data in an asynchronous fashion. The listing below roughly sketches the realisation.

```
 1  public class CompassClient {
 2    protected CompassConsumer comp = null;
 3
 4    public CompassClient() {
 5      this.comp = CompassConsumer.GetInstance();
 6      this.comp.Monc.Subscribe("CompassProvider", OnError);
 7      this.comp.In.Added += OnNewData;
 8      this.comp.Open();
 9    }
10    protected void OnError(MonitoringToken t, Service s) {
11      Console.WriteLine("Error: {0} from {1}", t, s);
12    }
13    protected void OnNewData(CompassInfo ci, Service s) {
14      Console.WriteLine("Compass: {0} from {1}", ci.Value, s);
15    }
16  }
```

Spica module stubs are all initialised in the same fashion. Additional steps have to be completed only if messages or monitoring data are requested. This is because the module stub needs to know where to deliver the respective information. The `CompassClient` therefore registers two callback methods: Monitoring data are sent to the `OnError` method (lines 6, 10–12), `CompassInfo` containers to the `OnNewData` method (lines 7, 13–15). Newly created and existing components may so be equipped with elaborate communication capabilities in only a few lines of code.

The architecture layout of CaNoSA already points at the resemblance to a SOA. A Spica-based communication infrastructure acts as the underlying communication layer: It interconnects components and cares about platform-dependency and interoperability issues. Because CaNoSA is the main testbed of the Spica development framework, it proves applicability and demonstrates Spica's capabilities. It further points to weaknesses in the development approach.

CaNoSA components directly relate to module definitions of a SpicaML model as shown in the example above. Relationships between modules are thereby based on offered or

[5]A singleton is not guaranteed to be instantiated only once, depending on the programming language. It furthermore introduces limitations and is overly used in most cases.

requested resources. CaNoSA does not prescribe a specific programming language. Every module may rather adopt the language or platform that is best suited for the respective application. This is where SpicaML's platform independence brings the most benefit. Module stubs further care about the configuration of the communication infrastructure and address interoperability issues. The abstract SpicaML models separate the organisational expense from the actual functionality of modules this way, realising a kind of model-view separation.

Spica effectively promotes the trend towards modularisation and clearly brings further advantages: Modules may be exchanged and replaced by other modules that exhibit similar resource offers but implement a different behaviour. Figure 9.2 hints at the possibility that Gazebo may replace the *CarpeSim* simulator component. Both thereby need to deliver a similar service, i.e. they need to share at least some matching message types. Section 9.3 below outlines the integration of Gazebo with CaNoSA. The Spica communication infrastructure further facilitates distributed operation of modules. A central server typically runs a simulator. The *LebtClient* modules, in contrast, are executed on different laptops, for example. The location of modules is irrelevant as Geminga cares about the automatic configuration of communication channels. Changes in the location are also handled transparently.

The decision for module bindings is based on offered and requested resources. The Geminga service is then able to deduce possible pairing relations. Priority arbitration may be applied in case of multiple alternatives. Thanks to the specification of allowed peer modules, it is further possible to restrict the number of viable peering partners as outlined in Section 8.5.

Messages are the most important elements in CaNoSA as they build the foundation for platform-independent information exchange. Modularity and heterogeneity are the main driving force towards more abstract specification capabilities. CaNoSA actually relies on SpicaML for the specification of its most important internally used data structures. On the one hand, this guarantees for a consistent data representation beyond the scope of a single language. Data exchange between different modules, on the other hand, is supported implicitly without the need for additional data conversion or reorganisation. When integrating third-party components, however, data conversion will most likely be required because of differences in the internal data representation. Not only different representations have to be considered but the development philosophy is an important factor as well.

Platform independence is clearly one of the major strengths of the Spica development framework. Geminga further contributes by providing automatic configuration of potentially complex communication infrastructures. It is nevertheless still not possible to bridge the gap between mutually incompatible software architectures in a completely automated fashion. Many approaches, Spica included, merely shift the development expense instead of reducing it. The Spica development approach is intentionally not completely automated. This is because the generated interface stubs need to be applicable for a broad range of components. As it is not yet possible to provide automatic data conversion in an efficient way,[6] we decided to leave the realisation of the data conversion to the developer.

We will not present the SpicaML model for CaNoSA here, mostly because of its complexity. Instead, we will have a look at a data-encoding scheme provided by Spica and adopted by CaNoSA: The Carpe Noctem Encoding Rules.

[6]Data conversion based on semantic annotation and ontology-based knowledge representation is an expensive and complex approach. It is supported but currently not used in Spica.

9.1.3 Carpe Noctem Encoding Rules

The Spica development framework provides a basic SERI scheme, suitable for the application in resource limited systems and computer networks. It realises a compact, binary data encoding that is comparable to the *Common Data Representation* (CDR) format used in the CORBA [98] middleware. The *Carpe Noctem Encoding Rules* (CNER) adopt this approach by slightly modifying the encoding of length fields.

Signed and unsigned integers as well as floating point values are encoded as is without altering their endianness. By following the RMR approach, CNER saves computation time for the sender and in most cases for the receiver as well. In the worst case – i.e. if the byte orders of the involved systems differ – only the receiver has to change the data representation. The sender does not have to take further action.

Strings and arrays represent variable-sized, bounded lists with less than 100 elements in average. Support for larger numbers of elements must nevertheless be available. CDR applies 32bit length fields in this case. Fixed arrays are encoded directly without declaring their length. CNER does the same but employs an indirect length declaration where the size of the length field is not fixed. Depending on the number of elements it requires at least 1B and at most 9B.

Encoding Length values of up to 2^7 are encoded directly without further processing. Length values greater than $2^{7 \cdot i}$, with $i > 0$, are split into i parts, each of which is at most 7bit in length. Bit 8 of parts $0 \ldots i-2$ is set to 1. For part $i-1$, Bit 8 is set to 0, marking the end of the indirect encoding. A part is typically stored in a byte.

Decoding As long as Bit 8 of the current byte equals 1 there is at least one more byte to consider. If the MSB equals 0, there are no further bytes that contribute to the length value. The lower 7bit of each such byte are added to the decoded length value accordingly.

The length encoding is limited to at most 9B which relates to a length value of 2^{63}. This should be enough for almost any application. With this approach, short arrays or strings with less than 128 characters only require one single character as a length identifier. The additional overhead introduced by the indicator field is insignificant in most cases. CNER encodes strings always in UTF-8; no other encoding convention is currently supported.

9.1.4 Summary

The CaNoSA RCA targets groups of AMRs that operate in highly dynamic, distributed environments. Its modular structure resembles characteristics of a SOA where several self-contained modules are composed to an integrated whole. The Spica communication infrastructure facilitates module interaction. It explicitly addresses heterogeneity issues and thus facilitates cooperation of mutually incompatible software solutions. A SpicaML model thereby captures data structures and the layout of CaNoSA in an abstract and platform-independent fashion. Geminga maintains the configuration of the Spica-based communication infrastructure and cares about the module composition. The CNER data-encoding scheme addresses efficiency in data representation and data processing.

The generated module stubs do not cover data conversion between mutually incompatible components. It must rather be implemented manually. Spica simplifies this task by providing appropriate measures that deal with heterogeneity issues and the configuration of the complex Spica-based communication infrastructure. A developer may thus concentrate on the functionality.

We will now roughly outline where and in which way CaNoSA adopts modules and SpicaML data structures. The overall number of SpicaML modelling elements thereby breaks down into the following numbers: There is 1 header defined for 22 messages that are exchanged between modules and robots. Together 9 enumerations and 62 containers are available, a part of which is used exclusively within the RCA of a single robot. CaNoSA finally employs 13 independent module definitions, including the ones required for external modules.

In the next section, we outline how to integrate the Miro middleware architecture into the Spica communication infrastructure, facilitating cooperation between the Miro and CaNoSA.

9.2 Miro

Miro [129, 128] is an object-oriented middleware architecture for robotic appliances. It relies on CORBA and so facilitates software distribution is a standardised manner. Miro further adopts the CORBA Notification Service that provides asynchronous event distribution channels (*Notification Channels*). A special data structure called *Structured Event* carries data in this case, facilitating data filtering in a Notification Channel. Section 3.1.1 further introduces and discusses the architecture of Miro.

In the course of this section, we will investigate whether it is possible to integrate Miro with a Spica communication infrastructure. More precisely, we want heterogeneous soccer robots to play together cooperatively by mutually exchanging world information.

9.2.1 Miro Communication Infrastructure

The starting point for the cooperation between Miro and CaNoSA is the CORBA Notification Service. This is because the entire communication within Miro relies on this facility. Communication with other robots basically follows the same principles as outlined in Section 3.1.1.

A *Structured Event* is only capable of carrying CORBA-based data structures. It consists of a header and a body. The header is divided into a fixed and a dynamic part. The fixed part specifies *a domain name*, *a type name*, and *an event name*. Miro initialises the domain name with the hostname of the sender. It further assigns the type name a description of the data structure to be distributed. This typically involves the data structure's CORBA type name. The event name is initialised with an empty string; the dynamic header is left empty.

The body of a structured event is divided into a list of filterable fields and the payload. Miro only uses the payload and leaves the list blank. The payload accepts CORBA Any objects, that is, serialisable data structures defined in CORBA IDL. It is used for event distribution.

Every Miro instance provides a private CORBA Notification Service. In order to achieve event distribution in a group of robots, all Notification Channels are required to connect to each other. Communication is hereby based on the connection-oriented TCP protocol. This results in an expensive, complex, and error-prone setup. The transmission complexity is about $\mathcal{O}(n)$ for a single host and $\mathcal{O}(n^2)$ for the entire group.

```
 1 module SpicaAdapter [monitorable] {
 2     offer message SharedWorldInfo
 3         -> SpicaAdapter,Base
 4         scheme otm/stream
 5         dmc ringbuffer swi [size=1; export];
 6
 7     request message SharedWorldInfo
 8         <- SpicaAdapter,Base
 9         scheme otm/stream
10         dmc ringbuffer[] shwm [size=1; ttl=5s; export]
11         dmc ringbuffer[] shwmBall [type=BallInfo; size=1; ttl=5s; export];
12 }
```

Listing 9.3: SpicaML model for a Miro adapter module

To overcome this situation, Miro introduces *NotifyMulticast*, a solution for event distribution that is much more efficient in terms of network usage and management overhead. By adopting multicast communication, it reduces the transmission complexity to $\mathcal{O}(1)$ for a single robot and $\mathcal{O}(n)$ for the entire group. This approach does not only save communication bandwidth but also addresses network unreliability and exhibits a fault-tolerant behaviour. Thanks to the unreliable transport protocol UDP, data are not delayed and transmissions will never block in case of network failures. However, this implies that data may get lost.

The *Shared Belief State* (SBS) specification [130] facilitates cooperation in a group of AMRs. It is used for the distribution of a robot's belief among the members of a group of robots. Belief in this case refers to what a robot thinks about its environment, that is, estimated positions of objects, their estimated covariances, or other information related to environmental sensing and belief. The SBS is one possible data structure for the Miro event distribution approach. It is typically distributed via NotifyMulticast.

9.2.2 Spica Integration

We will now present the measures that have to be taken for interoperation between Miro and Spica. First, a common agreement on the information exchange must be accomplished. Let us assume we have two teams of soccer playing robots: The first team adopts Miro, the second team uses CaNoSA. Two different procedures may be used then: Each team may use its very own measures for cooperation, thus keeping at their own information sharing data structures. For Miro this would imply SBS, CaNoSA would stick with its `SharedWorldInfo` message as defined in Listing 9.1. The information exchange between the two teams would then imply that at least one of the two teams had to implement the other team's structure.

This is also required for the second alternative where both teams agree on a common data structure for the entire information exchange. We decided in favour of this alternative, mostly because of the less complex communication behaviour: There is only one message type to be exchanged. We will therefore design a Spica adapter module for Miro that is able to communicate with the CaNoSA modules.

The SpicaML module specification shown in Listing 9.3 is required for this purpose. It consists of a module definition with two message handling directives: The first one (lines 2–5)

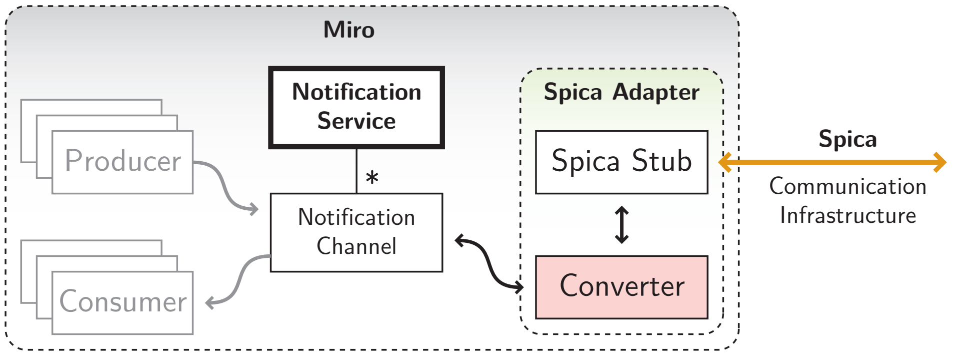


Figure 9.3: The Spica Adapter for Miro

offers the `SharedWorldInfo` message to other modules. The second one (lines 7–11) requests it respectively. The request statement furthermore exhibits an interesting characteristic: The ball position is extracted from the message and inserted into a dedicated DMC automatically (line 11). This way, the Spica adapter module does not have to traverse the message structure but may process the required information directly.

Both statements further restrict the interaction to some well known module types. Only the module types `SpicaAdapter` and `Base` are allowed to receive the message (line 3). A similar restriction addresses permitted senders (line 8). Communication is thus only allowed between Spica adapters for Miro and the *Base* module of CaNoSA.

Spica integrates into Miro just like NotifyMulticast: The Spica adapter module, i.e. a stub with the additionally required functionality for data conversion, attaches itself to the Notification Channel as consumer and supplier. Data then have to be converted between the SBS and the `SharedWorldInfo` message. Data distribution still relies on multicast. The only major difference between NotifyMulticast and the Spica approach is the automatic maintenance of the communication infrastructure. Developers furthermore do not have to take care of data-encoding and management. Figure 9.3 gives an overview of the Spica adapter layout. `Converter` here represents the glue component between the Spica stub and the Miro middleware. Only this one has to be implemented manually.

For global information exchange, the SBS is replaced by the `SharedWorldInfo` message. Even though both approaches exhibit similar objectives, the design of the `SharedWorldInfo` message is much simpler and it is more generic than the SBS. Hence, the Spica adapter is responsible for converting the contents of a SBS into suitable SpicaML containers and attach them to the `SharedWorldInfo` message. The same procedure but the other way round is required when posting information to the CORBA Notification Service. In the robotic soccer domain, mostly the estimated positions of the own robot, the ball, and opponents may be of interest. Uncertainty measures such as covariance matrices are provided by the SBS but are not required by CaNoSA. They are thus left out. Any further dispensable information is omitted as well. Miro still uses the SBS internally.

9.2.3 Summary

Miro is a mature, CORBA-based middleware architecture for robotic appliances. The communication infrastructure builds on the functionality of CORBA, except for event distribution between robots. Miro adopts the CORBA Notification Service but employs a self-developed software that takes care of channel federation. Data distribution is thereby based on multicast. For the interaction between Miro and CaNoSA, we decided in favour of a similar approach: A Spica adapter is provided that interfaces both, the CORBA Notification Service as producer and consumer, and the Spica communication infrastructure. The SBS is then converted into a `SharedWorldInfo` message by the Spica adapter and vice versa.

The integration of the Spica communication infrastructure into the Miro middleware proved to be straightforward, just like with established middleware approaches. Spica, however, explicitly targets communication infrastructures for mobile robots and thus provides appropriate measures to overcome failures. Merely the data conversion has still to be implemented manually. The SpicaML module specification is a generic adapter interface that may be used for other purposes as well.

Below, we discuss the integration of the Gazebo simulation engine with CaNoSA. It will be used as a drop-in replacement for CarpeSim.

9.3 Gazebo

We will now equip the Gazebo simulation engine [77] with an interface to the Spica communication infrastructure, and thus make it accessible for CaNoSA. As a part of the Player/Stage software framework, Gazebo is responsible for realistic, three-dimensional simulations. It adopts the *Open Dynamics Engine* (ODE),[7] a physics simulation environment that simulates rigid-body physics. Gazebo is capable of simulating several robots with sensors and actuators, as well as other objects that may be placed arbitrarily into a simulated environment. Realistic sensor feedback and physically plausible interactions between objects render Gazebo very attractive for robotic appliances. The most limiting drawback of Gazebo is its complexity and therefore the performance penalty it involves. This is not only due to the graphical user interface that renders the simulated world in OpenGL, but a result of the resource demanding physics simulation. Gazebo thus aims at the simulation of a limited number of robots, typically only a few dozens. It nevertheless depends on the available hardware resources if a simulation is feasible or not.

Several robot and sensor models are provided as predefined models that can be integrated right into a simulation environment. The configuration of the simulated world is based on an XML document in which the different components – robots or sensors, for instance – are combined appropriately. A plugin architecture finally promotes extensibility by providing the ability to load and integrate robot and sensor models dynamically during runtime.

9.3.1 Simulation Architecture

The architecture of Gazebo is kept quite simple and very generic, resembling a sort of container for models. It cares about instantiation, connects the models according to their configuration, and provides fundamental functionality such as an interface to the physics engine and simulation capabilities. Figure 9.4 roughly outlines the Gazebo architecture. The hierarchical models dominate the internals of the Gazebo architecture. An interface

[7] `http://www.ode.org/` (accessed 2008-08-30).

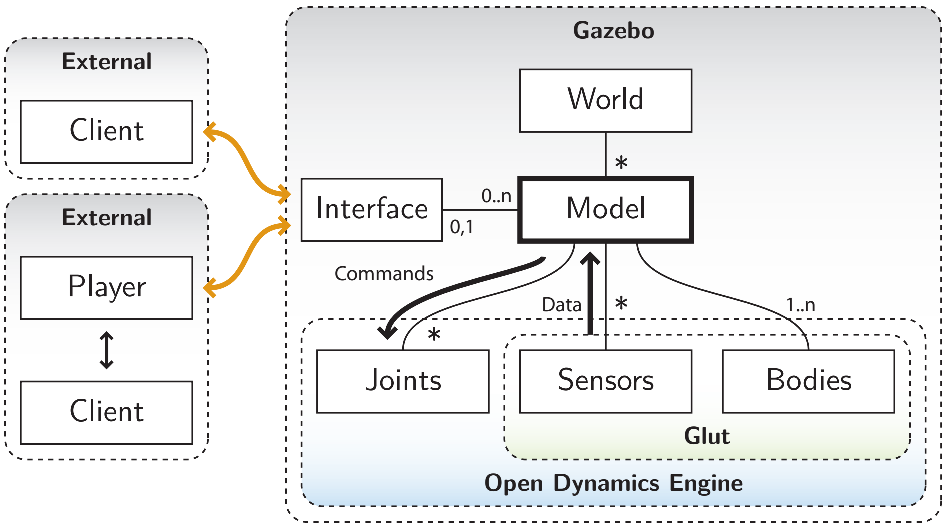


Figure 9.4: The Gazebo architecture with two clients. The first accesses Gazebo directly through shared memory, the other one adopts the Player server.

component provides access for external entities that connect via *Shared Memory*. The two external component packages represent the two possibilities through which Gazebo can be accessed. Our considerations below are based on Gazebo version 0.8.

Each instance of Gazebo maintains a world with several objects. Each object is characterised by a model and represents a self-contained, reusable entity with a dedicated functionality. Models may be as simple as fundamental geometrical entities with no further function. They may also resemble complex aggregations of primitive models or functional copies of real world objects. Examples for such models are laser scanners, infrared sensors, wheels, or even a complete robot platform. Models may be nested, so the complexity of models increases by hierarchical organisation. A laser scanner, for example, may be placed on top of a robot whereas a wheel together with a motor is typically placed thereunder. The relationship between the wheel and the robot is very important in this case as changes in the wheel's or the motor's states have direct influence on the robot. ODE provides so-called *joins* for this purpose. A join is a particular concept that specifies how to interconnect objects. It accepts commands that influence its state. The result then influences the connected objects.

How can an external program access a simulated object? Gazebo provides two different possibilities: Access via Player or through `libgazebo`, a library that accesses the Gazebo instance via shared memory. This choice is outlined in the top left corner of Figure 9.4. Interfacing via Player is the more convenient but also the most expensive alternative. Here, a client program can make use of Player's provided functionality such as remote operation and transparent access to devices. The simulated components appear as if they were real, so a client program will not notice any difference between real and simulated hardware. It is nevertheless more expensive as socket-based communication and further abstraction layers are involved. Even though `libgazebo` involves more effort by the programmer, it basically takes after the Player API.

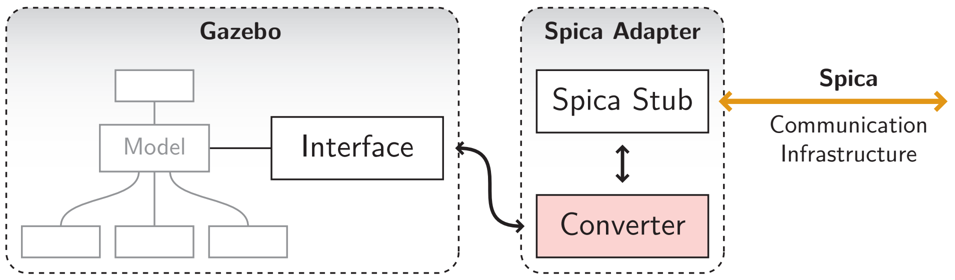


Figure 9.5: Gazebo Added to the CaNoSA RCA

9.3.2 Spica Integration

For integration with Spica we decided to use `libgazebo`, mostly because of its performance benefits. It may further be integrated directly into the respective adapter module. The following reasons clearly justify this decision:

- Gazebo should be used as a centralised simulation platform in CaNoSA. A Spica stub with supplementary functionality for data conversion cares about data distribution and interfacing with the Spica communication infrastructure. It is always located on the same host as Gazebo, typically even in the same address space. Remote operation as provided by Player is thus not required in this case.
- The Player approach would result in yet another process that must be running. Besides its lean appearance, `libgazebo` delivers higher efficiency that is important when simulating several robots. It may be embedded into the adapter module.
- The additional functionality provided by Player is not required within CaNoSA. The `libgazebo` solution thus provides the highest degree of freedom when interfacing a quite different RCA.

Integration with Player would nevertheless be possible with almost the same effort. We again introduce a Spica adapter module that bridges Gazebo and CaNoSA. The interface towards Gazebo is thereby based on `libgazebo`. The adapter resembles the approach used for the Miro integration above. Figure 9.5 outlines its basic structure. Just like the Spica adapter for Miro, the `Converter` component in the Spica adapter for Gazebo bridges between the two incompatible components. It has to be implemented manually as well.

9.3.2.1 Spica Module Specification

Specification of a new module in SpicaML is a straightforward task. For this application, however, it is not necessary to create a new module because CaNoSA already provides an interface module for a simulator. The respective SpicaML specification is shown in Listing 9.4.

The `Simulator` module provides one `WorldModelData` message (lines 2–4) and requests four others: `KickControl`, `SimulatorControl`, `MoveBallControl`, and `MotionControl` (lines 5–16). For the Gazebo integration, we will only discuss the `WorldModelData` and `MotionControl` message types in more detail. The `KickControl` message controls the kicking device and may release a kick. `SimulatorControl` is issued by the *Base* to further customise the simulator instance. `MoveBallControl` finally allows to move the ball within the simulator. The Simulator

```
module Simulator {
    offer message WorldModelData [enabled]
        -> Base [local] scheme otm/pubsub
        dmc ringbuffer simWmData [size=1];
    request message KickControl [enabled]
        <- Base [local], RemoteControl scheme otm/pubsub
        dmc ringbuffer simKickControl [size=1];
    request message SimulatorControl [enabled]
        <- Base [local] scheme otm/stream
        dmc ringbuffer[] simSimControl [size=1];
    request message MoveBallControl [enabled]
        <- RemoteControl scheme otm/stream
        dmc ringbuffer simMoveBall [size=1];
    request message MotionControl [enabled]
        <- Base [local], RemoteControl scheme otm/pubsub
        dmc ringbuffer[] simMotionControl [size=1];
}
```

Listing 9.4: The SpicaML Simulator module specification

module executes a callback method after retrieving a `MotionControl` message. This method has to be implemented by the Spica adapter module where the message is stored into a local variable.

Every offered or requested message type only requires a few lines of code in the Spica adapter module: Providing a new message instance needs only one line for triggering insertion of a message into the respective DMC. For retrieval, a new callback method or interface must be implemented and registered with the respective DMC. In general this requires less than 10 lines of code for management structure overhead.

9.3.2.2 Interfacing a Simulated Robot

The `MotionControl` message is issued by the *Base*, more precisely by behaviours running in the behaviour engine. The provided data must be interpreted and integrated with Gazebo appropriately.

CaNoSA specifies a motion request command as the three-tuple $[\alpha, v, \omega]$ where α is the heading, v the velocity, and ω the rotation of the robot. A robot's desired heading is given in radians in the range of $[-\pi, \pi)$. The velocity is given in mm/s, the rotation in rad/s. This motion value controls an omnidirectional drive. A vanilla Gazebo installation, however, does not directly support omnidirectional control. Even though this functionality can be patched into the source, we will stick with a differential drive for this application, resulting in less degrees of freedom.

Robot models provided by Gazebo implement the `position` interface, allowing to set the desired motion and to retrieve the respective odometry data. Motion requests in Gazebo consist of a three-dimensional velocity vector and roll-pitch-yaw angles. When mapping the movement of a differential drive in flat terrain, only two of these variables are required, namely the velocity in x direction (relates to v) and the yaw angle (relates to ω). As heading is not supported by a differential drive, we will take it as an additional component for the rotation. The concrete realisation is error-prone, as it requires a reduction of the dimensionality. We will therefore skip the implementation and concentrate on the communication interface.

The client implements a main loop in which the interaction with Gazebo takes place in three steps: First, sensor data are read from the simulated world, wrapped into suitable SpicaML data structures, and sent to the robot group as part of a `WorldModelData` message. Offering this message simulates the *Vision* that emits the same message type. Additional data may be wrapped into the `WorldModelData` message, such as the odometry data that is otherwise provided by the *Motion*. In the second step, the received message is processed. The third step synchronises the main loop to a fixed frequency of 33Hz or 100Hz, depending on the desired simulation speed.

No more measures are required in order to interface Gazebo with CaNoSA. The only difficult and error-prone issue is the conversion of different representations and physical units. We have already mentioned that this typically cannot be automated with low effort.

9.3.3 Summary

Suitable measures are provided by the Spica development framework to integrate third party applications into a Spica communication infrastructure. The Gazebo simulator is part of the Player/Stage RCA solution. It provides realistic three-dimensional simulation capabilities and different alternative for client programs to access simulated objects. We have discussed the structure of Gazebo and outlined a possible integration with CaNoSA. As CaNoSA does not depend on a third dimension in space but a third degree of freedom for motion control in the plane, the mapping between CaNoSA and Gazebo is not straightforward and error-prone.

A Spica adapter module was proposed that bridges the gap between Gazebo and CaNoSA. By adopting the existing SpicaML module specification for the CaNoSA simulator, only the implementation of the data conversion had to be provided. The transformation nevertheless involved a lossy conversion, as the motion models of the two components are different. The results nevertheless argue for the applicability of the Spica development framework. Even architectures that are incompatible in almost every respect may so be connected with low effort. Integrating the generated module stub furthermore requires only a few lines of code for every offered or requested message type.

9.4 Availability Information

The Spica development framework is freely available as part of CaNoSA which, in turn, is licensed under a BSD-style license.[8] It is tightly integrated into the software architecture and hence may require some adaptations if used for an entirely different application domain. Depending on the development progress of CaNoSA, Spica may be adapted or modified according to the respective requirements.

The most current version of CaNoSA and all related scientific publications are available online at `http://das-lab.net/research/` (accessed 2008-08-30). An official logo is

[8]The *Carpe Noctem User friendly BSD-Based License* (CNUBBL) is a license under German jurisdiction that grants all rights to a user as long as the author notice is retained. The most current version of the license is available online at `http://das-lab.net/pub/cnubbl.txt` (accessed 2008-08-30).

furthermore associated with the Spica development framework, visualising the name's origin: α Virginis in the constellation Virgo. All relevant image material is also available from the aforementioned address.

10 Evaluation

"Aristotle maintained that women have fewer teeth than men; although he was twice married, it never occurred to him to verify this statement by examining his wives' mouths."

— Bertrand Russell
The Impact of Science on Society, 1952

In this chapter we evaluate the Spica approach and verify compliance with the requirements established in Section 1.2. An extensive evaluation of the functionality provided by Spica and associated tools is essential in order to assess the benefits and simplifications provided by this approach. It is nevertheless difficult to verify to which degree the Spica development methodology contributed to ease of development as it has influence that is more non-functional.

The evaluation approach is twofold: Measurable transmission latencies accompanying a Spica communication infrastructure are assessed if possible. This includes the internal processing, data storage, and the characteristics of the communication infrastructure. The aspects of the Spica development framework that cannot be measured and interpreted empirically are opposed to alternatives. Their applicability and efficiency is justified and better-suited approaches are proposed.

The experimental evaluation is based on implementations written in only two languages: C# and C++. Java as the third possible language has a specific weakness that complicates evaluation: It is not possible to retrieve the current time in sub millisecond granularity. As we need this functionality to measure communication latencies and compare it to the other implementations, Java is not taken into account. A possible solution to this problem would be to use *Java Native Interface* (JNI) and implement a call to the `gettimeofday()` function in the standard C library. We will not go this way but assume C# approximately as efficient as Java.

This chapter is organised as follows. First, Section 10.1 discusses the applicability of the Spica approach, taking into account the pros and cons of MDSD, its extensibility, and the simplification it delivers. Section 10.2 then highlights Spica resource utilisation and compares these results to reference implementations or other approaches. Section 10.3 and Section 10.4 discuss the applicability of Spica in distributed architectures and its accordance to the transmission behaviour of autonomous mobile robots. Section 10.5 closes with a summary of the results.

10.1 Applicability

In order to justify the Spica development approach, it is important to scrutinise its underlying design paradigm at first. MDSD brings advantages and simplification but in return still implicates development and management effort to be accomplished.

10.1.1 Model Orientation

It is indisputable that model orientation and the MDSD methodology as a whole simplify the software development process. This is mostly because of the abstraction from underlying architectures that assist developers; they do not have to take care about highly platform-dependent characteristics anymore. Most development approaches employ model orientation anyway: Modelling, model transformation, and code generation are typical steps involved in high-level programming based on languages such as C/C++, C#, or Java, for instance. An MDSD approach now applies the same concept to these languages, placing another layer of abstraction on top. This creates a sort of meta-modelling approach, realising top-down development with increasing levels of detail where transitions between abstraction layers are carried out automatically where possible.

An increasing number of abstraction levels nevertheless hide an increasing number of details of the underlying architecture. The applicability of a highly abstract and thus generalised development approach is limited but may in return provide a better representation of the aforementioned application domain because of dedicated modelling capabilities. This implies that in order to cover a broader application domain, several specialised modelling approaches might be required, though.

The presented pros and cons for the model-driven approach are not the only ones. Several more voices assess model orientation in the one or the other direction. They are summarised in the following two subsections.

10.1.1.1 Weaknesses of MDSD

MDSD is a very generic methodology. Modelling as such is used in many areas and is well established. The expenses required by a model transformation and code generation approach, however, are not considered in the discussions above.

Sneed [124], for example, critically analyses the *Model-driven Software Evolution* (MDSE) process by giving several reasons why the model-driven approach should be considered harmful. The statement of the paper reduces to two main argumentation blocks that are considered the most important rationale against model-driven software development and evolution:

- Model orientation hides many implementation details from the developer. This includes computationally expensive operations and errors in the generated code. Due to the different levels of abstraction, it is hardly possible to apply rigid experimental and formal testing procedures. The generated code may further not behave as expected, depending on the model and the realisation of the generated functionality.

- Different abstraction levels imply redundancy and increased maintenance effort. This is partly because code generation typically resorts to some type of template-based code generation approach that obviously tends to promote redundancy. The other reason stems from the fact that precisely the code generation is complicated and error-prone because of its complexity. It is thus difficult, sometimes even impossible, to realise the code generation process so that it never has to be touched again. Assuming, for example, flaws in the generated code or API changes in an associated library that require the generated code to be modified. Changes like this or in whatever shape may again lead to unintended behaviour.

These reasons illustrate already that the development effort is shifted from the *software development* to the *creation of transformation rules* for unified, abstract models onto arbitrary target platforms. In the last decade, the Java technology introduced the very promising *bytecode* approach. It is a kind of hardware and software platform-independent representation of a more or less abstract model. A runtime environment typically maps bytecode to a concrete target platform before or during execution. Microsoft .NET refined this concept by allowing several programming languages to map to an intermediate code representation, called *Microsoft Intermediate Language* (MSIL) in the Microsoft .NET context. Both solutions depend on a runtime environment for every target platform. These approaches clearly link model orientation and traditional programming which itself is model oriented again.

With MDSD, an abstract model is interpreted, transformed, and mapped onto a concrete programming language and execution platform. The challenge is to keep the model consistent with the application and its application domain, retain the compatibility of the first transformation process to the intermediate model, and finally guarantee the applicability of the code mapping for the respective target platform. As programming languages and the functionality of underlying libraries are more likely to change over time than a computer architecture, operating system interfaces, or standard libraries, the adaptation of the code mapping is more elaborate in this case.

The code generation process is furthermore often based on a template-based code generation approach. This is currently one distinguished weakness of MDSD approaches as it still holds up the implementation process of ordinary software but with some major differences: Code has to be written only once and the implementation must be generic enough to be applicable to a wide variety of source models. The first mentioned difference is clearly an advantage whereas the second puts the first statement into perspective. Writing generic code for code templates is a very complex task, the result of which often cannot be used for other target platforms without modifications. Maintenance and adaptations are as well very difficult. This is why developers often shy at modifications of template code.

This has further implications: The higher the level of abstraction the less influence a developer has on the actually generated code. If influence on low-level operations is retained, abstraction is abandoned and the abstract modelling language is degraded to yet another programming language. The question is thus permitted whether a modular, component-based, or service-oriented development approach is not at least equally well suited. It seems that a combination of both approaches with a lean and clear model transformation process may outperform each single one. This is where the advantages of model-orientation take effect.

10.1.1.2 Strengths of MDSD

MDSD has its clear disadvantages that, if cared about adequately well, can play its strengths in combination with other techniques. Reuse of algorithms and code clearly qualify MDSD to be an efficient technology for simplifying software development. Even though porting a once implemented code generation template to another platform is difficult or sometimes impossible, MDSD has its clear strengths when dealing with reoccurring programming tasks and implementations that virtually never change. For the generation of data structures defined in SpicaML, for example, model orientation and the restriction to a very specific modelling domain are profitable. It thereby relieves developers of the manual implementation for arbitrary platforms. Platform independence is thus clearly a step towards software development that goes beyond specific platform requirements. Java and Microsoft .NET support platform independence on the language level. MDSD resides one level above without the need for a common runtime environment and with support for potentially any platform.

The availability of a consistent modelling interface for a well-defined modelling domain is a further important and beneficial aspect of MDSD-based approaches. Aiming at different programming languages, software architectures, and hardware platforms furthermore tightens the demand of MDSD to be platform independent. In the end, the applicability of a modelling language depends on whether modelling entities are suitable and if they are designed in such a way that the application domain is covered well.

10.1.1.3 The Spica way of MDSD

Taking into account the weaknesses and strengths of MDSD, and considering the fact that model orientation is well established, the aforementioned combination of MDSD with a service-oriented development approach seems to bring the most benefit. The design of the Spica development framework follows this proposition: It realises a MDSD-based approach that generates implementations for specific entities of an application domain. For Spica this would denote data structures and data flow. More complex and problem-specific functionality such as SpicaDMF, Geminga, and belonging accessories, for instance, is encapsulated in self-contained entities and provided as a ready-to-use implementation for every target platform. Spica thus pursues an approach that guarantees consistency between target platforms and combines it with a middleware-oriented backend.

These two aspects indeed provide a quite convenient modelling experience. They nevertheless do not set themselves apart from other approaches such as IDL, for example. The Spica development framework represents a unique technology for application in a specialised domain. For AMRs it is even one of the first solutions adopting modelling techniques. It therein addresses problems that arise with multi-robot interaction and highly distributed application scenarios. This is why not the methodology as such but the integrated concept provides the innovation.

We will now shortly discuss the code generation approach and the extensibility of the Spica development framework, before closing with a summary on its applicability.

10.1.2 Code Generation

Code generation in Spica is based on StringTemplate, as introduced in Section 6.6. Alternatives like the *Extensible Stylesheet Language Transformation* (XSLT) [141] are considered more heavy-weight and less intuitive for programming. It furthermore requires XML formatted input, involving the need for another model transformation step. After all, XSLT is no template engine but an approach that facilitates model-to-model transformations.

Spica enforces a strict guideline on the template interfaces and on how data are passed to templates. This clearly simplifies template creation and sustains consistency. Basic code structures and fundamental functionality are included in the main template. If applicable for other purposes as well, they may be added to supplementary templates.

Programming for a template is typically challenging because the realisation must be generic enough to remain shapeable by the model parameters. Spica thereby provides external libraries for more complex and self-contained functionality in order to simplify template creation. Castor, for example, provides Geminga and further platform-dependent functionality required by the Spica development framework.

Spica thus addresses the complexity issues of template creation with only structural measures. Anyway, this process is still confusing and complex, rendering template creation the most costly task in the Spica development framework. There is nevertheless no viable alternative available now that effectively reduces the amount of work. The only promising approach is the ontology-supported code generation [140, 50] as outlined in Section 6.7.3. It again shifts the development focus, this time from the creation of templates to the semantic annotation of low-level functionality in libraries. Annotating a library is, however, a relatively straightforward task for which a programmer only has to spend little energy. This effectively reduces the transformation effort. An abstract template is nevertheless still required, as the respective structure for the final code mapping must be declared. It is not clear whether a pure algorithm specification is sufficient for a mapping to different target platforms. Further details on the target platform and programming language will most likely be still required, though. The approach as such is not yet mature enough for application in Spica.

10.1.3 Extensibility

The most important element of the Spica development framework is the model transformation tool Aastra. It is responsible for interpreting the model and generating code for a specific target platform. Aastra separates responsibilities by providing different components that care about model parsing, transformation, and code generation. Other realisations providing similar service may replace them.

Aastra comes with suitable components for parsing and transforming SpicaML models. The parser builds on ANTLR v3, returning an AST tree that is carried over to a respective AIR representation by the transformation component. The AIR is further extended and completed as required by the code generation process. As model parsing and transformation are tightly bound tasks, they are applicable only in combination in most cases. The code generation component finally passes the AIR to the template engine or another code generation techniques, depending on the realisation. The respective component for Spica contains an appropriately configured StringTemplate instance.

Consequently, Aastra is only a framework for these components, exhibiting a fixed internal processing path. It thereby handles parallelisation automatically, depending on the availability of resources. Three steps have to be accomplished in order to add new modelling languages to Aastra: First, a new parser component has to be created. It must be capable of reading the new language and providing a reference to the respective in-memory representation. The transformation module is then responsible for model conversion, requiring a more or less complex procedure. It mostly depends on the language and the desired capabilities. As Spica already provides a generic code generation component, it may be used for the new language as well. This implies, however, that the transformation component has created an AIR tree. The code generation finally requires code templates to be written.

10.1.4 Simplification and Weaknesses

Spica is a development framework that targets the realisation of communication infrastructures for AMRs and thereby mostly the communication within groups. In this function, it is supposed to ease software development and facilitate the integration of components as well as robots. The degree of simplification in software development can hardly be measured for an approach like this. This is why we will evaluate the key functionality and contrast Spica with other development approaches.

10.1.4.1 Specification of Data Structures

The abstract specification of data structures is the most fundamental capability of SpicaML. Here, Aastra is in charge of creating concrete realisations for specific target platforms with the help of appropriate code templates. This is a widely accepted and common approach. Spica nevertheless exclusively aims at generating data structures for the Spica communication infrastructure. These are equipped with specific capabilities and exhibit semantic metadata for further processing. It is thus not possible to fall back to other abstract message specification approaches.

The modelling process thereby bears resemblance to the specification of data structures in ordinary programming languages. The following characteristics give reasons for its acceptance: Model transformation and the final code mapping processes are typically almost immutable and so no costly template modifications have to be expected. This approach furthermore clearly reduces programming overhead and contributes to lowering development expense once appropriate code generation templates are available.

10.1.4.2 Specification of Data Flow

The data flow specification is the second most important modelling capability of SpicaML. It provides developers with specification means for data handling and data flow in distributed, modular software architectures. Other approaches such as FlowDesigner [33] provide a similar functionality. However, SpicaML is tailored to a very specific application domain: A restriction to only required functionality and the dedication to appropriate solutions to striking problems render SpicaML especially valuable for the application in groups of AMRs. Decreased genericness compared to all-purpose middleware architectures furthermore contributes to the applicability and ease of use of Spica-generated module stubs.

Geminga takes care of the communication infrastructure and the mutual dependencies between components. It is a unique solution here, even though such service is available in other approaches as well. Geminga is resource-efficient and decentralised, explicitly aiming at the application in AMR communication scenarios.

10.2 Resource Utilisation

Resource utilisation is an important factor in a communication infrastructure geared towards real-time behaviour. Starting with the efficiency of the communication infrastructure, we evaluate the performance of the generated architecture and compare it to other approaches. Hereby, we emphasise the overhead imposed by internal processing. We do not regard network latency of an underlying network as latency and jitter characteristics mostly depend on the networking technology and physical constraints. Instead, availability is more important in this case: Mobile robots must be able to get along with the unreliability of the communication media and still operate normal in case of errors.

The memory consumption of the communication infrastructure is primarily of interest for resource-constrained devices. In this context, we will asses the requirements for computational memory as well as the overhead imposed by message encoding and message management structures. A discussion on the performance of Spica finally demonstrates conformance to the requirements.

10.2.1 Processing Overhead of the Communication Infrastructure

The processing overhead imposed by the supporting framework of a communication infrastructure is typically insignificant compared to network latency. For real network communication, it is indeed much more important that the communication protocol can handle errors appropriately. The processing overhead for sending a message carries weight in cases of mass delivery and real-time demands. We will thus hide the physical impact of network communication and analyse only the expense for the delivery and reception processes. Tests are therefore conducted via the loopback network device. Table 10.1 shows the system configuration for the experimental evaluation.

We will now compare the processing overhead of a Spica-based communication infrastructure to the overhead imposed by the *CORBA Notification Service*. CORBA is used in the robotics domain mostly because of its cross-platform availability [128, 52]. For the evaluation, we have adopted the *TAO Realtime CORBA ORB* that provides a high-performance and mature CORBA Notification Service implementation. Transmission latencies of raw BSD sockets provide the baseline data for benchmarking. We have implemented respective test sets for two programming languages (C++ and C#) using two transport protocols (TCP and UDP). The networking part of the C++ implementation relies on the *asio C++ library;* the Mono .NET framework runs tests for C#. All results have been conducted under Linux.

The basic test setup is the same for all three test candidates: Two programs are required, a sender and a receiver. The sender prepares a message with 512B payload that is sent in a loop to the receiver a given number of times, 25000 rounds in a typical test run. Bytes $0 \ldots 7$ of the message are initialised with the current timestamp, bytes $8 \ldots 11$ are initialised with the number of the current round, and the rest is random data. Upon reception, the

CPU	Intel Core 2 Duo 6320	
Memory	4096MiB	
OS	Ubuntu Hardy 8.04	
Kernel	Linux 2.6.24	(2.6.24-19-generic)
Compiler	GCC 4.2.3	(4.2.3-2ubuntu7)
Standard Library	glibc 2.7	(2.7-10ubuntu3)
Networking[1]	asio 1.0.0	
CORBA[2]	ACE+TAO 5.4.7	(5.4.6-13)
.NET Framework[3]	Mono 1.9.1	

1 `http://asio.sourceforge.net/`
2 `http://www.cs.wustl.edu/~schmidt/TAO.html`
3 `http://www.mono-project.com/`

Table 10.1: System configuration used for the experimental evaluation

receiver takes the current time, unpacks the timestamp and the round number, computes the transmission latency measured in μs, and stores the round number together with the latency value for later processing. All measurements are conducted with the same system configuration as shown in Table 10.1. Below, we outline the detailed test setup for each involved candidate.

- **BSD Socket**. The receiver listens to a port on the local host; the sender creates the message and starts the transmission loop. Timestamp and round number are updated once every loop iteration. Their values are copied to the payload before the new message is sent to the receiver. Afterwards, the loop is suspended for 100μs.
- **Spica**. Besides the two test programs, a local Geminga instance is required. It cares about the communication infrastructure. The sender and receiver place their requests and wait 2s for the resource discovery to have finished. The sender then commences with data transmission just like in the BSD socket scenario above.
- **CORBA Notification Service**. The test setup for the CORBA Notification Service comprises three additional programs: A *CORBA Naming Service*, a CORBA Notification Service factory, and a servant for a Notification Channel. The receiver first acquires a reference to the Notification Service, registers itself as a consumer (*Push Consumer*), and waits for incoming events. The sender also resolves the Notification Service instance but registers itself as a supplier (*Push Supplier*). It then commences with data transmission just like in the scenarios above.

10.2.1.1 Transmission Latencies Compared

The basic message latency test measures the latencies for the three test candidates implemented in C++ as introduced above. Table 10.2 outlines the measured latencies. The BSD sockets are most efficient with UDP being a touch faster than TCP. With respect to the system configuration outlined above in Table 10.1, UDP imposes about 8.00μs ($\sigma \approx 0.17\mu second$) and TCP about 11.00$\mu$s ($\sigma \approx 0.12\mu$s) message transmission latency.

	Median		RMS[1]	IQR[2]	Min	Max
CORBA	77.20μs	×9.65	2.02μs	0.20μs	75.80μs	108.40μs
Spica	49.40μs	×6.18	1.75μs	1.40μs	47.20μs	75.40μs
TCP	11.00μs	×1.38	0.12μs	0.00μs	10.60μs	15.40μs
UDP	8.00μs	×1.00	0.17μs	0.00μs	7.00μs	12.80μs

1 = Standard Deviation, Root-Mean-Square 2 = Interquartile Range

Table 10.2: Message Latencies caused by BSD Sockets, Spica, and the CORBA Notification Service. 25000 messages have been sent at 100Hz. UDP represents the baseline for the other protocols.

Spica ranks third with a median latency of 49.40μs ($\sigma \approx 1.75\mu$s). This is about six times as high as the possible minimum for UDP. The CORBA Notification Service induces a median latency of 77.20μs ($\sigma \approx 2.02\mu$s), about seven times as high as the possible minimum for TCP and about nine times above the baseline. Spica and the CORBA Notification Service thus share approximately the same overhead characteristics if compared to the measurements of the underlying protocol.

The latencies of all candidate protocols are quite low, so are the standard deviations. Interquartile range values are all in a reasonable range as well. Improvements in usability clearly have an impact on the performance of Spica and the CORBA solution. For all three solutions, the latency penalty is constant and quite low. It can so be disregarded in most cases.

We will now have a look at the impact of a managed language on the message latency. Due to the lack of a viable CORBA Notification Service implementation for C#, only the BSD socket and Spica will be evaluated this time. Both candidates use the UDP transport protocol. Table 10.3 outlines the results. The variance is a bit more distinctive, at least the maximal latency is clearly increased. The median latency for the BSD socket with UDP is 11.00μs ($\sigma \approx 3.01\mu$s), for Spica about 85.80μs ($\sigma \approx 23.03\mu$s). All results are still in a justifiable range but reveal minimal degradations in reliability. The interquartile range values, for example, are a bit increased. This is presumably due to the managed environment where the program flow is not as uniform as in native languages.

	Median		RMS[1]	IQR[2]	Min	Max
Spica	85.80μs	×7.80	23.03μs	2.40μs	81.00μs	268.00μs
UDP	11.00μs	×1.00	3.01μs	0.80μs	9.80μs	44.00μs

1 = Standard Deviation, Root-Mean-Square 2 = Interquartile Range

Table 10.3: Impact of a managed C# implementation on the transmission latency imposed by plain sockets and Spica. 25000 messages have been sent at 100Hz. UDP represents the baseline.

It must be mentioned here that the system load has strong influence on the latency variance and thus on the standard deviation. All the tests have been conducted on a standard desktop system with a typically low load. TCP communication showed to be more robust against load changes, so it may be preferable in some cases. The load problem is further analysed in Section 10.2.1.3.

10.2.1.2 Multiple Subscribers with Spica and CORBA

The following test aims at the scalability of the respective communication approaches. We only examine Spica and the CORBA Notification Service here. Spica is capable of using multicast to address multiple receivers in parallel. The Notification Service only implements TCP by default.[1] In order to provide reasonable data for comparison, Spica uses unicast as well. The setup for this experiment resembles the configuration of the last one except that three consumers are instantiated.

Table 10.4 shows the medians of measured latencies. All Spica consumers exhibit a quite constant statistic with an average delay of 52.50μs: The median latency for the first consumer is 49.40μs ($\sigma \approx 1.75\mu$s), for the second consumer 56.60μs ($\sigma \approx 2.91\mu$s), and for the third consumer 50.60μs ($\sigma \approx 3.88\mu$s). Surprisingly, the CORBA Notification Service exhibits steadily increasing numbers: The median latency for the first consumer is 77.20μs ($\sigma \approx 2.02\mu$s), for the second consumer 138.20μs ($\sigma \approx 3.81\mu$s), and for the third consumer 198.80μs ($\sigma \approx 5.67\mu$s). This issue seems to be caused by the TCP protocol but disabling the Nagle algorithm leads to exactly the same results. Maybe further buffering strategies influence the measurement in this case. This time shift is nevertheless not dramatic for few consumers, but it may be for tens.

	Consumer 1		Consumer 2		Consumer 3	
CORBA	77.20μs	×1.56	138.20μs	×2.44	198.80μs	×3.93
Spica (TCP)	57.40μs	×1.16	75.00μs	×1.33	139.80μs	×2.76
Spica (UDP)	49.40μs	×1.00	56.60μs	×1.00	50.60μs	×1.00

Table 10.4: Multiple subscribers for the CORBA Notification Service (TCP) and Spica (UDP and TCP). 25000 messages have been sent at 100Hz. Using several TCP channels involves additional latency.

In order to verify the assumption that the steadily increasing transmission latencies are due to TCP, we switched from UDP to TCP channels in the Spica test candidate. The results point to the same assumption that TCP carries out further internal buffering. The first consumer exhibits a median latency of 57.50μs ($\sigma \approx 1.29\mu$s), the second consumer 75.00μs ($\sigma \approx 1.15\mu$s), and the third consumer 139.80μs ($\sigma \approx 2.33\mu$s).

A multicast transmission in the local network would result in almost the same latency on all receiver hosts, of course depending on the system architecture and performance. The main advantage of multicast is the fact that only one message has to be sent for n. The number of messages for a unicast-based approach grows linearly with the number of recipients.

10.2.1.3 Sensitivity Issues

Even though the Spica communication infrastructure and the CORBA Notification Service are almost equally efficient, the prototypical Spica implementation has problems with varying system load. This problem mostly relates to UDP as TCP-based communication keeps transmission latency nearly constant in most cases. Figure 10.1 shows two examples of Spica-based communication where increased fluctuations in the transmission latency have

[1] A possible multicast event distribution approach (NotifyMulticast) is provided by Miro [128].

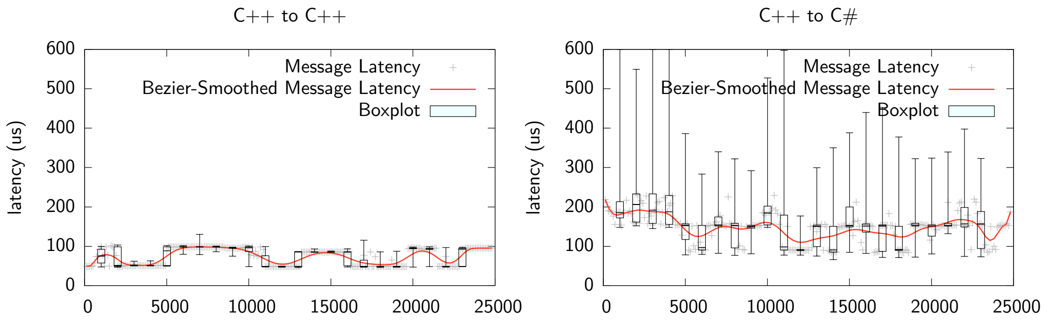


Figure 10.1: Sensitivity of Spica-based communication to changes in the system load. 25000 messages have been sent at 100Hz. The boxplots illustrate median-smoothed statistics of 1000 messages each.

been observed. The values of both plots were gathered from the experiment outlined in the last paragraph. In a few test runs, at least one out of three consumers misbehaved and thus negatively influenced the test.

Figure 10.1 illustrates less stable communication behaviour. The plot on the left hand side is based on data gathered from Spica-based sender and receiver programs both implemented in C++. The median latency is 85.20μs ($\sigma \approx 21.60\mu$s), basically a satisfying value. Even though the standard deviation is quite high, the fluctuation stays within a defined range; no extreme maxima influence communication negatively. The other plot exhibits more irregular latency characteristics. The sender is still the same but the receiver is now implemented in C# and run by the Mono .NET framework. With a median of 153.20μs ($\sigma \approx 70.10\mu$s) the performance is a bit worse and most notably the variance is much higher. Spica typically performs well in C++ and C#, except for slightly increased maxima as shown in the evaluation of managed BSD sockets in Table 10.3.

It is not quite clear why these fluctuations showed up in some measurements. This issue requires further evaluation. Indeterministic changes in the transmission behaviour have negative influence on the overall performance of a Spica communication infrastructure. Even if latencies are measured in μs, single latency spots might measure a ms or more (1.30ms in the example above). It is thus crucial that a basic service quality is guaranteed which might imply slightly increased average transmission latency because of additional caching.

10.2.2 Memory Footprint

The memory footprint of the Spica communication infrastructure is an important issue for most resource-constrained systems. We will thus estimate the memory consumption of important parts of the Spica development framework and the respective communication infrastructure. Computational memory and data serialisation space requirements are considered where required. As no concrete examples of automatically generated code have been discussed, a lower bound will be presented where possible instead.

Regarding the Geminga resource discovery approach, we will limit its evaluation to the memory consumption of the Geminga daemon. Section 10.3.2 discusses the communication overhead extensively. Aastra will not be regarded, as it does not influence the runtime behaviour of the Spica communication infrastructure.

10.2.2.1 Generated Data Structures

SpicaML supports four different types of data structures, each of which has very specific space requirements regarding the generated implementation and the respective serialisation.

An enumeration represents the simplest form of data structure. Its implementation therefore requires hardly more space than its SpicaML model: For C++ and C# it is mapped directly onto an enumeration. Even though newer Java versions support enumerations as well, a SpicaML enumeration is mapped onto several constants in this case. If serialised, an enumeration value relates to its respective primitive type and encodes with no additional overhead.

SpicaML containers represent complex types composed of primitives or other complex types. Apart from the fields, each container comprises several management variables that account for a static portion of space in both the implementation and the serialised data. For each container, 6B are required as outlined in Section 5.3. This includes the endianness, type, and state fields. Two strings furthermore provide the semantic identifier, i.e. the concept and the representation. Only the 6B of management information will be encoded along with each container during serialisation.

A header provides management information for a message. As discussed in Section 5.3, it involves 8B of static management information. This includes the magic number, endianness, and type fields. The semantic identifier fields are added only to the implementation, just like for the container. Depending on the activated message capabilities, some more information will be involved:

- **Authentication**. The authentication capability adds one byte (`authEnabled`) plus an array for the signature (`authSig`) and the rest of the message. A message authentication signature typically consists of at least 8B. Authentication thus adds at least 9B of overhead.
- **Encryption**. The encryption capability needs five byte (`encEnabled` and `encPosition`) plus an array for the encrypted message (`endData`). The data size of an encrypted message will be roughly about the same size as the original message. Hence, the overhead imposed by the encryption capability results in 5B. The original message will be replaced.
- **Compression**. The compression capability requires one byte (`compEnabled`) plus an array for the compressed message (`compData`). By allowing for a certain threshold, compressed data is typically shorter than the respective original. The overhead will nevertheless be counted as 1B.

A SpicaML message finally inherits the management overhead from its header. As headers are never serialised on their own, no more additional information counts in this case for messages.

Depending on the target, some more overhead will clearly be involved. Especially a concrete implementation of a message or container will include additional overhead imposed by the programming language and the platform. Such a structure relates to a class in object-oriented languages, for example. Several member functions are required, some of which for accessing the member fields. Inheritance in the SpicaML model thereby typically maps onto language-specific features. It is especially important to note that most member functions have to be overwritten because their functionality depends on the respective data structure. Each class thus brings a quite large code segment. Member fields, in turn, are typically derived and so count only for the super class.

Due to the hierarchical structure of SpicaML containers, the respective overhead of 6B may count several times, depending on the structure definition. The general lower bound of a SpicaML message thus computes to $8\text{B} + n \cdot 6\text{B}$ where n is the number of included containers. This estimation does not consider any active message capabilities in the header.

10.2.2.2 Generated Stubs

The module stubs generated from the SpicaML module definitions are tuned towards efficiency and ease of use. They are intentionally kept as simple and lean as possible, offering a clean interface towards the user. Each stub thereby contains a reference to a Geminga client instance. It optionally maintains an instance of a monitoring consumer, supplier, or both, depending on its configuration in the SpicaML model. A dedicated DMC is further instantiated for every offered and requested message type.

The Geminga client is kept very simplistic as well. It maintains a socket connection to the Geminga daemon and several other socket connections to modules. A buffer finally lists all issued GCNP protocol statements. They are replayed if the connection to the Geminga must be reinitialised.

The monitoring consumer consists of a single DMC, the supplier of two. Management overhead of a DMC involves the data structure and a given number of fields. It is therefore bounded by the size of the data structure plus much less than 100B. SpicaDMF employs only priority queues for its DMC instances. The maximum size may therefore grow and shrink dynamically. Because of the automatic purging mechanism, it will never grow unchecked.

10.2.2.3 Geminga Resource Discovery Daemon

The most essential component of the Spica communication infrastructure is the Geminga daemon. It is part of a distributed repository maintaining information on currently available resources and resource requests in the network. We will now examine the internal data structure of a Geminga daemon that maintains resource offers and requests. The resource discovery is carried out thereon locally.

Geminga is implemented as a daemon in C++, adopting the *asio* and some of the *Boost C++ Libraries*. For every resource offer or request, a new so-called *Resource Record* (RR) is created. It contains information on the resource itself (e.g. type, reachability, and capabilities), its provider (e.g. name, hostname, and address), and relationships with other resources. An RR requires about 500B of memory, depending on the number of available details on the resource. Geminga thus behaves quite conservative in terms of memory usage.

A specific indexing mechanism is employed for locating RRs in the local repository. Appropriate indexing is especially important for large amounts of records. This is why Geminga adopts `MultiIndex`, a data structure offered by Boost that maintains multiple indices on a set of elements. Geminga depends on seven indices, each covering a specific aspect of the resources:

- **Resource Key**. This aspect is a combination of several characteristics of a service. It includes the *Geminga identifier*, the *client identifier* that is unique within a Geminga instance, the channel mode, and optionally the address of the resource. This index builds the foundation for resource matching.
- **Resource Provider**. This aspect only takes into account the *Geminga* and the *client identifiers*. It is used for locating all offers or requests for specific client.
- **Resource**. This aspect matches on the resource identifier, i.e. the *numerical, textual*, or *semantic identifier* of a resource. It is used for locating all resources for a given resource identifier.
- **Address**. This aspect considers the *address* of the resource through which it is reachable. The address includes the *transport protocol*, so it is used for locating all resources that are reachable using a specific addressing scheme.
- **Timestamp**. The purging mechanism of Geminga requires this aspect. For each entry, a *timestamp* is maintained. Once an entry timed out, it is removed from the repository automatically.
- **Geminga Identifier**. This aspect matches on the *Geminga identifier* for locating all clients for a given Geminga instance. It may be used for querying all available resources for a given host.
- **Client Session**. Only local resources use this aspect. All resource records of a client in the local repository must be purged once the client disconnects. It thus filters on the *client identifier* and the identifier of the local Geminga instance.

The matching procedure depends on the resource key, supporting incomplete search queries. It is so possible to query for a service that was not fully specified. This is mostly required for manual subscription processes where a user does not have all the required information.

The memory requirement and a performance estimation of the `MultiIndex` data structure are provided along with the respective class documentation.[2] In a typical scenario with CaNoSA and six active robots, the Geminga daemon will require about 3MiB of computational memory.

10.2.3 Performance

Regarding the analysis of the performance characteristics, we evaluate two fundamental principles of the Spica communication infrastructure: Serialisation and deserialisation as well as message transmission and reception. Section 10.2.1 already examined the overall processing overhead.

10.2.3.1 Data Encoding and Decoding

Serialisation, the transformation of a SpicaML data structure into a flat byte stream is required for every kind of data structure that has to be sent to a remote system. As serialisation and deserialisation have to be carried out for every message and every container,

[2]`http://www.boost.org/doc/html/multi_index.html` (accessed 2008-08-24).

and all this whenever a message is delivered, they must be especially efficient in terms of processing expense. However, this fundamental functionality must be implemented in a modular fashion in order to retain interchangeability of serialisation techniques. These two, partly conflicting characteristics have to be harmonised in some way.

Spica relies on serialisation techniques to be provided as self-contained modules, implementing the `SERI` interface. They are typically loaded once during initialisation but it is also possible to directly link them into the client program. `SERI` requires two methods to be implemented for every primitive type supported by SpicaML, according to Section 5.3.4. One of them is responsible for encoding, the other for decoding the respective type. Additional functions are provided for encoding a length field or complex structures. As explicitly typed encoding routines, they contribute to the type safety in the generated implementation.

The individual methods are typically implemented as functions or methods in the target language. However, each call involves a considerable amount of additional costs caused by subroutine jumps in the program flow. This involves save and restore of variables and processor registers. One possibility to avoid this problem is to dispense with modularity and embed the encoding and decoding logic directly into the generated code. This should, however, be considered the ultimate alternative if the performance penalty grew too much.

Another solution that is applicable at least for C++ is to use *inline methods* for the serialisation and deserialisation modules. The compiler here cares about embedding the respective functionality into the implementation during compilation and so retains modularity. Many modern languages or runtime environments try to deal with this optimisation automatically where applicable. Such an optimisation is presumably not effective in this case as the generated code references methods several times. This typically disqualifies for inlining. It is thus up to the developer whether to dispense with modularity or not.

10.2.3.2 Transmission and Reception

Message transmission and reception are quite straightforward tasks in Spica. Figure 10.2 outlines the scenario of transmitting a message via the Spica communication infrastructure. When sending a message to one or more remote system, it is passed to the respective DMC instance (1) that is provided by the Spica stub.

The DMC first inserts the message into its underlying data structure where it may be deferred as a result or reordering. Given a uniform distribution, we assume the average delay to be $\Delta_{\text{reorder}} \approx \left\lfloor \frac{n}{2} \right\rfloor ts$. It depends on the size of the data structure (n), the time between two retrieve operations (t), and the average reordering offset that is assumed to be about half the structure size. The priority distribution of all messages is another important impact factor in this case. About half the messages will be reordered because we assume a uniform distribution here as well.

Depending on the synchronisation policy (**SYNP**) setting, the highest rated message is then retrieved either immediately or by the next retrieval operation. Given the uniform distribution again, the delay will be about $\Delta_{sync} \approx \frac{t}{2}$s where t refers to the synchronisation interval **SYNI**, defining the time between two retrieval operations. A DMC may thus add an overall delay of

$$\Delta_{\text{DMC}} = \Delta_{reorder} + \Delta_{sync} \approx \left(\left\lfloor \frac{n}{2} \right\rfloor t + \frac{t}{2} \right) = \left(\left\lfloor \frac{n}{2} \right\rfloor + \frac{1}{2} \right) ts$$

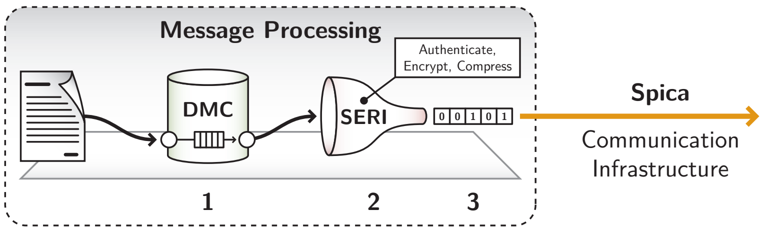


Figure 10.2: Three-step message transmission in Spica

A data structure of arbitrary size that does not synchronisation retrieval ($t = 0$) will result in $\Delta_{\text{ring}_0} = 0$. If synchronisation is enabled ($t = 100\text{ms}$) for a ringbuffer or a queue that are not capable of reordering ($\Rightarrow n = 0$), the average delay will be $\Delta_{\text{ring}_1} \approx \frac{t}{2} = 50\text{ms}$. This example shows that the message insertion frequency (t_{insert}) is not regarded here. Most incoming messages will not be delivered if $t_{\text{insert}} > t$. They will be instead overwritten sometime. The size of a queue will steadily increase in this case. Both structures are mostly useful only for balancing irregular insertion frequencies. The formula Δ_{DMC} holds for a priority queue of size n.

In the next step (2), the message is serialised. This involves the creation of a new instance of the `SERI` encoder module. Here, only some variables have to be initialised, basically resembling the allocation of a piece of memory. The data structure is the passed to the encoder that returns an array of bytes ready for delivery (3).

The reception of a message involves the processing steps shown in Figure 10.2 but the other way round. First, the message is received (3). The resulting byte array is then passed to the decoder (2). As all SpicaML messages contain a well-defined magic number at the beginning of their serialised data, accidentally delivered data will be sorted out here. The decoder then tries to extract the type identifier and compares it to the known message types. Decoding proceeds if the message type is known. A new instance is returned if the data were decoded successfully.

Delays induced by the DMCs are the predominant source of message deferral. DMCs may nevertheless be configured to operate in a time-invariant fashion. The `SERI` solution, however, cannot be completely removed so easily. It clearly has a strong influence on the performance, as it is required for every message. Internal serialisation in the data flow and costly callback functions introduce even more latency.

The Spica communication infrastructure relies on event-oriented communication. Hence, messages typically arrive asynchronously in an indeterministic fashion. In order to simplify the handling and to prevent race conditions, all incoming data are serialised, so only one message is processed at a time. This discretisation again accounts for additional latencies, even though they are negligible in most cases.

Callback methods furthermore introduce additional processing overhead by requiring special handling. Especially for C++ it is difficult to implement callback methods in an object-oriented manner. It involves several indirection steps, each involving at least one method call until an event is finally delivered. See Section 7.3 for details. Other languages may handle this differently but managed languages such as Java or C# clearly involve a still higher overhead. Spica tries to reduce the overhead by reducing the level of genericness and the number of hierarchically called callback functions.

10.3 Distribution

We will now evaluate how the measures provided by Spica behave in a distributed scenario and how they handle prevailing issues. Their performance is compared to the requirements introduced in Section 1.2.

10.3.1 Failure Tolerance

Using a Spica communication infrastructure must never lead to blocking calls or any other unintended behaviour that may influence the operation of the RCA in any way. As shown in Figure 10.3, a reliable protocol is not well suited for this purpose. This is why Spica defaults to connection-less, unreliable communication channels based on UDP. Message delivery is thereby at risk, as data may get lost. The only viable reliability measure allowed in this case is the repeated transmission of messages. The receiver requires appropriate duplicate handling in order to accomplish delivery in an idempotent manner. Spica therefore provides unique message identifiers that are initialised automatically during the instantiation of a message. Redundant transmission increases network load but dispense with sophisticated protocol operations.

Unreliability is also in important issue for sensor data exchange. Robots typically announce measurements periodically, depending on the respective sensor's delivery behaviour. In the majority of cases, only the most current measurement is of interest for potential receivers, as it reflects the most current state of a robot's environment. Spica follows a very trivial but effective approach to this purpose: It relies on the periodical transmission and does not take care of reliability. So a measurement is sent only once. This implies that receivers have to get along with incomplete data. They have to be able to cope with such kind instability anyway, so this should be no further handicap. This approach is furthermore highly efficient and thus well suited for the given application domain.

Network partitioning is yet another source of failure that is a common phenomenon in wireless networks. In a Spica communication infrastructure, mostly Geminga is vulnerable to this issue. However, severe consequences only have to be expected if the network was already partitioned before the initialisation of a Geminga daemon and is reunited some time thereafter. In this case, the remote possibility exists that two Geminga instances – they must be located in two separate parts of the network – have created the same identifier. This is because they were unable to deliver the identifier announcements and so no collision checking has been carried out. In this case, all involved Geminga instances have to restart themselves and regenerate their identifiers.

These measures guarantee that a Spica-based communication infrastructure can cope with unreliability and function well even if network media are unavailable. They are conservative in resource consumption and exhibit a low level of complexity.

10.3.2 Scalability

The scalability of a Spica communication infrastructure not only depends on the adopted communication schemes such as multicast but on the efficiency and scalability of Geminga.

We will therefore consider both measures, reviewing their contributions to the scalability of the automatically generated realisations.

Multicast is well suited for multilateral communication within a group of autonomous robots. A single message is thereby sent to several receivers in parallel. Hence, it exhibits characteristics like broadcast but in addition allows restricting the number of recipients by establishing logical *multicast groups*. A unique multicast address represents each group. Even routing beyond gateway boundaries is possible. See Section 2.4 for more information.

Within a Spica-generated communication infrastructure, multicast groups serve different purposes. Geminga maintains a multicast group for resource announcements. Multicast is further well suited for message distribution to several robots. This is why CaNoSA employs a dedicated multicast group for the delivery of the `SharedWorldInfo` message as shown in Listing 9.1.

Section 10.4.1 further analyses the advantages and drawbacks of multicast in a Spica communication infrastructure. Below we will review the size estimation of a Geminga announcement message and the application of a compression scheme for lowering communication overhead.

10.3.2.1 Geminga Announcement Message Size Estimation

Figure 8.2 outlines the layout of the Geminga announcement message. In order to assess its scalability, we will analyse its overall complexity and estimate the space requirements for serialisation.

A SpicaML data structure model builds the foundation for the Geminga announcement message. It therefore comprises a header that brings a default management overhead of 8B. The message identifier, Geminga identifier, and timestamp fields together add another 16B. Additional header fields that subsume the fields for the message capabilities typically require less than 32B and the hostname of the sender will involve 16B as most. The header thus requires about 72B.

The next logical block in the message structure deals with the addresses of the respective Geminga service. As Spica supports IPv4 [111] and IPv6 [35] addresses, the maximal address length is 16B. Considering the serialisation overhead of about 7B, each address will require at most 23B once serialised. This includes the protocol type and port values.

The rest of the message deals with the offered and requested resources. It contains an array of *Client Records* (CR) that provide information on the clients of the local Geminga service. Each CR is realised as a SpicaML container and thus involves 6B of management overhead. Actually all records of the Geminga announcement message are SpicaML containers and so impose the 6B overhead.

The *Geminga Identifier* in the header and the *Client Identifier* in the CR that both need 4B uniquely identify each Geminga client. The *Client Name* is assumed to be less or equal than 32B, even though it might be larger in rare cases. Each CR finally consists of two arrays: A *Publication Record* (PR) represents a client's resource offers whereas a *Subscription Record* (SR) summarises its resource requests. This results in an overall space requirement per CR of $(42 + k \cdot \mathrm{PR} + l \cdot \mathrm{SR})$B.

PRs exhibit almost the same structure as SRs except that SRs contain an additional array of *Remote Client Records* (RCR). RCRs allow a client to subscribe to a resource provided by one or

Component	Estimated Size (B)
Header	72
Address	23
Client Record (CR)	$42 + k \cdot \mathrm{PR} + l \cdot \mathrm{SR}$
Publication Record (PR)	14
Subscription Record (SR)	$14 + m \cdot \mathrm{RCR}$
Remote Client Record (RCR)	14

Table 10.5: Space requirements of Geminga announcement message components

more specific clients. The PRs and RCRs so count for 14B each, the SRs for $(14 + m \cdot \mathrm{RCR})$B where m is the number of RCRs. The management overhead of 6B is included already. Table 10.5 recapitulates the estimated space requirements for the Geminga announcement message.

The resulting sizes of all the remaining fields are quite clear. Based on this data, the overall space requirement estimate of a Geminga announcement message is:

$$\big(\mathrm{Header} + i \cdot \mathrm{Address} + j \cdot \mathrm{CR}\big)\,\mathrm{B}$$

where i denotes the number of local addresses and j the number of local clients. The variables i and m are typically quite small. Much more influence on the size of the message is exerted by j, k, and l. Because of their relation, the overall message size complexity can be approximated by $\mathcal{O}(n^2)$ where $j, k, l < n$.

Example Let us now assume some typical values: A robot has two local addresses ($i = 2$) and four clients connected to the local Geminga service ($j = 4$). Every service further offers and requests two resources each ($k = l = 2$) and the subscription is bound to one specific client ($m = 1$). The approximate size of the Geminga announcement message thus computes to as follows:

$$\begin{aligned} & (\mathrm{Header} + 2 \cdot \mathrm{Address} + 4 \cdot (42 + 2 \cdot \mathrm{PR} + 2 \cdot \mathrm{SR}))\,\mathrm{B} \\ = \; & (72 + 2 \cdot 23 + 4 \cdot (42 + 2 \cdot 14 + 2 \cdot (14 + 1 \cdot 14)))\,\mathrm{B} \\ = \; & 622\mathrm{B} \end{aligned}$$

10.3.2.2 Lowering Communication Overhead

By considering the size of the Geminga announcement and `SharedWorldInfo` messages, it is obvious that the amount of data exchanged between participants is a limiting factor for scalability. Other solutions must be found if messages are getting too big and their sizes cannot be restricted with available measures. SpicaML has built-in support for data compression, providing a filter component that adopts the deflate compression algorithm [37].

A quick survey covering message transmission in CaNoSA[3] has shown that compression is

[3]We enabled data compression in CaNoSA for Geminga announcement and `SharedWorldInfo` messages during the RoboCup German Open 2008 championships. A given amount of data must be available for compression to be effective that depends on the type of data and their representation. For the two messages we ascertained a value of about 160B.

effective: The Geminga announcement and the `SharedWorldInfo` messages have both been shrunk by about 40% in average.

Resource consumption is the main deficiency of this approach: The system load increases noticeable if messages are sent with a rather high frequency.[4] It has furthermore influence on the transmission latency as shown in Table 10.6. Both measurements were gathered with the same implementation in C#: No compression is used in the row named **None**, the latency variance (RMS) is due to the implementation language. The row named **Deflate** exhibits a greater dispersion and a much worse latency characteristic, about six times slower than without compression. It has to be pointed out that an implementation in C/C++ would perform better here. An influence on the message latency will nevertheless be still perceivable.

	Median		**RMS**[1]	**IQR**[2]	**Min**	**Max**
Deflate	591.00μs	×7.20	61.46μs	77.20μs	385.00μs	1391.20μs
None	81.00μs	×1.00	22.98μs	9.00μs	68.40μs	238.00μs

1 = Standard Deviation, Root-Mean-Square 2 = Interquartile Range

Table 10.6: Impact of Deflate compression in C#. 25000 messages have been sent at 100Hz. Besides increased latency, compression involves a slightly higher RMS. IQR is increased due to changes in the system load.

10.3.2.3 Security

The Spica development framework explicitly addresses security issues arising from eavesdropping and data manipulation in wireless communication media. It therefore provides developers with suitable measures for preventing such attacks: Authentication or encryption capabilities may be enabled for messages in a SpicaML model. The code generation process translates these capabilities into the respective functionality and embeds them directly into the message serialisation and deserialisation processes. A message is aware of its internal structure, so it can carry out the respective tasks appropriately.

The Castor library provides implementations for compression, encryption, and authentication. The previous section already discussed compression. We will therefore not consider it here. In terms of encryption, Castor provides an implementation of the *Advanced Encryption Standard* (AES) [89]. It is wide spread, well understood, and provides a satisfying performance even in software. Encryption of message streams adopts the CTR Cipher Mode. CTR is particularly well suited for unreliable communication and robust against message loss, as it does not rely on a feedback mechanism. Castor further adopts the HMAC algorithm for authentication purposes as outlined in Section 2.5. It is fast and provides a reliable symmetric authentication scheme for messages.

Message encryption shares the deficiencies of compression, namely an impact on the performance and degradation in message latencies. However, it is typically more efficient because it generates a key stream that is applied to the serialised message in a trivial manner. Compression, in turn, depends on quite expensive operations for each message. More precisely, the compression tree must be set up for each message anew.

[4]Messages have been sent with about 100Hz on a system with the configuration mentioned above.

Performance Testing In order to illustrate the differences between these approaches, we have conducted some basic performance tests. They are not representative for the algorithms themselves because the test cases are implemented in C# with no specific optimisations. The tests furthermore target the relative differences in execution performance for a set of very problem-specific algorithms.

Each algorithm represents a reversible function: The forward path processes the data. The backward path then typically reverts the processing. For this evaluation, an array of 500B is filled with placeholder text.[5] It has to be processed by the algorithms the following ways: *Forward* and *Backward*, F*orward-only*, *Backward-only*. This is repeated 102400 times.

Compression adopts the Deflate [37] algorithm provided by the *SharpZipLib*,[6] an open source compression library for the Microsoft CLR. AES encryption and HMAC-MD5 authentication rely on the standard implementation provided by the Mono .NET framework. A reference implementation serves as a baseline candidate: It applies a simple arithmetic operation to every element of the byte array.

Table 10.7 presents the comparison results. Compression is definitely the most expensive operation, even though decompression is fairly efficient. Encryption is still resource demanding but has a much less severe impact. Especially decoding is a bit more expensive than encoding. HMAC authentication is quite resource-efficient. Hence, it is well suited for the application in the Spica communication infrastructure even for large amounts of data.

	Two-way		**Forward**		**Backward**	
Deflate	345.83μs	×66.89	262.28μs	×126.71	63.20μs	×30.53
AES-128	159.16μs	×30.79	74.48μs	×35.98	82.41μs	×39.81
HMAC-MD5	33.45μs	×6.47	15.47μs	×7.47	15.47μs	×7.47
Baseline	5.17μs	×1.00	2.07μs	×1.00	2.07μs	×1.00

Table 10.7: Performance comparison: Compression, Encryption, and Authentication in Spica

10.4 Event Orientation

The Spica development framework promotes message-oriented communication as a foundation for an event-oriented communication model. Event orientation suits best for the given application domain because of several reasons: Sensor data acquisition is realised in one of two modes, namely *push* or *pull*. The push model resembles event-orientation to some degree as is provides new information asynchronously once available. Information exchange between animals or humans exhibits similar characteristics and robots acting in a group typically employ the push scheme for new discoveries or in case of emergency. Event-oriented communication is thus one of the most fundamental communication schemes available and well suited for real-time event distribution. The pull model typically involves polling for new information. It is well suited if asynchrony must be avoided.

[5]We used *lorem ipsum* for this purpose which is a common placeholder text used to demonstrate the graphic elements of a document or visual presentation. A lot of redundancy results in a good compression characteristics.

[6]`http://sharpdevelop.net/OpenSource/SharpZipLib/` (accessed 2008-08-30).

Event-oriented communication is further not limited to only bilateral interactions. Communication with a broader audience is typically realised through broadcasting or even multicasting data in the networking domain. Another advantage is the simplicity of the communication model. As in reality, events (messages) are not guaranteed to reach their destination. In a human football match, for example, information distribution capabilities are limited only to the distances and acoustic levels in the stadium. In the networking scenario, more or less the same restrictions apply. They will never disappear as they are due to the fundamental physical constraints of the communication media.

Event orientation as a basic communication strategy thus mimics the natural ideal and provides convincing real-time capabilities.

10.4.1 Messaging and Group Communication

Spica uses the multicast communication scheme for addressing group communication. Even though transmission is known to be unreliable, multicast realises the optimal distribution strategy when dealing with more than two individuals. Utz [128] opposed a multicast-based communication scheme to a traditional unicast-based communication scheme for group communication in the context of an experimental evaluation. Unicast communication clearly disqualifies for the current scenario because of a linear transmission complexity $\mathcal{O}(n)$, with n being the number of participants in the group.

Spica supports both unicast and broadcast. The channel mode `otm/stream` implements a multicast-based communication characteristic. It always exhibits an optimal transmission complexity of $\mathcal{O}(1)$. The downside is that all participants receive every message, regardless of their own preferences. This implies an increase in computational overhead. Multicast must furthermore be supported by the communication media, otherwise it is simulated through either unicast or broadcast semantics. The latter case implies a potentially high impact on the reactivity of all receivers.

Multicast and broadcast have a positive impact in wireless communication media because media access is one of the most expensive operations besides data transmission as such. Minimising the number of media accesses improves the overall network stability.

Spica also provides two unicast communication schemes: `oto` and `otm/pubsub`. They are required whenever data have to be transmit that are not of common interest. An example would be control information intended for a single receiver.

Communication semantics in Spica cover the most important interaction types, including group communication. The channel mode is selected with regard to the information type and the target audience. Given, for example, every member in the group provides a `SharedWorldInfo` message. The overall communication complexity, i.e. the number of required message transmissions of all participants combined, then results in the following complexity classes: For unicast communication, the message transmission count is in $\Theta(n^2)$ whereas the message transmission count of multicast is in $\Theta(n)$.[7] If only a subset of participants is addressed, the complexity boundaries reduce to $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$ respectively.

Event distribution systems like the CORBA Notification Service, for example, avoid direct transmission by introducing a hub component. The hub accepts events from senders and

[7] $f(n) \in \Theta(g(n))$ means there are positive constants c_1, c_2, and k, such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq k$. See `http://www.nist.gov/dads/HTML/theta.html` (accessed 2008-08-30).

forwards them to all interested receivers. This approach does not reduce traffic but increases it by n in the case discussed above.

10.4.2 Message-Oriented Communication and Reliability

Datagram-based communication is the preferred transport protocol in Spica-based communication infrastructures and reflects event orientation best. The unreliability of the communication media has a major impact on the communication behaviour and cooperation capabilities of the robots. For mass data transmission, as required for streams of sensor values or camera images, a stream-oriented, connection-based protocol seems to be well suited at first. It would further take care of transmission reliability. A closer examination reveals, however, that a reliable connection-oriented communication protocol is inappropriate here. This is especially obvious when regarding real-time guarantees for unreliable communication: Message retransmission because of reliable communication would lead to increased network latency. Data are furthermore buffered in the sender's network stack, implying little but noticeable delays.[8] It further has influence on the variance of the message latency (*jitter*).

Figure 10.3 demonstrates the packet loss sensitivity of the TCP protocol for three loss rates: 1%, 10%, and 20% message loss. The packet loss is simulated using the `iptables` statistics module.[9] Messages were delivered with a frequency of 100Hz, each message contained 512B payload.

The protocol still seems to perform well at an average loss rate of 1%. The median transmission latency is 14.2μs ($\sigma \approx$ 8.9ms). However, the standard deviation is already in the ms range, hinting at issues imposed by packet loss in combination with transmission reliability measures. Things obviously get worse with a loss rate of 10%. The median transmission latency is now 104.4ms ($\sigma \approx$ 293.17ms). This makes communication nearly impossible for mobile robots. A loss rate of 20% finally leads to a median transmission latency of 7.8s ($\sigma \approx$ 8.4s).

The measurement data and respective plots show quite clearly the window adaptation process of TCP. It is less perceptible in the plot for lower packet loss rates due to median smoothing used for creating the figure. The evaluation revealed what is obvious: The combination of periodical data transmission and reliable communication results in a poor protocol performance under packet loss conditions. Accompanying the increase in transmission latency, the time required for sending a message increases as well. This is because the TCP send operation blocks and so suspends processing in the sender. All these reasons clearly render a reliable communication protocol inappropriate for the given application domain.

By contrast, datagram-based transmission of messages does not share these drawbacks. Even though it may loose data in case of network errors, it is rather insensitive to network unavailability, at least with regard to increased latency or blocking. The primitive protocol operation contributes to its capability of pseudo real-time operation, adding no additional latencies.

[8]The Nagle algorithm in TCP may be disabled, resulting in immediate transmission. Notwithstanding this fact, reliable transmission is retained.

[9]We used the following rule to simulate a packet loss rate of the given probability, 20% in this example:

```
iptables -A INPUT -j DROP -p tcp -m statistic --mode random --probability 0.2
```

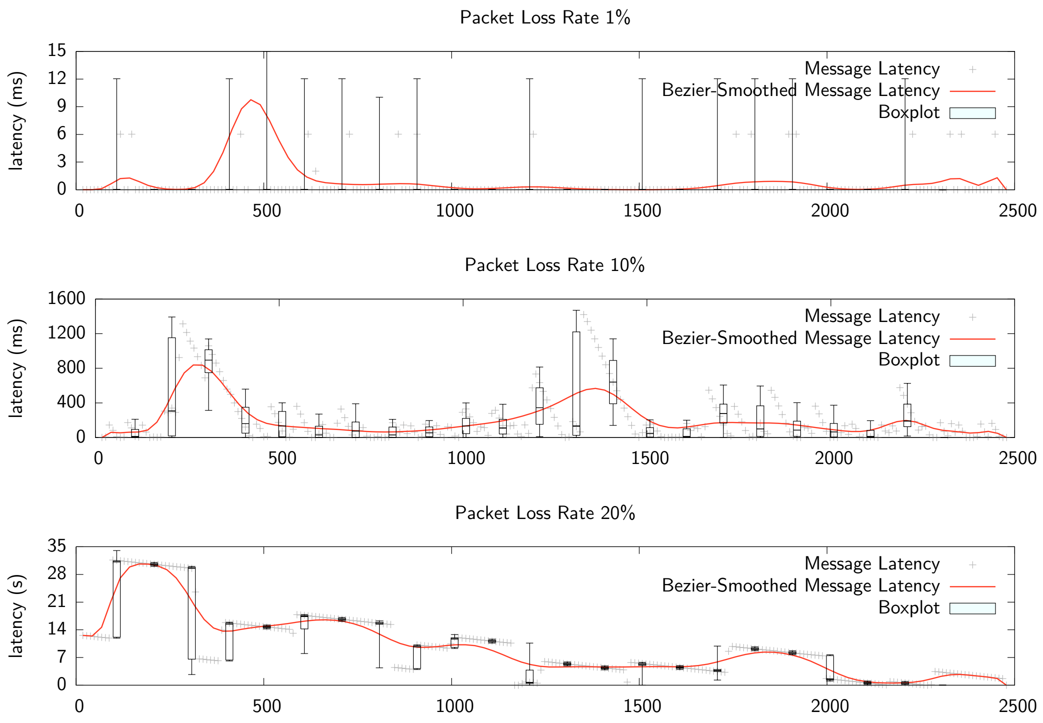


Figure 10.3: Packet loss sensitivity of event-based message transmission over TCP. 2500 messages have been sent at 100Hz. The boxplots illustrate median-smoothed statistics of 100 messages each.

10.4.3 Synchronisation and Arbitration

Event distribution inherently deals with asynchrony. Providers thereby do not have to take any specific measures but the consumers must be able to handle asynchrony appropriately. This is why Spica introduces the DMC concept. DMCs are integrated message buffers that are able to signal the availability of new data by executing a callback function, following a predefined synchronisation policy (**SYNP**). A DMC is further customisable in terms of storage semantics and storage handling. Support for three different synchronisation policies is available, each of which targets a specific type of application domain:

- **None**. No synchronisation at all implies signalling incoming messages immediately. The registered callback function must thus be able to process the data appropriately. This behaviour implements the strictest real-time level available in Spica. There is no synchronisation involved. Incoming messages are nevertheless inserted into the DMC for further processing later on.
- **Soft**. Soft synchronisation measures the relative time difference between successive message reception events. Messages that arrive within a synchronisation interval Δt are discarded. This synchronisation policy still delivers the same real-time characteristics as if no synchronisation is applied. It rather filters out messages that arrive too fast. This behaviour is important especially if the internal processing cannot cope with oversupplied data.
- **Hard**. Hard synchronisation forces messages to be stored to their dedicated DMC at first. A system timer periodically retrieves them. This approach may exhibit

further message latencies due to caching characteristics. It is, however, suited well for applications where messages have to be signalled in a uniform fashion.

Section 10.2.3.2 analysed the delay imposed by the DMC concept in detail. Every invocation policy has its dedicated application domain where no other scheme suffices. In combination with the invocation policy (**INVP**), parallel processing can be realised with only little effort. The advantage of DMCs over ordinary message buffers is the integrated concept they follow. A DMC cares about storage and retrieval as well as signalling. It realises a software pattern that simplifies implementation and code generation.

10.5 Summary

This chapter evaluated the Spica approach. It first discussed the general applicability of Spica and the model orientation in particular. It gave reasons why modelling is not suited equally well for all application domains. If used in combination with other software development approaches, however, modelling develops its full potential as a powerful development tool. The code generation part, however, turned out to be weakest link in the approach: It typically requires manual creation and maintenance of code templates that is a very costly task.

We further reviewed the resource utilisation, communication overhead, and processing latencies imposed by the Spica communication infrastructure. It also covered its memory footprint and performance characteristics. A discussion on the applicability in distributed environments and the measures regarding event orientation finally closed the evaluation.

The results confirm the power of the Spica development framework and prove its applicability in real-world appliances. The measures taken by Spica to promote a heterogeneous, platform-independent development framework within the robotics community have shown to be effective in real-world application during RoboCup tournaments with a multi-robot team.

11 Conclusions

"Conclusive facts are inseparable from inconclusive except by a head that already understands and knows."

— Thomas Carlyle
Past and Present, Chapter II, Statistics, 1848

11.1 Requirements Revisited

Section 1.2 established several requirements that focus the responsibilities of the Spica development framework on prevalent issues of its application domain. These requirements are classified into three thematic groups, namely *development methodology*, *communication infrastructure*, and *resource discovery*. This section will now bring together the contributions of the Spica development framework with these requirements.

11.1.1 Development Methodology

The development methodology is an important issue for developers and a fundamental indication of cross-platform applicability. Below, we review each of the requirements proposed in Section 1.2.1 and present the respective measures taken by Spica.

Abstract Modelling In the context of modelling and high-level programming, domain-specific languages provide a developer with the ability to specify its demands in an abstract, platform-independent manner. Hence, this allows translating solutions to several different platforms instead of only a single one.

Chapter 5 introduces SpicaML. It supports development in a platform-independent fashion by providing modelling means for important entities of a specific domain. Regarding communication infrastructures of AMRs, this involves data structures and data flow of modular software architectures. Together with the model transformation process and Aastra as introduced in Chapter 6, Spica also adheres to the platform independence requirement.

Model Transformation Abstract models apparently depend on an appropriate model transformation approach. The Spica development framework is therefore required to define a viable process that is able take the abstract model over to a concrete realisation for one of several target platforms. Modelling and model transformation must thereby be well understood and open for extensions.

Chapter 6 introduces Aastra that takes care of model transformation from SpicaML down to platform-dependent code. It thereby employs a template-based code generation process as shown in Section 6.6. SpicaML and the transformation process may be modified, extended, or replaced as required.

Applicability The applicability of generated code is a non-functional and yet vague characteristic. In the current context, it describes the suitability of an implementation for a concrete target platform, considering optimisation and resource efficiency.

Code templates provide the concrete and platform-dependent realisation of a model as shown in Section 6.6.2. As these templates must be able to deal with a wide range of input parameters, mostly the respective creator is responsible for the adherence to the requirement for tight integration and efficiency. Spica nevertheless provides measures that support developers in doing this.

Modularity A platform-independent software development approach is required not only for dealing with different hardware architectures but rather for supporting different software architectures and existing solutions. In this context, typically self-contained functional entities have to be combined to an integrated whole. Modular or service-oriented architectures with loosely coupled components are supposed to be well suited for this purpose.

SpicaML provides modelling support for data flow between independent modules. Automatically generated stub components and the Geminga resource discovery approach take care of the communication infrastructure and its dynamic reconfiguration. Section 3.3 introduces Geminga and Chapter 9 presents experimental results gathered from real-world applications.

11.1.2 Communication Infrastructure

The Spica communication infrastructure facilitates the interactions between distributed modules. It is responsible for handling heterogeneity issues and providing a consistent interface to group communication. The discussion below reviews the respective requirements defined in Section 1.2.2.

Message Orientation As motivated in Section 1.2.2, an event or message-oriented communication scheme suits the data delivery behaviour of sensors best. A certain operational frequency furthermore allows a system to mimic natural behaviour, the continuous character of which cannot be mapped directly onto computing machinery that operates in discrete domain. Fault-tolerant behaviour is required because of unreliable communication media and the possibility for dynamic reconfiguration.

Spica covers data loss robustness by only adopting measures that do without a history of past messages. AES in CTR mode as discussed in Section 2.5 is an example for this. Unreliable transport protocols mostly address failure robustness. Chapter 7 and Chapter 8 give examples that both rely on a multicast communication scheme. Section 2.4 builds the foundation for the communication approach followed by Spica.

Service Guarantees Specialised appliances depend on more advanced service guarantees for the communication infrastructure. This includes reliable data transmission, delivery of data with arbitrary length, and suitable group communication schemes.

Spica builds on repeated data transmission for simulating reliable data transmission and multicast for group communication. Message fragmentation is the most suitable measure when considering the delivery of data with arbitrary length. SpicaML allows enabling fragmentation conditionally on a per-message basis as shown in Section 6.4.1. More service guarantees may be added if required.

Compression Data compression is neither a real service guarantee nor a security measure. This is why it is an independent category. Spica provides a message capability for compression in SpicaML that may be enabled as required. Compression nevertheless makes quite high demands on the processing capabilities as shown in Section 10.3.2.2. For this reason, it is not equally well suited for all application scenarios.

Communication Security Security finally deals with the privacy deficiencies of communication media and in particular with weaknesses in the network configuration. Preventing forgery and unauthorised eavesdropping is thereby the primary goal.

Spica provides appropriate measures that address both requirements. Message authentication based on HMAC-MD5 guarantees authenticity of messages and their payload. As a highly efficient solution, it further meets the requirements of resource-efficiency. Confidentiality measures depend on per-message AES encryption. Both are introduced in Section 2.5 and thoroughly analysed in Section 10.3.2.3.

11.1.3 Resource Discovery

Section 1.2.3 discusses the requirements for a resource discovery approach in unreliable ad hoc networks. Fault tolerant operation and efficiency in resource discovery are the most prevalent demands in this case. We will now sum up to what extend Geminga addresses these requirements.

Fault-Tolerance As already demanded for the Spica communication infrastructure, error conditions must be handled in a fault-tolerant fashion. This partly implies that the resource discovery approach must be capable of unattended, dynamic reconfiguration.

Geminga provides fault-tolerance by renouncing elaborate protocol interactions. Instead, it settles for a periodical, announcement-based approach as outlined in Section 8.4. Dynamic reconfigurability is introduced in Section 8.1, combining a timeout-based approach with periodical announcements.

Distributed operation Application in highly dynamic scenarios argues for completely decentralised operation of a resource discovery approach. Configuration should furthermore be stored redundantly in order to anticipate network partitioning or other forms of communication breakdown. This retains the ability for participants to access to the configuration even in case of network unavailability.

Geminga assures both requirements by providing local storage for all received announcements. Resource discovery operates solely on this pool without further network interaction. Section 8.4 and Section 8.5 characterise this procedure in more detail.

Applicability Resource discovery in a Spica communication infrastructure is required to operate in a resource-efficient and platform-independent fashion. Section 8.1 introduces the Geminga announcement approach with the respective message format. Both adhere to resource efficiency as shown in Section 10.3.2. Geminga itself is a self-contained service. The matching process depends on an efficient data structure that maintains indices on announcement information. Clients may connect and interact with the service via GCNP as introduced in Section 8.3. This separation effectively minimises mutual dependencies and improves cross-platform applicability. Ready-to-use interface stubs are provided for several languages.

11.1.4 Summary

The Spica development framework addresses all the requirements in a largely satisfying way. Its services address several different architectures – thereby covering both hardware and software issues – but retain Spica's applicability in even challenging environments. It furthermore deals with the configuration of the automatically generated, distributed communication infrastructure in a straightforward manner, requiring almost no manual intervention.

The only weakness in the whole development approach is the template-based code generation. Code templates abdicate from the control of Spica by requiring developers to deal with their implementation and especially with the peculiarities of the target platform. As central parts of Spica, they depend too much on the skills of the template creator. Another solution is advised for this problem. Section 11.3 further discusses this topic.

11.2 Contributions

This thesis contributes to the ongoing research in robotic software development mainly by proposing a model-driven development approach. With Spica, we have presented an integrated development framework that aims at platform-independent development of an RCA. It furthermore provides developers with the ability to integrate third-party components into a consistent communication infrastructure with only low effort. We thereby pay special attention to the unreliability in the communication media.

Platform-independence is the first and by far most important characteristic of the Spica development framework. It pervades the whole framework, from the modelling down to the source layer, allowing developers to address several platforms with only a single model specification at once. Cross-platform development is thereby not only required for different hardware platforms. The composition of different software solutions is rather a much more common necessity in robotic software architectures. Spica adopts an MDSD approach with SpicaML as a domain-specific modelling language. SpicaML is tailored to the application domain of groups of AMRs, capturing important elements thereof in abstract modelling entities. It facilitates the creation of communication infrastructures for loosely coupled, self-contained components. The abstract definition of data structures further provides modelling means for exchangeable data.

This leads to another issue that is most evident in modular architectures made up of several independent processes: Communication behaviour and data representation are incompatible in most cases. Similar issues exist in monolithic programs where incompatibilities between APIs or ABIs (Application Binary Interface) are prevalent. The diversity of transports and communication schemes, protocols, and data representation approaches for interfacing modular software components nevertheless leads to an almost unmanageable number of possible solutions. This is where the Spica development framework and its modelling language SpicaML simplify interoperation: They abstract from the underlying communication behaviour and provide a unified view for all participating components. Most distributed middleware architectures provide similar simplifications that address multiple platforms. The most important difference of Spica directly anticipates the characteristics of communication in mobile robots: Event-orientation in Spica explicitly addresses the communication behaviour of mobile robots as well as the unreliability of the communication media.

Mobile robots typically interact through wireless links as outlined in Section 2.4. Most middleware architectures thereby intend to provide invocation transparency. Reliable communication is a base requirement in this case,[1] which instantly disqualifies RMI for application in unreliable ad hoc networks. Spica, in contrast, realises message delivery only and leaves the decision for a transport protocol to the developer. All viable protocols are allowed but unreliable data transmission is preferred. For addressing several recipients, a group communication scheme such as multicast may be used. The Spica communication infrastructure furthermore does without a central message hub or broker, aiming at full distribution and independent operation. The entirely message-oriented communication behaviour further takes sensor acquisition and processing characteristics into account. It is thereby possible to poll sensors or to let them deliver data asynchronously. Processing follows this behaviour. Spica takes the burden from developers having to care about this. Instead, they have to use a more or less generic interface that facilitates delivery and reception of messages.

This interface is generated automatically from the abstract SpicaML model specification. Aastra here passes the interpreted and transformed model to a template engine that generates concrete source code for a given target platform. Even though template creation is expensive, it simplifies the integration of distributed components. The Spica communication infrastructure handles interaction management. Geminga thereby cares about the configuration of the respective communication channels.

Geminga is a resource discovery approach that most notably facilitates the establishment of a Spica communication infrastructure. It is capable of further functionality but mainly used for this purpose. The module stubs generated from the abstract SpicaML data flow model automatically, register themselves with Geminga. It thereby informs the stubs of the availability of new resources and consequently triggers the establishment of new communication channels. DMCs – generic message buffers with more elaborate functionality – take care of message handling and synchronisation.

[1]RMI is typically capable of one-way method calls, degrading the call to a delivery of a message. Ordinary RMI calls, however, involve at least an answer-request protocol to be executed. This, in turn, depends on reliable transmission as RMI calls typically block.

Even though no concrete code template was introduced in this thesis, the Spica development framework demonstrated its effectiveness for real-world application in Chapter 9 and during RoboCup tournaments. With the Geminga resource discovery approach, Spica furthermore offers a distributed discovery approach that is in particular applicable for unreliable communication scenarios.

11.3 Open Questions

Autonomous robots software is still evolving; so is their hardware. The whole discipline of robotics will presumably need to fulfil additional requirements and adapt to new challenges. As a result, research will steadily grow and require new measures to overcome future's problems. Below, we outline some of the open questions that remain unsolved in this thesis. We cannot claim completeness but instead restrict our view on the scope of this application domain.

11.3.1 Superseding Template Creation

The creation of code generation templates is still a complex task in Spica. This stems from the fact that templates have to be generic programs that are capable of handling a variety of different parameters and configurations. The expressiveness of a template description language furthermore restricts programmers in their ability to create highly optimised code.

A separation that splits a template in an algorithmic and a platform-specific part would retain the use of templates by lowering the effort for reimplementation. An experimental realisation using the Spica code generation templates nevertheless revealed severe problems: Most implementations contain a huge part of platform or programming language-specific functionality that cannot be generalised in an algorithmic fashion. It is therefore difficult to find the least common denominator of a process description that is still applicable for code generation, covering all possible languages. Nevertheless, this approach should be further scrutinised, maybe with additional intermediate transformation steps.

Ontology-supported code generation [140, 50] is another approach that nevertheless hits the same issues: An abstract algorithmic description has to be mapped onto several concrete platforms. It has yet to be figured out whether this approach is practicable or not.

11.3.2 Automatic Data Conversion

Another open issue is the automatic conversion of data. Incoming messages from third-party components may be structured differently and include values in incompatible types or measures. The transformation between different structures is thereby possible but may require further knowledge or semantic annotations. Computing new values from series of data scattered all over one or more data structures is thereby challenging. This is nevertheless an important capability for fully automated code generation and adaptation. In particular, the automatic conversion of physical units, both the trivial and non-trivial ones, has to be examined.

This will presumably require semantic metadata to be available for each data type and each value. Automatic inference is only possible for a system that possesses additional information describing the data. This, in turn, implies increased specification expense, more complex data representation, and rising space requirements for serialised data.

The pros and cons of new results must be compared to the currently dominating approaches.[2] It is further still unsolved how to integrate a priori unknown data – i.e. the conversion rules may be determined for each value anew or created once and forever – and what performance penalty will be involved. An approach similar to that of Section 11.3.1 above [140, 50] might be applicable here as well.

11.3.3 Tighter Integration of Semantics

The integration of semantic annotations in Spica is mainly of conceptual nature, building merely the foundation for automatic conversion of physical units in combination with DADL. It would nevertheless improve data processing by adding custom filter routines. Together with a concrete ontology, inference would furthermore be possible, enabling developers to preprocess data in a platform-independent fashion. This, in turn, implies a reduction in implementation effort and maybe in the number of software bugs.

The downside of this functionality is clearly an increase in template complexity. A suitable solution to this problem is essential in order to retain the ability for adaptation and extensions of the generated code.

11.3.4 Tighter Integration with Existing Frameworks

The integration with existing RCAs is not yet satisfying in Spica. While MARIE [34] provides several interfaces for well-established software components, adapter modules in Spica still have to be implemented manually. This is mostly due to the need for manual data conversion. Automating this process as outlined above would clearly be profitable, yet a realisation is still not tangible.

We scheduled the development of predefined adapter modules for well-established software architectures and components for the near future. This will enable developers to prototype an RCA out of existing components. Interfaces towards the MSRDS [74], Player/Stage [51], Orocos [24], and Carmen [88] as well as more elaborate interfaces for Miro [129, 128] and Gazebo [77] are thereby of highest interest.

11.4 Outlook and Future Work

The Spica development framework in its current state substantially facilitates the integration of existing components. Automatically generated implementations provide a consistent development experience on heterogeneous platforms and self-configuration capabilities for the communication infrastructure. Desired functionality that would further improve Spica's value is nevertheless still missing.

[2]In the majority of currently available approaches, all systems within an application domain agree upon a common vocabulary and grammar.

Section 11.3.4 already mentioned the need for a tighter integration of Spica with existing solutions. Such support is scheduled for integration in the near future. Especially RCAs would benefit from the additional functionality and a simplified integration of third-party components. As an example, CaNoSA as outlined in Section 9.1 lacks an elaborate simulation engine. The integration of Gazebo as outlined in Section 9.3 is a first step towards more elaborate simulation capabilities.

Support for logging and debugging should be further extended. Work is currently in progress that deals with extended logging support, including remote logging capabilities based on SpicaDMF. We consider appropriate filtering of provided logging information indispensable, as it builds the foundation for knowledge generation from raw data. The filters thereby have to be somewhat smart, incorporating more or less elaborate heuristics or textual analysis, for instance.

Apart from error detection on the processing level, communicated data is a valuable source of information. A decoder module for Wireshark [80, 79] will be developed, based on the automatically generated data structure implementation. This module is then responsible for interpreting communicated data or capturing data stream from single modules.

A new component that makes heavy use of communication capabilities is currently under development for CaNoSA: A plan-based behaviour engine as a replacement for the current one. It explicitly addresses modelling of cooperative team behaviour that clearly depends on information exchange between participants. It will furthermore be offered as an independent module for use outside of CaNoSA. Access is thereby realised by SpicaML-based module stubs and the Spica communication infrastructure.

We plan to use Spica for other robotic and non-robotic appliances as well. The development of a CaNoSA game controller – a component similar to the `LebtClient` module shown in Figure 9.2 – is scheduled for application in PDAs. It will adopt an automatically generated module stub, realising the interface towards the Spica communication infrastructure. Embedded systems are another valuable target platform. Several electronic components in the Carpe Noctem hardware architecture may be equipped with SpicaML-based data structure implementations, enabling direct interaction with CaNoSA components.

The Spica development framework is a viable alternative to currently available middleware platforms. We demonstrated its applicability during several RoboCup tournaments. Platform independence and automatic configurability proved to be beneficial for the development of groups of mobile robots with heterogeneous hardware and software components in mobile ad hoc networking environments.

A ANTLR v3 Grammar for SpicaML

The ANTLR v3 grammar for SpicaML is shown below. The `output` option (line 8) instructs ANTLR v3 to generate a tree parser for the SpicaML grammar. The entry point is defined by the `model` rule (line 11) in this case. Lines 30–79 contain the parser rules for data structures, lines 82–135 the respective rules for data flow. Lines 138–252 furthermore define a grammar for DADL.

```
 1  // SpicaML grammar for ANTLR v3
 2  //
 3  // 2006-2008 by Carpe Noctem Robotic Soccer
 4  // Distributed Systems Group, Kassel University, Germany
 5  // http://carpenoctem.das-lab.net/
 6
 7  grammar SpicaML;
 8  options { language = CSharp; output = AST; }
 9
10  // Entrypoint for Aastra.
11  model
12      :   model_spec ( struct_spec | flow_spec )* ;
13
14  // Grammar rules for the global SpicaML model structure
15  model_spec
16      :   ( model_prefix )* model_namespace? model_hash?
17      ;
18  model_prefix
19      :   'urnpfx' model_urn_prefix model_urn_id_open ';'
20          -> ^( URNPFX model_urn_prefix model_urn_id_open )
21      ;
22  model_hash
23      :   'hash' ID ';' -> ^( HASH ID )
24      ;
25  model_namespace
26      :   'namespace' model_id ( '.' model_id )* ';' -> ^( NAMESPACE model_id+ )
27      ;
28
29  // Grammar rules for the data structure specification
30  struct_spec
31      :   ( struct_enum | struct_header | struct_container | struct_message )
32      ;
33  struct_enum
34      :   'enum' struct_sig ':' model_id_prim '{' struct_enum_value+ '}'
35          -> ^( ENUM model_id_prim struct_sig struct_enum_value+ )
36      ;
37  struct_header
38      :   'header' struct_sig ( '{' struct_block_field_default* '}' | ';' )
```

```
        -> ^( HEADER struct_sig struct_block_field_default* )
    ;
struct_container
    :   'container' struct_sig ( '{' struct_block_field_default* '}' | ';' )
        -> ^( CONTAINER struct_sig struct_block_field_default* )
    ;
struct_message
    :   'message' struct_sig ( '{' struct_block_field_message* '}' | ';' )
        -> ^( MESSAGE struct_sig struct_block_field_message* )
    ;
struct_block_field_default
    :   model_type struct_block_field
        -> ^( FIELD model_type struct_block_field)
    ;
struct_block_field_message
    :   model_type_container struct_block_field
        -> ^( FIELD model_type_container struct_block_field)
    ;
struct_block_field
    :   struct_array? model_id_all struct_default_value? model_ann? ';'
        -> struct_array? ^( NAME model_id_all ) struct_default_value?
           model_ann?
    ;
struct_sig
    :   model_type struct_super_type? model_ann?
    ;
struct_super_type
    :   ':' model_type -> ^( INHERIT model_type )
    ;
struct_default_value
    :   '=' model_value -> ^( DEFAULT model_value )
    ;
struct_enum_value
    :   ID ( '=' model_value )? ';' -> ^( ITEM ID model_value? )
    ;
struct_array
    :   (
         '[' ( T_INT ( ',' T_INT )* )? ']' -> ^( ARRAY T_INT* ) |
         '[]' -> ^( ARRAY )
        )
    ;

// Grammar rules for the data flow specification
flow_spec
    : flow_module
    ;
flow_module
    :   'module' model_type model_ann? '{'
            (
             flow_module_offer | flow_module_request |
             flow_define | analysis_filter
            )+
        '}'
        -> ^(
             MODULE model_type model_ann? flow_module_offer*
             flow_module_request* flow_define* analysis_filter*
            )
    ;
flow_module_offer
    :   'offer' 'message' model_type model_ann?
        '->' flow_peers? 'scheme' flow_scheme flow_dmc+ ';'
        -> ^(
             OFFER model_type model_ann? flow_peers?
             flow_scheme flow_dmc+
            )
    ;
flow_module_request
    :   'request' 'message' model_type model_ann?
        '<-' flow_peers? 'scheme' flow_scheme flow_dmc+ ';'
        -> ^(
             OFFER model_type model_ann? flow_peers?
             flow_scheme flow_dmc+
            )
    ;
flow_define
    :   'define' flow_dmc ';' -> ^(DEFINE flow_dmc)
    ;
flow_peers
    :   flow_peer_single ( ',' flow_peer_single )* -> flow_peer_single*
    ;

flow_peer_single
    :   model_type model_ann? -> ^(PEERS model_type model_ann? )
    ;
flow_dmc
    :   'dmc' flow_dmc_type flow_dmc_array? model_id model_ann
        -> ^( DMC flow_dmc_type model_id flow_dmc_array? model_ann )
    ;
flow_dmc_array
    :   '[]' -> ^( ARRAY )
    ;
flow_dmc_type
    :   ( 'ringbuffer' | 'queue' | 'priority_queue' )
    ;
flow_scheme
    :   ( 'oto' | 'otm/stream' | 'otm/pubsub' )
    ;

// Grammar rules for the Data Analysis Decription Language
analysis_filter
    :   'filter' model_id '{'
            flow_dmc+
            ( analysis_call | analysis_predefined )
        '}'
        -> ^(
             FILTER model_id flow_dmc+ analysis_call?
             analysis_predefined?
            )
    ;
analysis_call
    :   'call' analysis_func_name? model_ann '{' analysis_source* '}'
        -> ^( CALL analysis_func_name? model_ann analysis_source* )
    ;
analysis_predefined
    :   'predefined' analysis_func_name? model_ann '{' analysis_source* '}'
        -> ^( PREDEFINED analysis_func_name? model_ann analysis_source* )
    ;
analysis_func_name
    :   model_id -> ^( NAME model_id ) ;
analysis_source
    :   (
         analysis_identifier
         (
          '=' analysis_expression ';'
          -> ^( ASSIGN analysis_identifier analysis_expression)
         |
          ';'
          -> ^( CALL analysis_identifier )
         )
        |
         analysis_for -> analysis_for | analysis_if -> analysis_if
        )
    ;
analysis_expression
    :   analysis_expression_element
        (
         ( '+'^ | '-'^ ) analysis_expression_element
        )*
    ;
analysis_expression_element
    :   analysis_expression_element_part
        (
         ( '*'^ | '.*'^ | '/'^ | './'^ | '^'^ | '.^'^ )
         analysis_expression_element_part
        )*
    ;
analysis_expression_element_part
    :   (
         analysis_identifier -> analysis_identifier |
         '(' analysis_expression ')' -> ^( GROUP analysis_expression )
        )
    ;
analysis_function
    :   (
         model_id '(' first=T_INT? ':' second=T_INT? ')'
         -> ^( EALL model_id $first? $second? )
        |
         model_id '('
         ( analysis_expression ( ',' analysis_expression )* )?
         ')'
         -> ^( FUNCTION model_id analysis_expression* )
        )
    ;
analysis_identifier
    :   (
         T_INT -> ^( VALUE T_INT ) |
         T_FLOAT -> ^( VALUE T_FLOAT ) |
         T_STRING -> ^( VALUE T_STRING) |
         analysis_identifier_var ( '.' analysis_identifier_var )*
         -> ^( IDENTIFIER analysis_identifier_var+ )
        )
    ;
analysis_identifier_var
    :   (
         analysis_vector -> analysis_vector |
         model_id -> ^( VARIABLE model_id ) |
         analysis_function -> analysis_function
        )
    ;
analysis_vector
    :   '[' analysis_identifier ( ',' analysis_identifier )* ']'
        -> ^( VECTOR analysis_identifier+ )
    ;
analysis_for
    :   'for' model_id '=' analysis_identifier ':' analysis_identifier
            analysis_source*
        'end' ';'
        -> ^(
             FOR model_id
             ^( ARGS analysis_identifier* )
             ^( BODY analysis_source* )
            )
    ;
analysis_if
    :   'if' '(' analysis_eval ')' analysis_source* 'end' ';'
        -> ^( IF analysis_eval analysis_source* )
    ;
analysis_eval
    :   analysis_eval_element ( '||'^ analysis_eval_element )*
    ;
analysis_eval_element
    :   analysis_eval_element_part ( '&&'^ analysis_eval_element_part )*
    ;
analysis_eval_element_part
    :   (
         analysis_bool_expr -> analysis_bool_expr |
         '(' analysis_eval ')' -> ^( GROUP analysis_eval )
        )
    ;
analysis_bool_expr
    :   analysis_identifier
        ( '=='^ | '!='^ | '<'^ | '<='^ | '>'^ | '>='^ )
        ( analysis_identifier | '(' analysis_eval ')' )
    ;

// Grammar rules for miscellaneous purposes
```

```
model_ann
    :   ( '[' model_ann_kv ( ';' model_ann_kv )* ']' ) -> model_ann_kv*
    ;
model_ann_kv
    :   ( model_ann_kv_spec_address | model_ann_kv_generic )+
    ;
model_ann_kv_spec_address
    :   ( 'address' '=' T_STRING ) -> ^( ANNOTATION 'address' T_STRING )
    ;
model_ann_kv_generic
    :   model_id ( '=' ( model_value ( ',' model_value )* ) )?
        -> ^( ANNOTATION model_id model_value* )
    ;
model_value
    :   (
         T_UINT | T_INT | T_FLOAT | T_TIME | T_STRING | TRUE | FALSE |
         model_urn_id | model_type
        )
    ;
model_id_prim
    :    (
          BOOL | UINT8 | UINT16 | UINT32 | UINT64 | INT8 | INT16 |
          INT32 | INT64 | FLOAT | DOUBLE | STRING | ADDRESS | AUTO
         )
    ;
model_type_container
    :   model_id model_variant?
        -> ^( TYPENAME model_id ) model_variant?
    ;
model_type_cov
    :   'cov' '(' model_id ( ',' model_id )* ')'
        -> ^( COV model_id+ )
    ;
model_type
    :   model_id_prim | model_type_container | model_type_cov
    ;
model_variant
    :   '<' model_id ( ',' model_id )* '>'
        -> ^( VARIANT model_id+ )
    ;
model_urn_id
    :   ( '#' )? model_id_all ( ':' model_id_all )+
        -> ^( URN '#'? model_id_all+ )
    ;
model_urn_prefix
    :   ( '#' | model_id )
    ;
model_urn_id_open
    :   model_urn_id ( ':' )? -> model_urn_id
    ;
model_id
    :
        URN | URNPFX | HASH | VARIANT | HEADER | CONTAINER | MESSAGE |
        DEFAULT | COV | INHERIT | ARRAY | ENUM | ITEM | FIELD |
        ANNOTATION | VECTOR | VIEW | DMC | RINGBUFFER | QUEUE |
        MODULE | OFFER | REQUEST | ANNOUNCE | STATIC | FILTER | CALL |
        FUNCTION | VARIABLE | VALUE | ARGS | EALL | ASSIGN | BODY |
        NAME | IDENTIFIER | GROUP | TYPENAME | NAMESPACE | SCHEME |
        OTO | OTMS | OTMPS | PEERS | DEFINE | ID
    ;
model_id_all
    :   model_id | BOOL | UINT8 | UINT16 | UINT32 | UINT64 | INT8 |
        INT16 | INT32 | INT64 | FLOAT | DOUBLE | STRING | AUTO
    ;

URNPFX     : 'urnpfx'     ; HASH     : 'hash'     ; VARIANT   : 'variant'    ;
CONTAINER  : 'container'  ; HEADER   : 'header'   ; MESSAGE   : 'message'    ;
DEFAULT    : 'default'    ; COV      : 'cov'      ; INHERIT   : 'inherit'    ;
ARRAY      : 'array'      ; ENUM     : 'enum'     ; ITEM      : 'item'       ;
FIELD      : 'field'      ; VIEW     : 'view'     ; URN       : 'urn'        ;
ANNOTATION : 'annotation' ; VECTOR   : 'vector'   ; DMC       : 'dmc'        ;
RINGBUFFER : 'ringbuffer' ; QUEUE    : 'queue'    ; MODULE    : 'module'     ;
ANNOUNCE   : 'announce'   ; OFFER    : 'offer'    ; REQUEST   : 'request'    ;
STATIC     : 'static'     ; FILTER   : 'filter'   ; CALL      : 'call'       ;
FUNCTION   : 'function'   ; VARIABLE : 'variable' ; VALUE     : 'value'      ;
ARGS       : 'args'       ; EALL     : 'eall'     ; ASSIGN    : 'assign'     ;
BODY       : 'body'       ; IF       : 'if'       ; FOR       : 'for'        ;
PREDEFINED : 'predefined' ; NAME     : 'name'     ; GROUP     : 'group'      ;
IDENTIFIER : 'identifier' ; TYPENAME : 'typename' ; PEERS     : 'peers'      ;
DEFINE     : 'define'     ; AUTO     : 'auto'     ; NAMESPACE : 'namespace'  ;
SCHEME     : 'scheme'     ; OTO      : 'oto'      ; OTMS      : 'otm/stream' ;
OTMPS      : 'otm/pubsub' ; TRUE     : 'true'     ; FALSE     : 'false'      ;
BOOL       : 'bool'       ; UINT8    : 'uint8'    ; UINT16    : 'uint16'     ;
UINT32     : 'uint32'     ; UINT64   : 'uint64'   ; INT8      : 'int8'       ;
INT16      : 'int16'      ; INT32    : 'int32'    ; INT64     : 'int64'      ;
FLOAT      : 'float'      ; DOUBLE   : 'double'   ; STRING    : 'string'     ;
ADDRESS    : 'address'    ;

fragment UINT
    :   ( '0'..'9' )+ ;

ID
    :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
    ;
T_UINT
    :   UINT
    ;
T_INT
    :   ('-'|'+')? UINT
    ;
T_FLOAT
    :   ( T_INT '.' UINT? | '.' UINT)
    ;
T_STRING
    :   '"' (ESC | ~('\\'|'"'))* '"'
    ;
T_TIME
    :   ( ( T_INT | T_FLOAT ) ( 'm' | 's' | 'ms' | 'us' | 'ns' ) )
    ;

protected
ESC
    :   '\\' ('n' | 'r')
    ;
WS
    :   ('␣'|'\r'|'\t'|'\u000C'|'\n') { $channel=HIDDEN; }
    ;

COMMENT
 : (
   '/*' (options {greedy=false;} : .)* '*/' { $channel=HIDDEN; } |
   '//' ~('\r'|'\n')* { $channel=HIDDEN; }
  )
 ;
```

Nomenclature

AAS	Abstract Architecture Specification
AEAD	Authenticated Encryption with Associated Data
AIR	Abstract Intermediate Representation
AMR	Autonomous Mobile Robots
ASN.1	Abstract Syntax Notation One
AST	Abstract Syntax Tree
BCM	Block Cipher Mode
BTD	Block Type Definition
BTI	Blok Type Instance
CLR	Common Language Runtime
CNER	Carpe Noctem Encoding Rules
DADL	Data Analysis Definition Language
DFD	Data Flow Diagram
DMC	Data Management Container
DSL	Domain-Specific Language
DSML	Domain-Specific Modelling Language
GCNP	Geminga Client Notification Protocol
IDL	Interface Definition Language
INVP	Invocation Policy
ISM	Industrial, Scientific, and Medical radio band
MANET	Mobile Ad Hoc Network
MDA	Model-Driven Architecture
MDSD	Model-Driven Software Development

MDSE	Model-driven Software Evolution
MSIL	Microsoft Intermediate Language
NMC	NotifyMulticast
NV	Name Variant
RCA	Robot Control Architecture
RMR	Receiver Makes it Right
RR	Resource Record
SBS	Shared Belief State
SDP	Structure Duplicate Policy
SERI	SpicaML Encoding Rule Interface
SIP	Structure Insertion Policy
SMR	Sender Makes it Right
SOA	Service-Oriented Architecture
SOA	Service-Oriented Architecture
SpicaDMF	Spica Decentralised Monitoring Facility
SpicaML	Spica Modelling Language
SRP	Structure Retrieval Policy
SYNI	Synchronisation Interval
SYNP	Synchronisation Policy
TCB	Target Configuration Block
TC	Spica Target Configuration
URN	Uniform Resource Name
VI	Variant Identifier
WLAN	Wireless Local Area Network

Bibliography

A

[1] Hajime Akashia and Hiromitsu Kumamoto.
Random sampling approach to state estimation in switching environments.
Automatica, 13(4):429–434, July 1977.
doi:10.1016/0005-1098(77)90028-0.

[2] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose.
DNS Security Introduction and Requirements.
RFC 4033 (Proposed Standard), March 2005.
URL: `http://www.ietf.org/rfc/rfc4033.txt` (cited 2008-08-23).

[3] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose.
Resource Records for the DNS Security Extensions.
RFC 4034 (Proposed Standard), March 2005.
Updated by RFC 4470.
URL: `http://www.ietf.org/rfc/rfc4034.txt` (cited 2008-08-23).

[4] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose.
Protocol Modifications for the DNS Security Extensions.
RFC 4035 (Proposed Standard), March 2005.
Updated by RFC 4470.
URL: `http://www.ietf.org/rfc/rfc4035.txt` (cited 2008-08-23).

[5] Karl Johan Aström.
Optimal control of Markov decision processes with incomplete state estimation.
Journal of Mathematical Analysis and Applications, 10:403–406, 1965.

B

[6] Philipp A. Baer.
Group Authentication and Encryption in Distributed Environments.
Master's thesis, University of Ulm, June 2004.
URL: `http://npw.net/pub/phbaer/private/thesis.pdf` (cited 2008-08-23).

[7] Philipp A. Baer and Roland Reichle.
Robotic Soccer, chapter Communication and Collaboration in Heterogeneous Teams of Soccer Robots, pages 1–28.
I-Tech Education and Publishing, Vienna, Austria, December 2007.
ISBN: 978-3-902613-21-9.
URL: `http://purl.org/spica/robotic-soccer-ars` (cited 2008-09-03).

[8] Philipp A. Baer, Roland Reichle, Michael Zapf, Thomas Weise, and Kurt Geihs.
A Generative Approach to the Development of Autonomous Robot Software.

In *Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe 2007)*, pages 43–52, March 2007.
doi:10.1109/EASE.2007.2.

[9] Philipp A. Baer, Roland Reichle, and Kurt Geihs.
The Spica Development Framework – Model-Driven Software Development for Autonomous Mobile Robots.
In Wolfram Burgard, Rüdiger Dillmann, Christian Plagemann, and Nikolaus Vahrenkamp, editors, *Intelligent Autonomous Systems 10 – IAS-10*, pages 211–220. IOS Press, July 2008.
ISBN: 978-1-58603-887-8.

[10] Philipp A. Baer, Thomas Weise, and Kurt Geihs.
Geminga: Service Discovery for Mobile Robotics.
In *The Third International Conference on Systems and Networks Communications*. IEEE Computer Society Press, October 2008.
URL: `http://purl.org/spica/icsnc2008` (cited 2008-09-03).

[11] Krishnakumar Balasubramanian, Arvind S. Krishna, Emre Turkay, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt.
Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems.
International Journal of Embedded Systems, Special Issue on Design and Verification of Real-Time Embedded Software, 2(3/4):142–155, April 2005.
doi:10.1504/IJES.2006.014851.

[12] Genevieve Bartlett, John Heidemann, and Christos Papadopoulos.
Understanding passive and active service discovery.
In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement (IMC 2007)*, pages 57–70, New York, NY, USA, 2007. ACM.
ISBN: 978-1-59593-908-1.
doi:10.1145/1298306.1298314.

[13] Stefano Basagni, Marco Conti, Silvia Giordano, and Ivan Stojmenovic, editors.
Mobile Ad Hoc Networking.
Wiley-IEEE Press, August 2004.
ISBN: 978-0-47137-313-1.
doi:10.1002/0471656895.

[14] Pete Becker.
The C++ Standard Library Extensions: A Tutorial and Reference.
Addison-Wesley Professional, 2006.
ISBN: 978-0321412997.

[15] Mihir Bellare, Ran Canetti, and Hugo Krawczyk.
The HMAC Construction.
CryptoBytes, 2(1):12–15, 1996.
URL: `http://purl.org/spica/crypto2n1` (cited 2008-09-03).

[16] Mihir Bellare, Phillip Rogaway, and David Wagner.
The EAX Mode of Operation.
In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption*, volume 3017/2004

of *Lecture Notes in Computer Science*, pages 389–407. Springer Berlin/Heidelberg, 2004.
ISBN: 978-3-540-22171-5.
doi:10.1007/b98177.

[17] Ralph Bergmüller, Andrew F. Russell, Rufus A. Johnstone, and Redouan Bshary.
On the further integration of cooperative breeding and cooperation theory.
Behavioural Processes, 76(2):170–181, 2007.
doi:10.1016/j.beproc.2007.06.013.

[18] Andreas Birk and Cosmin Condea.
Mobile Robot Communication without the Drawbacks of Wireless Networking.
In Ansgar Bredenfeld, Adam Jacoff, Itsuki Noda, and Yasutake Takahashi, editors, *RoboCup 2005: Robot Soccer World Cup IX*, volume 4020 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 585–592. Springer Berlin/Heidelberg, 2006.
ISBN: 978-3-540-35437-6.
doi:10.1007/11780519.

[19] Ulysses D. Black.
Physical Layer Interfaces and Protocols.
IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.
ISBN: 0-818656-97-2.

[20] *Specification of the Bluetooth System v2.1 + EDR*.
Bluetooth SIG, July 2007.
URL: `http://purl.org/spica/bluetooth-v21-edr` (cited 2008-09-03).

[21] Eric Bonabeau.
Editor's Introduction: Stigmergy.
Artificial Life, 5(2):95–96, 1999.
doi:10.1162/106454699568692.

[22] Don Box and Ted Pattison.
Essential .NET: The Common Language Runtime.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
ISBN: 978-0201734119.

[23] Rodney A. Brooks.
A Robust Layered Control System for a Mobile Robot.
IEEE Journal of Robotics and Automation, 2(1):14–23, March 1986.
ISSN 0882-4967.
URL: `http://purl.org/spica/brooks1985` (cited 2008-09-03).

[24] Herman Bruyninckx.
Open Robot Control Software: the OROCOS project.
volume 3, pages 2523–2528. IEEE, 2001.
ISBN: 0-7803-6578-X.
doi:10.1109/ROBOT.2001.933002.

[25] Herman Bruyninckx, Peter Soetens, and Bob Koninckx.
The real-time motion control core of the Orocos project.
In *ICRA*, volume 2, pages 2766–2771. IEEE, September 2003.
ISBN: 0-7803-7736-2.

C

[26] Stuart Cheshire and Marc Krochmal.
DNS-Based Service Discovery.
IETF (Internet Draft), August 2006.
URL: `http://purl.org/spica/dns-sd` (cited 2008-09-03).

[27] Stuart Cheshire and Marc Krochmal.
Multicast DNS.
IEFT (Internet Draft), August 2006.
URL: `http://purl.org/spica/mdns` (cited 2008-09-03).

[28] Stuart Cheshire, Bernard Aboba, and Erik Guttman.
Dynamic Configuration of IPv4 Link-Local Addresses.
RFC 3927 (Proposed Standard), May 2005.
URL: `http://www.ietf.org/rfc/rfc3927.txt` (cited 2008-08-24).

[29] Alain Colmerauer and Philippe Roussel.
The birth of Prolog.
pages 331–367, 1996.
doi:10.1145/234286.1057820.

[30] Alan Colvin.
CSMA with collision avoidance.
Computer Communications, 6(5):227–235, October 1983.
doi:10.1016/0140-3664(83)90084-1.

[31] *Universal Serial Bus Revision 2.0 specification*.
Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips, April 2000.
URL: `http://purl.org/spica/usb-v20` (cited 2008-09-03).

[32] Mathew Scott Corson and Joseph Macker.
Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations.
RFC 2501 (Informational), January 1999.
URL: `http://www.ietf.org/rfc/rfc2501.txt` (cited 2008-08-24).

[33] Carle Côté, Dominic Létourneau, François Michaud, Jean-Marc Valin, Yannick Brosseau, Clément Raïevsky, Mathieu Lemay, and Victor Tran.
Code Reusability Tools for Programming Mobile Robots.
Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004), 2:1820–1825, September 2004.
doi:10.1109/IROS.2004.1389661.

[34] Carle Côté, Yannick Brosseau, Dominic Létourneau, Clément Raïevsky, and François Michaud.
Robotic Software Integration Using MARIE.
International Journal of Advanced Robotic Systems, Special Issue on Software Development and Integration in Robotics, 3(1):55–60, March 2006.
URL: `http://purl.org/spica/marie2006-ars` (cited 2008-09-03).

D

[35] Stephen E. Deering and Robert M. Hinden.
Internet Protocol, Version 6 (IPv6) Specification.
RFC 2460 (Draft Standard), December 1998.
Updated by RFC 5095.
URL: `http://www.ietf.org/rfc/rfc2460.txt` (cited 2008-08-30).

[36] Arthur P. Dempster.
A generalization of Bayesian inference.
Journal of the Royal Statistical Society, 30(B):205–247, 1968.

[37] L. Peter Deutsch.
DEFLATE Compressed Data Format Specification version 1.3.
RFC 1951 (Informational), May 1996.
URL: `http://www.ietf.org/rfc/rfc1951.txt` (cited 2008-08-24).

[38] R. Droms.
Dynamic Host Configuration Protocol.
RFC 1541 (Proposed Standard), October 1993.
Obsoleted by RFC 2131.
URL: `http://www.ietf.org/rfc/rfc1541.txt` (cited 2008-08-30).

[39] *EIA Standard RS-232-C Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Data Interchange*.
Electronics Industries Association, August 1969.

E

[40] Taher Elgamal.
A public key cryptosystem and a signature scheme based on discrete logarithms.
volume 31, pages 469–472, July 1985.
ISBN: 0-387-15658-5.

[41] Thomas Erl.
Service-Oriented Architecture: Concepts, Technology, and Design.
Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.
ISBN: 0-13-185858-0.

[42] Konrad Etschberger.
Controller Area Network. Basics, Protocols, Chips and Applications.
IXXAT Automation GmbH, October 2001.
ISBN: 978-3000073762.

F

[43] Roy T. Fielding and Richard N. Taylor.
Principled design of the modern Web architecture.
Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000), pages 407–416, 2000.
doi:10.1109/ICSE.2000.870431.

[44] Roy Thomas Fielding.
Architectural Styles and the Design of Network-based Software Architectures.
PhD thesis, University of California, Irvine, 2000.

[45] Lorenzo Flückiger, Vinh To, and Hans Utz.
Service Oriented Robotic Architecture Supporting a Lunar Analog Test.
In *Proceedings of the 9th Symposium on Artificial Intelligence, Robotics, and Automation in Space (iSAIRAS 2008)*, Los Angeles, California, February 2008.
URL: `http://purl.org/spica/sora2008` (cited 2008-09-03).

[46] Dieter Fox, Wolfram Burgard, and Sebastian Thrun.
Markov Localization for Mobile Robots in Dynamic Environments.
Journal of Artificial Intelligence Research (JAIR), 11:391–427, 1999.
URL: `http://purl.org/spica/markovloc1999` (cited 2008-09-03).

G

[47] Klaas Gadeyne.
Sequential Monte Carlo methods for rigorours Bayesian modeling of Autonomous Compliant Motion.
PhD thesis, Katholieke Universiteit Leuven, Faculteit Toegepaste Wetenschappen, Arenbergkasteel, B-3001 Heverlee (Leuven), Belgium, September 2005.
URL: `http://hdl.handle.net/1979/138` (cited 2008-08-24).

[48] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns: Elements of Reusable Object Oriented Software.
Addison-Wesley Professional, Boston, MA, USA, October 1995.
ISBN: 978-0201633610.

[49] Chris Gane and Trish Sarson.
Structured Systems Analysis: Tools and Techniques.
McDonnell Douglas Systems Integration Company, 1977.
ISBN: 0930196007.

[50] Kurt Geihs, Philipp A. Baer, Roland Reichle, and Jens Wollenhaupt.
Ontology-based Automatic Model Transformations.
Sixth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2008), 2008.

[51] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard.
The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems.
pages 317–323, Coimbra, Portugal, June 2003.
URL: `http://purl.org/spica/player2003` (cited 2008-09-03).

[52] Aniruddha S. Gokhale, Douglas C. Schmidt, Tao Lu, Balachandran Natarajan, and Nanbor Wang.
CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Applications.
In *International Middleware Conference, Workshop Proceedings*, pages 300–306, Rio de Janeiro, Brazil, June 2003. PUC-Rio.
ISBN: 85-87926-03-9.

[53] Yaron Y. Goland, Ting Cai, Paul Leach, and Ye Gu.
Simple Service Discovery Protocol/1.0, Operating without an Arbiter.
IETF (Internet Draft), October 1999.
URL: `http://purl.org/spica/ssdp` (cited 2008-09-03).

[54] Nada Golmie, Olivier Rébala, and Nicolas Chevrollier.
Bluetooth adaptive frequency hopping and scheduling.
IEEE Military Communications Conference (MILCOM 2003), 2:1138–1142, October 2003.
doi:10.1109/MILCOM.2003.1290352.

[55] Erik Guttman.
Attribute List Extension for the Service Location Protocol.
RFC 3059 (Proposed Standard), February 2001.
URL: `http://www.ietf.org/rfc/rfc3059.txt` (cited 2008-08-24).

[56] Erik Guttman.
Vendor Extensions for Service Location Protocol, Version 2.
RFC 3224 (Proposed Standard), January 2002.
URL: `http://www.ietf.org/rfc/rfc3224.txt` (cited 2008-08-24).

[57] Erik Guttman, Charles E. Perkins, John Veizades, and Michael Day.
Service Location Protocol, Version 2.
RFC 2608 (Proposed Standard), June 1999.
Updated by RFC 3224.
URL: `http://www.ietf.org/rfc/rfc2608.txt` (cited 2008-08-24).

H

[58] Sumi Helal.
Standards for service discovery and delivery.
Pervasive Computing, IEEE, 1(3):95–100, 2002.
ISSN 1536-1268.
doi:10.1109/MPRV.2002.1037728.

[59] Christian Henke, Carsten Schmoll, and Tanja Zseby.
Empirical Evaluation of Hash Functions for Multipoint Measurements.
ACM SIGCOMM Computer Communication Review, 38(3):39–50, July 2008.
ISSN 0146-4833.
doi:10.1145/1384609.1384614.

[60] Michi Henning.
A New Approach to Object-Oriented Middleware.
IEEE Internet Computing, 8(1):66–75, January/February 2004.
doi:10.1109/MIC.2004.1260706.

[61] Michi Henning.
The rise and fall of CORBA.
ACM Queue, 4(5):28–34, 2006.
ISSN 1542-7730.
doi:10.1145/1142031.1142044.

[62] *IEEE Std 1394b-2002 (Amendment to IEEE Std 1394-1995)*.
IEEE Standard for a High-Performance Serial Bus – Amendment 2.
Institute of Electrical and Electronics Engineers, Inc., 2002.

[63] *IEEE Std 802.15.1-2005 (Revision of IEEE Std 802.15.1-2002)*.

IEEE Standard for Information Technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 15.1: Wireless medium access control (MAC) and physical layer (PHY) specifications for wireless personal area networks (WPANs).
Institute of Electrical and Electronics Engineers, Inc., 2005.

[64] *IEEE Std 802.15.4-2006 (Revision of IEEE Std 802.15.4-2003).*
IEEE Standard for Information Technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs).
Institute of Electrical and Electronics Engineers, Inc., 2006.

[65] *ISO/IEC 8802-11:2005/Amd.4:2006(E) IEEE Std 802.11g-2003 (Amendment to IEEE Std 802.11-1999).*
IEEE Standard for Information Technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications Amendment 4: Further Higher Data Rate Extension in the 2.4 GHz Band.
Institute of Electrical and Electronics Engineers, Inc., 2006.

[66] *ISO/IEC 8802-11:2005/Amd.5:2006(E) IEEE Std 802.11h-2003 (Amendment to IEEE Std 802.11-1999).*
IEEE Standard for Information Technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 11: Wireless Medium Access Control (MAC) and Physical Layer (PHY) specifications Amendment 5: Spectrum and transmit power management extensions in the 5 GHz band in Europe.
Institute of Electrical and Electronics Engineers, Inc., 2006.

[67] *IEEE Std 1394c-2006 (Amendment to IEEE Std 1394-1995).*
IEEE Standard for a High-Performance Serial Bus – Amendment 3.
Institute of Electrical and Electronics Engineers, Inc., 2006.

[68] *IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999).*
IEEE Standard for Information Technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.
Institute of Electrical and Electronics Engineers, Inc., June 2007.

[69] *IEEE Std 802.15.4a-2007 (Amendment to IEEE Std 802.15.4-2006).*
IEEE Standard for Information Technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirement Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs).
Institute of Electrical and Electronics Engineers, Inc., 2007.

[70] *IEEE Unapproved Draft Std P802.11n/D4.00, Mar 2008.*
IEEE Draft STANDARD for Information Technology – Telecommunications and information exchange between systems – Local and metropolitan area networks –

Specific requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Amendment 4: Enhancements for Higher Throughput.
Institute of Electrical and Electronics Engineers, Inc., 2008.

[71] *IEEE Std 1394-1995*.
IEEE Standard for a High-Performance Serial Bus.
Institute of Electrical and Electronics Engineers, Inc., August 1996.

[72] *IEEE Std 1394a-2000 (Amendment to IEEE Std 1394-1995)*.
IEEE Standard for a High-Performance Serial Bus – Amendment 1.
Institute of Electrical and Electronics Engineers, Inc., 2000.

[73] *Abstract Syntax Notation One (ASN.1): Specification of basic notation*.
ITU-T Recommendation X.680.
International Telecommunication Union, July 2002.
URL: `http://purl.org/spica/asn1` (cited 2008-09-03).

J

[74] Jared Jackson.
Microsoft robotics studio: A technical introduction.
IEEE Robotics & Automation Magazine, 14(4):82–87, December 2007.
ISSN 1070-9932.
doi:10.1109/M-RA.2007.905745.

[75] Bob Jenkins.
Algorithm Alley: Hash Functions.
Dr. Dobb's Journal of Software Tools, 22(9):107–109, 115–116, September 1997.
ISSN 1044-789X.
URL: `http://purl.org/spica/jenkins1997` (cited 2008-09-03).

K

[76] Rudolph Emil Kalman.
A New Approach to Linear Filtering and Prediction Problems.
Transactions of the ASME – Journal of Basic Engineering, 82(Series D):35–45, 1960.
URL: `http://purl.org/spica/kalman1960` (cited 2008-09-03).

[77] Nate Koenig and Andrew Howard.
Design and use paradigms for Gazebo, an open-source multi-robot simulator.
IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004), 3: 2149–2154, September 2004.
doi:10.1109/IROS.2004.1389727.

[78] C. Ronald Kube and Eric Bonabeau.
Cooperative transport by ants and robots.
Robotics and Autonomous Systems, 30(1–2):85–101, January 2000.
doi:10.1016/S0921-8890(99)00066-4.

L

[79] Ulf Lamping.
The Wireshark Developer's Guide.
2008.
URL: `http://purl.org/spica/wireshark-dev` (cited 2008-09-03).

[80] Ulf Lamping, Richard Sharpe, and Ed Warnicke.
The Wireshark User's Guide.
2008.
URL: `http://purl.org/spica/wireshark-user` (cited 2008-09-03).

[81] Vincent Lenders, Martin May, and Bernhard Plattner.
Service Discovery in Mobile Ad Hoc Networks: A Field Theoretic Approach.
In *Proceedings of the Sixth IEEE International Symposium on World of Wireless Mobile and Multimedia Networks (WOWMOM 2005)*, pages 120–130, Washington, DC, USA, June 2005. IEEE Computer Society.
ISBN: 0-7695-2342-0-01.
doi:10.1109/WOWMOM.2005.93.

M

[82] David Q. Mayne and J. E. Handschin.
Monte Carlo techniques to estimate the conditional expectation in multi-stage non-linear filtering.
International Journal of Control, 5(5):547–559, 1969.

[83] David A. McGrew and John Viega.
The Galois/Counter Mode of Operation (GCM).
Technical report, National Institute of Standards and Technology, May 2005.
URL: `http://purl.org/spica/gcm2005` (cited 2008-09-03).

[84] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone.
Handbook of Applied Cryptography.
CRC Press, 2001.
URL: `http://www.cacr.math.uwaterloo.ca/hac/` (cited 2008-08-24).

[85] Joaquin Miller and Jishnu Mukerji.
MDA Guide Version 1.0.1.
Technical report, Object Management Group, Inc., June 2003.
URL: `http://www.omg.org/docs/omg/03-06-01.pdf` (cited 2008-08-24).

[86] Paul V. Mockapetris.
Domain Names – Concepts and Facilities.
RFC 1034 (Standard), November 1987.
Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592.
URL: `http://www.ietf.org/rfc/rfc1034.txt` (cited 2008-08-30).

[87] Paul V. Mockapetris.
Domain Names – Implementation and Specification.
RFC 1035 (Standard), November 1987.

Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343.
URL: `http://www.ietf.org/rfc/rfc1035.txt` (cited 2008-08-30).

[88] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun.
Perspectives on standardization in mobile robot programming: the Carnegie Mellon Navigation (CARMEN) Toolkit.
IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003), 3: 2436–2441, October 2003.
doi:10.1109/IROS.2003.1249235.

[89] *The Advanced Encryption Standard (AES)*.
Federal Information Processing Standard 197.
National Institute of Standards and Technology, Washington, 2001.
URL: `http://purl.org/spica/fips-197` (cited 2008-09-03).

[90] *Secure Hash Standard*.
Federal Information Processing Standard 180-2.
National Institute of Standards and Technology, Washington, 2002.
URL: `http://purl.org/spica/fips-180-2` (cited 2008-09-03).

[91] *The Key-Hashed Message Authentication Code (HMAC)*.
Federal Information Processing Standard 198-1.
National Institute of Standards and Technology, Washington, 2007.
URL: `http://purl.org/spica/fips-198-1` (cited 2008-09-03).

N

[92] Issa A.D. Nesnas, Reid Simmons, Daniel Gaines, Clayton Kunz, Antonio Diaz-Calderon, Tara Estlin, Richard Madison, John Guineau, Michael McHenry, I-Hsiang Shu, and David Apfelbaum.
CLARAty: Challenges and Steps Toward Reusable Robotic Software.
International Journal of Advanced Robotic Systems, Special Issue on Software Development and Integration in Robotics, 3(1):23–30, March 2006.
URL: `http://purl.org/spica/claraty2006-ars` (cited 2008-09-03).

[93] Landon Curt Noll.
Fowler/Noll/Vo (FNV) Hash.
Online, 2004.
URL: `http://purl.org/spica/fnv` (cited 2008-09-03).

[94] *UDDI Version 3.0.2*.
OASIS, October 2004.
URL: `http://uddi.org/pubs/uddi-v3.0.2-20041019.pdf` (cited 2008-08-23).

[95] *CORBA Component Model*.
Object Management Group, 2006.
URL: `http://www.omg.org/docs/formal/06-04-01.pdf` (cited 2008-08-24).

[96] *CORBA 3.0 – IDL Syntax and Semantics chapter*.
Object Management Group, 2007.
URL: `http://www.omg.org/docs/formal/02-06-07.pdf` (cited 2008-08-24).

[97] *Notification Service Specification*.
Object Management Group, 2004.
URL: `http://www.omg.org/docs/formal/04-10-11.pdf` (cited 2008-08-24).

[98] *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1*.
Object Management Group, 2008.
URL: `http://www.omg.org/spec/CORBA/3.1/` (cited 2008-08-24).

[99] *Model Object Facility (MOF) Core specification, Version 2.0*.
Object Management Group, January 2006.
URL: `http://www.omg.org/docs/formal/06-01-01.pdf` (cited 2008-08-24).

[100] *Unified Modeling Language 2.1.2 Super-Structure Specification*.
Object Management Group, November 2007.
URL: `http://www.omg.org/docs/formal/2007-11-04.pdf` (cited 2008-08-24).

[101] *Unified Modelling Language 2.1.2 Infrastructure Specification*.
Object Management Group, November 2007.
URL: `http://www.omg.org/docs/formal/2007-11-04.pdf` (cited 2008-08-24).

[102] *Meta Object Facility (MOF) 2.0 Query/View/Transformation*.
Object Management Group, April 2008.
URL: `http://www.omg.org/docs/formal/08-04-03.pdf` (cited 2008-08-24).

P

[103] Terence John Parr.
The Definitive ANTLR Reference: Building Domain-Specific Languages.
The Pragmatic Bookshelf, March 2007.
ISBN: 978-0-9787392-5-6.

[104] Terence John Parr.
Web Application Internationalization and Localization in Action.
In David Wolber, Neil Calder, Chris Brooks, and Athula Ginige, editors, *Proceedings of the 6th international conference on Web engineering (ICWE 2006)*, volume 263 of *ACM International Conference Proceeding Series*, pages 64–70, Palo Alto, California, 2006. ACM.
ISBN: 1-59593-352-2.
doi:10.1145/1145581.1145650.

[105] Terence John Parr.
Enforcing Strict Model-View Separation in Template Engines.
In *Proceedings of the 13th International Conference on World Wide Web (WWW 2004)*, pages 224–233, New York, NY, USA, 2004. ACM.
ISBN: 1-58113-844-X.
doi:10.1145/988672.988703.

[106] Terence John Parr.
A Functional Language for Generating Structured Text.
Technical report, University of San Francisco, 2006.
URL: `http://purl.org/spica/stringtemplate2006` (cited 2008-09-03).

[107] Judea Pearl.

Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.
ISBN: 978-1558604797.

[108] *The I2C-BUS Specification*.
Philips Semiconductors, January 2000.
URL: `http://purl.org/spica/i2c-spec` (cited 2008-09-03).

[109] Bernhard Plattner, Martin May, and Vincent Lenders.
Service discovery in ad hoc networks.
European Patent EP1677462, July 2006.
URL: `http://purl.org/spica/plattner2006` (cited 2008-09-03).

[110] John Postel.
User Datagram Protocol.
RFC 768 (Standard), August 1980.
URL: `http://www.ietf.org/rfc/rfc768.txt` (cited 2008-08-24).

[111] John Postel.
Internet Protocol.
RFC 791 (Standard), September 1981.
Updated by RFC 1349.
URL: `http://www.ietf.org/rfc/rfc791.txt` (cited 2008-08-24).

[112] John Postel.
Transmission Control Protocol.
RFC 793 (Standard), September 1981.
Updated by RFC 3168.
URL: `http://www.ietf.org/rfc/rfc793.txt` (cited 2008-08-24).

R

[113] Roland Reichle, Michael Wagner, Mohammad Ullah Khan, Kurt Geihs, Jorge Lorenzo, Massimo Valla, Cristina Fra, Nearchos Paspallis, and George A. Papadopoulos.
A Comprehensive Context Modeling Framework for Pervasive Computing Systems.
In Rüdiger Kapitza, Reinhold Kroeger, René Meier, Hans P. Reiser, and Bartosz Biskupski, editors, *8th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS 2008)*, volume 5053/2008 of *Lecture Notes in Computer Science*, pages 281–295, Oslo, Norway, June 2008. Springer Berlin/Heidelberg.
ISBN: 978-3-540-68639-2.
doi:10.1007/978-3-540-68642-2.

[114] Jeffrey Richter.
CLR Via C#, Second Edition.
Microsoft Press, Redmond, 2006.
ISBN: 0735621632.

[115] Ronald L. Rivest.
The MD5 Message-Digest Algorithm.
RFC 1321 (Informational), April 1992.
URL: `http://www.ietf.org/rfc/rfc1321.txt` (cited 2008-08-24).

[116] Ronald L. Rivest and Burton S. Kaliski Jr.
RSA Problem.
In Henk C. A. van Tilborg, editor, *Encyclopedia of Cryptography and Security*, pages 532–536. Springer US, 2005.
ISBN: 978-0-387-23473-1.
doi:10.1007/0-387-23483-7_363.

[117] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman.
A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.
Communications of the ACM, 21(2):120–126, 1978.
ISSN 0001-0782.
doi:10.1145/359340.359342.

[118] Phillip Rogaway.
Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC.
In Pil Joong Lee, editor, *Advances in Cryptology – ASIACRYPT 2004*, volume 3329/2004 of *Lecture Notes in Computer Science*, pages 16–31, Jeju Island, Korea, December 2004. Springer Berlin/Heidelberg.
ISBN: 978-3-540-23975-8.
doi:10.1007/b104116.

S

[119] Douglas C. Schmidt.
An Architectural Overview of the ACE Framework.
USENIX Login Magazine, Tools special issue, January 1999.
URL: `http://purl.org/spica/ace1999` (cited 2008-09-03).

[120] Douglas C. Schmidt, Aniruddha S. Gokhale, Timothy H. Harrison, and Guru Parulkar.
A High-Performance End System Architecture for Real-Time CORBA.
IEEE Communications Magazine, 35(2):72–77, 1997.

[121] Bran Selic.
The Pragmatics of Model-Driven Development.
IEEE Software, 20(5):19–25, September 2003.
ISSN 0740-7459.
doi:10.1109/MS.2003.1231146.

[122] Glenn Shafer.
A Mathematical Theory of Evidence.
Princeton University Press, 1976.

[123] Mark Smith and Tim Howes.
Lightweight Directory Access Protocol (LDAP): String Representation of Search Filters.
RFC 4515 (Proposed Standard), June 2006.
URL: `http://www.ietf.org/rfc/rfc4515.txt` (cited 2008-08-30).

[124] Harry Sneed.
The drawbacks of model-driven software evolution.
In *Workshop on Model-Driven Software Evolution at CSMR*, Amsterdam, March 2007.
URL: `http://purl.org/spica/sneed2007` (cited 2008-09-03).

[125] Daniel H. Steinberg and Stuart Cheshire.
Zero Configuration Networking: The Definitive Guide.
O'Reilly Media, Inc., 2005.
ISBN: 978-0-59-610100-8.
URL: `http://purl.org/spica/zeroconf2005` (cited 2008-09-03).

[126] Michael Steiner, Gene Tsudik, and Michael Waidner.
Diffie-Hellman Key Distribution Extended to Group Communication.
In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, pages 31–37, New Delhi, India, March 1996.
ISBN: 0-89791-829-0.
doi:10.1145/238168.238182.

T

[127] Sebastian Thrun, Dieter Fox, Wolfram Burgard, and Frank Dellaert.
Robust Monte Carlo localization for mobile robots.
Artificial Intelligence, 128(1-2):99–141, May 2001.
doi:10.1016/S0004-3702(01)00069-8.

U

[128] Hans Utz.
Advanced Software Concepts and Technologies for Autonomous Mobile Robotics.
PhD thesis, University of Ulm, Germany, 2005.
URL: `http://wm-urn.org/urn:nbn:de:bsz:289-vts-54004` (cited 2008-09-03).

[129] Hans Utz, Stefan Sablatnög, Stefan Enderle, and Gerhard K. Kraetzschmar.
Miro – Middleware for Mobile Robot Applications.
IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures, 18(4):493–497, August 2002.
doi:10.1109/TRA.2002.802930.

[130] Hans Utz, Freek Stulp, and Arndt Mühlenfeld.
Sharing Belief in Teams of Heterogeneous Robots.
In Daniele Nardi, Martin Riedmiller, Claude Sammut, and José Santos-Victor, editors, *RoboCup 2004: Robot Soccer World Cup VIII*, volume 3276/2005 of *Lecture Notes in Computer Science*, pages 508–515. Springer Berlin/Heidelberg, 2004.
ISBN: 3-540-25046-8.
doi:10.1007/b106671.

[131] Hans Utz, Gerhard K. Kraetzschmar, Gerd Mayer, and Günther Palm.
Hierarchical behavior organization.
pages 2598–2605, August 2005.
ISBN: 0-7803-8912-3.
doi:10.1109/IROS.2005.1545581.

[132] Hans Utz, Ulrich Kaufmann, Gerd Mayer, and Gerhard K. Kraetzschmar.
VIP – A Framework-Based Approach to Robot Vision.
Journal on Advanced Robotics, Special Issue on Software Development and Integration in Robotics, 3(1):67–72, March 2006.
ISSN 1729-8806.

V

[133] Richard T. Vaughan and Brian P. Gerkey.
Software Engineering for Experimental Robotics, volume 30 of *Springer Tracts in Advanced Robotics*, engineering Reusable Robot Software and the Player/Stage Project, pages 267–289.
Springer Berlin/Heidelberg, April 2007.
doi:10.1007/978-3-540-68951-5_16.

[134] Richard Volpe, Issa A.D. Nesnas, Tara Estlin, Darren Mutz, Richard Petras, and Hari Das.
The CLARAty Architecture for Robotic Autonomy.
In *Proceedings of the 2001 IEEE Aerospace Conference*, volume 1, pages 121–132, Big Sky, Montana, March 2001.
doi:10.1109/AERO.2001.931701.

[135] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, Krzysztof Czarnecki, and Bettina von Stockfleth.
Model-Driven Software Development: Technology, Engineering, Management.
Wiley, June 2006.
ISBN: 978-0-470-02570-3.
URL: `http://www.mdsd-book.org/` (cited 2008-08-24).

W

[136] Nanbor Wang, Chris Gill, Douglas C. Schmidt, and Venkita Subramonian.
On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE, volume 3291/2004 of *Lecture Notes in Computer Science*, chapter Configuring Real-Time Aspects in Component Middleware, pages 1520–1537.
Springer Berlin/Heidelberg, 2004.
ISBN: 978-3-540-23662-7.
doi:10.1007/b102176.

[137] Samuel Weiler and Johan Ihren.
Minimally Covering NSEC Records and DNSSEC On-line Signing.
RFC 4470 (Proposed Standard), April 2006.
URL: `http://www.ietf.org/rfc/rfc4470.txt` (cited 2008-08-23).

[138] Thomas Weise, Steffen Bleul, Diana Comes, and Kurt Geihs.
Different Approaches to Semantic Web Service Composition.
In *Proceedings of the Third International Conference on Internet and Web Applications and Services (ICIW 2008)*, pages 90–96, Washington, DC, USA, June 2008. IEEE Computer Society Press.
doi:10.1109/ICIW.2008.32.

[139] Doug Whiting, Russell Housley, and Niels Ferguson.
Counter with CBC-MAC (CCM).
RFC 3610 (Informational), September 2003.
URL: `http://www.ietf.org/rfc/rfc3610.txt` (cited 2008-08-24).

[140] Jens Wollenhaupt.
Ontologiegestützte automatische Modell-Transformationen.

Bachelor’s thesis, University of Kassel, 2008.
URL: `http://purl.org/spica/wollenhaupt2008` (cited 2008-09-03).

[141] *XSL Transformations (XSLT) Version 2.0*.
World Wide Web Consortium, 2007.
URL: `http://www.w3.org/TR/xslt20/` (cited 2008-08-30).

[142] *SOAP Version 1.2 Part 0: Primer (Second Edition)*.
World Wide Web Consortium (W3C), April 2007.
URL: `http://www.w3.org/TR/soap12-part0/` (cited 2008-08-24).

[143] *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*.
World Wide Web Consortium (W3C), June 2007.
URL: `http://www.w3.org/TR/wsdl20/` (cited 2008-08-24).

Z

[144] Lotfi A. Zadeh.
Fuzzy sets as a basis for a theory of possibility.
Fuzzy Sets and Systems, 100(Supplement 1):9–34, 1999.
ISSN 0165-0114.
doi:10.1016/S0165-0114(99)80004-9.

[145] John A. Zinky, David E. Bakken, and Richard E. Schantz.
Architectural support for quality of service for CORBA objects.
Theory and Practice of Object Systems, 3(1):55–73, 1997.
doi:10.1002/(SICI)1096-9942(1997)3:1<55::AID-TAPO6>3.0.CO;2-6.

Index