UNIVERSITÄT ULM
FAKULTÄT FÜR INGENIEURWISSENSCHAFTEN UND INFORMATIK
INSTITUT FÜR EINGEBETTETE SYSTEME / ECHTZEITSYSTEME

Self–description mechanisms for embedded components in cooperative systems

Dissertation
zur Erlangung des Doktorgrades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

der Fakultät für Ingenieurwissenschaften und Informatik
der Universität Ulm

Hubert–Marcus Piontek
aus Ulm

2007

Amtierender Dekan:	Prof. Dr. Helmuth Partsch
Gutachter:	Prof. Dr. Jörg Kaiser
	Prof. Dr. Franz J. Hauck
	Prof. Dr. Wolfgang Schröder-Preikschat
Tag der Promotion:	23. Juli 2007

Abstract

A new class of applications based on smart embedded devices is emerging thanks to advances in microelectronics. There is one common characteristic to most of these new applications: the devices are networked. They are expected to cooperate dynamically. As an example, consider yourself driving home in your car. Right as you pull up your driveway, your garage door opens, and the light in the house is turned on. Another example is a mobile robot moving through an instrumented smart environment, e.g. in a factory.

Dynamic cooperation requires awareness and knowledge of the services the environment provides. Device and service descriptions that are available on–line are suitable to provide the required information.

Current description languages, especially for embedded systems, provide only basic low–level information. They do not allow to describe a system in enough detail, and cannot provide the necessary semantic information.

Device and service descriptions can also have equally important uses during the complete device life cycle, i.e. they may be useful during the design and implementation phase of devices, as well as during their integration phase into larger systems. Most current description facilities fail to provide support in such a broad way.

The CODES (COsmic embedded DEvice Specifications) approach presented in this thesis provides a component description language that supports the complete life cycle of components, and lays the foundation for dynamic cooperation. Together with the underlying communication middleware COSMIC, which targets small autonomous components, it provides a description language for components and their services combined with a range of supporting tools and techniques. The description language is based on XML, and thus easily allows the integration with a number of well tested and widely used technologies. XML also eases the extension of the presented description language. CODES descriptions capture a component both in terms of the component as a whole with information about e.g. its manufacturer, and in terms of the services the component offers and relies on for its operation. The services' descriptions are based on the underlying COSMIC abstractions. While the COSMIC abstractions provide a high–level view of a component's interface, the descriptions are detailed enough to allow the completely dynamic setup of communication, including the encoding and decoding of the data exchanged. A seamlessly integrated parameterization facility allows the descriptions to be split up into a static part for

easy storage, and a variable part for customization. However, whenever the descriptions are actually used at run–time, the parameters are transparently merged into the descriptions, easing interaction for outside parties.

The supporting tools and techniques cover the component's life cycle. During the design phase, an editor for the description documents is provided. Part of the component's service implementation can be generated from the descriptions. Compatibility checking eases the integration of multiple components into larger systems. On–line discovery and query mechanisms allow the dynamic use of services. Discovery and query is important both during normal system usage, and during system maintenance.

Zusammenfassung

Dank Fortschritten in der Mikroelektronik bildet sich derzeit eine neue Klasse von Anwendungen heraus, basierend auf intelligenten eingebetteten Systemen. Die meisten dieser neuen Anwendungen haben eine Gemeinsamkeit: die Geräte sind vernetzt. Die Nutzer erwarten, dass sie dynamisch miteinander kooperieren können. Als ein Beispiel kann etwa die Fahrt nach Hause im eigenen Auto dienen. Sobald das Auto in die eigene Einfahrt einbiegt öffnet sich das Garagentor und das Licht im Haus geht an. Ein anderes Beispiel ist ein mobiler Roboter, der sich durch einen instrumentierten intelligenten Raum bewegt, etwa in einer Fabrik.

Dynamische Kooperation setzt voraus, dass die in der jeweiligen Umgebung verfügbaren Dienste wahrgenommen werden und genügend Wissen zur Nutzung der jeweiligen Dienste verfügbar ist bzw. gemacht wird. Online verfügbare Geräte– und Dienstebeschreibungen sind ein probates Mittel um diese Informationen zur Verfügung zur stellen.

Aktuell verbreitete Beschreibungssprachen, speziell aus dem Bereich der eingebetteten Systeme, bieten nur einfache low–level Informationen. Sie erlauben es nicht, ein System in allen nötigen Details zu beschreiben und sie können nicht die nötigen semantischen Informationen bereitstellen.

Geräte– und Dienstebeschreibungen haben außerdem wichtige Anwendungen während des gesamten Lebenszyklus eines Geräts, d. h. sie sind während der Entwurfs– und Implementierungsphasen von Geräten genauso nützlich, wie während der Integration in größere Systeme. Die meisten aktuellen Beschreibungsmechanismen bieten keine solch breite Unterstützung.

Der in dieser Arbeit vorgestellte CODES-Ansatz (COsmic embedded DEvice Specifications) bietet eine Komponentenbeschreibungssprache die den kompletten Lebenszyklus einer Komponente unterstützt und eine Grundlage für dynamische Kooperation legt. Zusammen mit der darunter liegenden Kommunikationsmiddleware COSMIC, die auf autonome kleine Systeme abzielt, bietet der Ansatz eine Beschreibungssprache für Komponenten und deren Dienste, kombiniert mit einer Auswahl an unterstützenden Werkzeugen und Techniken. Die Beschreibungssprache basiert auf XML und erlaubt daher die einfache Integration mit verschiedenen erprobten und weit verbreiteten Technologien. XML vereinfacht außerdem die Erweiterung der vorgestellten Beschreibungssprache. CODES Beschreibungen erfassen eine Komponente sowohl als Ganzes, z. B. mit Informationen über den Hersteller, als auch in Bezug auf die Dienste die die Komponente bereitstellt bzw. für ihren Betrieb benötigt. Die Beschreibungen der Dien-

ste basieren auf den zugrundeliegenden Abstraktionen von COSMIC. Die COSMIC Abstraktionen bieten eine high–level Sicht auf die Schnittstelle einer Komponente. Die Beschreibungen sind detailliert genug um einen komplett dynamischen Aufbau der Kommunikation zu erlauben, inklusive der Kodierung und Dekodierung der auszutauschenden Daten. Ein nahtlos integrierter Parametrisierungsmechanismus erlaubt es, die Beschreibungen aufzuteilen in einen statischen Teil, der sich leicht speichern lässt, und einen variablen Teil, der der Anpassung der Komponente dient. Immer wenn die Beschreibungen zur Laufzeit tatsächlich genutzt werden sollen, werden die Parameter transparent mit den Beschreibungen verschmolzen. Das erleichtert anderen Teilen die Verwendung der Beschreibungen.

Die unterstützenden Werkzeuge und Techniken decken den Lebenszyklus einer Komponente ab. Während der Entwurfsphase steht ein Editor für die Beschreibungen bereit. Ein Teil der Serviceimplementierung einer Komponente kann während der Implementierungsphase aus den Beschreibungen erzeugt werden. Kompatibilitätsprüfungen vereinfachen die Integration mehrerer Komponenten zu größeren Systemen. Online Discovery und Abfragemechanismen erlauben die dynamische Verwendung von Diensten. Discovery und Abfrage sind sowohl während des normalen Systembetriebs, als auch während Wartungsphasen wichtig.

Contents

Chapter 1

Introduction

Decreasing size and cost of microelectronics make a vision of a networked world a reality. In this world, arbitrary devices may interact with each other and their environment. As an example, consider yourself driving home in your car. Right as you pull up your driveway, your garage door opens, and the light in the house is turned on. Another example is a mobile robot moving through an instrumented smart environment, e.g. in a factory. The building blocks for such applications are smart devices. These devices are small, cheap, autonomous, and easy to replace. Units of use, e.g. mobile robots, are each composed of a number of such small devices. These small smart devices usually have limited capabilities. Most of the time, they are complemented by a few more powerful components for higher level functionality.

For entities, such as mobile robots, a centralized approach is often taken. Peripheral components are connected to a single central entity. The peripherals are essentially polled. This often leads to a big block of monolithic code on the central processor that is hard to maintain.

By moving towards networked smart components, such systems can be unitized into handy pieces. The pieces are easier to maintain, however the complexity of *wiring* them together in a functioning way increases. Further, distributed setups allow for much greater extensibility, adaptability and configurability. Such manifold flexibility also means, that reconfigurations are probably necessary from time to time.

COSMIC (COoperating SMart devICes) is a communication middleware. It targets small smart devices, such as mentioned in the previous paragraph. Despite their limited resources, these smart devices act autonomously. COSMIC's main goal is to support predictable interaction of devices. Unlike other communication middleware frameworks, COSMIC allows to specify predictability parameters at the application level using suitable abstractions. COSMIC's communication model is based on a publish/subscribe scheme. Such an anonymous communication pattern is very beneficial in constantly changing systems, as there is no need to change or register addresses in applications, i.e. changes of the system are transparent to the application as long as the required input is satisfied. The development of COSMIC was partly embedded in the CORTEX (CO–operating Real–time senTient objects: architecture and EXperimental evaluation) project (Project IST–2000–26031). The CORTEX project was concerned with autonomous applications, running without direct human control. As key characteristics, sentience, autonomy, large scale, time and safety criticality, geographical dispersion, mobility and evolution where devised [26]. Within the project, the design of an interaction model of so–called sentient objects (see section 2.1.2) was a large part. The development of COSMIC and the design of the CORTEX interaction model were closely related.

A clear and concise description of the individual components is crucial, especially when building large–scale systems. Once dynamic autonomous cooperation of systems is intended, the descriptions must be machine readable and processable. Unfortunately, such description mechanisms were not directly elicited during the CORTEX project. However, due to the large scope of CORTEX, the problem of achieving broad and dynamic interoperability became apparent. The experience gained during the CORTEX project, and during the development of COSMIC and applications based on COSMIC in particular, significantly inspired and contributed to this work. This result is a flexible, easily processable description mechanism. Unlike other mechanisms (see chapter 5), the descriptions not only contain technical details about e.g. data encoding, but also integrate higher–level information, such as the corresponding physical unit, and quality attributes in a machine–readable manner. Despite the focus on small systems, XML has been chosen deliberately as the basis for the description mechanism. A number of related techniques have been selected to achieve a seamlessly integrated tool–chain. This fosters the reuse of well–tested and widely employed techniques. Thus, there is no need to reinvent and reimplement the respective processing techniques. Further, this sparks familiarity, and therefore lowers the learning threshold necessary to successfully employ the provided mechanisms.

The description mechanisms should not only be useful for cooperation activities, but also during the rest of each component's life cycle. This way, the descriptions do not become an extra burden for the component developer, but can be used profitably as the central entity of the component's life cycle.

This thesis provides an approach towards specifying smart components in order to

- ease development of the smart components,
- document the components,
- configure components,
- ease the composition of smart components,
- check for compatibility,
- provide a basis for dynamic interaction of components.

The approach is called CODES (COsmic embedded DEvice Specifications). As the name suggests, it is based on COSMIC; the device specifications therefore are closely related to COSMIC notions. Current approaches, as presented in chapter 5 lack this broad support for the component life cycle.

CODES and COSMIC are not only suitable for the creation of mobile robots, as used for demonstration purposes during the CORTEX project. The approaches are also adequate in industrial automation, sensor networks, or automotive applications; or short wherever embedded systems interact with their environment.

A publish/subscribe based anonymous communication scheme ideally fits this world. It allows to decouple individual devices from each other, instead of introducing artificial dependencies between devices that may easily pose enormous problems.

The main target platforms are small systems, mostly 8 and 16 bit architectures with very limited RAM. Yet, the techniques presented are also usable with much more powerful machines. Most of the benefits apply to all architectures. Software development for such small devices is a tedious process. Usually, software for small micro controllers is written more or less from scratch for each device. Using C, and possibly a small executive is considered state of the art. High level tools, such as UML or MDA tools are rarely used, partly because the generated code is too bloated, and partly because the abstractions these tools offer are not suitable for devices like smart sensors. Tools like Matlab, however, are used more often. Yet, these tools focus on processing data. The burdensome task of successfully integrating the output of Matlab into a distributed system, that is supposed to use other components as well, is completely left to the developer. The CODES approach is different: the developer of a smart component is to be relieved of as much housekeeping work as possible. The device specifications are not only the basis for the component's implementation, but also for its documentation, and for proxy code that helps components to interact with each other.

The approach was born through work within the CORE group. When we were developing prototypes using the COSMIC middleware, we ran into a number of pretty much well–known problems:

- How to coordinate different developers?
- How to clearly specify a component's interface?
- How to implement more efficiently on small systems?
- How to manage configurations? Which component is there, and what version of its software is it running?
- Missing documentation in general, and sometimes even missing specification of components
- How to accomplish dynamic interaction?

- How to achieve generic monitoring and logging?
- How to emulate components before they are finished?

The CODES approach tries to tackle these challenges, specifically in the context of small distributed embedded systems.

1.1 Benefits of the device descriptions

The embedded device descriptions[1] are meant to be useful throughout the complete life cycle of a component (see section 2.1.1):

- During the *component design phase*, the device specifications allow the designer to formally specify a component's interface at a suitable level–of–detail early on. The tool–chain supports this with an editor for the device specifications. The editor ensures that the produced specifications adheres to the CODES grammar, and that a number of sanity checks beyond the grammar pass.
- During the *component development phase*, the device specification is used for code generation. While tools like Matlab produce data–processing code, the code generated from the device specifications is mostly house–keeping code relieving the developer from tedious recurring tasks like the declaration of data structures, the declaration of tasks for event publication, or proxies for use and generation of events. The aim is to automate as much house–keeping as possible, so the developer can keep his mind on the actual service to be provided, e.g. the handling of a sensor and the conversion of raw sensor data to meaningful values.
- During the *system–construction* or *system integration phase*, the device specifications are used to integrate multiple components into a larger system and to ensure the compatibility of the employed devices by cross–checking the specifications by a set of rules. Some parameters in the specifications are user–configurable in a configuration phase, e.g. the period with which an event is generated. This allows preparing a component for use in a specific system.
- During the *use phase*, the device specifications are available through a query service for dynamic use. They inform a user of the devices capabilities. Intelligent users (human operators, or possibly intelligent components) may dynamically analyze the specifications, and subsequently make use of the provided services. To achieve the possibility of dynamic usage, the specification needs to be sufficiently detailed

[1]The terms *specification* and *description* are used synonymously when referring to CODES descriptions/specifications, as these documents contain both a general descriptive part, and a detailed specification part.

and it needs to contain enough semantic information to be useful. While IDLs are capable of describing the technical details like parameter types and their encoding, they lack the possibility to contain semantic information, i.e. they do not transport the meaning behind an operation declared using some IDL. The CODES structure allows to extend easily the specifications in terms of semantics beyond the currently implemented level, e.g. by integrating semantic web technology. The specifications are also useful for an admission test where the system decides, whether a newly attached component is compatible with the existing system, and thereafter decides whether the new component is allowed to participate or not. This can effectively be done at system start–up, or generally as part of the configuration process that is taking place when a component is attached to the system.

- During *maintenance*, the specifications allow service personnel to determine what components are available, what services they provide, and what version they are.
- For *disposal* or preferably *recycling of hardware*, the manufacturer of the component is of interest for plain monetary reasons. Also, the specification contains information that allows efficient waste management by independent parties.

The specifications are meant to be the only interfaces between component designers, component developers, system designers, component users, and maintenance professionals.

Many parameters cannot be determined at design time. They may for example be fixed during the integration phase. Some may change during the life–time of the component. Some description mechanisms support only static descriptions and move such parameters out of the description itself, and into the operational interface of the component, e.g. UPnP (see section 5.2). Others make a distinction between a static description without parameters, and a fully configured description, e.g. LIN (see section 5.6). This work presents a different approach. The descriptions may contain parameterized elements. These parameters are stored separately from the static part of the description. Whenever the description needs to be processed, the parameters are included into a temporary copy of the description. This yields a description containing all parameters.

1.2 Contributions of this thesis

This work presents a description scheme for small embedded components. The description scheme distinguishes itself from comparable description schemes in a number of points. It provides maximum *flexibility* while preserving machine *processability*. This becomes apparent e.g. the way attributes for events are specified, and the way physical units are represented. Being based on XML, a wide range of supporting processing technologies are available, most notably XSLT. XML also ensures that the description language is *extensible*. The goal of enhancing descriptions of embedded devices with semantic information to achieve dynamic interaction, as briefly outlined in section 7.7, is unique. The simple, yet very powerful parameterization mechanisms described in section 7.2.2, combined with the concept of accessing complete descriptions including all parameters whenever handling components at run–time is also unique. Finally, supporting the complete component life cycle (section 2.1.1) with the description language and the associated tool–chain is a unique contribution.

1.3 Overview of this thesis

Chapter 2 introduces the terms, concepts and base technologies important for this thesis. It also formulates requirements for descriptions, which are important for the design of the description mechanisms described in this thesis. Chapter 3 discusses the software development process for small components, covering the phases design through implementation of the component's life–cycle. Beneficial support for these phases is one of the core aims of the presented approach. Chapter 4 is concerned with requirements for integrating components into larger systems. These requirements are meant to be met by the CODES approach. Chapter 5 presents other middleware approaches and their description techniques, and compares them to this work. Chapter 6 briefly introduces the main concepts of COSMIC, and it introduces the COSMIC middleware. COSMIC forms the platform on which the CODES approach is based. The CODES approach, comprised of a description language and a supporting tool–chain is presented in chapter 7. The thesis closes with a conclusion in chapter 8.

Chapter 2

Terms and Concepts

This thesis is about describing components, and the benefits of these descriptions. The first section will define the notion of a component in detail. Components were already mentioned, and one can probably get an intuitive grasp of the respective notion. Nevertheless, it is vital to define this notion in a more formal manner. An introduction to real–time systems forms the second section. Embedded real–time systems are the primary target of the proposed description mechanisms. Real–time environments not only pose special requirements on the software, but also on the description mechanisms to be beneficial. The third section then details about descriptions in general. Several requirements that are later used to classify different description mechanisms are presented. These requirements are meant to be fulfilled by the proposed description mechanism. Closing this chapter, the relevant key representatives from the XML family of technologies are presented. XML forms an increasing popular basis for description mechanisms. Besides its flexibility, its widespread use and the readily available supporting standards, and software make XML the prime choice to base component descriptions on.

2.1 Components

Components play a key role in this work. Each CODES description specifies a component, more specifically, it describes a *sentient component's*[1] interface to the world. Thus, not only a definition of what a component is is given. There is also a rather broad discussion about what a component is and how it relates to types, classes and objects known from object oriented programming.

The notion of *software components* dates back to at least 1968, when they were mentioned at a NATO sponsored conference on software engineering [93]. Even though the notion back then was rather vague, the criteria

[1]During the CORTEX project, sentient components where called sentient objects. It should become clear after this discussion that sentient component is a more precise term for the (same) concept. This is why the term sentient component is used instead of the term sentient object.

applied to components nowadays (e.g. exchangeability, unit of deployment) were already discernible.

Numerous definitions of what a (software) component is can be found in the literature (e.g. [128], [104], [35], [94], [109], [149]).

The most straight forward definition can be found in [128]. SZYPERSKI defines a software component as follows:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

In the following discussion of further details about components, and their relations to classes and objects, this definition will be clarified.

Unit of composition According to [109], component–based software development represents a shift from traditional statement–oriented coding towards building systems by plugging together components. To be composable, a component needs to be sufficiently self–contained. It must come with a clear specification of its requirements and its services. If there was only a single component world, it would be sufficient to enumerate all required interfaces of other components to specify all context dependencies. As there are competing component worlds (e.g. Microsoft's COM, Sun's JavaBeans), the supported component world must be specified, too [128]. Unfortunately, in most component worlds, not even the list of required interfaces is normally available, deferring the discovery of missing components to run–time. The emphasis in component descriptions is on provided interfaces. To achieve ease of composability, a component's external dependencies, and its own services, must be limited to a concise set. According to MEIJLER and NIERSTRASZ [94], composition of components is done by binding their parameters to specific values or other components. Compositions are seen as configurations that may name components to be used, but due to the black–box approach, actual implementation choices may be delayed to link or run–time. Black–box reuse of components refers to reusing an implementation without relying on anything but the component's interface and specification, much like a typical application programming interface [128].

Components and interfaces Components are not necessarily based on specific programming language concepts. It must be possible to implement components in any programming language, and the resulting entities must be able to interact. In other words, one of the goals of component–orientation is programming language independence. To achieve this, standards for describing components have to be established, e.g. in textual form as *interface definition languages* (IDLs) [109]. On the downside, the commitment to a specific programming language is traded for the commitment to a specific IDL. According to

[129], IDLs are only around to support the definition of interfaces sitting between interface implementations and interface clients — neither of which logically owns the interface. Interfaces written in an IDL define the interface's operations in terms of call signatures. They must be transformed to a given target programming language by a compiler. As the IDL definition does not capture semantics and documentation in general, any documentation referring to the interface's signature must be transformed to the target language, as well.

Components and programming languages Modules, well–known from languages like Ada and Modula–2 can be seen as minimal components [109] [128]. The main characteristics that such modules share with components are separate compilation and the possibility of type–checking across module boundaries. Abstractions like name spaces and packages also come rather close to the notion of components [128]. For components, the kind of programming language does not matter, i.e. they can be implemented using object–orientation, functional or procedural programming, or any combination of these.

In object–orientation, types, classes and objects are differentiated. Before discussing the relation between components and these terms, a short introduction to the terms type, class, and object is given. Each concept is exemplified in C#.

A type is an abstract notion defining attributes and operations on these attributes. Most types have a correspondence in the "real world", however not all do. As an example, consider a distance sensor. It can be modeled as a type *DistanceSensor*. As an attribute, there is the sensor's distance reading. As an operation, there is a method that will trigger a measurement. These abstract types are also called *specification types* [10]. Most OO languages have no means to represent specification types. At the other end of the spectrum of *types*, so called *implementation types* can be found. They are more commonly found in object–oriented languages like C++. Some modern object–oriented languages offer interfaces, which come closest to representing specification types in the programming language. The *DistanceSensor* type can be represented in C# as follows:

```
public interface DistanceSensor {
  public double distance {get;}

  public void trigger_measurement();
}
```

Any semantic information behind the attributes or operations cannot be represented in the programming language. It is usually documented in textual form, either as comments in the code, or in separate documents.

Types and classes are fundamentally orthogonal concepts. A class is an implementation of one or more types. Types need not be fully implemented by a certain class, i.e. a class may only implement a type partially. Classes are abstractions of the programming language. Languages like C++, however, do not distinguish between types and classes, and offer them as a single concept. To continue with the example of a distance sensor, consider the following C#–class that implements a distance sensor:

```
public class CDistanceSensor : DistanceSensor {
  private double distance _distance = 0.0;
  private int _adc_channel = 0;

  public CDistanceSensor(int adc_channel) {
  _adc_channel = adc_channel;
  }

  public double distance {
    get { return _distance; }
  }

  public void trigger_measurement() {
  InitADC();
  _distance = ReadADC(adc_channel);
  }
}
```

An object is an instance of a class. Objects only exist at run–time. They have identity, i.e. different instances can exist at run–time, and they can be identified [94]. Again, in C#, the following code instantiates a distance sensor and uses it:

```
static class MyProgram {
  public static void Main() {
    CDistanceSensor front_ir = new CDistanceSensor(7);

    while( true ) {
      front_ir.trigger_measurement();
      if( front_ir.distance < 100 ) {
        // perform emergency stop
      }
    }
  }
}
```

In object–oriented languages, classes are the unit of reuse. Composition of classes is done at the language level — usually limiting the use of classes to a single language [109].

While both components and classes encapsulate concepts, classes tend to be finer grained than components. Classes tend to have a vast amount of external dependencies, quite in contrast to the goal of component–orientation of minimizing external dependencies. Components are meant to abstract meaningful subsystems [109] — often represented by a number of tightly coupled classes [130]. Of course, some components may well be implemented using a single class. If classes are used to implement components, individual classes are confined within the component, i.e. an individual class cannot span multiple components. Just as classes can depend on other classes (via inheritance), components can depend on other components via their input interfaces [128].

Similar to classes, components are static entities needed at system build–time. At run–time, their structure is not necessarily visible, or even existent [94].

component weight Components can have widely varying complexity ranging from individual functions to complete applications. A component's size is not only depending on the complexity of the offered services, but also on the component's external dependencies. If all dependencies are included with the component, it is easily usable, yet fat and possibly inflexible. Also, such monolithic blocks thwart the idea of handy, pluggable components. If, on the other hand, all dependencies are external, reuse of the component is severely impacted by the explosion of context dependencies. Class hierarchies tend to be at this end of the spectrum [128].

If components have minimal dependencies on their environment, they are likely to be widely usable [129].

component deployment Components are a unit of independent deployment. They must be well separated from their environment and other components. A component is never deployed partially [128].

Components are shipped in a binary, executable format. They contain all information about their static dependencies beyond the component platform. Components may contain more meta–information, such as deployment attributes [130].

Currently established component technology (such as IBM's SOM, Microsoft's COM or Sun's JavaBeans) fails to offer meta information about the components within their description [109]. These technologies represent only low–level connection standards [128].

Components cannot have state It must not make a difference, which of a number of replicas of a component is actually used at run–time. Therefore, a component cannot have state that influences its functionality [130].

Reasons to use components

Reuse: Components are independent building blocks for systems. They are supposed to be reusable in different systems [109] [128] [94] [129] [130].

Extensibility: New features can be added to a system in such a way that the ”old” system can interact with the it [130].

Evolvability: Old components can be replaced by new ones to improve the system in terms of quality, functionality, or both [130].

Lean software: Instead of having huge monolithic applications, components need only be loaded on demand. This helps to slim down applications in terms of program size and memory use [109].

Closing the discussion about components, some prominent characteristics for this work are given:

- A component is like a traditional piece of hardware. Thus, it is universally usable, replaceable, interchangeable and reliable in terms of its services. A component may include software and hardware. In fact, the practical part of this work is greatly concerned with components including both software and hardware. Thus, the terms *component* and *node* are used synonymously.

- Components are implemented using the sentient component notion developed in the CORTEX project [29] [31] [27] [28].

- Components interact using events, as described in the CORTEX project [30] [33] [27] [28].

Conclusively, it can be noted that the notion of components fits a world of many networked, autonomous systems very well.

A component is only usable, if its description (low–level interface and semantics) is known. The component’s interface is the lowest level of description. It encompasses technical issues like calling conventions, parameter encoding, etc. Above these low–level details, the interface is described in terms of operations or services that the component offers. To be programming language independent, interface definition languages were introduced. They allow the definition of method signatures and there are compilers that transform the interface definitions to the programming language to be used. An interface definition in an IDL does not contain any information beyond the signatures of the supported operations. The semantics of the parameters, e.g. the physical dimension of a measurement value, and the semantics behind the complete component are missing from the traditional interface definitions. This information is usually represented in a textual notation separate from the interface definition. The work presented in this thesis provides a means of enhancing the component descriptions in a machine–readable manner.

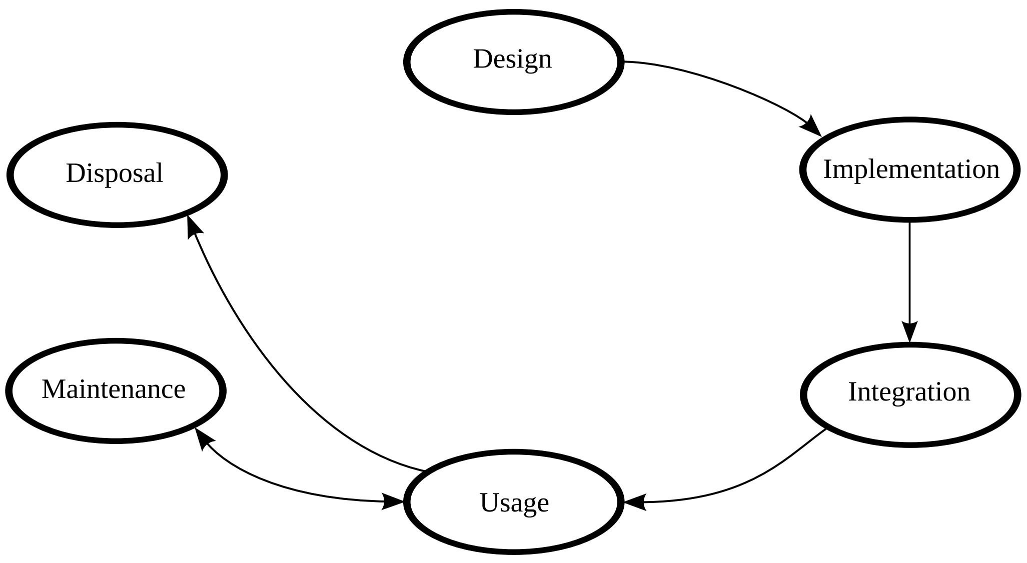


Figure 2.1: Component Life Cycle with typical transitions

A component can only be dynamically used, if its description is available dynamically. This is why this thesis proposes to store the description within the component itself. They are available trough a standardized interface. Beyond the definition of the event signatures, the proposed device specifications also contain semantic information for individual data fields, events, and the whole component. This semantic information is currently stored in a semi–structured way. Storing the specifications as XML allows to enhance these structures in the future without sacrificing compatibility to older software. This essentially takes extensibility and evolvability from components to their descriptions. At the far end, the specifications can be enhanced to include e.g. semantic web technology.

2.1.1 Component Life Cycle

A component's life cycle (see figure 2.1) starts with the component's design phase. The design phase includes the specification of the component. After the design is done, the development or implementation phase is next. Both design and implementation can occur in an iterative way, and both phases usually occur at the manufacturer of a component, be it hardware, software, or a combination of both. Of course, the implementation phase also includes a thorough testing stage. The outcome of the development phase crosses from the realm of the component manufacturer to the system integrator. In the construction or integration phase, individual components are integrated into larger systems by plugging them together in hardware, and by configuring their parameters in software to achieve the desired interaction among them. As with the implementation phase, the integration phase en-

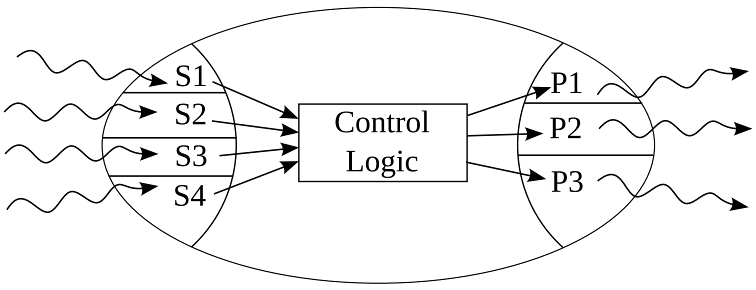


Figure 2.2: A sentient object

compasses system testing. The integration phases' output crosses from the integrator's realm to the user's realm. The user typically has few, if any, parameters to configure. Afterwards, the system is employed as intended. In case of failure, upgrade, or for other reasons, the system, or individual components of the system can go into maintenance. If the whole system needs maintenance, usually the integrator performs the maintenance tasks; if an individual component needs maintenance, the component's manufacturer is probably the right partner. Maintenance usually requires the identification of a component as a first step. Further, the components configuration is of interest; it might need to be changed. Once the system is decommissioned, or parts of it fail, it is taken apart. The individual components are then disposed of, preferably by their manufacturer. Of course, figure 2.1 cannot capture all possibilities, e.g. defective components going straight from maintenance to disposal, however these details would only clutter the presentation here.

Interface definition languages are useful throughout most of a component's life cycle. What is missing from current approaches is the support for disposal. As traditional IDLs are only concerned with software components, disposal might at first seam to be out of place. However, correctly uninstalling a component can be a complicated and tedious task on its own, depending on the component technology. While in .NET, it mostly suffices to delete the component's files, COM+[2] component uninstallation usually requires extensive cleanup of the system registry.

2.1.2 Sentient Components

In the CORTEX project (CO-operating Real-time senTient objects: architecture and EXperimental evaluation, Project IST-2000-26031), the notion of *sentient objects* has been a central entity. In [50], sentient objects are described as being *mobile, intelligent software components*. Defining sentient objects to be software only, however, is restrictive.

Sentient objects are able to sense their environment using sensors. These sensors may either be hardware sensors able to sense the physical environ-

[2]COM+ is the predecessor of the .NET technology.

ment, or software sensors, able to detect events in software. More details can be found below. Sensors are the only means of *input* to sentient objects. After *consuming* an input event via its sensors (left side of figure 2.2), a sentient object uses its internal control logic to decide, whether it will in some way react to the event, i.e. change its internal state, or not. When a sentient object changes its internal state, it may employ its actuators to *produce* outgoing events via its actuators (right side of figure 2.2), so other sentient objects can be aware of said change. This production may either manifest itself via a software event being transmitted via a communication network, or via a physical actuation of some kind, e.g. switching on a light or turning on a motor.

Sentient objects are the building blocks for applications according to the CORTEX paradigm. They can be composed recursively. Sentient objects are components in the sense of section 2.1: they interact with their environment solely through event consumption and production. Their internals are of no concern to the user. Thus, the term *sentient object* is misleading. This is the reason why they are called *sentient components* in this work.

Sensors and Actuators

Sensors and actuators are a sentient component's means of interaction with its environment, be it physical or virtual. A sensor for the real–world is an entity that produces (object internal) software events in reaction to occurrences in the physical environment. It consists of both hardware and software. The hardware converts real–world occurrences into software accessible measurements. The software uses these measurements to generate software events. A software sensor is a piece of software that monitors the system environment (e.g. for incoming events or variables like free memory).

Accordingly, an actuator is an entity that consumes object internal software events and causes a change in the environment. Again, this can be both the (virtual) system environment, as well as the real–world environment. Change to the system environment may e.g. be the transmission of an event message or the deletion of old temporary files to free up disk space. To achieve changes to the physical environment, hardware is controlled, e.g. a motor.

Each sentient component has sensors residing on its consuming side, and actuators sitting on its producing side. Sensors and actuators are small entities that are rather similar to sentient components, themselves: they consume input and produce output according to a usually linear and small piece of control logic.

Context Awareness

As sentient components interact with their physical environment, they need to be aware of the situation they are in, e.g. to properly support safety requirements.

In [50], context is defined as:

> Any information sensed from the environment that may be used to describe the situation of a sentient object. This includes information about the underlying infrastructure available to the sentient object.

Context–awareness is in turn defined as follows:

> The use of context to provide information, to a sentient object, which may be used in its interactions with other sentient objects, and/or the fulfillment of its goals.

In the CORTEX project, inference engines were proposed for implementing the components' internal control logic. Depending on the complexity of the context, this decision makes more or less sense. For very simple objects, e.g. an infrared distance sensor, this is certainly not necessary. Such sensors only require a linear flow of control.

Interaction

Sentient components are expected to use anonymous, event–based communication. This form of communication supports loose coupling between sentient components, and thus allows for a more dynamic interaction among components.

Sentient components communicate and cooperate with other sentient components both through software events and via the physical environment. A sentient component is not limited to consume only software or only real–world events. Neither is it limited to the production of only one of the two. Sentient components may arbitrarily consume and produce any combination of events.

Implementations of sentient components may use a range of different networking technologies. Middleware, such as COSMIC (see chapter 6), supports the creation of sentient components by transparently masking the actual networking technology in use while still providing the possibility to influence quality aspects of the communication.

2.2 Real–time systems

Real–time systems must exhibit special temporal behavior. The German DIN defines real–time processing as [38]:

> Real–time processing is a processing mode of a computer system, in which programs are ready constantly for the processing of new incoming data. The processes provide their results within a pre–defined time span.

> Depending on the application, the data may become ready according to a random distribution over time, or at pre–defined points in time.[3]

Processing must not necessarily be as fast as possible, but it must be completed within defined bounds in time. Safety critical applications strongly rely on this property. It must always be met. Outdated results or actions may even be harmful to the system. This class of applications is called *hard real–time*. For some applications, slightly outdated values may still be of use. These are classified as *soft real–time* [81]. Some applications may cope with a few missed deadlines by adjusting their behavior [32]. Also, in real–time systems, there may be processes that require only best effort services.

Writing real–time capable software is a non–trivial task. Compared to other software, the aspect of meeting a deadline is an additional constraint. The typical platforms running real–time software add additional burdens to the developer, mainly due to their limited processing power and memory. Embedded real–time systems are mostly produced in large quantities, and therefore cost is crucial. Many well–known technologies, e.g. libraries or communication protocols, are not suitable in embedded environments, as their temporal behavior is not predictable.

Most real–time systems must work autonomously without any or with only minimal interaction with a user or operator. Instead, they interact directly with their physical environment through sensors and actuators. The level of reliability necessary is mostly above that of other software, especially when the software is used in safety–critical control applications.

The special requirements of real–time environments also adds requirements to the description mechanism to be used. This mechanism must e.g. include support for temporal properties. The information is crucial especially during the integration phase. If it is not available, integration cannot rely on the descriptions only. Temporal information can be used to e.g. schedule communication on a network. The information is also necessary to check composability of (software) components into an application with its own temporal requirements.

2.3 Descriptions

Descriptions are common on different levels, ranging from informal textual descriptions with or without any given structure to mathematically verifiable specification languages. Depending on the entity to be described and the target audience of the description, different approaches are useful.

2.3.1 Formality

The most common form of description certainly is informal, plain text in a natural language. Its main advantages are that its use comes natural to

[3]The original definition is in German, translated by the author.

people. A natural language's expressiveness is enormous, yet most words do not have a fixed interpretation. Obviously, plain text descriptions are easily readable, but they may be hard to understand, even for humans, as misinterpretations cause confusion. Analyzing and processing plain text with a computer is especially hard. Even though programs like ELIZA or IVR systems tend to impress people, the technology is far from being able to understand the meaning of a piece of text, let alone automatically process it further in a way, suitable to e.g. generate application code.

Descriptions should be written in a formal language. This eases automated handling and allows automatic checking for errors. Having a concise semantics in formal languages prevents misunderstandings.

Most programming languages are based on context–free grammars, yet they are not really context–free. Variables e.g. must be of the proper type in many languages. Therefore it is easily possible to write syntactically correct programs that will not compile all the same. This approach still has been proven to work quite well, tremendously easing the construction of parsers. Checking a few rules outside the grammar is feasible and frequently done [121], [57].

2.3.2 Processability and readability

Whenever descriptions are meant to be used automatically, the processability of their format is of importance. Processability is strongly related to formality. Descriptions formulated in regular and context–free grammars are easier to parse and process than others. Descriptions in natural languages are especially complex to process, not only because of their grammar. The semantics of (key) words in formal languages is usually precisely defined, whereas in texts written in a natural language, the semantics behind a specific word may change in subtle ways throughout the text. Descriptions may also be given in non–textual formats, i.e. in binary formats that are extremely easy to process, as no parsing needs to be done. If the structure is fixed, access to a specific parameter can be done in constant time. Such descriptions are not human readable, however. They also tend to be inflexible and hard to extend.

Readability can be considered the inverse of processability. While binary descriptions are easily processable, they are unreadable for humans, unless special tools are provided. At the other end of the spectrum, descriptions in a natural language are easily readable, but hard to process. In between, descriptions based on technologies like XML provide an interesting trade–off: they are readable by humans — possibly requiring some training — and they can be processed by machines pretty well using well–known libraries and utilities — provided enough memory and processing power is available. Description formats that are human readable because they contain all necessary information in the description itself, e.g. the name of parameter together with its value, as is the case with XML formats, are often called *self–describing* [148], [36], [90]. As parameter names often are far from being truly self–describing, such description formats are called *self–contained* for this work. The term *self–description* is associated with a different meaning (see the following section).

2.3.3 Level of Detail

Descriptions should be written with a suitable level of detail, depending on the application of the description. While for an end–user, a rather coarse–grained description of a black–box unit, defining its service and functionality to the world, is enough, it will not suffice for the developer of said entity. Description languages should therefore provide the possibility to describe a system in varying, refinable levels of detail. This refinement can continue to a level where the description is equivalent to a (complete) program written in a traditional programming language. The key is to offer a description language that allows to describe a system at a level that is abstract enough, so there is a gain compared to programming languages that makes using the description language seem worthwhile. On the other hand, it should allow such a fine–grained description that it can be turned into useful pieces of code.

2.3.4 Completeness

To be attributed *complete*, a description must contain all aspects of the component described that are relevant for its application. This especially means that no ambiguities may be left. Determining the completeness of a specification is a non–trivial matter in general. For some areas, approaches to automatically verify completeness of some sort have been proposed, e.g. [55], [68], [56].

2.3.5 Extensibility

Besides advances in the component architecture, the transition to new application areas also demands for extensibility. Any description format that is intended to be usable over time should be extensible, so future features can be included. But identifying all conceivable attributes when introducing a description format is impossible. To properly support extensibility, a description format must allow the seamless integration of new attributes, and it should provide a versioning scheme, associating a description with the proper version of the description format. Binary description formats often obstruct extensibility. XML based formats have been proved to support extensibility in a seamless way, provided the tool chain does not completely choke on unknown elements. Here, the need for a versioning scheme becomes apparent: if a particular description lacks a certain element, is this because of a flaw in the description or because the description adheres to some older standard?

2.3.6 Standardization

The support framework for any description format should be kept as simple as possible to reduce complexity both for implementors of the framework, and for its users. Description formats should be based on well–known standards and mechanisms. This makes them easier to implement, as there are tested and trusted libraries available. The potential spread can be maximized, as users tend to be familiar with at least part of the concepts, easing migration. Proprietary technology tends to consume vast amounts of effort for successful market introduction. Further, using open standards ensures availability over time, even as manufacturers come and go.

2.3.7 Operation signatures

Operation signatures encompass three elements:

- The name of the operation. The operation's name is the only part of the signature that may contain hints towards the operation's semantics.
- The operation's return value's data type.
- The name, data types, and order of the operation's parameters.

Similarly, in message–based systems, the declaration of data structures lacks semantic information.

The operation's semantics is usually described in another language, be it as a comment right at the signature's declaration, or in a separate document. These descriptions usually lack a standardized structure. Therefore, they are not analyzable automatically.

If an operation's signature is written in a specific programming language, the resulting component is only usable in this language. If other programming languages need to interact with the component, compatible signatures must be written, mostly by hand. Because of technical differences, this might not even be possible, e.g. because of different calling conventions (who puts parameters on the stack, in which order are they put there, and who removes them from the stack), or because of differing type systems or type representations. As a remedy to this situation, *interface definition languages* (IDLs) were introduced. IDLs are commonly called language–independent, however, they represent nothing else than a language for declaring operation signatures. For IDLs, compilers are available to transform the IDL signatures to (multiple) programming languages.

To be dynamically usable, more information than typically can be described in signatures, is necessary; especially in systems that exchange sensor measurements: given a parameter *distance*, it is not clear, whether the values are metric or imperial, or with what scaling factor the values are encoded. If this were to be encoded in the type system, it would lead to an explosion of data types. Also, the value's scaling is conceptually not part of the data type, but rather an attribute of the data field.

2.3.8 Usage spectrum

The usage spectrum of descriptions can cover a wide range:

- *Documentation:* Documentation for end users and system integrators describes the component as a black box. It should contain all information necessary to successfully run the component in a user–defined environment. Therefore, the documentation must contain information about its parameters, and how to set them. Documentation also contains a description of what the component is supposed to do, and sometimes, how this is accomplished. A human user is thus able to determine, which component may be used for what purpose.

- *Specification:* During the development process of a component itself, a specification of the component is produced. This specification not only contains information about the component's external interface (as the documentation), but usually also contains information about the internal setup of the component. Such specifications are produced in varying levels of detail. During the development process, they are usually refined iteratively, especially when the development process employs the MDA approach. Specifications mostly are written in languages situated above the programming language level, i.e. in either a specification language like Z or UML, or in a natural language.

- *Code generation:* For some formal specification languages, transformation to other languages — most importantly programming languages — are available [4]. Code generation from specifications makes sense, if the specification is detailed enough. Choosing the optimal point to transform a specification into code (that still needs to be completed) is non trivial. The amount and type of code produced may vary greatly, ranging from empty function stubs, or class structures to almost complete applications. Also, the size and speed of the code produced can vary [14].

- *Testing:* With tool support, assisted or even automated generation of tests for a component is possible, whenever the component's interface is specified completely in the description. This includes not only input and output parameter ranges, but also timing properties, and possibly other quality attributes at the interface level. There are even approaches for gray–box testing, e.g. [5].

- *Plug and Play:* If a component can be identified, and information about its communication capabilities are available (in its description), it can be integrated into a system automatically, i.e. low–level setup of communication parameters can be completely automated. Of course, this not only requires access to the description, but also that the description is machine–readable and that there is a least common denominator for interacting with the component. This e.g. includes automatic bit rate detection on a communication bus. Also, if a TDMA communication scheme is involved, it is potentially necessary to create a new communication schedule. This can also be automated, if all respective information is contained in the component descriptions.

- *Discovery and dynamic cooperation:* Besides the basic component identification necessary for plug and play, it is desirable to be able to discover new components at a higher level. Once components can be discovered by applications, they may be used dynamically. To dynamically use a component, knowledge about the component and how to use it is necessary. Making the component's description available to applications at least provides the basic information necessary for dynamic use. Depending on the kind of (semantic) information, varying degrees of dynamics can be achieved, ranging from hard–coded rules, or manual assistance by an operator to true dynamic interaction.

- *Configuration management:* In complex systems a tool based configuration management is essential to handle complexity. Having machine–readable descriptions — including possible and current configuration settings — readily available for the configuration management tool, is a prerequisite for efficient configuration management.

Descriptions that include all necessary information to achieve at least plug and play and discovery are called *self–describing*.

2.4 XML

In this section, XML technologies important for this work are briefly recapitulated. This description of course cannot be exhaustive. It provides a coarse overview that will ease the understanding of CODES in chapter 7.

Since its introduction in 1998, XML [140] has been a rising star. XML is a simplified subset of SGML. XML is used to structure data and documents. XML files — also called XML documents — contain nested *tags*. Tags are elements that structure the XML document. They are enclosed in angle brackets (<, >). For each element, there is an opening tag and a closing tag: `<tag>` is the opening tag, and `</tag>` is the closing tag. Empty tags can be written either as `<tag></tag>` or as `<tag />`. XML documents start with an XML header, such as `<?xml version="1.0" encoding="utf-8"?>`. The header is followed by a document element. The document element may contain child elements. If the document element does not contain siblings, and otherwise adheres to the previous description, the document is called well–formed. Further, if the document can be successfully validated against a document type definition (short DTD), or against a XML Schema document (see chapter 2.4.2), it is called *valid*. Validation ensures that a given XML document is compliant with the *grammar* defined in either the DTD or the schema document. XML dialects defined by XML Schema documents or DTDs are mostly called *languages*, e.g. XSLT (see section 2.4.4). While some people argue that the XML dialects still are XML, calling them *languages* is common practice in the XML world. Thus, the XML Schema presented in chapter 7 also presents a *language*.

An element's content can either be text, child elements, or both. Further, an element can have attributes. Attributes are directly included in the tags: `<tag attribute1="value" attribute2="value">`.

An XML document's hierarchical structure of elements can be represented in a document tree — called *Document Object Model* (DOM). While there are other possibilities to parse and process XML documents, the representation as document tree is very useful, especially for technologies like XSLT and XPath.

XML is as successful because it constitutes a simple, standard way to exchange structured textual data between applications. Data is also readable and writable by humans. XML satisfies two important requirements:

- *Separating data from presentation:* if data is to be used ubiquitously, it must be separated from its presentation. Weather forecast data, e.g. is rather useless if it comes wrapped in HTML or other presentation centered formats.

- *Transmitting data between applications:* data must be exchanged between applications every day. As software integration projects tend to consume enormous amounts of time and money, more flexible ways to exchange data are needed.

2.4.1 XML name spaces

As XML was being widely used, conflicting tag names became a problem. The problem of colliding names is already well–known from programming languages. In programming languages, the notion of name spaces is a proven cure to the problem. It stands to reason that name spaces where also introduced in XML [139]. Name spaces are referenced in documents using attributes; usually the document element lists all name spaces to be used, e.g. `<tag xmlns:prefix="uri1" xmlns="uri2">`. Here, `xmlns` introduces a new name space. To distinguish tags from different name spaces, a prefix is prepended. The prefix and the tag name are separated by a colon. There can be a default name space for a document, e.g. the name space `uri2` in the previous example. All tags without prefix are assumed to belong to the default name space. Name spaces are identified by a URI. Usually, the name space identifiers look like URLs, however they do not need to actually point to anything.

2.4.2 XML Schema

Both DTDs and XML Schemas [133], [141], [142] are used to define the grammar of an application specific XML dialect. DTDs lack expressiveness, e.g.

it is not possible to specify that an element must contain an integer, as DTDs essentially lack the notion of data types.

XML Schemas are a more powerful replacement for DTDs. XML Schema documents are XML documents themselves. The usage of XML Schemas requires using XML name spaces, specifically the name space `http://www.w3.org/2001/XMLSchema`.

XML Schema introduces a type system. Elements of the target language must adhere to one of the predefined types, or the schema developer must derive user–defined data types.

The type system differentiates among *simple types* and *complex types*. Simple types cannot contain any child elements.

Types can be derived in three ways

- **restriction:** Types are created by adding constraints to a given type. XML Schema itself uses derivation by restriction, e.g. `xs:positiveInteger` is derived from `xs:integer`.

- **list:** List datatypes allow defining structures within the content of a single element. List data types are rarely used because current XML APIs do not allow accessing individual elements of the list's structure.

- **union:** Derivation by union allows defining new data types by combining the lexical spaces of several other data types.

2.4.3 XPath

XPath [77], [143] is a language that allows processing of values conforming to the XPath data model. Among other things, the XPath data model specifies how XML documents are represented as trees.

XPath is intended to be embedded in another language; notably it is embedded in XSLT (see chapter 2.4.4) and XQuery [89]. XQuery is a data retrieval language, much like an "SQL for XML". In both XSLT and XQuery, XPath's purpose is to select a set of nodes from a document tree.

Node selection is done by specifying a path into the document tree. Several shortcuts are available, e.g. `//name` selects all elements named `name`, no matter where in the document hierarchy. Further, regular expressions can be used in XPath expressions. Indexing for result sets is also possible, e.g. `//name[5]` selects the fifth element called `name`. XPath expressions may be nested. They may also rely on context provided by the containing language, e.g. in XSLT stylesheets, XPath expressions can rely on XSLT context. While this feature appears rather strange, it is the key to seamlessly embed XPath into other languages. The integration of XPath into XSLT is discussed in great detail in [77] and [78].

XPath version 2.0 is in many ways superior to XPath 1.0. Many of the path expressions written for this work would be nearly impossible to write in XPath 1.0.

2.4.4 XSLT

XSLT (Extensible Stylesheet Language for Transformations) [144] is language for processing XML documents. [78] defines XSLT as follows:

> XSLT is a language for transforming the structure and content of an XML document.

An XSLT processor takes an XML input document, an XSLT stylesheet, and produces an output document by applying the XSLT stylesheet to the XML input document. XSLT distinguishes three output methods:

- **XML:** the XML input is transformed into XML output. This can e.g. be useful to transform the output of program A into a structure that is readable by program B.

- **HTML:** the XML input is transformed into HTML. This is usually used to produce web pages. Currently, it constitutes for the main usage of XSLT.

- **Text:** the XML input is transformed into arbitrary text. This is e.g. used to produce application code skeletons in CODES (see chapter 7).

The XSLT transformations developed during this work relies heavily on features introduced with version 2.0 of XSLT. XSLT version 1.0 still was pretty focused on rendering XML documents in HTML. Applying structural changes to a document often were tedious to do. The pattern recognition capabilities were much weaker (cf. XPath 2.0 vs. XPath 1.0).

XSLT transformations heavily rely on pattern matching and subsequent application of transformation templates. The patterns to be matched are written in XPath.

As XSLT is a declarative language, there are no side effects. This permits the XSLT processor to apply templates in any order, and for any number of times without changing the result. Another reason why declarative languages are desirable is that they are optimizable. This was first claimed in 1970 in [22]. In the meantime, this has been proven correct with relational optimization techniques in relational databases. Unfortunately, current XSLT processors use little, if any optimizations.

The transformation process (see figure 2.3) has multiple steps:

- *Parsing of input:* both the XML input, and the XSLT stylesheet are parsed and represented as trees. The document tree model of an XSLT processor is somewhat similar to the DOM, yet they are not equivalent.

- *Structural transformation:* the input data tree is converted to the output tree according to the rules of the XSLT stylesheet. Conversion is done by applying the template rules of the stylesheet to the document input tree.

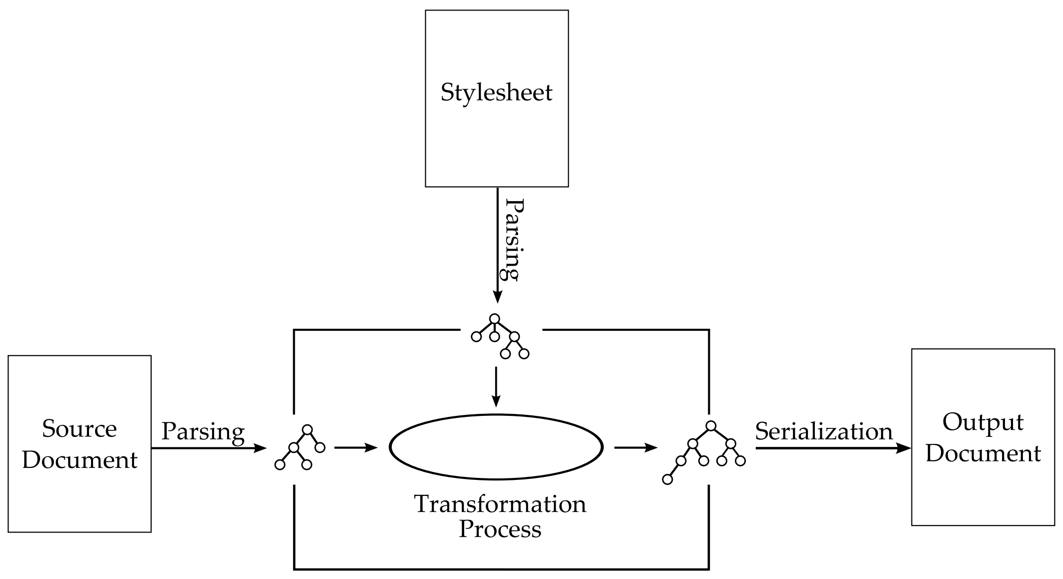


Figure 2.3: The XSLT transformation process

- *Formatting:* the output tree is serialized into the requested format, i.e. to XML, HTML, or text.

Usually prefixed with `xsl`, XSLT tags are in the name space `http://www.w3.org/1999/XSL/Transform`.

A typical XSLT stylesheet consists of a sequence of template rules. Each rule describes how a particular construct should be processed. The rules can appear in arbitrary order.

Chapter 3

Development for small components

Traditionally, small embedded systems were used in isolation, e.g. to control a washing machine, or similar appliances. Today, small embedded systems are used ubiquitously not only in appliances, but also in machinery and cars. They are not used in isolation anymore, but they are interconnected using various networking technologies. As networking became mandatory, software complexity exploded. Context awareness is on the uprise, also increasing complexity. Typical tasks of such embedded devices are:

- data acquisition
- data filtering
- data conversion
- data fusion
- communication in a timely manner

All of these tasks may need to be performed in a different way, depending on context.

The technology employed in such devices is mainly chosen by cost and size factors. Energy consumption often also plays an important role. Thus, these devices tend to have very limited capabilities in terms of processing power and memory. Power consumption limits the possible clock frequency and memory sizes. Cost limits pretty much every parameter in system design.

Powerful chips become cheaper, smaller and more energy efficient. Yet, older, less powerful chip designs are pushing forward into markets and application areas where the usage of micro controllers was unthinkable only a short time ago. Extremely limited architectures, such as the MARC4 [7] are newly developed and introduced. The MARC4 features a 4 bit CPU core. Even members of this family provide communication facilities, e.g. the ATAM682-x [8], which provides UHF ASK/FSK communications.

It is essential, that adequate development support is provided, even for the most cumbersome platforms.

3.1 UML and MDA

The *Unified Modeling Language* (UML) is a standardized modeling language. It includes a standardized graphical notation. This graphical notation is used to create the so–called *UML model* of a system. Different diagrams each represent a partial view of the complete model [3], [119]. The concepts in UML are based on object–orientation [3].

UML is defined via a meta model [3]. The UML meta model is expressed using the OMG's *Meta Object Facility* (MOF). Notably, the MOF specifies how models can be accessed and interchanged. It defines the *XML Metadata Interchange* (XMI) format. XMI is used to exchange UML models with different tools.

In UML, multiple views of a system are defined, and each view has a number of diagram types associated with it [3]:

- The *User View* shows the system from a user's perspective. This view is composed of *Use Case Diagrams*, which describe the system's functionality.
- The *Structural View* contains the static aspects of a system. Two diagram types are used in this view:
  - *Class Diagrams* contain classes and associations. They describe the static structure of a system.
  - *Object Diagrams* describe the structure of a system at a particular time during its life. They represent examples of structure.
- The *Behavioral View* comprises a system's dynamic aspects.
  - *Sequence Diagrams* are used to specify the system's behavior towards its interactors.
  - *Collaboration Diagrams*, which were renamed to *Communication diagrams* in UML 2.0 show which classes of a system realize its behavior.
  - *Statechart Diagrams* specify the states and changes of objects participating in behavior due to outside interaction.
  - *Activity Diagrams* describe a class' behavior in response to internal processing, rather than external stimuli. They communicate flow of control and information.
  - *Interaction Overview Diagrams* are a mixture of activity diagrams and sequence diagrams [52].
  - *UML Timing Diagrams* are used to show timing behavior in real–time systems.

- The *Implementation View* covers structural and behavioral aspects of the model's realization. *Component Diagrams* define the organization of software components, and their dependencies.

- The *Environment View* covers the aspects of the domain in which the system must be realized. *Deployment Diagrams* describe the processing resource elements, and the mapping of software components to the processing elements.

Extensibility is provided by the means of so–called *UML profiles*. New concepts may be introduced to the language via so–called *stereotypes*. These define new concepts via the means of UML itself [119].

UML profiles are used to define domain–specific languages. This fosters the view of UML not as a single language, but as a family of languages, e.g. the UML Profile for Framework Architectures [51], the Real–Time UML [39], [40], or Executable UML [95].

UML forms the necessary basis for the evolution of model–driven approaches, specifically the *model–driven architecture* (MDA).

In MDA, the focus is on transforming *Platform Independent Models* (PIMs) into *Platform Specific Models* (PSMs). A PIM can possibly be transformed into a number of different PSMs, one for each platform. The generation of code from a model can essentially be seen as another PIM to PSM transformation. Transformations are performed by applying *mapping functions* to the input models [96].

The OMG defines four modeling layers, termed M0 to M3. Layer M0 represents the running system in which actual instances exist. Layer M1 contains models, e.g. UML models. It defines the concepts instantiated in layer M0. In other words, each element at layer M0 is an instance of an element at layer M1. Layer M2 specifies the elements available for instantiation at layer M1, e.g. the concept of a class or an attribute is defined at layer M2. Again, each member at layer M1 is an instance of an element at layer M2. The relationship of layer M2 and layer M1 is the same as the relationship of layer M1 and layer M0. The model residing at layer M2 is called a *meta model*. E.g. the UML meta model resides at layer M2. Every UML model at layer M1 is an instance of this meta model. Likewise, layer M3 contains the model of layer M2, called the *meta meta model*. Within OMG standards, the MOF resides at layer M3. Modeling languages like UML are instances of the MOF. To prevent an infinite recursion, the OMG defined that all elements of layer M3 must be defined as instances of concepts at layer M3 itself [79].

This meta modeling structure is used in MDA to

- provide a mechanism for defining modeling languages and to

- allow transformations of models in a source language into models in a target language using transformation rules. These rules employ the meta models of both the source and the target language to define the transformation.

MDA endorses *Domain Specific Languages* (DSLs) based on the MOF. In practice, MDA encourages using UML profiles [137]. By providing a DSL based on the MOF and a transformation definition to another language, MDA can be extended.

Modeling smart real–time components (as targeted by this work) using UML was not deemed desirable. The requirement for a readable description format (as discussed in section 7.1), paired with easy processability are the main reasons for not using UML. Further, the modeling of pure data structures in UML is awkward. Data structures would have to be modeled via classes without functionality [101]. The amount of machine–readable detail necessary in the descriptions would further complicate UML modeling. Nevertheless, the outlook of basing the CODES language (see chapter 7) on OMG's MOF would essentially make it a DSL in terms of the MDA.

3.2 Software development for small components in practice

3.2.1 Resource constraints

In practice, it is still very common to program micro controllers without the help of any high–level tool. On very small devices, it is even necessary to use assembly language, as these micro controllers do not offer any RAM or stack space to speak about. The Atmel ATtiny15L [6], e.g. has only 32 8 bit registers, and not further RAM. It is equipped with a small stack capable of storing three return addresses. Its program memory can hold up to 512 instructions. There is no tool beyond an assembler available for this class of microcontroller. Programs are usually debugged in software simulators; an interface to a in–circuit emulator is not very common. This is tedious work, but it can be done, as these programs certainly are limited in size and complexity. Of course, all advantages and drawbacks of using assembly language become apparent: the code is very efficient, but it is hard to maintain, as a lot of optimizations and side effects are used up to their possible extents.

A little further up the scale, e.g. for the Freescale HC08[1] or Atmel AVR[2] families of micro controllers (8 bit cores, a few kilo bytes of RAM, several kilo bytes of instruction memory), compilers are available. The most common programming language for these devices is C. Some vendors also offer stripped down versions of C++, Pascal or even Basic. Most of these compilers come with declaration files that define the locations of Special Function Registers on common devices. Language support is often limited to a subset of the corresponding standards (especially for C++). On the other hand, the compiler vendors enhance the language to make certain features of the

[1]http://www.freescale.com
[2]http://www.atmel.com

hardware platform available[3]: some devices offer a special addressing mode for certain locations of the address range[4]. Especially C compilers tend to have default settings that will not put local variables onto the stack, but onto the heap. While this allows to limit the stack size, it breaks with the semantics that programmers are used to, and provides a common pitfall when using some sort of recursive calling of a function. Most compilers also come with some libraries for common tasks. Despite the premium price vendors usually charge for embedded compilers, these and also their libraries often show serious bugs that a programmer has to work around. The Cosmic C Cross Compiler for the HC08, Version 4.1 [34], e.g. has an undocumented off–by–one (byte) bug when computing the amount of stack space needed for functions with more than four parameters. Its C library has similar bugs throughout its memory management functions (e.g. `calloc`, `free`)[5]. On this device class, compact operating systems can be found, e.g. ProOsek from 3soft [1]. Some of these provide real–time scheduling capabilities, e.g. the real–time executive developed by Kilian Rall at the Core group in Ulm [117]. Furthermore, communication middleware like COSMIC is available for this class of devices, though sometimes with restrictions. COSMIC, e.g. supports only Hard Real–Time and Non Real–Time communication on the HC08. Soft Real–Time would use up too much CPU resources. The availability of operating systems and communication middlewares greatly eases software development. The developer can reason in terms of tasks, and does not need to ensure correct scheduling manually over and over again. A communication middleware provides the developer with an API that is hopefully easy to use and relieves him of the duty to write device dependent functions that must manipulate device specific registers. If both operating system and middleware are integrated with each other, as is the case with COSMIC, the developer is also freed from the task of scheduling messages manually on the network. Higher level tools such as UML designers and code generators are used rarely, mostly because of the limited device resources and missing language features of the compilers. Application software complexity on these types of system, however, tends to grow with the degree of distribution.

Climbing the next step in terms of micro controller performance, 16 bit devices offer not only more CPU power, but often also considerably more memory. Compilers are common, assembly language is used only occasionally. Compiler vendors still enhance their tools with device specific specialties, however their number tends to be lower than with 8 bit controllers. As the peripherals become richer in functionality, so does their complexity. Therefore, device manufacturers offer configuration tools that try to ease application specific device setup. Infineon e.g. offers DAvE (Digital Appli-

[3] Unfortunately, this usually happens in an incompatible way — even for different compilers for the same device.

[4] The HC08 family, e.g. has a special addressing mode for the first 256 bytes of memory.

[5] The impact of these bugs is rather low, as these functions are rarely useful on devices with only 2 kilo bytes of RAM.

cation virtual Engineer) [64], a tool to interactively configure their C166 and C166V2 families of controllers. Rudimentary *device drivers* for on–chip peripherals are available, e.g. Infineon offers a CAN driver [65] for its C166 family. Operating systems are commonly used. Supporting development tools and debugging facilities are readily available and sometimes well–known as older PC technology makes its way to the embedded market, e.g. x86 compatible controllers and the MS–DOS operating system.

Currently, state–of–the–art micro controllers are equipped with a 32 bit core. ARM[6] and PowerPC[7] are among the most popular families in this category. Memory and processing constraints are almost unheard of, as RAM sizes reach mega bytes and clock frequencies reach a few hundred megahertz. Operating systems are readily available from various vendors. Compilers mostly do not have any limitations when compared to workstation compilers. More recently, powerful 32 bit micro controllers with only some kilo bytes of RAM started to become available. These allow for rather cheap devices that need lots of processing power for small applications. Even though operating system support is limited because of the limited memory, compiler and tool support is the same as for full–blown 32 bit systems.

Most advances in software engineering require rather large amounts of resources. Therefore, they are only available for powerful platforms, mainly in the 32 bit class. Often people reason that computing power will be cheap and vastly available soon, even for embedded applications. While it is certainly true that prices for hardware equipment are constantly falling, tiny devices like 8 bit or even 4 bit micro controllers are still extremely widespread. 8 bit micro controllers still accounted for the largest market share in terms of units shipped[8]. They will surely not be replaced by more powerful devices, as they continue to be cheaper and more energy efficient than their counterparts. Volume prices for 8 bit micro controllers in low pin count packaging average around 40 US cents per part, whereas low end 32 bit controllers cost around five US dollars in volume quantities. Of course, more sophisticated parts are substantially more expensive, e.g. the 68HC908AZ60A, which provides an 8 bit core, combined with 60k of Flash, 2k of RAM, and a CAN controller, comes in a 64 pin TQFP package and currently (April 2007) averages around 9 US dollars per part an 1000+ quantities. 32 bit designs, e.g. the MAC7101, providing an ARM7 core, 512k Flash, 32k RAM, and four CAN controllers, cost an average 16 US dollars per part in 1000+ quantities[9]. Thus, it is important to come up with new ways of developing software efficiently on those limited controllers, as they continue to be used in ever more challenging ways, especially in terms of networking.

[6]e.g. from http://www.nxp.com
[7]e.g. from http://www.freescale.com
[8]Source: Gartner Dataquest August 2003
[9]Source for prices: Freescale semiconductor

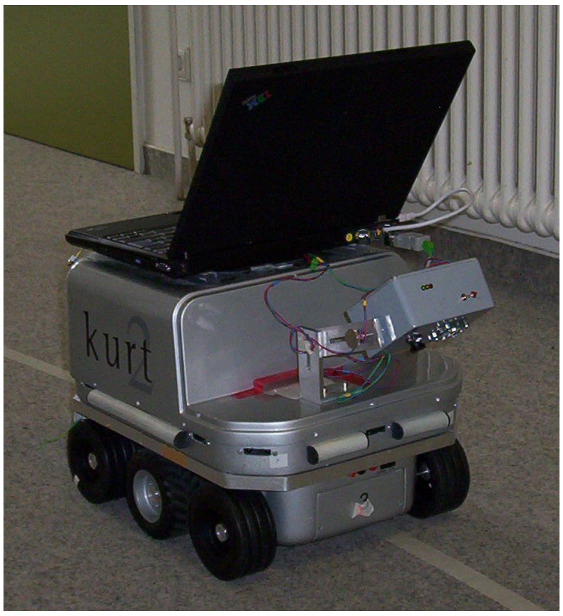


Figure 3.1: Kurt, one of our robot testbeds

3.2.2 Examples of common problems

While a middleware API like COSMIC's (see section 6) is helping a lot, there are still open problems, especially when different devices from different developers need to interact. Insufficient documentation and misunderstandings provide a rich ground for hard–to–find bugs in the software. Each developer will test his component until it works to his satisfaction. When putting different components together, strange things can happen, however.

As an example, think of a mobile robot application where two autonomous robots (called Kurt and John Silver) are supposed to move in coordination. Both robots are equipped with a tank–like drive system that employs one motor per side (see figure 3.1). There are two developers involved — say Carlos and Hubert — and they make use of the COSMIC middleware.

Among others, the following components are involved:

motor control: the motor control component takes the desired speed for both motors as an input event. It will keep both motors running at the given speed. As the involved robots differ mechanically, the motor control component differs on both robots, but it has the same "interface" (see below).

coordination control: this high–level component coordinates both robots and "instructs" their motor control component by generating appropriate events. As both motor control components have the same interface, only one coordination control module that can control both robots is to be developed.

Carlos specifies (in a clear–text document) the "common" interface for the motor control component: the desired speed is to be given in encoder ticks per second[10] for convenience. He gives a function that relates encoder ticks per second to actual wheel speed in centimeters per second for John Silver. Further, he writes that in the event's contents, there are two fields, the first one for the left motor, the second one for the right motor. Both fields are encoded as a 16 bit integer. Positive values are to be mapped to driving forwards, while negative values will make the motor turn backwards. Carlos programs his motor control component for John Silver and hands the specification to Hubert, so Hubert can write the motor control component for Kurt.

Later, Hubert writes the higher–level coordination control module. For testing, Hubert uses his own motor control component on Kurt. It is basically equivalent to Carlos'.

When testing the coordination control module on John Silver, nobody notices anything. However, once both robots engage in their cooperation task, the problem becomes apparent: the two robots move at different speeds. How could this happen? Unfortunately, Hubert did not pay enough attention to the function that maps encoder ticks per second to centimeters per second, which is slightly different for both motor control components. So even though the coordination control module works correctly, and in principle, so do the motor control components, the application fails. While this bug is rather easy to find, the causes for subtle differences in behavior may be extremely hard to find in other cases.

Yet another thing could be done wrong in this example: imagine that Hubert mixed up the order of the fields within the speed event. The result would be that Kurt would turn exactly the opposite way as it is supposed to. Suppose he programmed his motor control component in C, and the event's content is declared as a simple array: `int16_t event_content[2]`. As he is a lazy programmer, he does not introduce any named variables for access that would make the code quite a bit more readable. Also, for his internal operations, he uses yet another array `int16_t internal_desired_speed[2]`. According to Murphy's law, the association of *left* and *right* is just the other way round for both arrays, but Hubert does not pay attention and simply copies the array's contents. This will result in the behavior stated earlier: Kurt will do all turns exactly in the opposite direction.

From this example, two important lessons can be extracted:

1. All physical entities should be represented in universally usable format derived from the SI units, e.g. speed should be represented in meters per second, optionally multiplied by a constant factor, or offset by a constant value. If specified explicitly, this ensures that misconceptions like two different "ticks per second" units in the example, are avoided.

2. Access to fields within data structures should only be done via named members.

[10] On each motor, there is an optical encoder that is used to get a speed feedback.

Though these points might seem rather obvious, and most developers will state that they adhere to them, they are violated in practice, if they cannot be enforced or automated. In COSMIC data sheets, physical entities must be specified with a unit derived from the SI units. It is simply impossible to define “ticks per seconds” as the unit for a speed field. Instead, a common reference is enforced. Also, named fields and automatic code generation ensures that the programmer will get a code skeleton with named members as a start. Even though this cannot completely solve the second point, it provides a good starting point and at least enforces the programmer to use named access to input or output fields when receiving or publishing events.

Chapter 4

Requirements for successful system integration

The construction of a system based on components is ideally limited to *wiring* the selected components together into a specific configuration. During this integration phase, it is crucial to have tool–based support for (1) determining whether the selected components will be able to cooperate successfully and for (2) managing the created configuration. The presented requirements should be met by the CODES approach as presented in chapter 7.

4.1 Compatibility Checks

Compatibility checks are situated on a number of differing levels ranging from basic low–level issues concerning technicalities of communication up to semantics behind certain operations:

- All components must be from an *identical component world*. Of course, this is rather obvious. Components that are not based on the same abstractions will not be able to cooperate, e.g. .NET components and JavaBeans. *Bridging components* are a means of integrating different component worlds, yet it is extremely hard, if not impossible, to retain real–time and other end–to–end properties across such gateways, as some necessary abstractions may be missing on either end.

- *Communication must be technically possible* among components, i.e. they must share a common physical layer, i.e. network media, voltages, bit rates, etc. Even if the components run on a single node, they must agree on the communication protocols to be used.

- *Data encoding* in messages exchanged must either be known and agreed on by all components, or it must be deducible. The latter is easily possible by either exchanging the encoding to be used before operational communication, or by using a universally decodable message format, e.g. suitable XML encoded messages.

- *Interfaces* must match for both the provider and the user of an interface. This includes the name of the operations, and the name and type of the parameters involved, i.e. the operations' signatures. The usage of only valid ranges on parameters cannot be enforced by most approaches. Unfortunately, many components do not check their parameters profoundly on invocation of an operation. This not only leads to malfunctioning components, but also to a wealth of security problems, as documented in the news almost daily. For an embedded system where safety of operation is a major issue, special precautions must be taken. Any tool–based support greatly eases this error–prone and tedious task.

- Whenever (hard) real–time communication on a TDMA basis is involved, the respective *communication schedules* must match, or it must be possible to modify them at run–time without introducing faults.

- The use of *bandwidth* on the communication bus must be kept at feasible levels. This is an issue especially on field busses which have rather low bandwidths. When weak nodes are participating, the bus load may not even reach 100%, as these nodes are not able to process messages at full bus speed. The Freescale HC908AZ60A, e.g. is perfectly able to run the CAN bus at 250kbps, 500kbps, and even 1000kpbs. Yet, it is not able to process messages at full rate. It is capable of processing only about 100 to 200 CAN messages per second using the COSMIC middleware. Since it has only a single receive buffer, message loss will occur on burst transfers that otherwise would be well below the possible limit. Therefore, a feasible bandwidth use for such a node also means that communication must adhere to a minimum inter–arrival time in order to keep the CPU load at a feasible level.

- Cooperating components must agree on the *semantic meaning* of messages and operations. For truly dynamic cooperation, this is the hardest problem that needs to be solved, as the semantics must be encoded and processed by a machine, preferably without any intervention of a human user.

While most of these issues seem obvious and are handled pretty well already on "full–size" computers like PCs, they still are very current problems in the field of small networked embedded systems. Here, the overhead of communication middleware must be kept to a minimum; technologies like CORBA and .NET are too bloated. Also, in embedded systems, there is a much greater heterogeneity of communication media, communication protocols, and computational platforms. This situation has become especially visible in the automotive domain, and there are efforts to remedy the situation. Most notably, the AUTOSAR[1] organization published their first specifications including a sort of communication middleware in October 2005.

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

Unfortunately, there is basically no material publicly available without an NDA.

Compatibility checking is essential not only during system integration. If possible, it should be done at the following points in time:

- At *system design time* a number of system and application constraints are defined. All components in the system must comply with these constraints. Further, it must be ensured, that all designated system components are compatible with each other.
- At *System start up*, automated compatibility checking should be executed. This will ensure that a system remains in a compatible state throughout its life–time. Checking on start up addresses replaced or added components.
- On *dynamic integration*, e.g. when a new component is added to an already running system. The main concern for safety critical systems is that the existing system can continue to operate without being affected or even compromised.

4.2 Configuration management

Configuration management is an essential part for the maintenance of distributed systems. For any service personnel, it is vital to determine, what nodes are present in the system, and what their particular configuration is. A maintenance task may consist of changing the configuration because the system is about to be used for a new application.

The configuration information mostly consists of the following information:

- Some form of *unique identification of each node*. Mostly the identifiers are meant to be unique worldwide, so the name space must be large enough. 48, 64, and 128 bits are common sizes. 48 and 64 bit identifiers usually require a centralized scheme where a certain number of bits is reserved for a manufacturer id. This manufacturer id must be allocated with some registry. One well–known example is the Ethernet family, where each NIC has its own unique 48 bit identifier. 24 bits are used to identify the manufacturer. The manufacturer identification is called *Organizationally Unique Identifier* (OUI). Manufacturers must register with IEEE [2] to allocate an OUI [151]. 128 bit and larger identifiers are mostly produced locally in a decentralized fashion. Collisions are statistically improbable. One such scheme, the *universally unique identifier* (UUID), has been proven to work reliably for more than a decade now [152].

[2]OUI assignments can be done at http://standards.ieee.org/regauth/oui/index.shtml.

Even though in some systems, e.g. in COSMIC systems, it is not necessary to be able to identify a certain node during operation, it is essential that each node can be identified and addressed for configuration. It is e.g. possible to have two functionally identical components in a system. Yet, each of them must have a slightly different configuration for the application at hand. Therefore, the configuration facilities must provide access to each device individually.

- *Communication parameters*. All parameters necessary for successful communication are part of the device configuration, e.g. the bit rate used on the bus. Further, any predefined schedules, e.g. for hard real–time communications are also part of the configuration. Communication parameters may also include quality attributes.

- *Component specific parameters*. Depending on the kind of component, differing parameters are part of the configuration, e.g. the sampling period of an infrared distance sensor. For this reason, the configuration facilities must provide a generic way of specifying parameters and their values.

Configuration management tools assist the user in the following areas:

- *Planning and building* distributed systems. During the planning phase, the configurations of the individual nodes are taking on form. Parameter by parameter, settings are defined. *Simulation* of the system in this stage is often a desirable feature in this stage.

  As the system is being constructed, it may be desirable to emulate those parts that are not yet implemented "for real".

- *Automated configuration*. It is desirable to automatically define as many parameters as possible. Interesting candidates are low–level communication parameters, such as the bit rate. With the knowledge of e.g. how many messages need to be exchanged per second, these parameters can be set to a valid starting point.

- *Discovery* in the system. Whenever the system needs maintenance, it is necessary to detect all components in the system. Also, for diagnostic purposes, it is important that all (active) components are found. This provides a fast way of detecting components in a fail–stop state.

- *Queries* to the system. If components carry their description and their configuration, an on–line query mechanism can be included in the system. Such a query mechanism tremendously eases maintenance and analysis of an hitherto unknown system, as the relevant information can be obtained quickly.

Configuration management facilities may either be included in the normal operational interface of a component, or they may be provided via a dedicated interface. Having these facilities included in the normal operational interface has the advantage of keeping the number of access methods down, and therefore reducing complexity on the technical level. However, this bears the danger of hiding the configuration management facilities, which should at least provide a distinct interface on the conceptual level.

Chapter 5

Related Work

There is a large amount of description and specification languages. For most uses, one can even choose among a number of them for the same purpose. It would be impossible to discuss all languages. Therefore, a number of interesting technologies featuring description mechanisms have been chosen. Web services, Universal Plug and Play (UPnP), and Jini represent general purpose approaches. Web services are mostly not written according to a centrally defined standard. As interchangeability suffers in such decentralized worlds, UDDI was introduced to provide central registries, which also enable the discovery of Web services. UPnP's most interesting features are its centrally defined device architectures that ensure basic compatibility among vendors, and its integrated discovery mechanism allowing for the dynamic integration of new devices. The complete network setup is automated and completely configuration–free. Jini is a middleware framework bearing resemblance to UPnP. In contrast to UPnP, Jini is limited to the Java language, and in practice, to RMI mechanisms for interaction. Further, Jini does not offer a dedicated description language for the services offered. Instead, Java interfaces are used to identify services of interest. While this demonstrates that an extra description language may not be necessary in all cases, it also shows that programming languages are inadequate as component descriptions in some cases. More specifically, whenever detail information beyond e.g. the data type of a parameter is necessary, solutions based on programming language type systems tend to be inadequate. If the necessary information can be encoded at all with the required flexibility, the number of types necessarily explodes. The second part of this chapter is concerned with examples from the embedded systems domain. IEEE 1451 is focused on low–level issues of sensors and actuators without direct access to a network. The flexible and machine–readable representation of physical units is of special interest. CANopen is especially interesting because it is running on CAN. Optionally, CANopen devices may store their own description within themselves. While CANopen's description mechanism is concerned with individual devices, LIN also features the description of clusters. Further, LIN provides the possibility to seamlessly simulate not–yet–existent nodes. In contrast to most other description languages, it is possible to define in–band flagging explicitly in the LIN signal definition. TTP/A may be viewed as a direct counterpart to COSMIC. Besides the communication protocol, a description language for TTP/A nodes exists. Unlike the preceding examples from the embedded world, TTP/A smart transducer descriptions are XML based. Similar to LIN, complete clusters are described in cluster configuration descriptions. An extensive tool–chain exists, and continues to be extended.

5.1 Web Services

According to the W3C, the programmatic interface of an application, made available on the web, is referred to as a *Web service*[1].

Because of the inherent distributed and dynamic nature of the web, discovery and description of Web services are essential for their success. A Web service's programmatic interface is described using WSDL (see section 5.1.1). The WSDL descriptions contain all necessary information to technically interact with the service. It does not, however, contain any semantic information. UDDI provides discovery facilities for web services. The data stored in the UDDI registries contains information describing web services on a higher level, e.g. using UNSPSC (see section 5.1.3).

5.1.1 Web Services Description Language

The Web Service Description Language (WSDL) [15] is supposed to provide documentation and the possibility for automating details of communication between the service and its clients.

Essentially, WSDL captures web services in a way comparable to method signatures. WSDL is a XML grammar structuring web service descriptions into seven sections:

Types: The Types section contains all relevant types needed for the message exchange. Types are defined in terms of XML Schema types.

Messages: Messages are composed of *Parts*. Parts have a name and a type associated with them. Parts can roughly be compared to parameter declarations of methods.

Operation: Operations are individually invokable entities of a web service. Operations can be likened to methods.

PortType: A port type groups operations on an abstract level.

Binding: A binding specifies exactly one protocol to be used for a port type, e.g. SOAP.

Port: A port specifies exactly one network address for a binding.

[1]http://www.w3.org/2002/ws/, last modified 2006/06/29

Service: A service groups a number of ports.

Currently, bindings to SOAP 1.1, HTTP GET/POST, and MIME are defined in [15], whereas [156] leaves the definition of bindings to an extra standard [157]. While WSDL 1.1 was defined using an informal syntax, WSDL 2.0 is defined using the Z specification language [124].

5.1.2 Semantic Annotations for WSDL

The Web Services and Semantics Project[2] recently published a first working draft of their *Semantic Annotations for WSDL* (SAWSDL) recommendation [158]. SAWSDL introduces extension attributes to WSDL (both 2.0 and 1.1). These attributes allow the linking of WSDL constructs to concepts in semantic models defined outside of WSDL. SAWSDL is independent of any specific ontology expression language, as the extension attributes to WSDL only contain URIs to the respective concepts. It is recommended that these URIs resolve to a document containing the referred concept's definition. There are three extension attributes defined:

modelReference: Each *modelReference* attribute specifies the association between a WSDL entity and a concept in a semantic model. It can be used to annotate XSD types, elements, attributes, as well as WSDL operations and interfaces. The purpose for annotating services at the interface level is to enable dynamic discovery, which is only possible when services are published, cataloged, and annotated with semantics. The categorization defined using SAWSDL can be used as input when publishing the service in a UDDI registry (see section 5.1.3). The annotation of operations provides a high level description of said operation.

liftingSchemaMapping: This is list of URIs that reference mapping definitions defining how an XML instance document conforming to the element or type defined in a schema is transformed to data that conforms to a semantic model. In other words, the output of this operation is semantic data.

loweringSchemaMapping: Similar to the *liftingSchemaMapping*, the mappings referred to by *loweringSchemaMapping* allow transforming semantic data into XML instace data. XSLT is proposed as a schema mapping language.

5.1.3 Universal Description Discovery & Integration

Universal Description Discovery & Integration (UDDI) [9] is a directory service for web services, and in principle, any other services. It is itself a web service, accessible via SOAP. Conceptually, UDDI is composed of three distinct directories:

[2]IST-FP6-004308

- The *White Pages* is a name register.
- The *Yellow Pages* is indexed by taxonomies, e.g. the *United Nations Standard Products and Services Code* (UNSPSC).
- The *Green Pages* gives information about technical details of services.

Entries in the directory service provide meta information about web services. Without this information, a web service remains uninteresting. The information targeted by UDDI encompasses providers and services. It is written in an XML grammar defined by the standard. The XML structures describe *business entities*, *business services*, and technical information needed for interaction with the target service. The technical information is contained in so–called *binding templates* that refer to *tModels* which in turn describe *unique concepts* [9]. Further, a large API for accessing the directory is defined by the standard.

Discovering a service based on a client's criteria, such as support interface, and access method is the main use of UDDI. This includes gaining access to a service's WSDL description.

5.2 Universal Plug and Play

Universal Plug and Play (UPnP) [98], [97] aims at enabling zero–configuration, invisible peer–to–peer networking for all kinds of devices ranging from portable devices to full–grown PCs. The main focus of UPnP is home automation in a very broad sense from managing one's LAN and Internet connectivity (the traditional IT domain) to lighting, monitoring and heating (the traditional home automation domain) to home entertainment systems (covering TVs, DVD players, and DVRs).

UPnP was introduced in 2000 with Microsoft Windows ME. It is managed by the UPnP Forum. The UPnP Forum was formed on October 18, 1999. On their web site[3], the UPnP Forum describes itself as follows:

> The UPnP Forum is an industry initiative designed to enable simple and robust connectivity among stand–along devices and PCs from many different vendors. As a group, we are leading the way to an interconnected lifestyle.

5.2.1 UPnP key features

no binary device drivers: UPnP components can interact without traditional device drivers. UPnP uses declarative network protocols along with machine readable descriptions to eliminate the need for traditional device binary drivers.

[3] http://www.upnp.org

device and service descriptions: Devices are described electronically in device description documents and service description documents. Written in XML, they play a key role in enabling autonomous peer–to–peer interaction between devices. These description documents can be seen as a form of device drivers: The description allows to dynamically call any of the offered operations.

based on common Internet protocols: All Interaction is based on widely used Internet protocols like IP, UDP, TCP, HTTP, and SOAP.

platform and programming language independence: Contracts are solely based on declarative protocols. Therefore, UPnP is platform and programming language independent. All messages are expressed in XML and communicated via HTTP.

network media independence: Because UPnP uses IP, any kind of base network can be used if there is an IP stack available and the medium provides sufficient bandwidth.

no API: UPnP does not specify API design on its components. Similar to TCP/IP, UPnP only specifies a network protocol, not an access method like e.g. the Berkeley socket interface for TCP/IP.

device control: UPnP enables device control in two ways. Remotely invokable service actions enable programmatic control. An optional HTML presentation page hosted on the device itself is intended for direct user control.

interoperability: Standardized device types and service types enable device interoperability across different vendors.

designed to work with firewalls: By using common Internet protocols like HTTP, UPnP works in a firewalled environment, in fact, the standard firewall of Windows XP is UPnP enabled[4].

extendability: Usage of the Flexible XML Processing Profile (FXPP) ensures that UPnP is extendable by non–standard or not–yet–standard features. FXPP parsers ignore any unknown tags in an XML document.

zero–configuration networking: UPnP features zero–configuration networking by using DHCP and Auto IP. All UPnP components feature automatic device discovery.

device architecture: Device and service template definitions are done by the UPnP Forum according to a common device architecture [131] provided by the UPnP Forum. According to this device architecture, UPnP components are structured into three basic building blocks: (1) *devices*, (2) *services*, and (3) *control points*. Devices can be containers

[4]Which can be a problem of its own from a safety point of view.

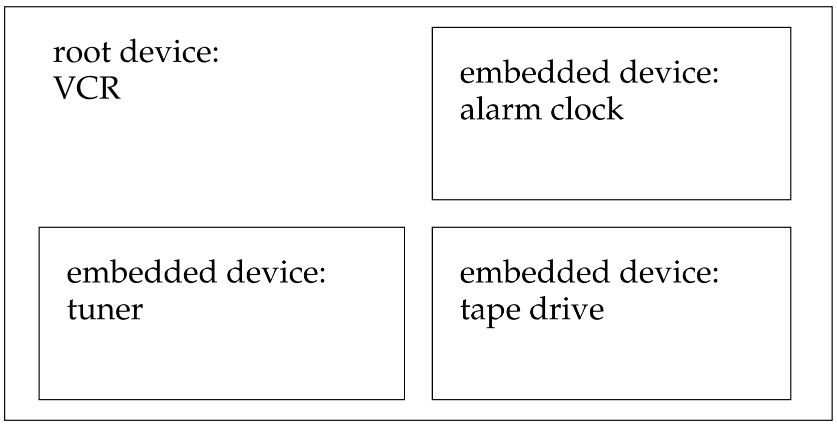


Figure 5.1: VCR device structure

for other devices. They can offer any number of services. There is a set of services defined for every standardized type of device. The structure of a device is captured in the device description document. A service is the elementary unit of control. It exposes actions and models its state with state variables. A service is defined by its service description. Services consist of three parts: (a) the state table, (b) the control server, and (c) the event server. The state table represents the service's state and updates its state variables. The control server receives and executes action requests. The event server publishes events to subscribers on state variable changes. Control points are capable of discovering and controlling other devices. Most UPnP components are expected to incorporate control point functionality to enable true peer–to–peer networking without the need for a large central system like a PC running Windows or Linux. Control points can subscribe to events of services.

legacy device integration: Non–native components can be connected to a UPnP scenario via *UPnP bridges*, e. g. a legacy weather station can be connected to a UPnP bridge and thus become UPnP enabled.

5.2.2 Description Documents

In UPnP, the electronic description of a (physical) component is partitioned into two logical parts: (1) device description and (2) service description. Both parts together authoritatively document the implementation of the device.

The device description structures a UPnP component into its logical devices, and lists the services that these devices offer. To give a short example (see figure 5.1), a VCR can be split into the VCR itself as a root device, and a tuner, a tape drive, and an alarm clock as embedded devices. Each of these logical devices has its own set of services associated with it. The list of services in the device description contains a URL for each service that points to the service's description document. Service descriptions specify

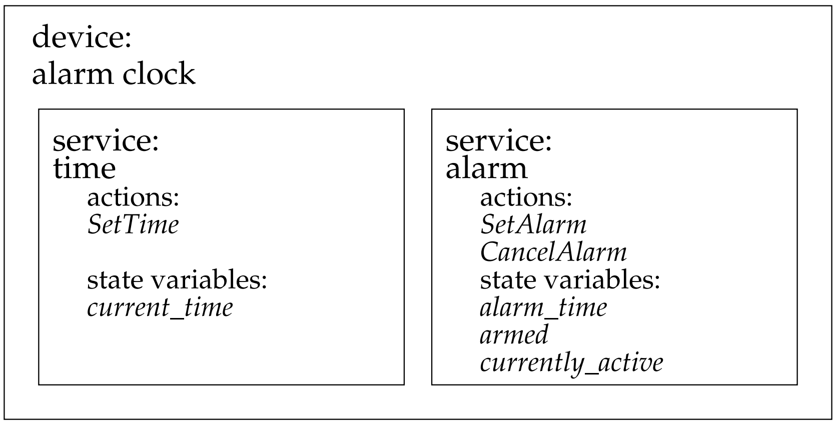


Figure 5.2: Device: alarm clock

what can be done with a certain device. A service can offer any number of actions that can be invoked from the outside. It also offers at least one state variable that can be queried or subscribed to. Again, as an example, consider the mentioned VCR, more specifically its embedded alarm clock (see figure 5.2). This (extremely simple) alarm clock offers two services: (1) time, and (2) alarm. The time service offers one action *SetTime* that allows some other device to synchronize the VCR's clock with its own. The time service also offers the current time as a state variable that can be subscribed to. The second service — alarm — offers the two actions *SetAlarm* and *CancelAlarm*. In terms of state variables, it offers the alarm time, whether the alarm is armed, and a variable called *currently_active*. While the first two are rather obvious, the third one deserves a closer look. When the current time matches the alarm time, and the alarm is armed, then any user of the alarm clock will obviously expect some sort of alarm, or notification. In UPnP, if one subscribes to the alarm clock events, all state changes of any state variable are generating an event message. Such an event message can be mapped to the expected alarm signal. To do so, an extra state variable *currently_active* of type boolean is introduced. It's value changes once the current time equals the alarm time. Before another alarm can be generated, this variable needs to be reset. This can be done in various ways, for now, it can be simply assumed that the alarm clock itself resets this variable once the current time and the alarm time do not match anymore. This will generate another event message.

Both device and service descriptions are written in XML by the device vendor. There are templates available for standardized devices and services. These templates are written in an XML dialect called UPnP Template Language. The UPnP Template Language (for devices and services) is derived from XML Schema (Parts 1 and 2). This ensures that UPnP device and service descriptions are machine–checkable in terms of completeness, syntactical correctness, and data types of values.

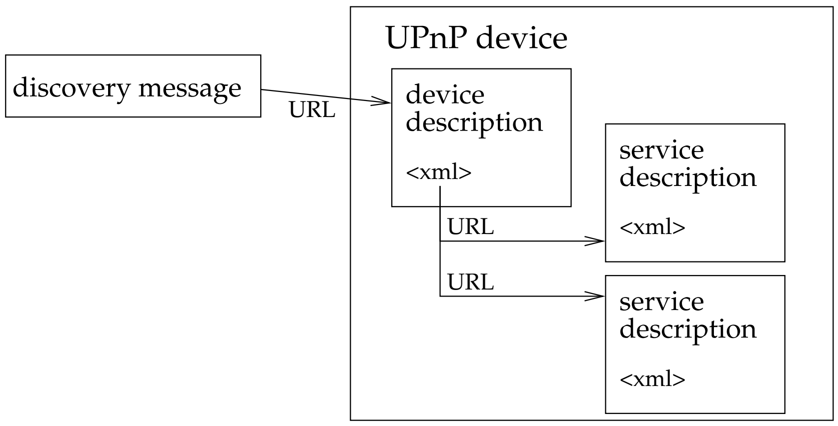


Figure 5.3: device and service descriptions

Device Descriptions

The device description document can be found at the URL given in the discovery response message or the presence notification message (see figure 5.3). Retrieval is done by making a standard HTTP GET request with the given URL. TCP is used as the transport layer protocol. Device description documents structure the UPnP component at hand into logical devices. The outermost device is called *root device*. The root device can contain a number of *embedded devices*. Besides this structural information, device descriptions contain information like the type of (root) device, its name, its model number, its serial number, and the like. Important from a UPnP point of view are the *UDN* (Unique Device Name), which is a UUID for the device that must survive reboots, the *serviceList* that lists all services offered by the device, and the *deviceList* that lists any embedded devices. These descriptions are written in an XML dialect called UPnP Device Template. This UPnP Device Template is produced by a working group of the UPnP Forum. All UPnP device templates are written in the *UPnP Template Language for devices*. The main goal of having UPnP device templates is to standardize devices. Currently (February 2007) standardized UPnP device templates are freely available at the UPnP Forum's website.

Service Descriptions

Service descriptions contain a list of all supported *actions* the service supports, a list of their *arguments*, and a list of its *state variables* (called service state table), as well as their data type and either range or list of allowed values. Each argument of an action should correspond to a state variable to enforce simplicity of the service model. A service can have any number of actions, including zero, but it must have at least one state variable. Such a service without an action forms an autonomous information source. Like device descriptions, service descriptions are written in an XML dialect called UPnP Service Template (also standardized by the UPnP Forum). UPnP Service Templates in turn are written in the *UPnP Template Language for services*.

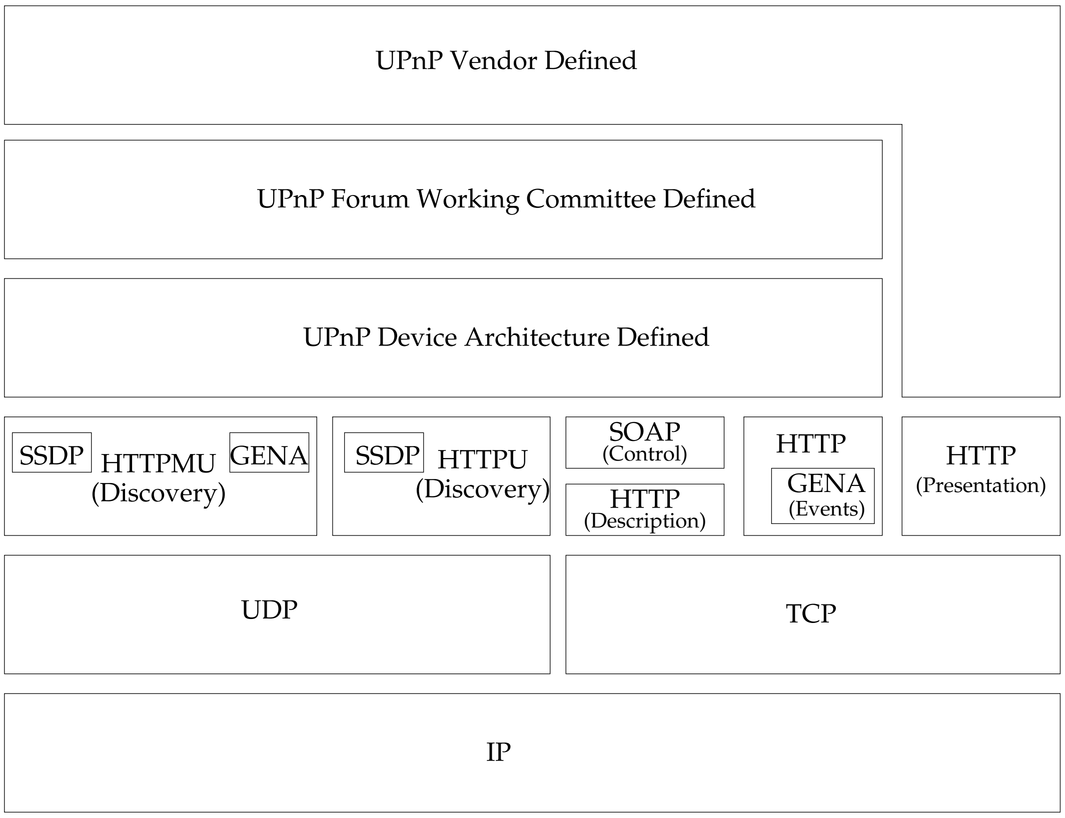


Figure 5.4: The UPnP Protocol Stack

Again, the main goal is to standardize services. Standardized UPnP service templates are freely available at the UPnP Forum's website.

5.2.3 Networking

The UPnP protocol stack

As a common base for networking, the IP protocol is used. Each device must obtain a unique IP address on connection to the network. To do so, each device must incorporate a DHCP (Dynamic Host Configuration Protocol) client, and an Auto IP implementation. If a DHCP server is available, the UPnP device must use the IP address assigned by the DHCP server. If no DHCP server is available, the device must choose an IP address according to the Auto IP standard. If a DNS server is available, the device should make use of it to provide more user friendly network addresses ("VCR (living room)" is easier to remember than "192.168.1.211"). For all discovery communication, UDP is used on top of IP. Device and Service Advertisements as well as search requests are made using a standard multicast address/port combination. HTTP in various variants plays a core part in UPnP networking. Advertisements are delivered via a multicast HTTP variant that is extended by GENA methods and headers, and SSDP headers. Search requests are delivered via a multicast HTTP variant that is extended by SSDP methods and headers. Search responses are delivered via a unicast HTTP vari-

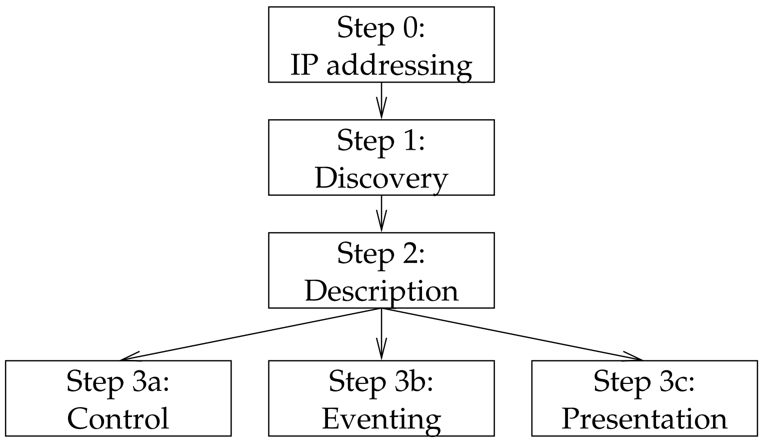


Figure 5.5: UPnP communication steps

ant that is extended by SSDP. For any other communication, TCP is used. Specifically, for description document retrieval, plain HTTP is used. For action invocation, SOAP over HTTP is used. For eventing, HTTP extended by GENA is used. For any optional presentation pages, plain HTTP is used. These optional presentation pages are entirely vendor specific. For all other parts, the UPnP Device Architecture defines their usage in general terms. Depending on the device type, more specific requirements are made. The vendor finally can enhance its devices by introducing further non–standard services.

Communication steps

UPnP communications take place in four steps (see figure 5.5). Communication starts by setting up the lowest level: the IP level. After that, device discovery generates a model of the reachable network. Third, descriptions are retrieved as needed. Then, the setup is complete and UPnP devices of interest can be put to use by three different means: control, eventing, and presentation.

0. *IP addressing*: UPnP devices use either DHCP or Auto IP to obtain a unique IP address. DNS is optional, but should be used if available.

1. *Discovery*: On connection to a network, the UPnP discovery protocol requires the newly connected device to advertise its presence and its services to the network. As advertisements are given with a timeout, they must be renewed by the device before the timeout expires. Also, the device should send a bye–bye notification before leaving the network to enable graceful shutdown. Whenever a control point is added to the network, it can search for other devices by issuing search requests. The control point can repeat searches at any convenient time. Both in advertisements, and in search request responses, only few essential specifics about the device(s) are transmitted. Among these, most important, is a pointer (URL) to more detailed information.

2. *Description*: Once a control point has discovered a device, it needs to retrieve that device's device and service descriptions before the device can be used. The discovery messages contain the URL of the device description document, which in turn contains the URLs of all offered services. Device and service descriptions are covered in more detail in section 5.2.2.

3a. *Control*: Usage of a UPnP device can be partitioned into three parts: (1) control, (2) eventing, and (3) presentation. (Remote) control is an active way of putting a device to use. Control points can invoke actions on any of the services provided by a UPnP device. Invoking an action is similar to a traditional RPC: the control point sends an action request to the service, which executes the specified action, and returns any results or error codes via a response message. Just like an RPC, UPnP device control is primarily designed to be done programmatically. Action request messages are expressed in XML and packed in a SOAP envelope. They contain the name of the action, and all arguments that the action needs, as well as their values. The response is also expressed in XML and packed in a SOAP envelope. It contains any return values and out arguments specified in the service description, as well as their values. If the service encounters any errors, it returns an error response, also as a SOAP encoded message. A service must complete an action and respond within 30 seconds. Actions that take longer must be defined to return early and send an event when complete. Error handling of missed response deadlines is application specific. If an action has any effects, these are modeled by changes in the state variables of the service. Besides invoking actions, control points can also query the values of state variables. State variables must be queried one at a time. While many state variables can also be subscribed to, some are not *evented*. These must be explicitly queried if their value is of interest. In case a state variable is *moderated* (see below), querying the variable can yield more up–to–date data than those provided by eventing. Again, a service must respond within a time frame of 30 seconds.

3b. *Eventing*: If one or more state variables are *evented*, the service publishes updates to all subscribed control points whenever any of these variables change. Control points subscribe to eventing by sending a subscription message to the service. The service response includes a duration for the subscription. To keep the subscription active, the control point must renew the subscription before it expires. Subscriptions can be canceled if they are not needed any more. Subscribers should also monitor discovery messages: if a publisher cancels its advertisements, subscribers must assume that their subscriptions have been effectively canceled. After first subscription, an *initial event message* is sent to the subscriber that contains the names and values of all evented variables. This allows the subscriber to initialize its model of the state of the service. Whenever at least one state variable changes, the service sends an update message containing the name and new value of the changed state variable(s) to all subscribers. This implies that a control point cannot subscribe to a subset of the state variables, but only to the complete set. Event messages contain a 32 bit sequence number called *event key*. Event keys are assigned per subscription. The initial event message has its event key set 0. For every event message, the event key is incremented. If a subscriber detects a gap in the event key, it must cancel its subscription and resubscribe to get a complete status update. The event key must wrap to 1. All event messages must be acknowledged by the receiver within 30 seconds. If the receiver fails to do so, the service should abandon the current message, but keep the subscription active until it expires. Some state variables may contain values that are too large and would generate too much traffic when used with eventing. These can be marked as *non evented* and are never sent to subscribers. However, they can be queried as described above. Some state variables will change too often to be useful for eventing. Such variables can be *moderated*. For moderated values, the service description can contain the optional annotations *maximumRate* or *minimumDelta*. *maximumRate* specifies that the concerned variable will not be part of an event message more often than the specified number of seconds. This is useful for continuously changing variables, such as e.g. CPU temperature. *minimumDelta*–annotated variables will not be part of an event message unless the value has changed by more than the specified number. This is only useful for variables that are defined as number or real data types, e. g. for an interrupt counter.

3c. *Presentation*: Unlike control and eventing, Presentation is targeted at human operators. Presentation is an optional feature that allows UPnP devices to hold one or more HTML page(s) that can be displayed in a browser for user interaction. The contents and capabilities of the presentation feature is completely up to the device vendor, but should be written in HTML version 3.0 or later. Presentation pages can contain anything from static information about the device to current device status and remote control features. Localizing presentation pages is recommended. The HTTP standard's ACCEPT-LANGUAGE and CONTENT-LANGUAGE features are used to determine the correct language.

5.2.4 Developing UPnP components and applications

The fact that no APIs are specified by the UPnP Forum is both a blessing and a curse. Such a specification is not necessary for UPnP components to be interoperable. UPnP stacks [99], [122], [66], [67] can be tailored to different device categories, or even single devices. However, when developing a UPnP enabled component, as a best–case, the developer has to choose among different implementations with their specific advantages and disadvantages, which are possibly unknown or undocumented. As a worst–case scenario, one has to develop a new UPnP stack without any guidelines. In practice, if a device is powerful enough to run a standard operating system such as Linux, Windows CE, or Windows XP, a developer can choose among different UPnP implementations from different vendors, e.g. the open source libupnp project hosted on SourceForge [67], Microsoft's UPnP stack [99], Intel's UPnP Microstack [66], or Siemens' UPnP stack [122].

5.2.5 Discussion

The major drawback of UPnP is its rather large protocol stack and the need for XML processing. Protocols like TCP/IP and HTTP are easily handled on PCs or devices with 32 bit micro controllers and a memory size of a few megabytes. However, when moving to 8 bit micro controllers, they put a serious burden on those devices in terms of CPU and memory load. On very limited devices with only a few kilobytes of RAM (e. g. 2 kilobytes on a common Motorola 68HC908AZ60A), it is practically impossible to run these protocols together with any other application, especially an XML processor.

UPnP does not take real–time into account. The only timing–related requirements are request timeouts of 30 seconds. Also, UPnP does not require any clock synchronization[5]. Using TCP/IP as a base protocol already defies real–time guarantees.

Media–independence is one of the main features claimed by UPnP. However, this is constrained by the use of the TCP/IP stack and therefore the availability and — more important — the suitability of a TCP/IP stack for a given medium. Running TCP/IP over a CAN bus is certainly possible, however, TCP/IP stacks for the CAN bus are not readily available. Even more, TCP/IP is not suitable on the CAN bus, as the maximum data payload per CAN message is 64 bits — so a single CAN message is barely able to hold a TCP/IP header structure. TCP/IP over CAN generates an enormous overhead.

UPnP does not take security into account. The Microsoft Platform SDK [99] states:

> Security Note: Because a UPnP service can potentially be remotely activated without authentication, it presents an area of vulnerability for a networked system. When UPnP services are deployed in a controlled environment, such as a home or business intranet where all the users are trusted, the risk of malicious attack is lessened.

UPnP is not proven to scale well. In practice, scalability is not an issue currently, as there are not very many UPnP enabled devices around. Until now, the UPnP Forum defined only twelve device types apart from the *basic device* that offers no features. The standardized device types do not even

[5]Of course, a clock synchronization service could be constructed.

cover all scenarios mentioned in the UPnP white paper (dated June 2000), specifically a generic storage device is missing. Other scenarios mentioned are not even described in enough detail to determine all device types necessary.

For some purposes, e.g. the increasingly popular streaming media services, UPnP is not able to handle all necessary communication. Other communication protocols are then used to handle this situation. The respective architecture documents [118], e.g. lists a number of so–called *out–of–band transfer protocols*. Currently published service descriptions are not capable of referring to these external dependencies.

5.3 Jini

The Jini middleware framework [145] is completely based on Java, and just like Java, Jini strives to overcome heterogeneity in distributed computing environments.

Jini's focus is on connecting appliances, such as PDAs, printers, cameras, and PCs. As Sun's Jini implementation relies on the Java VM, and HTTP and RMI over TCP/IP for network communication, it consumes vast amounts of resources, currently limiting its use to rather powerful devices like PCs.

Jini is designed to be robust, evolvable and administration–free, to support plug–and–play, and to allow spontaneous networking. It is built on a number of core principles [41]:

Discovery: Jini supports the discovery of services, most notably, the lookup service. Discovery works in three distinct ways:

- Discovery requests are sent using UDP Multicast.
- Discovery announcements are also sent using UDP Multicast.
- *Direct* discovery uses TCP unicast to known addresses. Jini lists this as discovery, even though there is no real discovery involved.

Services offer functionality to clients. Services are accessed by Java interfaces.

Communities: A community is formed by the services that are close to each other in a network–specific manner. A community usually stretches as far as UDP Multicast packets are routed.

Lookup Service: A lookup service aggregates information about services in the vicinity, as well as proxies to access the services. Clients can query the lookup service for what they need.

Leasing: To conserve and reclaim resources, Jini consequently employs leasing to manage the life–time of resource allocations. Jini uses a more prominent term for resource reclamation: *self–healing*.

Remote events: Somewhat similar to (local) Java events, Remote events allow the asynchronous notification of applications.

Transactions: Jini provides a model for distributed transactions, including a transaction manager service.

5.3.1 Lookup Service

The lookup service deserves closer attention for how lookups can be performed. Jini clients present the lookup service with a *template* of what they are looking for. This template has three properties:

- A known *Service ID*. The Service ID uniquely identifies a service in time and space. It allows to identify a (well–behaved) service doubtlessly that is registered with multiple lookup services.
- An array of *interfaces* that the service must all support.
- An array of *attributes* that the service must comply with.

The matching algorithm is that each field that is *null* is treated as a wildcard, i.e. all potential services match this field, whereas each field that is not *null* must match exactly.

When searching for a yet unknown service, a client can characterize its needs in terms of Java interfaces and attributes. Attributes conceptually are name–value pairs that can describe a service beyond its interface. Jini defines a set of standard attributes. Their semantics are well–known. The standard attributes are

- *Address:* Provides geographical information.
- *Comment:* Allows to provide a free–form string.
- *Location* within an organization, e.g. floor, or office.
- *Name:* Human–readable name for the service.
- *ServiceInfo:* Provides generic information about the service, e.g. its manufacturer, and model name.
- *ServiceType:* Human–readable description that can describe the service.
- *Status:* Provides information on the current operating state of a service. Status is meant to be sub–classed to service–specific types of status.

Other attributes may be defined by the service vendor. Attributes can be *ServiceControlled*, i.e. the service is able to automatically keep their information up to date, e.g. the status of a printer can automatically be changed to out–of–paper once the printer runs out of paper. Other attributes can not be *ServiceControlled*, such as location or address. These must be administered manually through the service's administration interface.

5.3.2 Resource requirements

Sun's Jini implementation is resource intense. According to [59], Jini on a Java2 VM has a memory footprint of almost 18 MB. This is divided up as follows: The VM requires 0.5 MB, the Java core classes take up some massive 17 MB, and the actual Jini communication package requires 258 kilo bytes. This is obviously too large for small devices, even for most PDAs.

A number of projects, e.g.[59] and [37], therefore tried to reduce memory requirements. The project described in [59] implemented *Mini* (which stands for minimalistic Jini). Mini requires only 26 kilo bytes, and the VM Kaffe that it is based on requires 100 kilo bytes for the VM and 1 MB for the core classes. While this is a significant reduction in size, it is still too much for most small devices. Further, Mini is not compatible with Sun's Jini anymore, as a few protocol changes where introduced.

5.3.3 Discussion

While the attribute mechanism that can be used with the lookup service is flexible, there is no defined set of standard attributes. In other words, the mechanism is only useful within an organization that manages the attributes in a consistent manner.

Further, it is rather awkward to have a standard *administrable* interface in a system with the explicitly declared goal of being administration–free. While it is understandable that some services may need administration, it is astounding that the lookup service itself needs administration, even though it is a central entity of Jini.

Jini's requirements for resources prohibits its usage on very small devices. This is mainly due to the fact that Jini is based on Java. Jini devices are also limited to the Java world, i.e. it is not possible to build a Jini device not running Java.

Similarly, the use of RMI is not mandatory. Yet, implementations using different modes of transportation will not be able to communicate with each other. Therefore, Sun's implementation sets the standard. All other vendors will adhere to this standard to avoid a commercial fiasco.

Due to its non real–time capable basis, Jini is not able to offer quality of service features.

Physical units for measurement values cannot be encoded directly. They would have to be integrated into the type system, leading to an explosion in the number of types.

In terms of dynamic cooperation, the classic approach with Jini is to know the interfaces of interest a–priori. If this is not the case, calls to previously unknown interfaces can still be made via the help of the Java reflection services. Yet, it is hard to imagine a successful machine based dynamic cooperation using only reflection without any further meta–data.

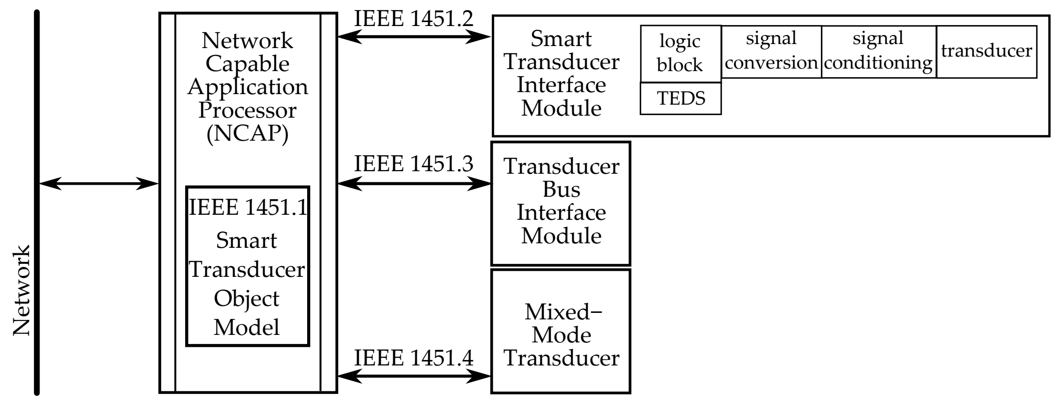

Network
Network Capable Application Processor (NCAP)
IEEE 1451.1 Smart Transducer Object Model
IEEE 1451.2
IEEE 1451.3
IEEE 1451.4
Smart Transducer Interface Module
logic block
TEDS
signal conversion
signal conditioning
transducer
Transducer Bus Interface Module
Mixed–Mode Transducer

Figure 5.6: The IEEE 1451 family of standards and their relations

5.4 IEEE 1451

The IEEE1451 family of standards [60], [61], [62], [63], [86] aim towards standardizing *smart transducer communication*. Figure 5.6 presents the relationships within the standards.

In the IEEE 1451 world, a transducer is defined as

> a device that converts energy from one domain into another [88].

A transducer is typically connected to a microcontroller running a suitable application. The microcontroller is also connected to a network. This setup is then termed *networked transducer*. A *smart transducer* delivers more than just a correct representation of a sensed or controlled physical quantity. [88] lists the following requirements for a smart transducer:

- It provides additional functionality easing the transducer's integration into a networked application environment.
- It is able to perform self–identification to the system.
- It has the capability to provide key information about itself, e.g. its performance parameters. The information is represented in a standardized format, preferably using as little memory as possible.
- On power up, or on query, the transducer identifies itself to the host processor. This enables the automation of diagnostic, configuration, and identification procedures.

More specifically, smart transducers are connected to a microcontroller via a defined digital interface. The processing element, termed *Network Capable Application Processor*, is connected to the network. The individual standards in the family define these building blocks:

- IEEE 1451.1 [60] defines a *common object model* for the components of a networked smart transducer. It provides two interfaces:
  - The interface to the transducer block mapping the details of the transducer hardware implementation to a simple programming model.
  - The interface to the NCAP (*Network Capable Application Processor*) block and ports, which encapsulates the different network protocols behind a small set of communications methods, i.e. IEEE 1451 devices are independent of any specific network technology.

  The object model permits an application to retrieve a transducer's TEDS [120].

- IEEE 1451.2 [61] defines the *Transducer to Microprocessor Communication Protocol*, the *Transducer Electronic Data Sheet* (TEDS), and the *Smart transducer Interface Module* (STIM). Being the most interesting part of IEEE 1451 for this work, the TEDS are described in more detail below. The Transducer to Microprocessor Communication Protocol specifies how a smart transducer can be accessed right down to the pin layout of the connector. The STIM consists if the transducer itself, a signal conditioning stage, the conversion circuitry, the memory chip containing the TEDS, and the necessary logic to implement the digital interface. A STIM may host up to 255 transducers, each of which is defined as a *channel*.

- IEEE 1451.3 [62] introduces so–called *Multidrop Systems*. Multidrop systems connect multiple physically separated transducers to a single NCAP by using a bus. The main feature is self–identification of transducers on the bus.

- IEEE 1451.4 [63] defines so–called *mixed-mode transducers*. A mixed–mode transducer is an analog transducer, connected via a two–wire interface. It has the ability to use its two–wire interface for both digital communication, and analog signal transmission; however, not both at the same time. This enables transducers to transmit their TEDS, and subsequently switch to their analog mode of operation.

- The IEEE P1451.5 group will propose a standard for wireless communication protocols [146].

5.4.1 Transducer Electronic Data Sheet

The TEDS is stored within the STIM itself, using a nonvolatile memory chip. It contains the transducers description in a binary format, requiring only 178 bytes for the mandatory data [87]. Because the TEDS identifies a transducer, and contains all important sensor parameters, plug–and–play of

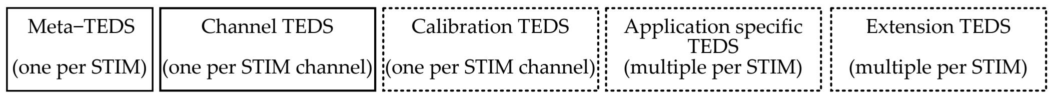


Figure 5.7: Overview of the TEDS structure

transducers is possible, e.g. when upgrading a temperature sensor to a new version with improved accuracy.

The TEDS is structured into five parts (see figure 5.7):

- *Meta–TEDS*: This part contains the information common to all transducers on the STIM, e.g. overall description of the TEDS data structure, worst case STIM timing parameters, etc.
- *Channel TEDS*: There is one Channel TEDS per transducer. It contains information like physical units, upper/lower range limits, warm–up time, etc.
- *Calibration TEDS*: Contains calibration parameters, and the calibration interval. This part is optional.
- *Application–specific TEDS*: Reserved for end–users. This part is optional.
- *Extension TEDS*: Reserved for implementing future and industry extensions. This part is optional.

Meta–TEDS

The Meta–TEDS contains information about the STIM and its interface. The following table[6] lists all defined fields.

Meta TEDS		
Field #	Description	Field Type
	Data Structure related information	
1	Meta–TEDS Length	U32
2	IEEE 1451 Standards Family Working Group Number	U8
3	TEDS Major Version Number	U16
4	Future Extensions Key	U8
5	CHANNEL_ZERO Industry Extensions Key	U8
6	End User's Application Specific TEDS Key	U8
7	Number of Implemented Channels	U8
8	String Language Code	U8
9	Bytes per Character	U8

[6] All TEDS structure tables are reproduced exactly as in [155].

Meta TEDS		
Field #	Description	Field Type
	Timing Related Information	
10	Worst Case Channel Data Model Length	U8
11	Worst Case Channel Data Repetitions	U16
12	Worst Case Channel Update Time	F32
13	Worst Case Channel Write Setup Time	F32
14	Worst Case Channel Read Setup Time	F32
15	Input/Output Response Time	F32
16	Calibration TEDS Write Time	F32
17	Worst Case Data Clock Frequency	U32
18	Worst Case Channel Sampling Period	F32
19	Worst Case Unit Warm Up Time	F32
	Channel grouping related information	
20	Channel Groupings Data Sub–Block Length	U16
21	Number of Channel Groupings	U8
22	Group Name Length	U8
23	Group Name	STRING
24	Group Type	U8
25	Number of Group Members	U8
26	Member Channel Numbers List	array of U8
	Data integrity information	
27	Checksum for Meta–TEDS	U16
	Data Structure related information	
28	Meta–Identification TEDS Length	U32
	Identification related information	
29	Manufacturer's Identification Length	U8
30	Manufacturer's Identification	STRING
31	Model Number Length	U8
32	Model Number	STRING
33	Revision Code Length	U8
34	Revision Code	STRING
35	Serial Number Length	U8
36	Serial Number	STRING
37	Date Code Length	U8
38	Date Code	STRING
39	Product Description Length	U16
40	Product Description	STRING
	Data Integrity Information data sub–block	
41	Checksum for Meta–Identification TEDS	U16

The flat, static structure of a TEDS is clearly visible. Strings are stored in a Pascal–like manner with their length preceding the actual string.

Channel TEDS

For each sensor or actuator, there is one channel, and hence one Channel TEDS. Most sensors measure some physical entity, so there is the possibility to express physical units in a TEDS. Physical units are encoded as products of the SI base units, each having its exponent encoded [155]:

Physical Unit Encoding		
Field #	Description	Field Type
1	ENUMERATION 0: Unit is described by the product of SI base units raised to the powers recorded in fields 2 — 10. 1: Unit is U/U, where U is described by the product of SI base units raised to the powers recorded in fields 2 — 10. 2: Unit is $\log_e(U)$, where U is described by the product of SI base units raised to the powers recorded in fields 2 — 10. 3: Unit is $\log_e(U/U)$, where U is described by the product of SI base units raised to the powers recorded in fields 2 — 10. 4: The associated quantity is digital data and has no unit. Fields 2 — 10 are set to 128. 5 — 255: Reserved.	U8
2	(2 * <exponent of radians>) + 128	U8
3	(2 * <exponent of steradians>) + 128	U8
4	(2 * <exponent of meters>) + 128	U8
5	(2 * <exponent of kilograms>) + 128	U8
6	(2 * <exponent of seconds>) + 128	U8
7	(2 * <exponent of amperes>) + 128	U8
8	(2 * <exponent of kelvins>) + 128	U8
9	(2 * <exponent of moles>) + 128	U8
10	(2 * <exponent of candelas>) + 128	U8

The enumeration field contains more detailed information as to how the physical unit must be interpreted, and if the data has a physical units associated with it at all. There is no further description of *digital data*. Exponents of the SI base unit are encoded as $(2 * exponent) + 128$. A value of 128 in the TEDS therefore eliminates the respective SI base unit from the result. Further, it is possible to specify exponents in increments of 0.5, allowing the declaration of e.g. $\sqrt{\mathrm{Hz}}$. As each exponent is encoded in eight bits, the maximum representable exponent is 63.5, the minimum representable is -64.

The Channel TEDS itself contains information about the channel's constraints, like its lower and upper range limit. The corresponding physical units are encoded as described above. If available, a connection to the corresponding Calibration TEDS is included:

Channel TEDS		
Field #	Description	Field Type
	Data Structure related information	
42	Channel TEDS Length	U32
43	Calibration Key	U8
44	Industry Extension Key	U8
	Transducer related information	
45	Lower Range Limit	F32
46	Upper Range Limit	F32
47	Physical Units	UNITS
48	Unit Type Key	U8
49	Unit Warm Up Time	F32
50	Self Test Key	U8
51	Uncertainty	F32
	Data Converter related information	
52	Channel Data Model	U8
53	Channel Data Model Length	U8
54	Channel Model Significant Bits	U16
55	Channel Data Repetitions	U16
56	Series Increment	F32
57	Series Units	UNITS
58	Channel Update Time	F32
59	Channel Write Setup Time	F32
60	Channel Read Setup Time	F32
61	Data Clock Frequency	F32
62	Channel Sampling Period	F32
63	Timing Correction	F32
64	Trigger Accuracy	F32
	Data Integrity Information	
65	Checksum for Channel TEDS	U16
	Data Structure related information	
66	Channel Identification TEDS Length	U32
	Identification related information	
67	Manufacturer's Identification Length	U8
68	Manufacturer's Identification	STRING
69	Model Number Length	U8
70	Model Number	STRING
71	Revision Code Length	U8
72	Revision Code	STRING
73	Serial Number Length	U8
74	Serial Number	STRING
75	Channel Description Length	U8
76	Channel Description	STRING

Channel TEDS		
Field #	Description	Field Type
	Data Integrity Information	
77	Checksum for Channel Identification TEDS	U16

Calibration TEDS

In [155], the sophisticated representation of calibration parameters is presented. The Calibration TEDS has the familiar flat structure:

Calibration TEDS		
Field #	Description	Field Type
	Data Structure related information	
78	Calibration TEDS Length	U32
	Calibration related information	
79	Last Calibration Date–Time	U32
80	Calibration Interval	U32
81	Number of Correction Input Channels	U8
82	Correction Input Channel List	U8
83	Correction Input Channel–Key List	U8
84	Channel Degree List	U8
85	Number of Segments List	U8
86	Segment Boundary Values Table	F32
87	Segment Offset Values Table	F32
88	Multinomial Coefficients	F32
	Data Integrity Information	
89	Checksum for Calibration TEDS	U16

The layout of the TEDS parts is strict, inflexible, and not extensible. All extensions must be implemented using Extension TEDS, or Application–specific TEDS. Extension TEDS must be approved by the IEEE 1451 body. Any fields that are not used still use up space. As all TEDS are encoded in binary format, the resulting structure is very compact.

5.4.2 Discussion

TEDS are not readable without a special tool. Without such a tool, or the standard at hand, decoding is impossible. Dynamic use is virtually impossible, especially considering extension TEDS — there is no self–describing information encoded in the TEDS itself.

5.5 CANopen

CANopen is a higher level protocol based on the CAN bus. It's main target area is industrial automation. CANopen is looked after by the CAN

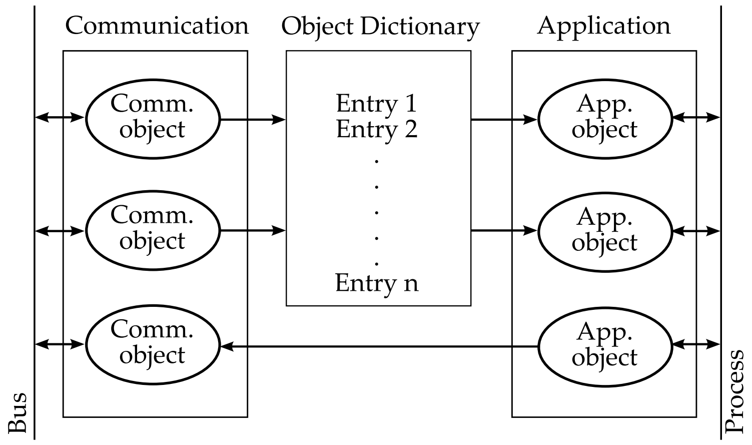


Figure 5.8: CANopen device model

in Automation (CiA) e. V. There are a number of standards that make up CANopen, starting at the physical layer [21], which essentially references the original CAN specification from Bosch [11]. On top, the standards 201–207 [16] define an application level communication model. This communication model most notably takes care of the automatic assignment of CAN identifiers, and introduces network management capabilities for initializing, configuring, starting, stopping, and resetting nodes. DS 301 [17] introduces the CANopen device model with the *object dictionary*. The standard also encompasses the definition of data types, and their representation on the network. The object dictionary is accessed using *Service Data Objects* (SDOs), while real–time communication is handled using *Process Data Objects* (PDOs). Finally, the standard introduces the notion of *device profiles*. A number of device profiles have been standardized by the CiA, each of them having a standard document number in the 400 range, e.g. the *Device Profile for Generic I/O Modules* [20]. The device profile classifies the type of device, and defines what parts of the object dictionary must be present, and what kind of functionality the device provides. As the object dictionary is the central interface of a CANopen device, this is the part covered by the CANopen *Electronic data sheets* (EDS) [19]. The module concept, as described in [19], with a *bus coupler device* that can be extended by *modules*, is very similar to IEEE 1451 (see section 5.4), where the NCAP corresponds to the bus coupler, and the STIM corresponds to the module.

5.5.1 Object Dictionary

According to the device model introduced in [17], a device is generally structured into three parts (see figure 5.8):

- The *communication* function unit provides communication objects and functionality to transport data via the network. The actual CAN controller hardware provides most of this functionality.

- The *object dictionary* is a collection of all data items having an influence on the behavior of the application objects and the communication objects.

- The *application* comprises the functionality of the device with respect to the interaction with the process environment.

What parts of the object dictionary address space are implemented is defined in the device profile. The dictionary is addressed using a 16 bit index; each entry may in turn be indexed by an 8 bit sub–index. The dictionary's overall structure is given in the following table:

Index (hex)	**Object**
`0000`	not used
`0001–009F`	Data Types
`00A0–0FFF`	Reserved
`1000–1FFF`	Communication Profile Area
`2000–5FFF`	Manufacturer Specific Profile Area
`6000–9FFF`	Standardized Device Profile Area
`A000–BFFF`	Standardized Interface Profile Area
`C000–FFFF`	Reserved

The data types section contains the definition of standard data types, and manufacturer or profile specific data types composed of the standard types.

The communication profile area contains communication related parameters, such as bitrate and other low–level timing parameters, the device profile, general node information´and identification, and PDO parameter mappings, which essentially define the real–time communication. PDOs, unlike SDOs, do not incur any protocol overhead, so communication partners must agree on the contents of PDOs beforehand.

Optionally, a device may choose to implement index 0x1021, allowing the storage of the device's description file within the device itself. [17] states three advantages to this scheme:

- The EDS does not need to be distributed using disks.

- Management of different EDS versions for different software versions is less error prone, if both are stored together.

- The complete network settings may be stored in the network, easing analysis and reconfiguration.

It remains a mystery, why this feature was not declared mandatory, given the stated advantages.

The manufacturer specific profile area is completely vendor dependent, only the access methods to the respective functionality makes use of the common access via the object dictionary.

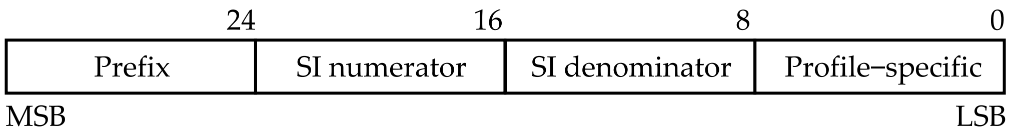


Figure 5.9: Physical unit representation structure

The contents of the standardized device profile area is defined in the individual device profile standard documents, e.g. [20] CANopen DS 401 for general I/O modules offering digital and/or analog I/O.

As CANopen devices usually interact with the environment, measurement values are communicated. For each relevant channel, e.g. for each analog input channel of a generic I/O module according to [20], there is one optional entry in the object dictionary describing the physical unit associated with the respective channel. The physical unit is represented in a 32 bit structure defined in [18] (see figure 5.9). The *prefix* is an 8 bit signed integer encoding the exponent of ten, by which the value is scaled. Valid values range from -18 to +18, representing factors from atto (10^{-18}) to exa (10^{18}). Both *SI numerator* and *SI denominator* are essentially an index to a lookup table mapping these entries to physical units. The lookup tables not only include the SI base units, but also a vast number of derived units. While this scheme is not as flexible as e.g. IEEE 1451's scheme, it results in a very compact representation. The complete lookup table is defined in [18].

5.5.2 Electronic Data Sheets

Software tools play an important role in managing the complexity of CANopen networks. These software tools greatly benefit from having electronic descriptions of the connected devices, as these eliminate the cumbersome and error–prone task of manual input [19]. The *Electronic Data Sheet* (EDS) serves as a template for a certain type of device A from vendor B. The *Device Configuration File* (DCF) contains information about a specific configuration of a device, i.e. it is an instantiation of an EDS. The DCF actually contains values for objects described in the EDS, e.g. for the bitrate, or the node–ID of a device. The EDS is supplied by the vendor, together with the device. Usually, the EDS is shipped on a disc accompanying the device. Vendors may optionally choose to store the EDS on the device itself (see section 5.5.1).

The EDS is a text file. It is structured into *sections*. A section is started with the section's name in square brackets. Within a section, name–value–pairs form the section's entries:

```
[section name]
name1=value1
name2=value2
```

There are a number of restrictions to formatting the EDS, e.g. lines may be up to 255 characters in length. Of course, this eases parsing, yet the format is extremely intolerable even trivial *noise* characters like spaces.

Besides the description of the object dictionary, the EDS contains information about the file itself (e.g. the file's name, its version, and its last modification time), and some general device information (e.g. vendor information, product name, and supported baudrates).

The description of the object dictionary defines which objects of the dictionary are supported. This list is structured into mandatory, optional, and manufacturer specific objects. Further, each object with each supported sub–index is described in a separate section containing (among other fields) the object's name, its data type, its range limits, and its access type.

Comments may be included in the same format as other information by defining a section called *Comments*:

```
[Comments]
Lines=2
Line1=This is a two
Line2=line comment.
```

A DCF is constructed by enhancing an EDS by all configuration values for a specific, configured device. The DCF is structured exactly like the EDS, but there are some additional entries, most notably *ParameterValue* entries for each configured object in the dictionary description.

5.6 LIN

LIN is a communication concept developed in the automotive domain. The LIN Consortium started out in 1998 as an initiative of Audi, BMW, DaimlerChrysler, Volvo, Volkswagen, VCT, and Motorola. LIN's objective is to provide a standard for low–cost communication in vehicles, usually as a cheaper sub–bus to CAN [123].

The LIN 2.0 specification [25] includes the transmission protocol, the transmission medium, the interface between development tools, and the interface for software programming.

Communication is handled in a publish/subscribe fashion. Messages have a unique identifier defining their contents. In a LIN network, there can only be one publisher for a given identifier, but there can be an arbitrary number of subscribers. The bus itself follows a simple master/slave concept. This simple scheme includes a self–synchronization feature. On the physical layer, a single–wire implementation with speeds up to 20 kbit/s is used.

A signal is the basic unit of communication. Either a signal is a scalar value between one and 16 bits long, or it is a byte array between one and eight bytes in length. Each signal is transported in the data field of a frame. Several signals can be packaged into one frame. Actual communication takes place in two steps:

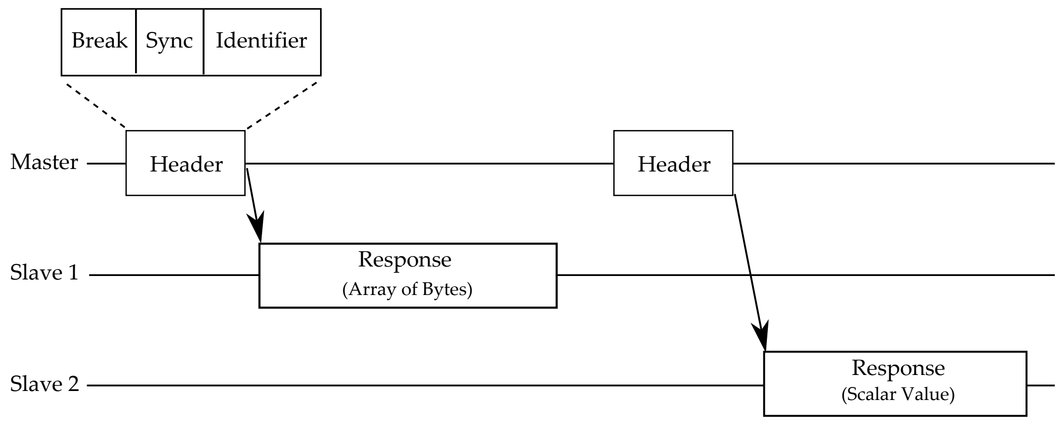


Figure 5.10: Master/Slave communication pattern

- The master node sends the frame's header,
- and the corresponding publishing slave the response with the actual contents. The header consists of a break and a sync pattern, followed by an identifier essentially defining which slave node will be responding (see figure 5.10).

Plug–and–play of pre–manufactured nodes is one of the major goals. To achieve this, each LIN node is accompanied by a *node capability file* (NCF) [24].

Development of a LIN cluster is partitioned into three phases (see figure 5.11):

Design: During the design phase, individual node capability files are combined to create the *LIN Description File* (LDF). This process is called *System Definition*. For nodes to be newly created, NCFs can be created either manually or via the help of a development tool. From the LDF, communication schedules, and low–level drivers for all nodes in the cluster can be generated (*System Generation*).

Debugging and Node Emulation: Based on the LDF, the LIN cluster can be emulated and debugged.

System Assembly: In the system assembly phase, the final system is assembled physically, and put to service.

A node's description contains six main parts (see figure 5.12):

1. The node's name.
2. Properties that specify the general compatibility with the cluster, e.g. the supported protocol version, the supported bitrates, and the LIN product identification. The LIN identification is composed of a 16 bit supplier ID assigned by the LIN Consortium, a 16 bit function ID

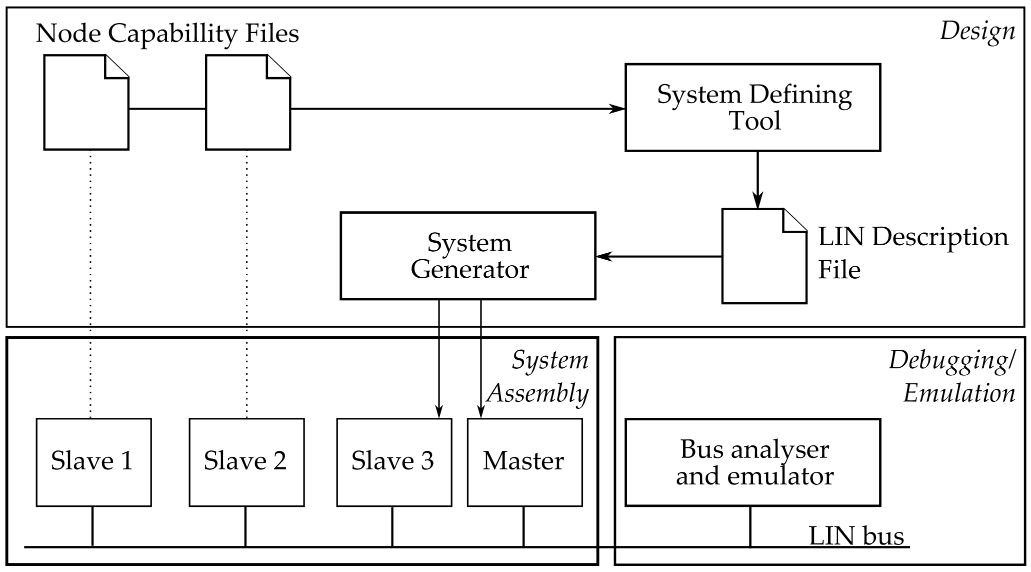


Figure 5.11: Development phases of a LIN cluster

defining the type of device, and a 8 bit variant ID, essentially encoding a version number for otherwise functionally unaltered devices.

3. Diagnostic properties, e.g. the minimum time between a master request frame and the following slave response frame.

4. Frame definition: all frames that are published or subscribed by the node are declared. The declaration includes the name of the frame, its direction, the message ID to be used, the length of the frame in bytes. Optionally, the minimum period and the maximum period can be specified. The frame's declaration closes with the associated signals' definition. Each signal has a name, and a number of properties associated with it:

   Init value specifies the value used from power on until first set by the publisher.

   Size specifies the signal's size in bits.

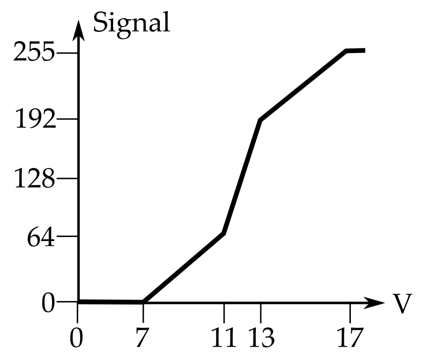
   Offset specifies the position within the frame (number of bits).

   Encoding specifies the signal's representation. The representation may be given as a combination of the four choices logical value, physical value, BCD value, or ASCII value. Declarations of physical values include a valid value range (minimum and maximum), a scaling factor, and an offset. Optionally, this can be accompanied by a textual description, mostly to document the value's physical unit. An example is given in figure 5.13[7]

[7]The example is taken from [23], page 14.

```
Node name
├General
│ ├Protocol version
│ ├Bitrates
│ └...
├Diagnostic
│ ├P2_min
│ └...
├Frame
│ ├Name
│ ├Direction
│ ├Message ID
│ ├Length
│ ├Minimum Period
│ ├Maximum Period
│ └Signal
│   ├Name
│   └Properties
│     ├Init value
│     ├Size
│     ├Offset
│     └Encoding
│       ├Name
│       ├Logical value
│       │ ├Signal value
│       │ └Textual information
│       ├Physical value
│       │ ├Minimum
│       │ ├Maximum
│       │ ├Scale
│       │ ├Offset
│       │ └Textual Information
│       ├BCD value
│       └ASCII value
└Status management
  └Error Response
```

Figure 5.12: NCF structural tree



```
V_battery {
    logical_value, 0, "under voltage";
    physical_value, 1, 63, 0.0625, 7.0, "Volt";
    physical_value, 64, 191, 0.0104, 11.0, "Volt";
    physical_value, 192, 253, 0.0625, 13.0, "Volt";
    logical_value, 254, "over voltage";
    logical_value, 255, "invalid";
}
```

Figure 5.13: LIN signal definition

5. Status management: This section specifies which published signals the master node should monitor to detect if the slave is operating as expected.

6. The free text section allows the inclusion of any help text, or more detailed, user–readable description.

The description file is a text file. The syntax is simple and rather similar to C. Properties are assigned using `name = value;` pairs. Subelements are grouped together using curly braces, equivalent to blocks in C.

5.6.1 Discussion

The LIN Configuration Language lacks the possibility to formally specify physical units in a machine–readable manner. Currently, the language does not allow the specification of non–functional properties.

Tool manufacturers must write their own parser. Though the syntax is simple, this requires extra work (as opposed to using readily available XML parsers). This is also the reason, why the language is harder to extend. Legacy tools will not be able to successfully parse an enhanced version of the configuration language. In contrast, with XML, the tool can simply skip over unknown tags, enabling it to read even an extended version.

The NCFs are not stored in the nodes themselves, therefore one must keep track of two items in parallel: the node, and its description file.

LIN clusters are a static setup. They cannot be dynamically extended. There is no dynamic discovery possible.

Nodes are not addressable individually.

5.7 TTP/A

TTP/A [49] is a low–cost communication protocol that follows the time–triggered paradigm [44]. It was mainly developed at the Vienna Technical University as part of the Time Triggered Architecture (TTA). The purpose of TTP/A is the interconnection of smart transducers [84], preferably using only a single wire [46], or some other means of low–cost communication infrastructure. Despite lowest cost, hard real–time communication facilities are provided.

Smart transducers[8] provide the infrastructure between sensing elements and a digital field bus [107]. A smart transducer consists of either a sensing or actuation element, combined with a microcontroller that in turn contains a processing core, memory, and a network controller in one unit. With MEMS (micro electro mechanical systems), even the sensing elements can be integrated on one die together with the microcontroller [82]. Such tight integration significantly lowers cost, as well as noise problems on analog signals, that are now restricted within a single chip. Using the processing

[8]This term was first introduced in 1982 in [80].

core, sensor data is pre–processed. The smart transducer can perform self–calibration and self–testing. Further, spurious inputs are filtered out. Even in case of a network failure, a node may potentially continue to retain some degree of local control of its environment.

TTP/A is designed to provide all services essential for the successful employment of such smart transducers, including timely communication, remote on-line diagnostics, sensor parameterization, and plug–and–play capabilities [83]. Further, TTP/A harmonizes with the fault–tolerant bus system TTP/C. TTP/A is designed to form a cost–effective sub–bus to TTP/C. TTP/A can handle up to 250 slave nodes per cluster [43].

In [46], five key requirements to success for field buses are given:

Cost: Cost obviously must be kept as low as possible. This includes an adequate plug–and–play scheme to reduce system set–up costs.

Bandwidth: The bandwidth provided should be as high as possible to serve as many applications as possible.

Real–time capabilities: For real–time applications, a timely communication facility is a prerequisite.

Error diagnostics: Even for non safety–critical applications, an on–line diagnostic feature is relevant to keep maintenance cost low.

Extensibility: The structure of a system often is not constant during the system's lifetime. It must be possible to connect, detect and correctly integrate new nodes.

As the name already suggests, TTP/A nodes use a TDMA scheme for colision–free media access. Obviously, TDMA schemes require a common notion of time in all connected systems [83]. Using TDMA also ensures message ordering and a low maximum jitter. Architecturally, TTP/A is a master/slave scheme. Communication is organized into rounds. The master node initiates these communication rounds and provides synchronization for the smart transducers, or slave nodes, for that matter. Each communication round is started with a so–called fireworks byte. This byte determines the type of the round and is a reference signal for clock synchronization. The communication pattern within each round is predefined via the RODL (ROund Descriptor List). The RODL is stored in a distributed fashion across all nodes. TTP/A supports eight different fireworks bytes. Their redundant encoding allows for error checking and startup synchronization [54]. There are two types of rounds: Multipartner rounds and Master/Slave rounds. The former is used for the transmission of real–time service data, the latter is used to access and configure single nodes for diagnostics and configuration.

More specifically, multi partner rounds consist of a configuration dependent number of slots. For each slot, a different sender (and receiver) is assigned (see figure 5.14). The complete round is defined in the RODL data

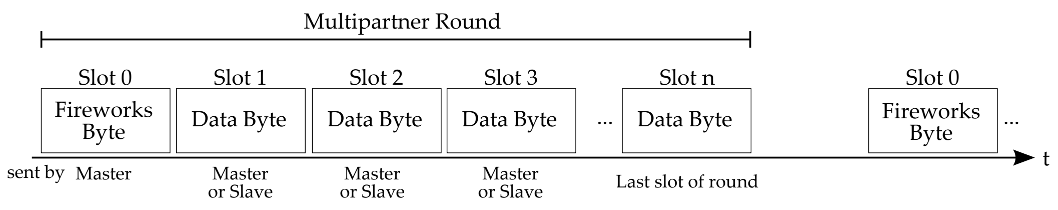


Figure 5.14: TTP/A Multipartner Round

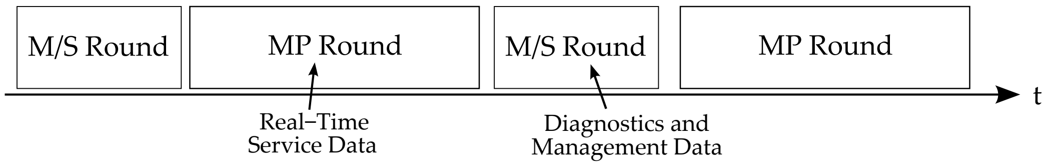


Figure 5.15: Interleaved TTP/A schedule

structure. Obviously, the RODL must be configured in all slave nodes prior to the execution of the corresponding multipartner round. In contrast, Master/Slave rounds have a fixed layout that establishes a connection between the master and a particular slave for accessing a node's *interface file system* (IFS). The IFS e.g. can be used to access the node's RODL information. In a Master/Slave round, the master addresses a data record in the IFS and specifies an action (reading, writing, or executing). Master/Slave rounds are scheduled periodically between multipartner rounds (see figure 5.15). Thus, maintenance and monitoring is enabled during normal system operation without a probe effect. Addressing of nodes in a master/slave round is done using a short logical node ID. This ID is only valid in the cluster. It is either assigned at compile time or on–line when the node is integrated into the cluster. The on–line assignment of this ID is called *baptizing*. Baptizing enables true plug–and–play in TTP/A. The basis for baptizing a node is the node's physical name, which is stored in the node's *documentation file* (see section 5.7.1).

5.7.1 Interface File System

The *Interface File System* (IFS) [84] is a distributed file system shared by all nodes of a TTP/A cluster. It supplies a structured name space for the data–elements that are exchanged among the nodes of a cluster. It serves as a basis for all higher–level protocols that assign meaning to the contents of IFS file records. The assignment of meaning is done in TTP/A node descriptions, termed *Smart Transducer Descriptions* (STDs). The address of an IFS record is composed of the following three parts:

<file name><node name><record number>

A record is the smallest addressable unit in the IFS. All records have a constant length of 32 bits [49]. This simple structure is beneficial when implementing TTP/A on very small nodes.

There are three operations on the IFS records: *read*, *write*, and *execute*. Four different file types are distinguished:

Documentation files: These read–only files contain documentation about a node. Two records of a node's documentation file are reserved for the node's unique physical name that is the basis for TTP/A's plug–and–play features.

Input–output files are used to store observations and parameters.

RODL files obviously contain the round descriptions. The RODL file name is also the round name. The master node can initiate the execution of a specific RODL by naming it within the fireworks frame. RODL records have the following structure:

<round position><op–code><file–name><length><file record address>

Such a record tells the node, at what *position* in the round an action (*read*, *write*, or *execute*) is required. The *op–code* field specifies which of these actions is required. *File–name* identifies the file that is involved in the action. The *length* field specifies the length of the data frame, and the *file record address* specifies which record is involved in case of a read or write operation.

Since individual RODL files can be addressed and manipulated just like any other files, programming of a new RODL can be performed using the standard file access methods without the need to introduce any special concepts and mechanisms.

Special command files contain executable program modules that can be executed. The transmitted data of a round forms the input parameters for such an invocation.

Single reads and writes from and to the IFS are atomic. If a higher level of atomicity is needed, the user has to design an own concurrency control protocol.

TTP/A gateways provide access to a TTP/A system via three distinct interfaces views:

- The *Real–time service interface (RS)* provides time sensitive information like sensor values.
- The *Diagnostic and management interface (DM)* enables monitoring and administration functionality. This is even possible during normal system operation, if rounds are scheduled appropriately. The master can e.g. access and modify a nodes internal diagnostic data or calibration values.

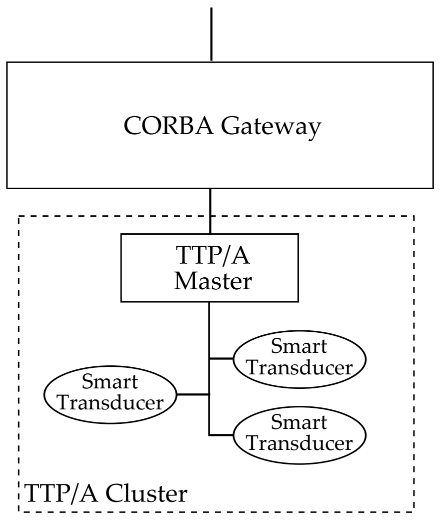


Figure 5.16: Integration of TTP/A with CORBA

- The *Configuration and planning interface (CP)* is used to initialize a network. It can also be used to automatically detect nodes on the bus, to *baptize* nodes, and to program the RODL files on the slave nodes.

5.7.2 Description mechanisms

Description is used twofold in TTP/A: *Smart Transducer Descriptions* (STDs) describe smart transducers concerning their static properties. *Cluster Configuration Descriptions* (CCDs) are concerned with configuration information about a cluster of smart transducers. Both description types are written in XML. The structure of these descriptions is defined with XML schemas. Both STDs and CCDs are presented in the following sections. This presentation is mainly based on [107], where all elements of the descriptions are presented in detail.

STDs

Via the smart transducer interface specification [103], CORBA applications gain access to smart transducers. The TTP/A clusters themselves do not need to be changed. The TTP/A master node is connected to the CORBA gateway using any convenient interface (see figure 5.16), e.g. RS232 [85]. Originally, CORBA was not concerned with hard real–time requirements. In [91], an approach towards extending CORBA for hard real–time systems is presented.

Communication within a TTP/A cluster is without redundant information such as the name of an observation that is transmitted in a specific time slot, as this information is intrinsically known via the RODL. At the CORBA interface, the complete information about an observation (name, time of observation, and value) is made available.

Since smart transducers have only very limited storage capacity, the metadata describing the semantics of the smart transducer is stored externally to the smart transducer. It is accessible through a register service. The metadata is composed of a description of the smart transducer in plain English, and an XML fragment describing the RODLs.

The register service takes care of linking a smart transducer to its description via the smart transducer's physical name. In [103], the subdivision of physical names into a *series number* defining the type (and manufacturer) of device, and a *serial number* is proposed. Further, a yellow–pages–like service is suggested.

Static STDs describe a type of node. Further, STDs may be used to represent individual nodes. These STDs can be seen as instances of a static STD. They carry the dynamic properties of a node.

The STD's structure is presented in figure 5.17. All elements termed *DMV* are actually three distinct elements grouped together because of space considerations. The three distinct elements are named *Documentation*, *MetaInformation*, and *View*, short *DMV*. These general description elements are used to supplement the document. The *Documentation* element is optional. It may contain an arbitrary combination of text and sub–elements. This appears to be mainly useful for additional user–readable documentation. The *MetaInformation* element is supposed to hold additional information for e.g. the presentation of an element in a tool. Finally, the *View* element allows locally specifying stylesheets for the presentation of an element.

STDs are structured into four major blocks. The first two blocks mostly contain information for identifying and classifying a node. The *processor block* is for documentation purposes. It contains detailed information about the processor used, e.g. the processor vendor's name, the processor's name, and a link to the processor's data sheet. Further, details about the processor's clock, like its frequency, type, source, and even clock drift information is available. The *node block* contains properties important for node identification, like node vendor's name, device name, device version and revision. In case of an STD describing a specific node, the unique node identifier is contained. If the STD is part of a CCD, the short local node identifier is also included. The *protocol block* consists of low–level information necessary for the successful integration of a node into a network. It contains information about the supported protocol version, and baud rates. Further, the type of UART implementation (software or hardware), and even the UART's jitter are given. If the STD is part of a CCD, the RODLs relevant for the current node are given. The description of the IFS file layout concludes the protocol block. It describes what parts of the IFS are available, with what type of memory individual parts are implemented (ROM, EEPROM, or RAM), and how these parts can be accessed. The concluding *node service block* contains a detailed description of a node's capabilities. For each capability, the general description structure is similar to a method signature: it consists of the capability's name, and its input and output parameters. For TTP/A, this en-

```
SmartTransducerDescription
├ProcessorBlock
│├ProcessorVendor
│├...
│└DMV
├NodeBlock
│├NodeVendor
│├...
│└DMV
├ProtocolBlock
│├ProtocolVersion
│├...
│├RODLBlock
││├RODLFormatVersion
││├rodl
│││├node
││││├slot
│││││├operationCode
│││││├fileName
│││││├recordNumber
│││││├recordAlignment
│││││├messageLength
│││││└valid
││││└DMV
│││└DMV
││└DMV
│├IFSFileLayoutBlock
││├IFSFile
│││├fileName
│││├fileLength
│││├fileStorage
│││└DMV
││└DMV
│└DMV
├NodeServiceBlock
│├LocalTypes
│├NodeService {ServiceType}
││├ServiceName
││├fileName
││├InputParameters
│││├Parameter
││││├DataType
││││├PhysicalUnit
││││├Constraints
│││││├ValueRange
│││││├Precision
│││││├Resolution
│││││├MART
│││││├MINT
│││││└DMV
││││└DMV
│││└DMV
││├ConfigurationParameters
│││└...
││├OutputParameters
│││└...
││├InternalStateInformation
│││└...
││└DMV
│└DMV
└DMV
```

Figure 5.17: STD structural tree

compasses the IFS entries relevant for the particular capability. Individual parameters are typed using the XML schema type system. This type system can be enhanced by specifying *local types*. Each *node service* structure describes one distinct capability of a smart transducer. For services, a distinction is made between *fixed* services that cannot be changed, *default* services that can be changed at run–time, and *application* services that are created by the user application. The *ServiceType* defines, of which type a specific service is. The *ServiceName* is used to assign meaning to a specific service. For each of the blocks *input*, *output*, *configuration*, and *internal state information*, parameters can be specified using identical structures. While the first two sections are rather obvious, the latter two require a short explanation: *configuration* parameters are used to configure a service to a user's need, usually on system start up, e.g. setting a threshold value. *Internal state information* provides information beyond a traditional method signature and specifies, what internal parameters are accessible via the IFS, e.g. for diagnostic purposes. For each parameter in these four sections, its data type is specified in terms of XML schema types — including local enhancements. Further, if the parameter represents a measurement value, its physical unit is given as a string. To complete the parameter information, constraint information on the parameter is given, e.g. value range, precision, and resolution.

CCDs

The main focus of TTP/A's description framework is support for configuration management [107] — the reason being the multitude of configuration parameters in a modern field bus network, and the resulting complexity of managing such systems. CCDs contain information about all relevant system properties — static and dynamic — of a particular cluster, e.g. communication baud rate, communication schedules, and a list of nodes in the cluster.

The configuration information can be sub–classified into the following categories:

Communication schedules: Time–triggered protocols need an a priori definition of static communication schedules. In TTP/A, this information is stored in a distributed fashion. Each node holds that part of the overall configuration that involves itself.

Node parameterization information consists of parameters specifying working modes, calibration information, and threshold values.

Local node application code can also be seen as part of the configuration information, if it is dynamically replaceable.

A CCD is an XML file containing this information. With the help of a software tool capable of accessing the cluster, it is possible to (re–) configure a cluster.

```
ClusterConfigurationDescription
├ClusterDescriptionMetaBlock
│ ├ClusterDescriptionFormatVersion
│ ├ClusterDescriptionFile
│ │ ├DFilePosition
│ │ ├FileSize
│ │ ├LastModified
│ │ └DMV
│ ├ClusterIdentificationID
│ ├MaintainerInformation
│ │ └DMV
│ └DMV
├ROSEBlock
│ ├RODLref
│ │ └DMV
│ └DMV
├ConfigurationBlock
│ ├ClusterRODLBlock
│ │ ├rodl
│ │ │ ├node
│ │ │ │ ├slot
│ │ │ │ │ ├operationCode
│ │ │ │ │ ├fileName
│ │ │ │ │ ├recordNumber
│ │ │ │ │ ├recordAlignment
│ │ │ │ │ ├messageLength
│ │ │ │ │ └valid
│ │ │ │ └DMV
│ │ │ └DMV
│ │ └DMV
│ ├BaudRate
│ ├MaxSigRunTime
│ ├MinSigRunTime
│ ├BusInterface
│ ├UARTFormat
│ ├InterframeGapLength
│ └DMV
├ClusterNodeBlock
│ ├STD / STDref
│ └DMV
├ViewDisciplines
│ ├ViewDefinition
│ │ ├Viewer
│ │ ├Accesses
│ │ └DMV
│ └DMV
└DMV
```

Figure 5.18: CCD structural tree

The CCD's structure is presented in figure 5.18. As with the discussion of the STD's structure above, the elements termed *DMV* stand for the three distinct elements *Description*, *MetaInformation*, and *View*.

The first of the five major parts, called *ClusterDescriptionMetaBlock* is concerned with the description document itself. Notably, it contains the version number of the description format in use (*ClusterDescriptionFormatVersion*), and the cluster's identification number (*ClusterIdentificationID*), which is used to access a specific cluster from CORBA. A *Round Sequence* (ROSE) defines the order of the executed rounds. It is defined in the *ROSEBlock*. Individual rounds are described as references to the respective RODLs that are defined in the following block. The *ConfigurationBlock* contains the definition of all RODLs, as well as all necessary low–level parameters for communication, like the cluster's baud rate. All parameters in the *ConfigurationBlock* are valid for the whole cluster. The *ClusterNodeBlock* contains a list of all nodes in the cluster. The list entries can either be references to STDs, or instances of STDs containing a snapshot of the current configuration state of each node. Finally, the *ViewDisciplines* block provides the definition of views of the system. Each view consists of a viewer and a list of elements that are shown in this view.

One of the tools making use of CCDs is the *Configuration and Planning Tool* presented in [108]. It uses CCDs to visualize and configure clusters semi–automatically. CCDs can be fairly large in size, [108] states 124 kbytes for a demonstration setup composed of 8 smart transducers. Unfortunately, it is unclear, whether the individual transducers' STDs are included in these 124 kbytes, or not.

Application descriptions

In [48] and [47], a description scheme for distributed applications is presented. Distributed applications are composed of individual jobs. These jobs run on nodes in a cluster. Jobs in an application interact with each other via the real–time interface. This models the functional and data flow linking between jobs. In addition, configuration dependent properties — such as the sample rate of a sensor — and end–to–end requirements — such as end–to–end signal delays in control loops — need to be specified. These end–to–end requirements are expressed as dependencies. Three types of dependencies exist:

Connection dependencies represent data flow. Each dependency is directed from its source to its target. Inputs may have only one connection to another output, whereas outputs may feed an arbitrary number of inputs.

Causal dependencies represent the subsequent execution of jobs.

Phase dependencies represent non–causal time–related dependencies.

As stated in [48], these properties are mapped to XML descriptions in a simple and straight–forward manner. Application descriptions are currently not based on XML schema.

[47] introduces the *TTP/A scheduler* to facilitate the automatic creation of (static) communication schedules for TTP/A clusters using these application descriptions. The generated communication schedules are verified. The verification process also uses information beyond the contents of the application description, e.g. the intended communication bandwidth. Any unused bandwidth is allocated for diagnostic and management purposes.

To produce the communication schedules, the TTP/A scheduler transforms the application description into a graph representing all jobs and their dependencies. This graph is called *Precedence Graph*. Its vertices represent jobs. Directed edges stand for causal dependencies, i.e. direction of data flow and precedence of that dependency. Undirected edges represent phase dependencies. Phase dependencies are transient, which simplifies the graph. The scheduling algorithm will then produce valid RODLs if

- there are no loops in the graph, i.e. a job must not depend on results from itself (Control loops are still possible, as the feedback path is external to the model.),
- only one directed causal edge exists between a pair of jobs, and
- no phase dependency is given, if already a causal path exists between two jobs, i.e. mutually dependent jobs must not be in phase.

5.7.3 Discussion

Software development for TTP/A slave nodes is independent of the descriptions. The application developer is conceptually offered a shared memory interface to the communication layer. On the AVR platform, a hardware abstraction layer and the TTP/A execution environment is available (Details for TTP/A on the AVR architecture are given in [45]).

CCDs contain valuable configuration information in an automatically processable way. STDs in contrast, are currently more suitable for documentation purposes: Data types of parameters are defined in terms of XML schema types. While this provides valuable information if data is exchanged in XML, it does not specify how these data types are encoded in binary form on the TTP/A bus. Further, physical units of measurement values are encoded as strings. There is no further meta information, or even just a set of recommended designations available, so it is virtually impossible to process this information automatically. It is also not easily possible to represent arbitrary scalings and offsets of values.

5.8 Analysis and Comparison

5.8.1 Component orientation

LIN, IEEE 1451, CANopen, TTP/A, Jini, and Web services can all be thought of as dealing with components in the sense defined in section 2.1. For LIN, IEEE 1451, CANopen and TTP/A, components are actually hardware components. Here, the terms node and component may be used interchangeably. For Jini and Web services, components are software entities. Possibly, each component may be running on individual nodes, but this is neither a prerequisite, nor the common case. All of these technologies make no assumptions about any other dependencies between their components other than the interface specified. With Web services, this interface is very flexible. For the field bus technologies LIN, CANopen, TTP/A, and essentially IEEE 1451, the technical, operational interface of the components is rather fixed, only a number of parameters will influence the component's behavior, and the kind of operation actually carried out. Of course, the semantics behind the operations always depend on the type of nodes, and the application they are used in.

UPnP might look like a real component–oriented approach, even more than Jini. Yet, UPnP is seldom capable of providing the complete interface necessary for a component. Consider the popular streaming media servers. Most of the current devices provide UPnP access. Yet, UPnP is only used for discovery, remote control, and setup purposes. Media transport is done using different protocols, e.g. RTP streams [118], [111], [112], [113], [114], [115], [116], [110].

5.8.2 Component life cycle support

This section discusses the presented technologie's support for the component life cycle as introduced in sectio 2.1.1.

None of the technologies presented in this chapter mentions *component design* as part of their user support. Necessarily, vendors support the component designer with tool chains, yet, these are proprietary. Data exchange among tools of different vendors is only given by the means of component descriptions, if available, e.g. in IEEE 1451, CANopen, and TTP/A. Notably, LIN specifies the data format used to interchange information among tools.

Component implementation is inherently supported by Jini, as the component's interface is defined using Java interfaces. Obviously, this support remains of a rather basic nature. It currently goes as far as having an IDE automatically enhance a class with empty methods that will implement interfaces. The UPnP standard does not address implementation issues at all. The standard explicitly states that UPnP is a network protocol, not an API. The UPnP stacks available on the market offer implementation support at varying degrees, i.e. their API is more or less comfortable to use. For IEEE 1451, and CANopen, very little information is publicly available in terms of programming support. Vendors usually provide a protocol stack with a defined, proprietary API. In the case of IEEE 1451, pure hardware solutions for the smart transducers themselves may be favorable, completely eliminating the need for software development support. With LIN, the API is defined in the standard. For TTP/A, protocol stacks are available for a wide range of platforms. To implement Web services, there are a number of frameworks available. They mostly differ in terms of comfort offered to the developer. To date, the most comfortable tool certainly is Visual Studio 2005, which will turn a .NET component into a Web service by simply adding a `[Webservice]` annotation to the component's class, and a `[WebMethod]` annotation to each operation that will constitute to the Web services interface. Even the necessary WSDL descriptions are produced automatically from the class' interface.

Component integration is the phase that almost all presented technologies are most focused on. Both UPnP and Jini provide means to discover services at run–time. Services with known interfaces may be dynamically made use of. Usually, service usage is based on user interaction, e.g. a user of a digital camera may want to print a picture. The user can then either be presented a list of available devices, or he simply invokes a function like *print on any available printer*. There are no widely spread planning and simulation tools available for either UPnP or Jini. In terms of compatibility, UPnP devices must adhere to at least one of the device profiles defined by the UPnP forum. No checking beyond the supported device profile at run–time is possible. Jini ensures compatibility via the service's interface. For both, there are no means to check a system for completeness in terms of available services, devices. They are designed to be inherently open and dynamic networking technologies. In a LIN tool chain, the *System Defining Tool* is used to set up a cluster. Individual nodes in the cluster are meant to be replaceable, yet the cluster itself is not extensible without a complete reconfiguration. Compatibility checking across nodes is not covered by the standard. Smart transducers according to IEEE 1451 are meant to be replaceable parts in fixed configurations. As they are not directly connected to a network, the integration phase for smart transducers is part of the design phase of the IEEE1451.1 device. The TEDS ensures that the IEEE1451.1 device can cope with slightly differing smart transducers. CANopen does not provide any standardized tools or methods for integrating a number of nodes into a system. TTP/A provides a tool for cluster planning, configuration and simulation. Network schedules are generated automatically from application descriptions. Verification tools, including bandwidth usage, are provided. It would be possible to verify the correct data types and data ranges across multiple nodes, however, verifying the correct physical unit remains problematic due to their encoding as strings in the STDs. For Web services, the quality of integration support varies with the tools used. UDDI strives to provide a lookup service for finding the right service.

During *Component usage*, the components usually work invisibly in the system. On some occasions, it may be desirable to retrieve information about a component for reference. UPnP devices not only provide their (XML) descriptions on–line, but they also feature an optional HTML page describing the device in a convenient format. IEEE1451 TEDS may be retrieved using special tools. For CANopen, the ability to store the device's description in the device itself is only optional. Otherwise, the description file needs to be retrieved off–line, as with LIN. TTP/A STDs are stored on a web server; their URLs can be retrieved from the devices. For Web services, some servers are able to deliver a web service's WSDL description. Jini does not provide any on–line documentation facilities beyond the attribute mechanism at the lookup service. Attributes are not retrievable from the services themselves, however, unless they provide an appropriate, yet non–standardized interface.

In section 2.1.1, two important steps of the *maintenance* phase are mentioned: identification and retrieval of the configuration information. It is of great value, if the system can provide the maintenance personnel with this information. Keeping configuration information only externally poses the risk of loosing the data, and the risk of having outdated information. UPnP devices and Jini services are identified using a UUID. Neither technology provides a default interface for accessing configuration information. With LIN, individual devices cannot be identified, and their configuration is only accessible externally in the LDF. IEEE 1451.2 smart transducers support both identification (through several entries in the Meta TEDS), and retrieval of configuration information (via the respective TEDS entries). CANopen provides similar, yet less powerful facilities. It is important not to loose the off–line DCF, otherwise the information is incomplete and those parts retrievable from the device cannot be interpreted correctly. Of course, CANopen devices able to serve their own EDS provide an exemption to this problem. With TTP/A, most necessary meta information is kept off–line in the STDs and application descriptions. Web services obviously are easily identifiable via their URL. Their configuration is usually off limits outside the server machine.

Disposal is not addressed by any of the approaches presented in this chapter.

5.8.3 Descriptive features

Jini lacks any special description facilities beyond the attribute facility used with the lookup service. The achievable level of detail (section 2.3.3) is up to the user, but since there is no standardized (section 2.3.6) default set of attributes, the automated processability (section 2.3.2) is limited. On the other hand, the set of attributes is easily extensible (section 2.3.5). Comments in the source files may adhere to the Javadoc conventions [127] and thus include the operations signatures (section 2.3.7). They can then be transformed into human readable HTML documentation. Javadoc comments are not available in the compiled byte code, i.e. they are external to the component.

UPnP description documents are (possibly large) XML files. They can only be retrieved in one piece at run–time from the respective devices. Device and service descriptions are machine checkable against their respective template languages. Thus, UPnP descriptions are written in a formal language (sections 2.3.1 and 2.3.2). They can be checked for completeness in terms of the underlying standards document (sections 2.3.4 and 2.3.6). The descriptions do not concern themselves with low level communication details, as they are setup automatic (section 2.3.3). The description of service operations is essentially a description of the operation's signature (section 2.3.7). It contains its name, its parameters, their names, types and direction. Errors are often reported using a parameter typed as integer. Valid ranges are defined not in the description documents, but in the respective UPnP service standards. The semantics behind all operations, and in fact, the available set of operations, is also defined in the standards documents. As operations cannot be sufficiently described in the service descriptions, user defined services (i.e. services not approved and published by the UPnP Forum) are of very limited use (sections 2.3.5 and 2.3.6). True dynamic usage is not possible.

Web service descriptions in WSDL (a formal language, sections 2.3.1 and 2.3.2) provide the operations' signatures, i.e. its name, and parameters including their details (section 2.3.7). Any semantics can only be encoded in the name of the operation. Approaches like SAWSDL try to enhance WSDL to include semantic information (section 2.3.5).

LIN descriptions are much more detailed in comparison: they provide meta information about the node, and a detailed specification of all in– and outgoing messages (section 2.3.3). The message definition covers the complete parameter range. Each value can either be assigned to carry a logical value, or a physical value. For physical values, the accompanying units are encoded as strings, providing maximum flexibility, yet limited processing possibilities (section 2.3.2). The possibility to describe the combination of logical and physical values within a single parameter is unique. While this in–band signaling certainly is highly efficient, it tends to complicate automated processing. Most encoded logical values will present some kind of error condition, e.g. as shown in the example in figure 5.13. However, as not all logical values are necessarily error conditions, automatic classification is difficult to achieve: it can only rely on the logical value's description. Extensibility is rather hard to achieve in a backward compatible way due to the descriptions' syntax (section 2.3.5).

IEEE 1451.2 TEDS are focused around the features and properties of sensors and actuators, they lack any communication properties (section 2.3.3). What makes them stand out is the possibility to specify physical units in a sophisticated, machine–readable manner (section 2.3.2). There is a slight trade–off towards keeping the description very compact: exponents of SI base units can only be given in increments of 0.5, and they are limited to a range of -63.5 to 64. Of course, in practice, this is not a serious limitation. The missing possibility to specify an offset weighs heavier. Otherwise, the TEDS may even include detailed calibration information. This feature is not necessary in other schemes, such as COSMIC. COSMIC devices will do calibration internally. Due to its binary notation, strict standardization is necessary (section 2.3.6). The TEDS are complete in terms of the TEDS standard (section 2.3.4).

CANopen EDS is focused on describing what parts of the device's object dictionary are implemented, and what parameters apply to the entries. The EDS will mostly contain the description of a defined set of object dictionary entries, as specified in the applicable device profile (section 2.3.6). It is possible to represent physical values. The representation is even more compact than in IEEE 1451.2 TEDS: it uses only 4 bytes compared to the 10 bytes for IEEE 1451.2. Clearly, quite a bit of flexibility must be sacrificed to achieve this.

Besides providing general information about nodes, TTP/A STDs cover two main topics. The first is concerned with protocol issues. It contains information about the RODLs on interest for the device (section 2.3.3). The second describes the node's functionality in terms of signatures containing the function's name, its address within the IFS, and its parameters (similar to operation signatures, section 2.3.7). For each parameter, the corresponding physical unit is given as a string. This seriously limits the possibility for automated processing (section 2.3.2). It is possible to enhance each information block by a block of meta information for documentation purposes. Further, tools may store tool–specific information in the respective area for each block. There are three features clearly distinguishing TTP/A STDs from the other approaches:

- The integration of communication information.
- The possibility of documentation for each data block.
- The structural integration of tool–specific information.

While the first two are sensible, integrating tool–specific information into a node description is not a very good idea. This information tends to bloat the description significantly, as the tools grow more powerful, and increase in numbers. Also, the information stored there is only useful for the specific tool the developer used. Any maintenance personnel or other users will not be able to make use of it.

LIN, CANopen, and TTP/A distinguish between template descriptions and instantiations of these to cover properties of a specific incarnation with configured parameters. Further, LIN and TTP/A support descriptions of whole clusters. These cluster descriptions make use of the structure of individual node descriptions. Essentially, cluster descriptions provide a wrapper around multiple node descriptions.

TTP/A currently is moving forward another step by introducing application descriptions. Application descriptions model an application in terms of jobs running on different nodes, and their interaction. These descriptions allow to automatically generate TTP/A schedules.

5.8.4 Configuration management

This section discusses the suitability of the presented technologies in terms of their support for configuration management as introduced in section 4.2.

Planning and building of distributed system is not an issue for UPnP and Jini. These technologies aim at selling individual nodes to consumers who will subsequently connect them to a loosely coupled system. Web services follow a similar approach, only the business model is different. IEEE 1451.2 does not concern itself with planning smart transducers, as such devices are kept simple enough to be easily deployable. They are connected to their host processor using a point–to–point connection. The CANopen standards do not describe any methodology towards cluster planning. For LIN, tools for planning and simulating clusters are available from different vendors. The wide availability of such tools was part of the goals behind LIN. The data exchange among those tools is done using the LDF. For TTP/A, similar tools exist ranging from manual scheduling to application planning with automated schedule generation. Tools for simulating and verifying schedules are available.

Automated parameter configuration is partly provided by TTP/A application planning and scheduling tools, and by LIN network design tools. IEEE 1451 does not provide any automatic configuration. CANopen devices may be able to auto–configure a few low–level communication parameters, most notably the bitrate used. UPnP devices automatically configure their networking parameters on connection to a network. Jini services and Web services usually do not have any needs for auto–configuration.

Service *discovery* plays a central role in both UPnP and Jini. Both technologies employ UDP Multicast for discovery. UPnP does so via the standardized SSDP. In Jini, discovery of the lookup service is done via a proprietary protocol. Web services do not offer any discovery facilities. The same applies to LIN, IEEE 1451, CANopen. TTP/A provides the possibility to baptize new nodes within the system, which requires finding out their unique ID. A binary search algorithm across the name space is performed. This can be seen as a discovery protocol. However, it is rather hard to subsequently integrate the new node into the existing cluster without a reconfiguration of the cluster.

The discovery process only reveals the presence of a component. The component starts to be useful only if it, or some proxy service, accepts *queries* about the component's capabilities. UPnP devices only provide coarse query capabilities. Device and service descriptions must be retrieved in whole. Analysis is left to the retriever. UPnP devices handle queries themselves, there are no central or proxy servers. In contrast, Jini services register with a (local) lookup service. The lookup service essentially is a proxy that will handle queries about other services. The granularity of the queries is also rather coarse, though finer than with UPnP: Query properties can either be set to a don't care state, or they must match exactly. Any finer matching must be done by the queree. For Web services, technologies like UDDI were introduced to query services. They all represent proxy services;

a Web service does not support direct queries in a standardized way. LIN, IEEE 1451, CANopen and TTP/A do not offer any query services.

5.8.5 Compatibility Checking

Jini relies on Java interfaces to guarantee compatibility among services and clients. Obviously, this will not be enough to enforce properties beyond those inherent to the language, e.g. timing constraints cannot be checked. UPnP devices mostly adhere to standardized profiles. Even though their interface is declared in the service descriptions, and therefore dynamically invoking calls to an arbitrary service would be possible, it is seldom useful if the semantics are not known. Mostly, known services are searched for. On–line compatibility checking might be performed, yet it is not part of the standard. One of LIN's goals is to facilitate the construction of systems from pre–manufactured components. These are accompanied by their description files. Yet there are no means towards enabling automated or assisted replacement of components by other, similar ones, i.e. the LIN specification does not talk about compatibility checking. With IEEE 1451 smart transducers, replacement by successors, or otherwise "better" versions are one of the goals. Yet, the standard does not discuss any compatibility checking. It is up to the NCAP to handle e.g. the connection of a sensor of the wrong type, or of an insufficiently precise sensor. Similarly, CANopen and TTP/A are not concerned with compatibility checking. TTP/A descriptions would offer enough detail for such checking, though. For Web services, no compatibility checking is provided.

5.8.6 Testing

Black–box testing (or any other approach) is not directly supported by any of the approaches presented. For approaches that feature declarative descriptions, assisting a tester is possible, if the descriptions contain relevant information, e.g. valid data ranges. Specifically LIN and TTP/A come closest to this.

5.8.7 Real–time capabilities

UPnP, Jini and Web services are not real–time capable, as defined in section 2.2. Strict real–time guarantees were not a design goal. LIN, IEEE 1451 smart transducers, CANopen and TTP/A are able to provide real–time guarantees. IEEE 1451 transducers and CANopen devices are polled. They guarantee to answer within bounded time. LIN and TTP/A have predefined schedules. A master initiates the communication round sequences and provides clock synchronization signals. Both are not able to provide any other service class like SRT or NRT.

5.8.8 Logging

It is extremely hard to provide central logging facilities capturing all communication in UPnP, Jini and Web service systems. For IEEE 1451 smart transducers, logging can easily be done at the NCAP. In clusters of LIN or TTP/A logging is best done at the master node which has significantly more power and storage available than other nodes. Because the master usually works as a gateway node to another backbone network, it is also the natural place where one would expect logging information. Generally, logging is easy if there is a shared communication bus, such as a field bus. Of course, this holds for CANopen. Yet, logging is only mentioned explicitly with LIN and TTP/A.

5.8.9 Simulation and Emulation

A system can either be simulated as a whole, or it can be emulated in parts. LIN and TTP/A provide system simulation. With LIN, even system emulation and debugging is described. This eases the system integration phase by allowing to build the system piece by piece. Also, in case an individual device is not yet available, the system can already be tested and debugged. All other approaches do not elaborate on system simulation.

5.8.10 Summary

The following table summarizes the presented related work to give a compact overview. The details behind this short summary can be found in the previous sections.

Criterion	**Web services**	**UPnP**	**Jini**	**IEEE 1451**	**CANopen**	**LIN**	**TTP/A**
Component oriented	yes	no	yes				
Life cycle support	incomplete						
Description language	XML based		none	binary	text based		XML based
Configuration management	none			vendor specific		yes	
Compatibility (checking)	no	standard device types	Java type system	no			
Testing	not mentioned						
Real–time capabilities	none			limited			HRT
Logging	no				possible		yes
Simulation / emulation	not mentioned					yes	in work
Resource usage	high			very low	low	very low	
Communication model	pull	push/pull		pull			push/pull

Chapter 6

COSMIC

The vision behind COSMIC is that of a networked physical world. In this world, all devices may interact with each other and their physical environment. As an example, consider yourself driving home in your car. Right as you pull up your driveway, your garage door opens, and the light in the house is turned on. Smart devices, such as the controller for the garage door, populate this world. They may encapsulate both software and hardware, so they are able to interact with their environment. Interaction with the environment is done via sensors and actuators. Smart devices can also communicate among each other using COSMIC interaction abstractions discussed below. Conceptually, smart devices are modeled as sentient components. The notion of sentient components was introduced in section 2.1.2. Each device acts autonomously and context–aware in response to aspects of its environment. When interacting with the environment, the devices' interaction and reaction must conform to real–time constraints, especially when operating in safety critical environments. Independent subsystems must be able to take local decisions and interact with other subsystems via networking capabilities, or via the physical environment. Interaction among devices often must also conform to real–time constraints. Special interaction patterns are necessary to combine timely communication with local autonomy. An event–based anonymous communication scheme is able to meet the requirements. In case of defects, the smart devices must be easily replaceable. Cost is also a critical issue. Of course, power consumption must also be limited, especially when operating on batteries or other limited means of supply. The smart devices often also must adhere to space constraints, so they can be integrated seamlessly into the environment.

Typically, applications will group a number of cooperating devices inside a wider network universe, which is composed of different physical networks with widely varying characteristics. The networking technologies may range from embedded field buses to high speed backbones. Often, a hierarchy of communication networks is present, e.g. a field bus network inside the car from the example above, and a wireless link enabling it to communicate with the "infrastructure", e.g. the garage. Generally, the hierarchy of networks can be classified into a tighter coupled network within an entity (the car), and a possibly loosely coupled network linking the entity with other entities (the garage). The former is called *controller area network* (CAN) in a broad sense, not necessarily based on the CAN field bus [11]. In more abstract terms, a CAN is also called *island of control*. The latter is called *wide area network* (WAN), independent of the actual network technology employed. The CAN and the WAN are connected via gateways. This hierarchical structure enables cooperation of islands of control via the WAN. Cooperation must be achieved in a timely and reliable manner. This model is called a WAN–of–CAN structure. It may be composed recursively over an arbitrary number of levels. Often, islands-of-control are composed of a number of small and severely constraint devices. Further, a more powerful node is often part of the system. This node may be connected to different networks, i.e. it may form the gateway to another network.

6.1 Interaction model

COSMIC uses an event–based interaction model. Hence, *events* are the main abstraction of the interaction model. Rather than specifying addresses, a content–based scheme is used. *Producers* and *consumers* of events specify what type of events are produced, or of interest, respectively. The result is an anonymous many–to–many communication scheme. Events do not imply any specific model of synchrony. They may be spontaneously generated and disseminated immediately, triggered by an occurrence in the environment or the system itself. Asynchronous communication also avoids artificial control flow dependencies among participants. This leads to autonomy of the participants.

Autonomy and anonymity are advantageous for dependable systems. Yet, these properties alone cannot guarantee the timeliness of the required information under all anticipated load and fault conditions. The underlying network needs to provide a certain level of predictability. As a WAN–of–CAN structure may include many physical networks with widely differing predictability properties, the event model cannot provide a single degree of predictability for all events in the system. Instead, it must be possible to explicitly express different quality requirements. This problem is tackled in COSMIC via the notion of *event channels*. An event channel encapsulates the dissemination properties of the underlying network in a high level abstraction in terms of application requirements. This is especially important when interaction is dynamic and spontaneous, e.g. when mobile devices want to interact with a new or changing environment. Different predictability classes are beneficial even on a single network, as predictability comes at a price, and not all events need the same reliability and timeliness guarantees. The middleware supports a high level of abstraction, yet it should be appropriate for resource constraint smart devices.

6.1.1 Events

As mentioned above, events are the main communication abstraction used in COSMIC. COSMIC components interact by producing and consuming events, i.e. they share information without transferring the flow of control. Producers and consumers do not specify source or destination addresses, instead, a content based scheme is employed, i.e. events are typed information carriers. Because analyzing the complete contents of an event, as in systems like e.g. Siena [13], requires a large amount of resources, a subject–based scheme is used. This significantly lowers resource requirements and enables fast filtering of events based on their subjects. Events distinguish themselves from simple messages by carrying information about the context in which they where generated, and quality attributes defining requirements for their dissemination. Thus, event instances are specified as:

$$\text{event} := <\text{subject, attributes, content}>$$

The *subject* corresponds to the event type and thus is related to the event contents. *Attributes* is a list of name/value pairs complementing the event contents. The attributes capture context information about the event, like location, or time of generation. Events are time/value entities that eventually expire and may become harmful for the system if used after expiration — a phenomenon well known from real–time systems [12]. Therefore, the event's validity interval is an important quality attribute. The omission degree, another quality attribute, defines under which conditions a correct transmission is guaranteed. The event's *content* part carries the actual data to be shared, e.g. temperature, in degrees centigrade.

Both observations of real–time entities, and programmed changes within the system are uniformly treated as events. This is supported by the notion of smart devices, which encapsulate all low–level computations and do not allow direct access to their internal structures. From an architectural perspective, smart devices are only visible as networked components with an event interface, a view also shared by [84]. On the respective abstraction level, these smart components are represented as active objects, publishing and consuming events. A generic architecture called *GEAR* (generic event architecture) has been presented in [135]. In this architectural model, objects interact exclusively via an event layer, which hides both the actual network and the transformation process of real–world events. Thus, it is possible to reason in terms of generic events. COSMIC, and the event and event channel abstractions in particular, provide a specific way to realize such an architecture [136].

In COSMIC, events are used for another important abstraction. Instead of communicating raw measurement values, data is provided in events using a more useful level of abstraction. Considering even simple devices like e.g. temperature sensors, it is beneficial to transmit not raw data. Raw data would include e.g. all measurement noise. The data is highly specific to a certain type of sensor, and possibly even to an individual sensor, if the sensors vary significantly. Therefore, an application using such sensors must be tailored to specific sensors, and do all calibration and filtering itself. Replacement of such a sensor would greatly be obstructed. Switching from a PTC to a NTC sensor would even require changes to the application. COSMIC devices provide temperature events instead. The data disseminated is not provided as raw data, but correlated to a corresponding physical unit, e.g. degrees centigrade for temperature events. COSMIC devices also filter, calibrate, or otherwise pre–process the data in a suitable manner. This allows the creation of more generic high–level applications. The underlying smart sensors can easily be replaced as long as the provided data meets the application's requirements.

6.1.2 Event Channels

Event channels enforce the temporal and reliability requirements of the communication. Event channels are unidirectional entities, i.e. a component can either publish to an event channel, or subscribe to it, but not both. Thus, similar to events, event channels are defined as

$$\text{event channel} := \langle\text{subject, attributes, handlers}\rangle$$

The *subject* defines the type of events that may be communicated via the channel. *Attributes* is a list of (quality) properties defining the channel's dissemination properties, e.g. periodicity, deadline, maximum latency, or reliability. While the attributes of an event describe the properties of a single individual occurrence of an event, the attributes of the event channel abstract the properties of the underlying communication network. There are two *handlers*: a regular event notification, and an exception handler.

Event channels allow specifying quality attributes on the application level. Whenever a publisher makes an announcement for publication, or when a consumer subscribes for an event notification, an instance of an event channel is created locally. On publication announcement, the publisher specifies the quality attributes it is able to deliver. When a consumer subscribes to an event channel, it may specify context attributes of an event which are used to filter events locally, e.g. if only events generated at a certain location are of interest. An advanced filtering scheme working on event attributes in an efficient manner has been proposed in [74]. Additionally, the consumer specifies quality properties of the event channel.

COSMIC distinguishes three event channel classes according to their synchrony properties:

- *Hard Real–Time.* This synchrony class guarantees event delivery within the validity interval in the presence of a specified number of omission faults. A deadline for delivery is derived from the validity interval.

- *Soft Real–Time.* Soft real–time event channels derive deadlines for scheduling from the validity interval. In overload situations, these deadlines cannot be guaranteed. In such a case, it is possible to detect

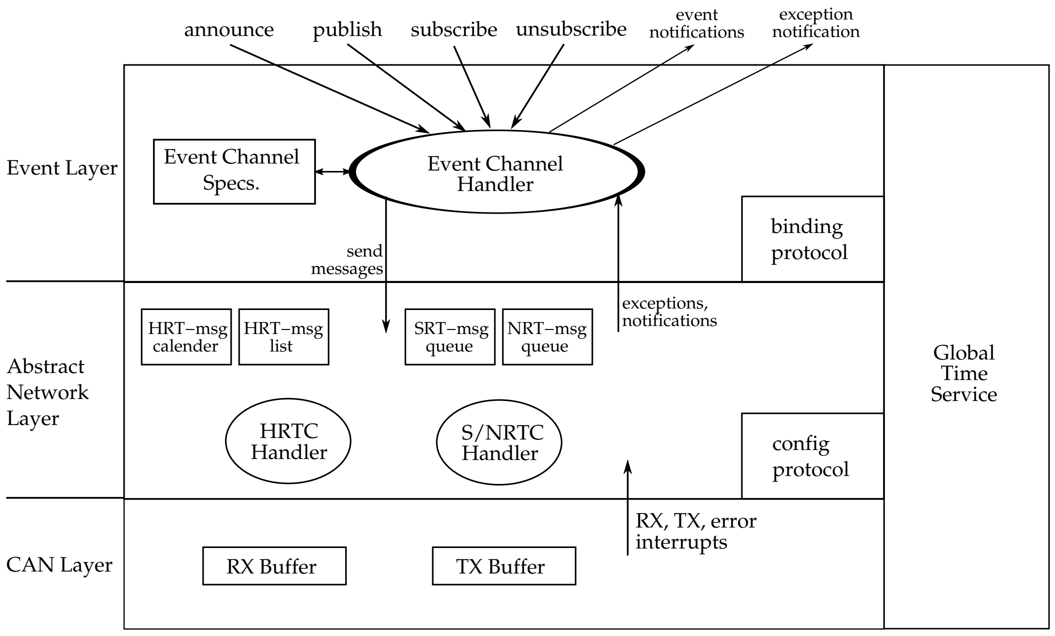


Figure 6.1: Architectural layers of COSMIC

the temporal inconsistency using the validity interval. Thus, awareness can be raised in the application.

- *Non Real–Time*. Non real–time event channels do not provide any temporal specification and disseminate events in a best effort manner.

The notion of event channels not only allows an application to specify its dissemination quality attributes on a high level, but also provides the middleware implementation with the possibility to reserve necessary resources when a channel is created. The middleware takes care of mapping these channel properties to lower level protocols which will meet the requested requirements.

6.2 The COSMIC middleware

Each node runs an instance of the middleware locally. It's internal architecture is presented in figure 6.1. The respective components are discussed in the following sections. The COSMIC middleware is subdivided into three distinct layers. The uppermost layer provides the application with the event channel abstraction. It takes care of routing and filtering events, and maps events into appropriate event channel classes. The middle layer is responsible for providing the respective guarantees for these classes. The lowest layer consists of a hardware driver.

Since COSMIC supports a hierarchical network structure, an artifact connecting different networks is required. Nodes connected to multiple networks hence function as transparent *gateways*.

6.2.1 Event Channel Handler

On each node hosting COSMIC components[1], the middleware is represented by an artifact called *event channel handler* (ECH). This event channel handler provides the COSMIC component with a standard API to the event channel abstraction on all platforms. The ECH is responsible for event routing and filtering, depending on the application's requirements. The ECH's API is composed of the four calls described below:

`announce( subject, attribute_list, exception_handler );`

> Before publishing events, the producer must `announce` them. The announcement is necessary to set up the respective event channel at the local event channel handler. This includes the reservation of data structures, lower level bindings, and the preparation of routing and filtering. `Subject` is a unique identifier specifying the type of events that are going to be published. The attribute list contains the quality properties that define the event channel, such as latency, omission degree, and periodicity.
>
> If the channel cannot be set up in a way compatible with the application's requirements stated via the `attribute_list`, i.e. if not enough resources are available, the call to `announce` fails.
>
> After successfully calling `announce` once for each subject to be published, the application can subsequently call `publish` to actually disseminate events.
>
> Later on, when operating, the `exception_handler` will be called in case of faults (e.g. timing faults with a `publish`).

`publish( event );`

> `Publish` commits the the `event` to the ECH. The ECH enters the `event` into the respective outgoing transmission queue, as set up on `announce`ment.

`subscribe( subject, attribute_list,`
`notification_handler, exception_handler );`

> Before receiving event notifications, the application must `subscribe` to the `subject` of interest. Respective quality properties are specified in the `attribute_list`. On subscription, the respective event channel is set up at the local event channel handler. This also includes setting up the necessary routes for event delivery from the producers to the consumer.
>
> Subsequently, whenever an event arrives, the `notification-handler` is called by the middleware. In case an event does not arrive on time, or other failures are detected, the `exceptionhandler` is called.

[1] For smart devices, each node typically hosts exactly one COSMIC component.

Only events from producers providing equal or stricter quality properties are delivered.

`unsubscribe( subject );`
If an application is not interested in a specific type of event any more, it can `unsubscribe` from the respective `subject`. This helps clean up both local resources, and remote resources: routing table entries and network bandwidth may be released if there are no further subscribers on a given branch in the network.

It can be seen that this API provides general access to event channels. Low–level details, such as the selection of the event channel class, are invisible. The middleware will determine the appropriate event channel class based on the quality attributes specified on `announce` or `subscribe`, respectively. Even subscribing with very lax quality attributes that are mapped to Non Real–Time, while the publisher produces events using a Hard Real–Time scheme, the application does not need to consider such change in semantics on the communication side — the ECH handles everything transparently as long as the requested quality attributes on the consuming and on the producing side are compatible.

In case of failures, the application is notified using the specified exception handlers. This leaves the responsibility of dealing with errors to the application; exactly the place it belongs. The COSMIC middleware should not and cannot know how to react to such failures.

6.2.2 Abstract Network Layer

Below the ECH, the *abstract network layer* (ANL) provides media independent abstractions for the different real–time classes.

Conceptually, the ANL incorporates three handlers for the three different real–time classes. Their job is to provide the ECH with communication facilities guaranteeing certain quality:

HRTC Handler: Hard real–time events are disseminated using a TDMA scheme. All necessary resources are reserved a priori. The HRTC handler stores a calender according to which events must be disseminated. Thus, hard guarantees for event delivery can be enforced. The middleware always delivers events to the consumer at the deadline. This guarantees constant latency and the smallest jitter possible for periodically produced events.

SRTC Handler: Soft real–time events are disseminated using dynamic priorities. Their dissemination priority is increased when time is nearing their deadline. Any events that expired, i.e. that are already older than their validity interval before being sent, are discarded. Thus, no a priori reservations are necessary. Events are extracted from the queue for dissemination based on their priority.

NRTC Handler: Non real–time events are disseminated in a best effort manner. Events are processed in a FIFO manner.

As an optimization, the SRTC handler and the NRTC handler may be combined into a single component while the respective event queues remain separated mainly due to performance reasons.

6.2.3 Communication Driver

At the lowest level of the COSMIC architecture, a network driver resides. It provides primitives for handling the actual communication hardware.

6.2.4 Global Time Service

The global time service provides the middleware with a synchronized clock. A synchronized clock is necessary to support a calender based dissemination scheme for hard real–time event channels, as well as for determining the expiration of events disseminated through soft real–time channels.

6.2.5 Gateways

Nodes connected to multiple physical networks can function as gateways from one network to the other. The gateway has separate instances of the ANL and driver layer for each network it is connected to. To applications, gateways are transparent. This is a required since the notion of event channels is end–to–end, and not concerned with intermediate entities.

6.2.6 The implementation

Currently, a mapping of the previously discussed architectural abstractions to the CAN bus, and to TCP/IP networks is implemented on a number of platforms. The supported platforms cover a wide range from cost–effective small 8 bit microcontrollers, such as the Freescale HC08 family, to PCs running Linux and RT–Linux. Figure 6.2 classifies well–known middleware frameworks and COSMIC in terms of usually implemented device potentials[2].

For a communications middleware, the supported network systems are also important. Most middleware frameworks, e.g. Jini, UPnP, and Web Services, rely on TCP/IP as the network protocol suite of choice. This enables them to make use of any kind of network medium that can be provided with an IP stack. While IP stacks are available for a lot of network media, not all of these are suitable for widespread use. Useful and widespread media include Ethernet, 802.11a/b/g, and Token Ring, to name a few. When designing a middleware framework for small embedded components, TCP/IP usually is not a good choice:

[2]The figure does not hint any minimum or maximum system requirements. COSMIC is currently implemented on all platforms written in italics.

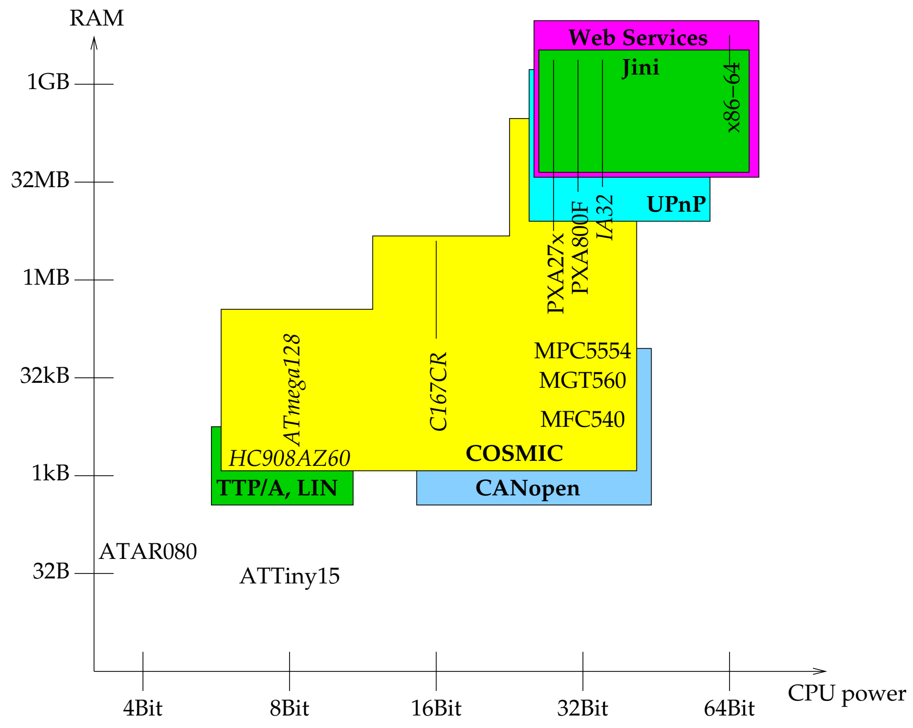


Figure 6.2: Middlewares and their usual range of application

- Its protocol stack is rather large and implementation is unfeasible in most components. On an 8 bit microcontroller with only a few kilobytes of RAM, a complete TCP/IP stack eats up a substantial part of the resources. The popular Nut/OS[3] for the Atmel AVR family e.g. requires 12 kBytes of Flash and 49 bytes of RAM for the TCP/IP stack, excluding the low–level hardware driver[4].

- In embedded systems the networking stacks tend to be very minimalistic, traditionally. This saves resources for the application and allows for better predictability. With TCP/IP, predictability is basically impossible to achieve. In the automotive and the factory automation domains, the CAN bus is a popular networking subsystem.

The current COSMIC implementation makes use of both the CAN bus and a TCP/IP network for inter–node communication. Real–time communication is only implemented on the CAN bus; the TCP/IP implementation works for Non Real–time channels.

[3]Nut/OS and the Ethernut platform can be found at http://www.ethernut.de. All presented numbers are taken from that website, too.

[4]To run the provided HTTP server, the on–chip 4 kByte RAM is not enough; about 10 kBytes of external RAM are required. Also, the HTTP server requires another 3 kBytes of ROM. The Ethernet driver requires about 4kBytes of ROM and more than 100 bytes of RAM

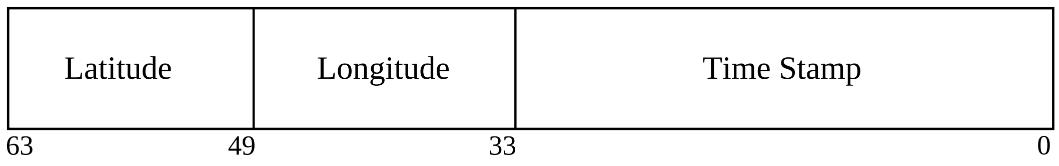


Figure 6.3: COSMIC identifierlayout

COSMIC Unique Identifiers

COSMIC uses globally unique identifiers for node and subject identification. While standard 128 bit UUIDs are widely used, the current COSMIC implementations use 64 bit identifiers for both historical and technical reasons. A length of 64 bit can conveniently be handled on the CAN bus. To achieve uniqueness with 64 bit identifiers, the 64 bits are split up as follows (see figure 6.3): 31 bits of geographical position and 33 bits of time stamp. The geographical position is further split up into 15 bits of latitude, and 16 bits of longitude. The time stamp is the number of seconds since January, 1st, 2005, 12:00am UTC.

Using 15 bits for latitude and 16 bits for longitude yields areas of at most 1.2km by 1.2km at the equator, assuming the earth is a perfect sphere. As most of the earth's surface is covered with water, and little software development takes place on the sea, it would be possible to shrink the area covered by a single latitude/longitude vector further. Essentially, a lookup table based approach could be used, provided all developers use the same table. As 70% of the earth's surface is covered with water, the areas making up the lookup table can be made smaller than 700m by 700m. The 33 bits for the time stamp will provide enough address space for the next 272 years. Where generating only a single identifier per second is not enough, older "unused" identifiers can be used. Unused identifiers are identifiers that did not get generated while the generating entity — e.g. a corporation — was occupying a certain spot in the grid.

CAN identifiers

The CAN bus [11] uses *message identifiers* instead of traditional addresses. The message identifiers define what the message contains. They can be assigned arbitrarily in a CAN system, as long as no identifier is sent by more than a single device.

The bus arbitration also makes use of the message identifiers: Access to the bus is controlled in a CSMA/CR fashion. This is achieved by exploiting the wired–AND characteristics of the physical bus, where a logical 0 is *dominant*, and a logical 1 is *recessive*, i.e. a node transmitting a logical 0 will "overwrite" a logical 1 on the bus. This is used during the arbitration phase. When a node wants to transmit a message, it waits for the bus to be idle, like in traditional CSMA schemes. The first part of the transmission is the message identifier. During each bit's transmission, the node monitors the bus.

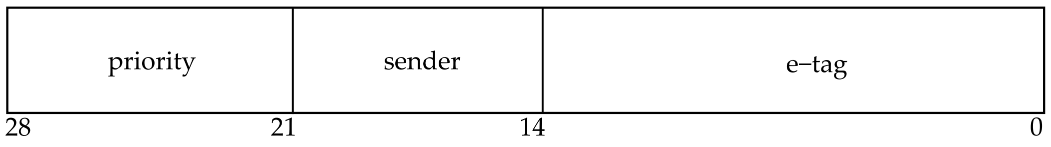


Figure 6.4: Layout of the CAN identifiers

If it detects that the bus does not carry the same bit it tries to transmit, the node just lost the bus arbitration and aborts its transmission. Thus, after the complete message identifier has been sent, it is clear which node may proceed to send the rest of the message. No collisions can occur, if no identifier is sent by more than one node, as stated above. To summarize, the message identifier with the lowest numerical identifier has the highest priority and wins access to the bus.

COSMIC exploits this arbitration mechanism. The message identifiers are subdivided into three sections. There are two kinds of message identifiers: 11 bit *standard identifiers* and 29 bit *extended identifiers*. The 29 bit extended identifiers provide enough space to include all three fields with feasible sizes. The extended identifiers cause some extra overhead. However, this ″overhead″ is made up for by encoding useful information in the identifiers [70]. Specifically, the identifiers are partitioned into (see figure 6.4; MSB is sent first):

- a **priority** field,
- a **sender** field, and
- a **e–tag** field.

The *priority* field is used to distinguish the different real–time classes. The highest priority is reserved for hard real–time communication. The HRT channels are scheduled according to a TDMA scheme, which is described in more detail in the following section. The NRT channels have a fixed priority of 255, the lowest possible. Messages belonging to SRT channels use dynamic priorities in between. The SRT event's deadline is dynamically mapped to a message priority by the SRTC handler, as described in section 6.2.2. The priority increases while the event is nearing its deadline.

The *sender* field is used to keep the identifiers unique. During start–up, each node is configured. During this configuration process, the node's unique identifier (which is a COSMIC unique identifier as described in the preceding section) is mapped to a short identifier. This short identifier is subsequently used in the sender field of each message leaving the respective node. This ensures that all message identifiers remain unique in the system, even though their priority and e–tag fields may match. As seven bits are used to encode the sender, a maximum of 128 nodes are possible on a CAN bus with the current implementation.

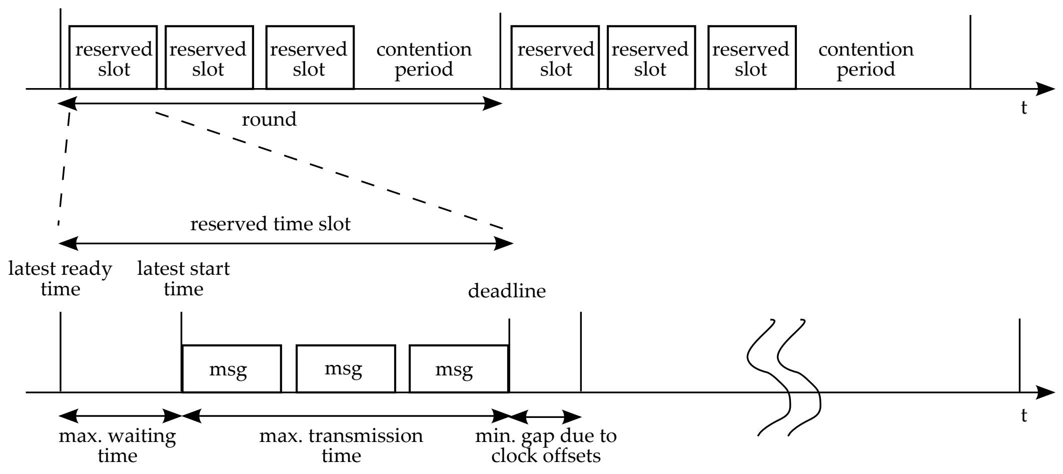


Figure 6.5: COSMIC communication rounds

The *e–tag* field is used to encode the event channel that the message belongs to. Similar to the sender field, the e–tag is a shortened event channel's subject (which again is a COSMIC unique identifier as described above). The e–tags are assigned on announcement of publication or subscription to an event channel. A component called *event channel broker* (ECB) is responsible for assigning these short nodes. The details about the ECB are not important for this work; more details can be found in [70]. The encoding of the event channel within the message identifier provides a means to filter incoming messages efficiently. On many platforms, this can even be done by the hardware. Many CAN controllers incorporate filtering facilities for the message identifiers. By filtering the 14 LSBs based on the event channels that are locally subscribed to, this can tremendously lower the CPU load.

Hard Real–Time communication on CAN

Figure 6.5 (top part) gives an overview how communication is arranged in COSMIC in the time domain. This section covers the basic characteristics; more details are presented in [72] and [69].

As previously mentioned, a TDMA based scheme is used for Hard Real–time operation. Communication is partitioned into rounds that are repeated. For each hard real–time event, a communication slot is reserved a priori to avoid conflicts [71]. During the non–reserved time (and during the unused reserved time), soft and non real–time events are communicated.

The reserved time slots are further partitioned (lower part of figure 6.5. At the beginning, a waiting time is introduced. This waiting time corresponds to the time needed to transmit a single message across the CAN network. Some nodes that do not participate in HRT communication may not know about the reserved slots, and transmit messages at will. This requires said waiting time, because messages cannot be preempted on the CAN bus. The special layout and distribution of the CAN identifiers (see figure 6.4) ensures that any HRT message will win the bus arbitration over any other SRT or NRT message. After the waiting time, the actual HRT message is transmitted. Depending on the allowable omission degree, it may be repeated a number of times. Repetitions are omitted if the message was transmitted successfully. This is also possible because of the distinct features of the CAN bus. Further, reserved slots are only used when new data is available. Taken together, these features help keeping the overhead to a minimum. Between any two reserved slots, a gap has to be inserted to account for clock offsets.

Chapter 7

The CODES Approach

COSMIC provides suitable abstractions for small smart devices, specifically for their interaction. Still, the development of such smart devices is a non–trivial task. The COSMIC middleware provides an easy to use communication infrastructure. Still, software development for such small devices is an error–prone task, as described in chapter 3. The problems are obviously not limited to developing a component's software. More problems arise when integrating individual components into a larger system. Traditionally, it is hard to ensure compatibility of the components in the system. COSMIC e.g. assumes that all devices subscribing or publishing a specific subject know how to correctly encode and decode the respective events. Yet, there is no way of enforcing this. Cross–checks are traditionally done checking the components' specifications manually.

In the highly dynamic vision of COSMIC, a system may have to cooperate with other systems dynamically. Consider a robot as a COSMIC system. This robot is moving through e.g. different factory halls. Over time, these buildings incrementally may be equipped with indoor localization systems similar to GPS, e.g. as proposed in [106]. The existing robots should then be able to dynamically make use of the localization systems, as they become available. Such dynamic cooperation of systems requires awareness of the potential cooperation partners, as well as knowledge about how to make use of those potential partners. Currently, a wide range of research is expended in this area, e.g. the SAWSDL project as introduced in section 5.1.2. Many projects provide a means for declaring semantic information, and its relations in a machine–processable manner, e.g. in the form of ontologies, such as with OWL [138], [58].

The CODES (COsmic embedded DEvice Specifications) approach tackles these problem areas. It has a number of goals that aim at supporting a smart component's life cycle from the design phase to disposal. The central element of the approach is the component's description. In detail, these are (see also figure 7.1):

- *Support for component design and development.* The descriptions form the component's *documentation*. They can be used to *generate* parts of the component's *code*. Also, they can be used to assist the developer in

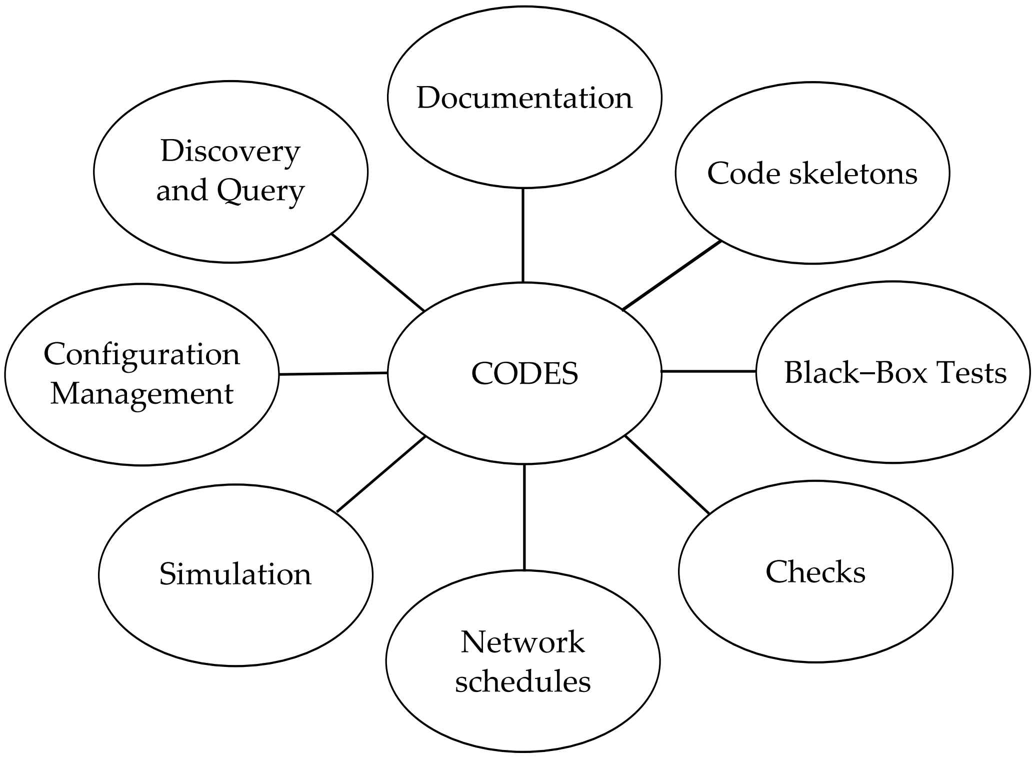


Figure 7.1: Benefits of using CODES

black–box testing the components.

- *Support for component integration.* The descriptions provide the details necessary to run *compatibility checks* across a number of components. They can be used to *generate* the *scheduling* calenders necessary for hard real–time operation on the CAN bus. Also, other application specific parameters can be accessed and set on integration. Further, the description of components can be used to *simulate components* not yet available.

- *Support for system maintenance.* When systems are in need of maintenance, it is crucial to know the exact composition of the system. The online availability of the component's description provides the maintainer with all necessary information for *configuration management*.

- *Support for dynamic cooperation.* Dynamic cooperation of components requires awareness of possible cooperation partners, and the knowledge what these partners may provide. Awareness is raised via a *discovery and query mechanism*. The CODES descriptions provide all details necessary to communicate with partners. Currently, semantic information is contained in names, e.g. the subjects or events. In the long run, these names can function as pointers into ontologies. Thus, CODES descriptions can be integrated with e.g. semantic web technology to enable autonomous dynamic cooperation of components without any user interaction.

7.1 Requirements for the descriptions

In order to achieve these goals, the requirements from section 2.3 have been detailed.

The descriptions must be able to function as documentation of the components (see section 2.3.8). More specifically, a description must specify a component as a black–box, i.e. the component's network interface must be specified completely (see sections 2.3.7 and 2.3.4). This view is shared by all presented approaches from the embedded domain in chapter 5. To complete the documentation, general information about the component, such as its name, serial number, or manufacturer must also be part of the description. This allows the identification of the component.

The descriptions should be in a text–based format, enabling their use even in the absence of special tools. It is probable that even in the long run, ASCII or Unicode based text files will be readable (see sections 2.3.2 and 2.3.6), as opposed to some proprietary binary format, such as e.g. the IEEE 1451 TEDS. Using an arbitrary text format is not sufficient. Consider a format that would only list the values of parameters necessary for some component. The index within this sequence of values will determine what parameter any given value is associated with. Such a format will be useless without the accompanying document describing the sequence of parameters (see section 2.3.2). Further, it is very prone to errors like mis–ordering of values, omission of values, or extra values. LIN descriptions partly use such parameter lists, specifically in the signal definitions, as presented in figure 5.13. Thus, the format must explicitly list each parameter, and its associated value. Finally, the parameters should be named in a way that at least domain experts will be able to deduce their meaning.

The descriptions must be machine–readable and checkable for grammatical correctness and completeness (see sections 2.3.1 and 2.3.2). All technologies presented in chapter 5 share this requirement. It provides the basis for successfully using the descriptions in automated processes.

To avoid the creation of a language that is extremely universal, yet hard to use, and that hinders the creation of efficient tools, the language must be based on a suitable set of abstractions. All description formats presented in chapter 5 are naturally tailored to the abstractions and technologies they strive to support. CODES is no exception to this rule. It is based on COSMIC abstractions. This results in a description format that defines a component's interface at a suitable level, both for the designer and for the users of the component (see section 2.3.3).

To achieve interchangeability of devices, e.g. when a failed device needs to be replaced, a detailed description of the component's capabilities is required. Compatibility checks then allow to determine, whether a given device can be used as replacement. Thus, the descriptions must contain low–level technical details, such as e.g. the layout of the events' data structures, as well as high–level quality attributes (see section 2.3.3).

The description format should be flexible enough to allow for future extensions (see section 2.3.5). Extended descriptions should be backwards compatible with older tools and devices. Again, consider the replacement of a failed device. It should be possible to integrate a device with an extended description without upgrading the rest of the system.

To enable autonomous dynamic cooperation of devices, the (machine–readable) descriptions must be available at run–time. This raises awareness about any available services. In contrast to simple plug–and–play schemes, dynamic cooperation requires the presence of semantic information that can be used to find relevant services (see section 2.3.8).

The descriptions must be stored and manageable on small systems. Static descriptions are obviously best suited in that respect. This is antipodal to the need of integrating possibly dynamic parameters that may change from time to time. The direct integration of such parameters into the description is vital for dynamic cooperation, e.g. the update frequency of the stated indoor localization system is relevant to the maximum speed the robot may safely drive with.

7.2 Design of the description language

XML was chosen as the basis for the description language. XML supports a number of the previously given requirements for the description language:

- It is text–based, and thus readable, even without special tools. A text editor basically suffices.

- It can be seen as a form of documentation, at least for domain experts with some knowledge of XML. The tags' names, if chosen carefully, will be an important hint for the domain expert to the respective tag's semantic meaning.

- Well–tested tools and transformation languages exist, e.g. XSLT to transform the descriptions into HTML documentation, or XSLT–FO to transform them into PDF. These transformations are also used otherwise, e.g. for code generation (see section 7.3.2) and the query mechanisms (see section 7.3.3).

- XML allows a structure that ensures that for each parameter, its name and its value is given: the parameter's name is the XML tag's name, and the tag's contents represents the parameter's value.

- XML files can be checked for grammatical correctness, if an appropriate XML schema document or an appropriate document type definition exists.

- An appropriate XML dialect, based on the COSMIC abstractions, can be defined with an XML schema or a document type definition.

- XML documents are machine–readable and processable. A wealth of well–tested tools to process XML are available, ranging from parsers to transformation engines.

- Extensibility can easily be achieved with XML. According to the FXPP, current software may simply ignore unknown tags. This ensures backwards–compatibility of revised descriptions with existing tools.

The XML family of technologies offers two possibilities for defining a grammar for an XML dialect: document type definitions (DTDs) and XML schema documents. As DTDs lack support for a type system, XML schema was chosen [100].

A syntax designed from scratch would require the development of own parsers and processing tools. This would not only generate a lot of work, but also be error–prone. Even if parsers exist, a number of other drawbacks still remain. A *C*–style language e.g. could easily be confused with code. Section based formats, such as used with CANopen, are unable to support hierarchical structures. Finally, the readily available transformation technologies, most notably XSLT, are not easily usable outside XML. Thus, a number of uses of the CODES descriptions would be much harder to implement, e.g. the code generation, and the documentation transformations.

On the downside, XML documents tend to be rather large, and a lot of resources are necessary to process them. This opposes the use of XML with small devices. Still, XML was deemed feasible on small devices. Small devices are able to handle static descriptions, if their size is not too large. Thus, two problems needed to be tackled:

1. The descriptions need to be static, i.e. the device cannot dynamically change its description. As some of the entries in the descriptions are not static, a parameterization scheme is required. This scheme is presented in section 7.2.2.

2. The descriptions need to be shrinked in size. This can easily be achieved as XML documents compress well. WBXML was tried, but the compression ratio of about 2:1 was insufficient [100]. GZip is readily available and on average achieves a compression ratio of about 10:1 (see table 7.1). This enables the storage of the descriptions even on very small devices, as only a few kilo bytes of ROM are necessary.

Like many programming languages, CODES is defined by a context–free grammar [121], [57]. Yet, there are meta rules not included in the grammar that make the CODES language context–sensitive.

The CODES language is used to describe COSMIC components. Each component is described in a separate document. Earlier versions of the language were presented in [105] and [76].

Device	XML size	GZIPped size
Infrared distance sensor	35.2K	2.7K
Acceleration sensor	38.6K	2.8K

Table 7.1: Device description sizes

To allow the validation of description documents, a XML schema document defining the CODES grammar was developed. The following sections present the details of the description language. The respective parts of the CODES XML schema are presented in each section. Thus, the descriptions are presented in plain text along with their formal definition. Details and elements used in multiple places throughout the descriptions are presented when they first occur. Elements from the XML schema name space are given with the commonly used `xs` prefix.

7.2.1 Contents and structure of the description language

CODES descriptions are based on COSMIC abstractions. Thus, the description of events and event channels form the core of the descriptions documents. Further, some general information about the component is included in the documents. This gives meta–information about the components and allows their identification. These three parts are reflected directly in the description documents. Finally, a versioning entry is included in the documents. Thus, the overall structure is:

```
<xs:element name="CODES">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="DocumentVersion" type="xs:string" />
      <xs:element ref="GeneralInformation"/>
      <xs:element ref="EventDefinitions"/>
      <xs:element ref="EventChannelDefinitions"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

It would be possible to combine the definition of an event type and its associated event channel into a single part. This would eliminate the need to cross–reference the event channels' definitions with the respective event definitions, thus easing the process of validating, whether the description is correct. The combination of aspects that were deliberately separated on the conceptual level is not acceptable, however. The description format was designed to resemble the underlying abstractions closely. The added overhead for checking the cross–references was deemed acceptable.

General Information

To allow the unique identification of a device, information about the device itself is necessary. This section of information thus contains a globally unique identifier for each device. Additionally, the section describes the device in terms of its name, its type, its manufacturer, and its supported event channel classes allowing the identification and classification of the device. The section about the device's `OperationalConnections` documents the device's communication capabilities. As the nature of these capabilities differs significantly depending on the actual networking technology, they are represented as a list of `CommunicationParameters`. The setup of the basic communication is done without the knowledge from the descriptions. It cannot be deduced from the descriptions, since they can only be retrieved after communication is technically possible, i.e. after the devices can successfully exchange network messages. This boot–strap problem is shared by all self–describing technologies. The basic setup can be achieved through either a static scheme, such as with IEEE 1451 transducers, where all communication parameters are fixed a priori. On the other end of the scale, systems like UPnP heavily rely on automatic configuration on various levels, e.g. automatic link configuration of an ethernet network, and automatic setup of IP communication over ethernet. The COSMIC implementation for TCP/IP delegates the setup to the TCP/IP stack. The CAN implementation currently uses a static setup. On CAN, the most important parameter is the bit rate. Traditionally, the bit rate is fixed a priori by the system designer. Still, auto–detection techniques have been developed for this, e.g. as presented in [147]. Thus, COSMIC for CAN could also employ a truly dynamic setup. Further information, like the employed processor, hardware and software version, and a plain text description of the device and its features are mainly for documentation purposes. The `FullInformation` element contains a URI to a site that provides at least a copy of the description, and possibly further information about the device, e.g. a set of pictures, and some application notes.

The `GeneralInformation` block is of particular interest during the integration and maintenance phases of the component life cycle. However, it also contains information of how to dispose of the device. This includes hints for dismantling, as well as information about hazardous materials used in the device, and their handling. For pure software components, the `RecyclingInformation` element holds removal instructions. In most current systems, disposal information is neglected. None of the approaches presented in chapter 5 considers the disposal of the respective components. For software systems, this often results in symptoms described as *bit rot* [153] or *DLL hell* [154] [42].

```
<xs:element name="GeneralInformation">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="NodeUID" type="CODESID"/>
      <xs:element name="DeviceName" type="xs:string"/>
      <xs:element name="DeviceType" type="xs:string"/>
      <xs:element name="Manufacturer" type="xs:string"/>
      <xs:element name="Processor" type="xs:string"/>
      <xs:element ref="OperationalConnection"
```

```
                           maxOccurs="unbounded"/>
      <xs:element name="HardwareVersion" type="xs:string"/>
      <xs:element name="SystemSoftwareVersion"
                                   type="xs:string"/>
      <xs:element name="Description" type="xs:string"
                                      minOccurs="0"/>
      <xs:element name="FullInformation" type="xs:anyURI"/>
      <xs:element ref="SupportedChannelTypes"/>
      <xs:element ref="RecyclingInformation"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="RecyclingInformation">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Hint" type="xs:string" minOccurs="0"
                             maxOccurs="unbounded"/>
      <xs:element name="HazardousMaterial" type="xs:string"
                      minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="Dismantling" type="xs:string"
                      minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="OperationalConnection">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element ref="CommunicationParameters"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Normally, each element listed in a `sequence` must appear exactly once. The addition of `minOccurs="0"` marks an element as optional; the addition of `maxOccurs="`*x*`"` allows to specify an arbitrary number *x* (larger than 0) or appearances. If an indefinite number of appearances is allowed, *x* is given as `unbounded`. Both attributes can be combined for an element.

```
<xs:element name="SupportedChannelTypes">
  <xs:complexType>
    <xs:element name="EventChannelType" type="channeltype"
                                      maxOccurs="3"/>
  </xs:complexType>
</xs:element>

<xs:simpleType name="channeltype">
  <xs:restriction base="xs:string">
    <xs:enumeration value="HRT"/>
    <xs:enumeration value="SRT"/>
    <xs:enumeration value="NRT"/>
  </xs:restriction>
</xs:simpleType>
```

Nodes and events are distinguished based on their unique IDs. These IDs must be globally unique for unique kinds of events, and for each device. Currently, the implementation uses 64 bit identifiers. As longer identifiers may be desirable for future implementations, e.g. standardized and well–known 128 bit UUIDs, it should be possible to move from one to the other. As XML Schema allows the definition of user defined types, the identifiers' type can easily be adjusted. The currently used 64 bit identifiers must be given in hexadecimal notation:

```
<xs:simpleType name="CODESID">
  <xs:restriction base="xs:string">
    <xs:pattern value="0x[0-9A-Fa-f]{16}"/>
  </xs:restriction>
</xs:simpleType>
```

As mentioned above, the list of `CommunicationParameters` is a list of name–value–pairs. For each parameter, its name, the associated value and the value's unit is given. Like the attribute lists presented later on, the communication parameters' units are represented in a special, machine–readable manner.

```
<xs:element name="CommunicationParameters">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="CommunicationParameter"
                       maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="CommunicationParameter">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Value" type="xs:string" />
      <xs:element ref="Dimension" minOccurs="0"/>
      <xs:element name="Description" type="xs:string"
                                    minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Event Definition

The `EventDefinitions` part contains a list of all events that the device exchanges, both incoming and outgoing.

```
<xs:element name="EventDefinitions">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Event" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Each event is described according to COSMIC's model as discussed in section 6.1.1. The event's subject is given both as a string and a unique ID. The plain text subject carries a hint towards the event's semantics. The unique ID is the identifier used by the COSMIC middleware, as discussed in section 6.2.6. Across the world, there may be multiple UIDs for the same subject. Each UID, on the other hand, uniquely identifies a certain subject. The `Description` element complements the subject by more detailed information for the user. The structure of the list of attributes and the event's data structure (its *contents*) is discussed below. The attributes necessary for each event are presented in section 6.1.1.

As introduced before, COSMIC devices act autonomously. They consume and produce events. For some applications, e.g. for control loops, it is necessary to have components that exhibit a certain behavioral pattern that cannot be expressed by COSMIC primitives. Consider a regulation component consuming some measurement as input, and producing output that is in turn used by an actuation component. The `MayTrigger` and `WillTrigger` elements can be used to model a flow path for such control loops. `MayTrigger` contains subject UIDs that name the types of events that may be created as a result of the appearance of the current event. Accordingly, `WillTrigger` lists the types of events that will be created as a result of the appearance of the current event.

```
<xs:element name="Event">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Subject" type="xs:string" />
      <xs:element name="SubjectUID" type="CODESID" />
      <xs:element name="Description" type="xs:string"
                                       minOccurs="0" />
      <xs:element ref="Attributes" />
      <xs:element ref="DataStructure" />
      <xs:element name="MayTrigger" type="SUIDList"
                                       minOccurs="0" />
      <xs:element name="WillTrigger" type="SUIDList"
                                       minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="SUIDList">
  <xs:sequence>
    <xs:element name="SubjectUID" type="CODESID"
                                 maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

The attributes associated with an event are grouped together in a list:

```
<xs:element name="Attributes">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Attribute" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

There are two possibilities to specify attributes:

1. specify a generic `<Attribute>` element
2. specify an element for each kind of attribute, e.g. `<Expiration-Time>`.

The former keeps both the description and the schema readable, but completely looses the possibility to check the description for missing attributes, e.g. an event's expiration time, by simply validating it against the CODES schema. In this case, the need for an extra tool to check the descriptions arises. This tool would need to have an implicit knowledge about needed event attributes, or it would need this knowledge explicitly written down somewhere — both of which is not desirable as it splits up the needed information into two parts: the tool and the schema. The latter way largely retains the possibility to check for all needed attributes by validating against the CODES schema, as all needed information is explicitly stated. Unfortunately, it also enlarges the CODES schema considerably. It therefore becomes more unreadable and more difficult to maintain.

Attributes need to be defined for several entities in CODES: Events, data fields, and event channels. To allow extensibility and ease of implementation, attributes are encoded directly as generic name–value–tuples. In most cases, the values have a unit associated with them, which is also part of the attribute's description.

The unit is represented in a machine–readable way to enable dynamic interaction. An attribute's definition is complemented by a clear–text description of its meaning. Finally, a tag marking the attribute as either *functional* or *non–functional* is used to further classify the attribute:

```
<xs:element name="Attribute">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element ref="Dimension" minOccurs="0"/>
      <xs:element name="Value" type="xs:string" minOccurs="0"/>
      <xs:element name="Description" type="xs:string"
                                                minOccurs="0"/>
      <xs:element name="Functional" type="emptytag"
                                                minOccurs="0"/>
      <xs:element name="Nonfunctional" type="emptytag"
                                                minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:simpleType name="emptytag">
  <xs:restriction base="xs:string">
```

```
    <xs:length value="0">
  </xs:restriction>
</xs:simpleType>
```

The physical units are represented in a way similar to IEEE 1451 TEDS. The unit itself is represented in terms of the base units from the *système international d'unités* (SI) [150], complemented by the pseudo–units *radians* and *steradians*[1] for angular values, as well as *bits* for information. The latter allows to encode e.g. the speed of communication. While this is not a traditional item encountered in real–world system models, it is often needed in digital systems. For each SI base unit, its exponent is encoded:

$$<unit>=<baseunit_1>^{exponent} \quad <baseunit_2>^{exponent} \ldots$$

For logarithmic values, the logarithm's base must be specified. Some entities may be dimension–less technically, as they are of the kind $\frac{unit}{unit}$. In this case, the optional element `IsDimensionless` must be set to `true`. For practical use, the magnitude, a scaling factor, and possibly a constant offset are necessary, yielding:

$$<value> \cdot scaling \cdot 10^{magnitude}[log_{base}] <unit>$$

$$+offset \cdot scaling \cdot 10^{magnitude}[log_{base}] <unit>$$

```
<xs:element name="Dimension">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="SIUnit"/>
      <xs:element name="Magnitude" type="xs:double"
                                           minOccurs="0"/>
      <xs:element name="Offset" type="xs:double"
                                           minOccurs="0"/>
      <xs:element name="Scaling" type="xs:double"
                                           minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="SIUnit">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Radians" type="xs:double"
                                           minOccurs="0"/>
      <xs:element name="Steradians" type="xs:double"
                                           minOccurs="0"/>
      <xs:element name="Bits" type="xs:double"
                                           minOccurs="0"/>
      <xs:element name="Meters" type="xs:double"
                                           minOccurs="0"/>
      <xs:element name="Kilograms" type="xs:double"
                                           minOccurs="0"/>
```

[1]Steradians are e.g. used in radar technology [125].

```
      <xs:element name="Seconds" type="xs:double"
                                             minOccurs="0"/>
      <xs:element name="Amperes" type="xs:double"
                                             minOccurs="0"/>
      <xs:element name="Kelvins" type="xs:double"
                                             minOccurs="0"/>
      <xs:element name="Moles" type="xs:double"
                                             minOccurs="0"/>
      <xs:element name="Candelas" type="xs:double"
                                             minOccurs="0"/>
      <xs:element name="LogarithmicBase" type="xs:double"
                                             minOccurs="0"/>
  <xs:element name="IsDimensionless" type="xs:boolean"
                                             minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The following three examples illustrate the flexibility and expressiveness of this construction. This encoding technique is not only used for attributes, but also for the individual data fields of an event's contents, as described below.

Consider a temperature sensor publishing its data in degrees Centigrade, which would be encoded as:

```
<Dimension>
  <SIUnit>
    <Kelvins>1</Kelvins>
  </SIUnit>
  <Offset>273.15</Offset>
</Dimension>
```

The second example is a distance in inch:

```
<Dimension>
  <SIUnit>
    <Meters>1</Meters>
  </SIUnit>
  <Magnitude>-2</Magnitude>
  <Scaling>2.54</Scaling>
</Dimension>
```

Finally, an acceleration sensor producing milli–G is encoded as:

```
<Dimension>
  <SIUnit>
    <Meters>1</Meters>
    <Seconds>-2</Seconds>
  </SIUnit>
  <Magnitude>-3</Magnitude>
</Dimension>
```

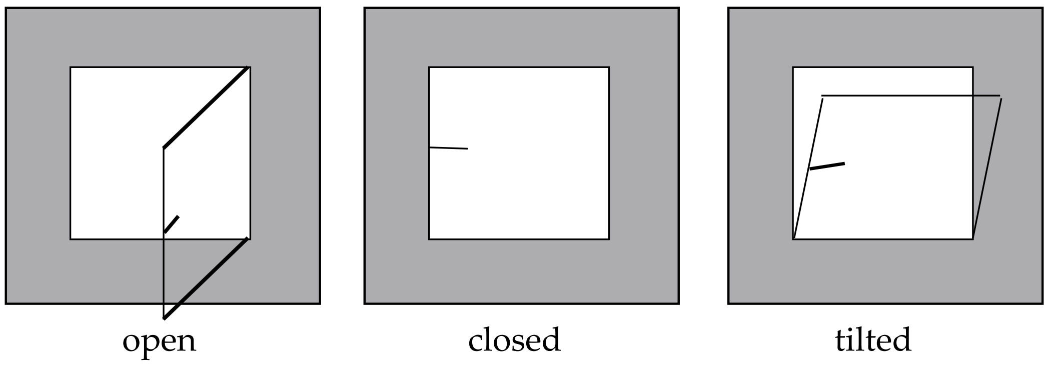


Figure 7.2: A window sensor that can determine the state of the window.

The final part of the event definition is concerned with the event's contents. Since in COSMIC environments communication bandwidth mostly comes at rather high cost, messages used for operational communication must be compact. Thus, binary data structures are communicated. This requires a detailed, non ambiguous definition of the data structures.

For each event, the data structure is described as a list of consecutive data fields. Each data field has a name carrying a hint towards the field's semantics. A plain–text description complements this information for the user. The field's data type, as well as the employed byte order (if the field is larger than one byte) is specified. Thus, any communicated value can be decoded.

To correctly interpret the decoded values, more information about their meaning must be given. Depending on the type of entity represented by the field, different information is necessary. There are two types of entities: continuous and discrete entities. The first type corresponds to most measurements of real–world entities, e.g. temperature or voltage measured by an A/D converter (ADC). Discrete entities on the other hand convey some form of "digital" data. Discrete entities can be of two sub–types: One sub–type groups together entities where the individual values do not exhibit any relations among each other. As an example for this type of entity, consider a mode switching facility. A data field of an event is used to switch the receiving node into either normal operations mode, or a special debug mode. The second sub–type groups together those entities where the individual values exhibit relations among themselves. As an example, consider a window sensor. The window can either be open, closed, or tilted (see figure 7.2). The window sensor will produce an event with a single data field that contains a representation of the current window state, i.e. whether it is open, closed, or tilted. Note that it is physically not possible for the window to change directly from the open state to the tilted state. A state machine can be used to model the relations among the individual values of such an entity.

In CODES descriptions, the corresponding physical unit is given for continuous entities. The same structures as discussed above with attributes' units are used.

Discrete entities with no relations among the individual values are represented by `Enum` fields. The structures for `Enum` fields are introduced below the schema part representing the data fields and available data types.

Discrete entities with relations among the individual values are represented by `StateMachine` fields. Their structures are introduced below the `Enum` fields.

Finally, for each field, the list of attributes contains information about the valid data ranges, and the field's resolution and precision.

```
<xs:element name="DataStructure">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Field" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Field">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Description" type="xs:string"
                                          minOccurs="0"/>
      <xs:element name="DataType" type="COSMICDataTypes"/>
      <xs:element name="ByteOrder" minOccurs="0">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="BigEndian" />
            <xs:enumeration value="LittleEndian" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element ref="Dimension" minOccurs="0"/>
      <xs:element ref="Enum" minOccurs="0"/>
      <xs:element ref="StateMachine" minOccurs="0"/>
      <xs:element ref="Attributes" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

For the description of a field's data encoding, the list of data types is defined:

```
<xs:simpleType name="COSMICDataTypes">
  <xs:restriction base="xs:string">
    <xs:enumeration value="u_int_8"/>
    <xs:enumeration value="u_int_16"/>
    <xs:enumeration value="u_int_32"/>
    <xs:enumeration value="u_int_64"/>
    <xs:enumeration value="int_8"/>
    <xs:enumeration value="int_16"/>
    <xs:enumeration value="int_32"/>
    <xs:enumeration value="int_64"/>
    <xs:enumeration value="ieee_754_32"/>
    <xs:enumeration value="ieee_754_64"/>
  </xs:restriction>
</xs:simpleType>
```

The large number of integer types, both in signed and unsigned variants, compared to floating point types is due to COSMIC's focus on small devices. These devices cannot handle floating point formats efficiently, and thus they are avoided whenever possible. Integers, however are available in all common widths to support a compact layout of the data structure. As a trade–off towards keeping the number of types down and ease of handling of the available types, the possibility to define fields of arbitrary bit width was dropped. Thus by keeping the smallest unit of information to a byte, this avoids confusion about bit numbering within a byte. Since most events are typically composed of only a few data fields, the resulting overhead was deemed feasible. In some cases, the use of floating point numbers may be justified. Therefore, two standard floating point formats have been introduced.

In many modern programming languages, the valid range for a given variable can be encoded in the type, e.g. in C++[2] or Pascal–derived languages. For the CODES descriptions, the data type specified for a data field is only of interest for the encoding of the field within the event. The valid data ranges are specified in the attributes of the event. This was chosen for better readability, and less processing overhead compared to the introduction of a more flexible type scheme based on XML schema data types.

Note that there is no native data type for handling strings. There is usually no need to communicate using strings in systems as targeted by COSMIC. The usage of strings would be contrary to the usage of compact, binary–encoded messages. If the need for strings should arise, a fixed–length buffer of `u_int_8` fields could be reserved, similar to handling strings in C.

Discrete field types

Enum data fields, as mentioned above, can be handy for implementing technical artifacts, such as simple mode switch commands for a COSMIC device. Such a command could e.g. enable a debugging mode. While this example command would not be necessary for the operation of a system running flawlessly, it is often necessary in practice to allow the tracing of an error. Since such commands are part of the device's interface, they should also be part of the device's description. The description of an enum field can either be a list of value–description–pairs listing all valid values individually, or it can be a range specifying the lower and upper bounds of the valid data range, complemented by a description of the field.

[2]For an example on how to achieve this in C++, see [126], chapter 11.4

```
<xs:element name="Enum">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Item" minOccurs="0"
                                 maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Value" type="xs:integer" />
            <xs:element name="Description" type="xs:string"
                                         minOccurs="0" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Range" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="LowerBound" type="xs:integer" />
            <xs:element name="UpperBound" type="xs:integer" />
            <xs:element name="Description" type="xs:string"
                                            minOccurs="0" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The individual valid values of an enum field do not have any relation among themselves. If such relations exist, it is better to specify such a field as a state machine. The main application of state machine fields are discrete sensors, however. Consider the window example from above, again. Whenever the state of the window sensor changes, an event is generated. The event contains the new state of the window sensor. Thus, if the receiver of the event gets the event, it not only knows what the current state is, but from the memory of the previous state and the description, it can also deduce what transition has been activated. More importantly, awareness of errors and omissions is raised. If the receiver does not find a valid transition from the previous state to the current state, it knows that it either missed an event, or that the producer is erroneous. By sending the current state instead of the activated transition, it is ensured that the system will re–sync itself again after an event omission, independent of the location of the omission. Technically, a state machine field is described by a list of all known states. Each state is defined by its name, its internal representation (a number), and a description explaining the state's purpose. All valid transitions are described in a separate list. Each transition also has a name. The originating and destination states are obviously part of the transition's definition. Additionally, a condition that must be met for the transition to be activated can be given. This condition is only informative; there is currently no formal way to assert that the condition was met when a transition was activated. Also, the transition's definition is complemented by a clear–text description.

```
<xs:element name="StateMachine">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="State" maxOccurs="unbounded"/>
      <xs:element ref="Transition" maxOccurs="unbounded"/>
      <xs:element name="InitialState" type="xs:integer"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="State">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Representation" type="xs:integer"/>
      <xs:element name="Description" type="xs:string"
                                                minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Transition">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name" type="xs:string" minOccurs="0"/>
      <xs:element name="FromState" type="xs:integer"/>
      <xs:element name="ToState" type="xs:integer"/>
      <xs:element name="Condition" type="xs:string"
                                                   minOccurs="0"/>
      <xs:element name="Description" type="xs:string"
                                                   minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Note that the rule that all states named in transitions must actually be declared cannot be expressed using XML schema. This is a nice example where context–free grammars alone are not enough.

Event Channel Definition

The `EventChannelDefinitions` section of the descriptions associates the previously defined events with their event channel properties. Thus each entry in the list of event channels consists of the subject ID linking the entry to the corresponding event definition, the type of event channel used, the direction as seen locally, i.e. whether the device produces or consumes this type of events, and a list of dissemination properties. The event channel type is one of the three channel classes hard real–time, soft real–time, and non real–time, as introduced in section 6.1.2. The dissemination properties of interest are listed in the same section.

```
<xs:element name="EventChannelDefinitions">
  <xs:complexType>
```

```
    <xs:sequence>
      <xs:element ref="EventChannel" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="EventChannel">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="SubjectUID" type="CODESID"/>
      <xs:element name="EventChannelType" type="channeltype"/>
      <xs:element name="Direction" type="direction"/>
      <xs:element ref="Attributes" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:simpleType name="channeltype">
  <xs:restriction base="xs:string">
    <xs:enumeration value="HRT"/>
    <xs:enumeration value="SRT"/>
    <xs:enumeration value="NRT"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="direction">
  <xs:restriction base="xs:string">
    <xs:enumeration value="consuming"/>
    <xs:enumeration value="producing"/>
  </xs:restriction>
</xs:simpleType>
```

7.2.2 Parameterization

A number of parameters cannot be fixed at design time when the description of a component is created. Specifically, this includes:

- The node's UID. If more than one node of a specific kind is produced, each must have its own unique ID. It would be inefficient to manually change the description for each new device that is being manufactured.

- Some event attributes. An event's expiration time, and its criticality are application dependent and therefore cannot be specified before the integration phase.

- Event channel attributes. The period, with which an event is disseminated is also not available before the integration phase.

As parameters throughout the description must be variable, a generic method for marking parameters in the otherwise static description must be introduced. The solution is to specify `<parameterized>` instead of a specific value in the description, e.g. to have a parameterized node id, one would specify:

```
<NodeUID>
  <parameterized/>
</NodeUID>
```

Such ”incomplete” descriptions will not validate against the presented schema. However, once these descriptions are completed with the missing information, they will validate. How this integration can be achieved is presented during the following discussion.

The corresponding parameter definition file contains pairs of path expressions to the parameter in the device description, and their actual value, e.g.:

```
<Parameter>
  <Path>/CODES/GeneralInformation/NodeUID</Path>
  <Value>0xc4d70e32024b776d</Value>
</Parameter>
```

The path expressions are XPath expressions (see section 2.4.3). This allows to easily specify access to the correct parameter. The parameter file, and more importantly the correct path expression can automatically be generated by a description editor (For this work, a description editor called *CODESCreator* has been developed. It is presented in section 7.3.1). For parameters embedded deeply, this path expression can grow rather complicated, and having it automatically generated avoids many errors. Consider e.g. the expiration time of an event. As each device may specify an arbitrary number of events, the path is not as straight forward as in the example above. Fortunately, XPath 2.0 provides enough expressiveness to tackle the task without complications. The following describes the path to the expiration time attribute of a specific event:

```
/CODES/EventDefinitions/Event[SubjectUID='0xc4d70e32024b77b9']/
    Attributes/Attribute[Name='Expiration time']/Value
```

The availability of an XPath expression addressing the correct element, and its configured value, allows to create a (XSLT based) tool that replaces the `<parameterized>` markings in the description with their actual values. Having such a complete description without any external dependencies available is important for e.g. validation, the query service, and compatibility checks across a number of nodes.

Just like the description file, the parameter file is stored within the device itself, and it can be retrieved in the same fashion. Additionally, it must be possible to store updated parameter files on the device.

Since COSMIC targets small systems, some devices are not powerful enough to process these XML pieces. Thus, a suitable adjustment to the parameterization scheme is proposed: Instead of specifying the parameter’s value in the parameter file, an address, type and encoding information for parameter storage are given. The address is a 16 bit address relative to a parameter storage area in the device. A 16 bit address provides plenty of space. This mechanism is only needed for small devices incapable of processing XML. Mostly, this concerns 8 bit devices, which mostly only have a 16 bit address space, anyway. Even for 16 bit devices, 64 kilobytes of parameter storage will be more than enough; it would almost suffice for the complete description. 8 bit addresses, on the other hand, could easily become too restricting. Assuming an average parameter size of 6 bytes, only 42 parameters could be stored. The parameter's type information describes the parameter value's data type in terms of COSMIC data types. Finally, for values larger than 8 bits in size, the encoding is given. Taken the above example, the parameter file would contain the following entries:

```
<Parameter>
  <Path>/CODES/GeneralInformation/NodeUID</Path>
  <Address>0xf000</Address>
  <DataType>u_int_64</DataType>
  <Encoding>BigEndian</Encoding>
</Parameter>
```

These parameter files are static, and they can be stored in the device without the need of changing them. A tool will then be able to retrieve and set the corresponding parameter values via a dedicated interface on the device.

This parameter access interface can e.g. be handled using three dedicated non real–time event channels. The *parameter request events* would carry four data fields:

- An identification of the target device.
- The address at which the targeted parameter starts.
- The size of the parameter in bytes.
- A request tag for tracking multiple requests.

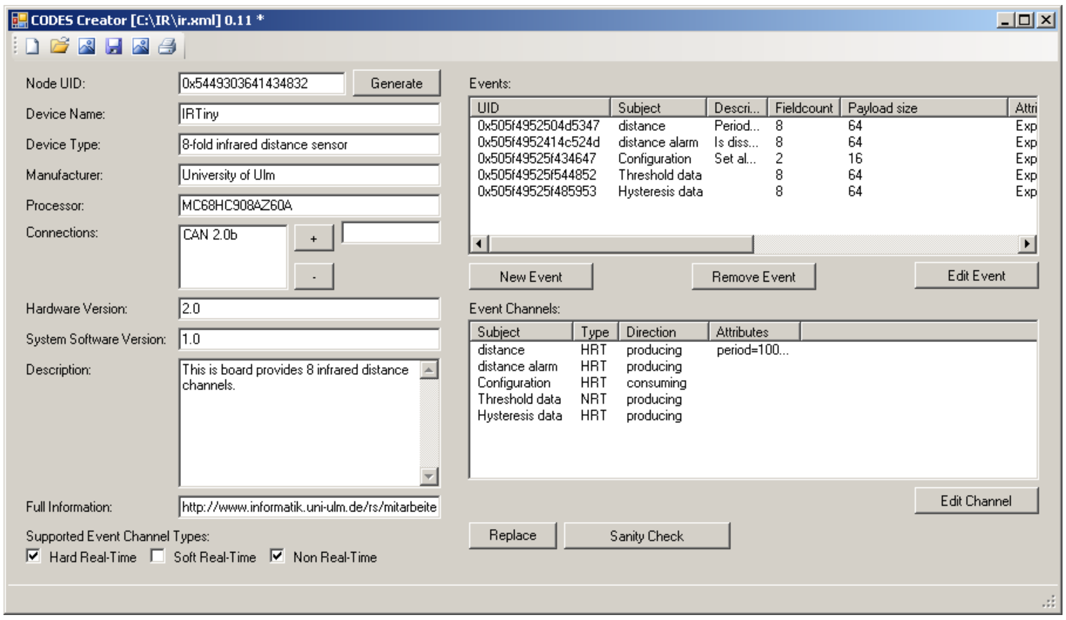
The target device would then answer on the data reply channel with events carrying two fields:

- The corresponding request tag.
- The requested data.

Finally, the *set parameter data events* carry the same information as the parameter request events. Additionally, they carry the new parameter data.

Thus, to build a complete description of a small device, three steps need to be taken:

1. Retrieve the static description and the parameter file.
2. Retrieve each parameter listed in the parameter file.
3. Combine the static description and the parameters into a single document that contains a current snapshot of the complete device configuration.


CODES Creator [C:\IR\ir.xml] 0.11 *

Node UID:	0x5449303641434832
Device Name:	IRTiny
Device Type:	8-fold infrared distance sensor
Manufacturer:	University of Ulm
Processor:	MC68HC908AZ60A
Connections:	CAN 2.0b
Hardware Version:	2.0
System Software Version:	1.0
Description:	This is board provides 8 infrared distance channels.
Full Information:	http://www.informatik.uni-ulm.de/rs/mitarbeite

Supported Event Channel Types:
[✓ Hard Real-Time] Soft Real-Time [✓ Non Real-Time]

Events:

UID	Subject	Descri...	Fieldcount	Payload size	Attri
0x505f4952504d5347	distance	Period...	8	64	Exp
0x505f4952414c524d	distance alarm	Is diss...	8	64	Exp
0x505f49525f434647	Configuration	Set al...	2	16	Exp
0x505f49525f544852	Threshold data		8	64	Exp
0x505f49525f485953	Hysteresis data		8	64	Exp

Event Channels:

Subject	Type	Direction	Attributes
distance	HRT	producing	period=100...
distance alarm	HRT	producing	
Configuration	HRT	consuming	
Threshold data	NRT	producing	
Hysteresis data	HRT	producing	

Figure 7.3: CODES Creator

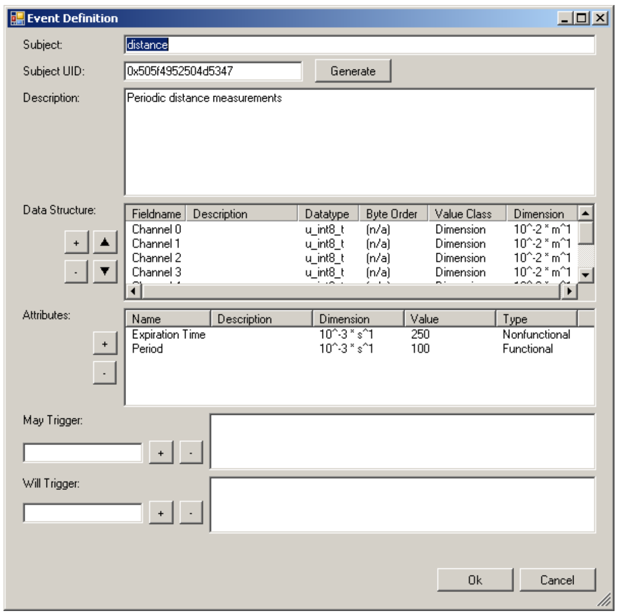
7.3 The CODES Tool–Chain

The CODES tool–chain complements the CODES descriptions to support the life cycle of COSMIC components as outlined at the beginning of this chapter and section 2.1.1. *CODESCreator* is an editor for CODES descriptions. Thus, the design phase of a component is supported. The implementation phase is represented by a set of XSLT transformations *generating* parts of the component's *code*. The *Query Service* is beneficial during the integration, usage, and maintenance phases. For a given system, the Query Service is the central instance where all descriptions — including all parameters — are readily available. *COSMICMonitor* is a general purpose monitoring and logging tool which is mainly useful during integration and maintenance. The *LogPlayer* is able to replay previously recorded logs, e.g. for debugging purposes.

7.3.1 CODESCreator

CODESCreator is an editor for CODES descriptions, i.e. it is part of the component design phase. It requires the .NET 2.0 framework to run. Instead of having to write the specifications in a text editor, the component designer is provided with CODESCreator (see figure 7.3) that ensures syntactical correctness and (technical) completeness of the produced specification. Technical completeness means that all elements required by the schema are present in the description document.

Besides the simple and straight forward editing facilities for the section on general information about the component, CODESCreator manages an



Subject: distance

Subject UID: 0x505f4952504d5347

Description: Periodic distance measurements

Data Structure:

Fieldname	Description	Datatype	Byte Order	Value Class	Dimension
Channel 0		u_int8_t	(n/a)	Dimension	10^-2 * m^1
Channel 1		u_int8_t	(n/a)	Dimension	10^-2 * m^1
Channel 2		u_int8_t	(n/a)	Dimension	10^-2 * m^1
Channel 3		u_int8_t	(n/a)	Dimension	10^-2 * m^1

Attributes:

Name	Description	Dimension	Value	Type
Expiration Time		10^-3 * s^1	250	Nonfunctional
Period		10^-3 * s^1	100	Functional

Figure 7.4: Event Definition

automatically increased version number of the specification file. For events and event channels, more sophisticated functionality is available. Event definitions and Event Channel definitions are automatically kept in a consistent state. More specifically, event definitions can be created and removed. Event Channels cannot be created or removed manually; this is done automatically.

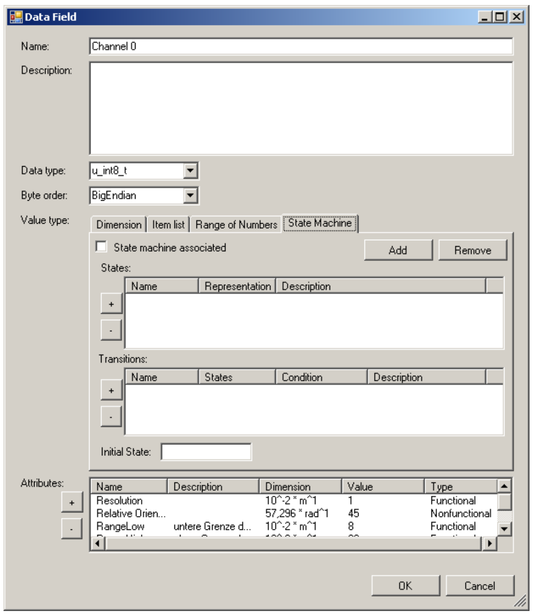
Defining events (see figure 7.4) is supported by providing the user with functionality for specifying individual data fields (see figure 7.5), rearranging them within the data structure, and specifying event related attributes (see figure 7.6).

The same dialog is used for data field related attributes and event channel attributes.

Attributes that are required for data fields, events, and event channels can be pre–defined in configuration files using the same XML fragments that are part of the description at the respective position. These configuration files are read on startup. Whenever a new data field, event, or event channel is created, the pre–defined attributes are automatically included in the description in their respective place. For data fields, the following attributes are currently required:

- The *lower bound* of the valid data range.
- The *upper bound* of the valid data range.
- The *resolution* of the data.
- The *precision* of the data.

Similarly, for events the following attributes are required:



Data Field

Name: Channel 0

Description:

Data type: u_int8_t

Byte order: BigEndian

Value type: Dimension | Item list | Range of Numbers | State Machine

State machine associated

States:

Name	Representation	Description

Transitions:

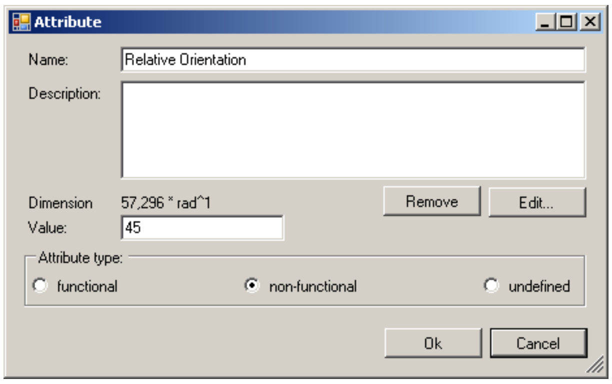
Name	States	Condition	Description

Initial State:

Attributes:

Name	Description	Dimension	Value	Type
Resolution		10^-2 * m^1	1	Functional
Relative Orien...		57,296 * rad^1	45	Nonfunctional
RangeLow	untere Grenze d...	10^-2 * m^1	8	Functional

Figure 7.5: Field Definition



Attribute

Name: Relative Orientation

Description:

Dimension 57,296 * rad^1

Value: 45

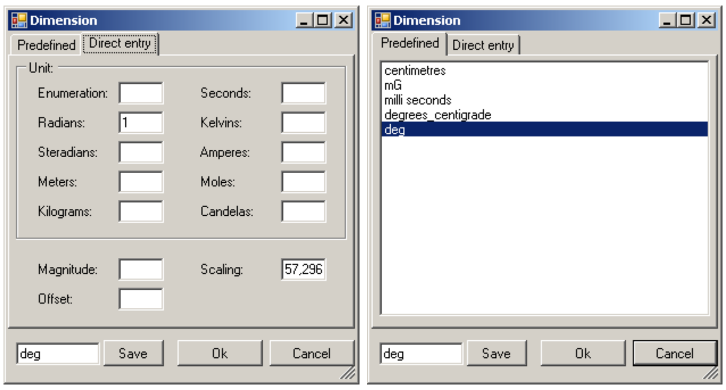
Attribute type: functional [✓ non-functional] undefined

Figure 7.6: Attribute Definition

- The *expiration time* of the event. If the event expires, it may potentially have harmful effects on the system, as described in section 6.1.1.
- The *periodicity* of the event.

Finally, for event channels, currently only the dissemination *period* is required.

Similarly, wherever physical dimensions are needed, the dialog in figure 7.7 is used. It allows the user to either select a dimension from a predefined list, or to specify all parameters directly. Using pre–defined entries saves the user from tedious work and avoids typos. Entries to this list can be added. They are saved to an XML file that is read on application startup, exactly like the files with the default attributes. The XML file contains a list of XML fragments adhering to the respective part in the CODES schema.


Dimension

Predefined | Direct entry

Unit:

Enumeration:		Seconds:	
Radians:	1	Kelvins:	
Steradians:		Amperes:	
Meters:		Moles:	
Kilograms:		Candelas:	
Magnitude:		Scaling:	57,296
Offset:			

deg Save Ok Cancel

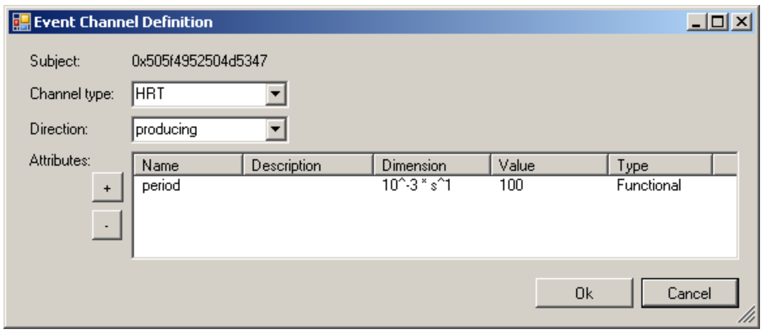
Dimension

Predefined | Direct entry

centimetres
mG
milli seconds
degrees_centigrade
deg

deg Save Ok Cancel

Figure 7.7: Dialog for physical dimensions


Event Channel Definition

Subject: 0x505f4952504d5347

Channel type: HRT

Direction: producing

Attributes:

Name	Description	Dimension	Value	Type
period		10^-3 * s^1	100	Functional

Figure 7.8: Event Channel Definition

Event Channels and their properties, along with their associated attributes are editable (see figure 7.8). CODESCreator automatically synchronizes events and their corresponding event channels, i.e. whenever an event is created, CODESCreator automatically creates the corresponding event channel. The same behavior is implemented in case of event deletion.

Additionally, CODESCreator is able to conduct consistency checks on the current description. The consistency checks are described in section 7.4.1.

7.3.2 Code Generation

As part of the implementation phase, code skeletons for the component in development can be generated from its CODES description [75]. The type of code generated from the descriptions is mainly house–keeping code. It is not the goal to produce complete applications or sophisticated algorithmic code as tools like Matlab do.

Code generation is done using XSLT transformations (see chapter 2.4.4). XSLT transformations are ideally suited for this job. They allow the programmer to focus on the task at hand. The example presented below shows that such a transformation can be written in the form of a template of the target language. This is a major advantage compared to writing custom conversion tools in traditional programming languages like C, C++ or Java where the programmer has to deal with rather low–level concepts, such as the APIs of XML parsers. Also, modifying such tools is harder than changing the target templates.

The following example is part of the prototype implementation. Since C is used for most projects in our group, it has been chosen as the target language for the prototype implementation, and hence, this example. Also, C is the most common programming language for microcontrollers. Specifically, the example shows how the `struct`s representing the events are created. Each field is annotated by a comment documenting the physical dimension associated with that value:

```
<xsl:template match="//Event">
struct <xsl:value-of select="replace(string(Subject),'\s+','_')"
/>_event {
  <xsl:for-each select="DataStructure/Field">
    /* <xsl:value-of select="Description"/> */
    <xsl:value-of select="DataType"/><xsl:text> </xsl:text>
      <xsl:value-of select="replace(string(Name),'\s+','_')"/>;
      // <xsl:call-template name="physicaldimension">
        <xsl:with-param name="varname"
                    select="replace(string(Name),'\s+','_')"/>
        </xsl:call-template>
  </xsl:for-each>
};
</xsl:template>
```

The XSLT processor matches this template to any `Event` elements in a CODES description document. The resulting `struct` will be called the event's subject, appended by `_event`. Since C does not allow identifiers to include spaces, a regular expression is used to replace any spaces in the subject with underscores. Next, the template iterates over all fields in the event's data structure and generates the corresponding field declarations. Notably, it is possible to combine pattern matching (as shown here in the form of the top–layer template matching all `Event` elements), and imperative programming style (as shown with the `for-each` loop covering all data fields of the event) in XSLT. This demonstrates how different paradigms can beneficially be integrated into a powerful language, allowing the developer to select the paradigm best suited for the occasion. The comment with the data field's unit shows how even sub–routine like calling is possible via the `call-template` facility. The corresponding template generating the unit's string representation is shown below:

```
<xsl:template name="physicaldimension">
  <xsl:param name="varname" required="yes"/>
  (<xsl:value-of select="$varname"/>
  <xsl:if test="exists(Dimension/Offset)">
    + <xsl:value-of select="Dimension/Offset"/></xsl:if>)
  <xsl:if test="exists(Dimension/Scaling)">
    * <xsl:value-of select="Dimension/Scaling"/> </xsl:if>
  <xsl:if test="exists(Dimension/Magnitude)">
    * 10^<xsl:value-of select="Dimension/Magnitude"/> *</xsl:if>
  <xsl:call-template name="SIUnit"/>
  <xsl:if test="exists(Dimension/SIUnit/IsDimensionless)">
   / (<xsl:call-template name="SIUnit"/></xsl:if>)
</xsl:template>

<xsl:template name="SIUnit">
  <xsl:if test="exists(Dimension/SIUnit/LogarithmicBase)">
   log_{xsl:value-of select="Dimension/SIUnit/LogarithmicBase"/>
   </xsl:if>
  <xsl:if test="exists(Dimension/SIUnit/Radians)">
   Radians^<xsl:value-of select="Dimension/SIUnit/Radians"/>
   </xsl:if>
  <xsl:if test="exists(Dimension/SIUnit/Steradians)">
   Steradians^<xsl:value-of select="Dimension/SIUnit/Steradians"/>
   </xsl:if>
  <xsl:if test="exists(Dimension/SIUnit/Meters)">
   Meters^<xsl:value-of select="Dimension/SIUnit/Meters"/>
   </xsl:if>
  <xsl:if test="exists(Dimension/SIUnit/Kilograms)">
   Kilograms^<xsl:value-of select="Dimension/SIUnit/Kilograms"/>
   </xsl:if>
  <xsl:if test="exists(Dimension/SIUnit/Seconds)">
   Seconds^<xsl:value-of select="Dimension/SIUnit/Seconds"/>
   </xsl:if>
  <xsl:if test="exists(Dimension/SIUnit/Amperes)">
   Amperes^<xsl:value-of select="Dimension/SIUnit/Amperes"/>
   </xsl:if>
  <xsl:if test="exists(Dimension/SIUnit/Kelvins)">
   Kelvins^<xsl:value-of select="Dimension/SIUnit/Kelvins"/>
   </xsl:if>
  <xsl:if test="exists(Dimension/SIUnit/Moles)">
   Moles^<xsl:value-of select="Dimension/SIUnit/Moles"/></xsl:if>
  <xsl:if test="exists(Dimension/SIUnit/Candelas)">
   Candelas^<xsl:value-of select="Dimension/SIUnit/Candelas"/>
   </xsl:if>
</xsl:template>
```

To conclude the example, consider the following (shortened) temperature event:

```
<Event>
  <Subject>Temperature</Subject>
  <SubjectUID>0x1234567890ABCDEF</SubjectUID>
  <Description>Sample temperature event. Contains three fields.
     </Description>
  <DataStructure>
    <Field>
      <Name>outsidetemp</Name>
      <Description>Outdoor temperature</Description>
      <DataType>u_int_8</DataType>
      <Dimension>
        <SIUnit><Kelvins>1.0</Kelvins></SIUnit>
        <Magnitude>0.0</Magnitude>
        <Offset>273.15</Offset>
        <Scaling>1.0</Scaling>
      </Dimension>
      <Attributes> ... </Attributes>
    </Field>
    <Field>
      <Name>outsidetemp_err</Name>
      <Description>Outdoor temperature error</Description>
      <DataType>u_int_8</DataType>
      <Dimension>
        <SIUnit><Kelvins>1.0</Kelvins>
          <IsDimensionLess>true</IsDimensionLess></SIUnit>
        <Magnitude>0.0</Magnitude>
        <Offset>273.15</Offset>
        <Scaling>1.0</Scaling>
      </Dimension>
      <Attributes> ... </Attributes>
    </Field>
    <Field>
      <Name>outsidetemp_log</Name>
      <Description>Outdoor temperature, logarithmic
           </Description>
      <DataType>u_int_16</DataType>
      <ByteOrder>BigEndian</ByteOrder>
      <Dimension>
        <SIUnit><Kelvins>1.0</Kelvins>
          <LogarithmicBase>2</LogarithmicBase></SIUnit>
        <Scaling>10</Scaling>
      </Dimension>
      <Attributes> ... </Attributes>
    </Field>
  </DataStructure>
  ...
</Event>
```

The above XSLT transformation will produce the following `struct` from this description:

```
struct Temperature_event {
  /* Outdoor temperature */
  u_int_8 outsidetemp; // (outsidetemp + 273.15) * 1.0 * 10^0 *
                                                  Kelvins^1.0
  /* Outdoor temperature error */
  u_int_8 outsidetemp_err; // (outsidetemp + 273.15) *
                                  1.0 * 10^0 * Kelvins^1.0 /
                                                 (Kelvins^1.0)
  /* Outdoor temperature, logarithmic */
  u_int_16 outsidetemp_log; // (outsidetemp) * 10 *
                                            log_2 Kelvins^1.0
};
```

Similarly, callback functions, event handlers and tasks handling the communication are part of the prototype implementation. There is one callback function per event being produced by a component. The callback functions are functions that the developer needs to complete by filling the respective event's contents. For each callback function, a task invoking the callback function is created. This task also handles the casting towards the lower–level COSMIC API, specifically for the HC08 in the prototype implementation. For events being consumed, event handler functions are generated that also need to be complemented by the developer. The generated code relieves the developer from doing these low–level housekeeping duties repeatedly. If these parts where programmed manually, it would be very easy to make mistakes, some of which could stay hidden for a long time.

7.3.3 Query Service

In situations of dynamic cooperation, awareness of the available services must be raised. Consider a mobile robot entering a factory building. The robot needs to know, whether the building offers a localization service with a certain precision and update rate. This information can e.g. be used by the robot to adapt the speed with which it moves inside the building. In other words, a fine–grained method of discovering services is necessary.

To limit network usage, any advertising of services is kept to a minimum and usually only contains a reference to a more sophisticated service. This service then allows to either retrieve the service descriptions as such, or it provides more fine–grained mechanisms for processing them at the discovery service itself. The former requires all requesting devices to possess enough processing power to analyze the descriptions, whereas the latter only requires the node running the discovery service to do so. Processing of the description is necessary, as the description needs to be analyzed to determine, whether the corresponding service matches a user's requirements. Independent of where the descriptions are processed, they, or the results of the analysis are only accessed or transmitted on request.

If service discovery would use COSMIC's publish/subscribe communication scheme, there would have to be two event channels, one for sending the requests and another one carrying the results of the requests. Communication on both channels would be correlated in a request/reply style, which is rather uncommon in event–based models [134]. Both channel identifiers would have to be standardized. This shared knowledge would eliminate the need for an initial basic discovery phase that is needed otherwise. The robot entering the building from the example above would then request the information needed whenever appropriate. It would then wait for an answer on the result channel. Whenever more than one device is requesting service information, each device must be able to correctly identify the information targeted at itself, and it must be able to filter out any other information. Due to the anonymous nature of COSMIC's communication scheme, a special identifier relating an event on the result channel to a specific recipient or request would have to be included in each event on the result channel. This requires a device to process all events on the result channel and filter out those events that it actually needs, leading to an unwanted high load on each device, especially during phases of high activity. Thus, a scalability problem arises on all devices listening on the result channel.

The solution discussed below is different from the previously presented alternative. It deliberately does not use COSMIC communication features. Finally, as the technical realization of the discovery mechanisms is not the primary topic of this work, the solution presented has the advantage of being composed of common and tested components that tremendously eased its implementation. The main service discovery is handled by a web service, called the *Query Service*. Before the Query Service can be accessed, its address must be known. This is handled using the *Simple Service Discovery Protocol* (SSDP) [53]. SSDP is based on Multicast, limiting its scope to nearby network segments in terms of Multicast packet routing. As Multicast packets are either limited to the local sub network, or are limited to institutional boundaries, e.g. departments or companies, the reach of Multicast packets mostly corresponds to the physical surroundings and is thus suitable in scoping the discovery mechanisms.

As discussed above, the actual service discovery can either be done locally, after retrieving any service descriptions available, or it can be done at a central service. In order to allow less powerful devices to take advantage of the discovery mechanisms, the latter mechanism was chosen. In order to reflect its capabilities and to distinguish it from the SSDP, the central service was called *Query Service*. It provides an interface for querying the CODES descriptions in COSMIC systems. Conceptually, only a single operation — to execute a transformation on the descriptions — is necessary. Such a central service requires the use of a suitable query language that is used to process the descriptions. The query language must be able to take full advantage of the descriptions. It should also be relatively easy to use. Instead of reinventing such a language, XSLT transformations where chosen. XSLT has several advantages:

- It is a common technology, and therefore rather easy to use for many users.
- Parsers and processors are readily available and well–tested.
- Both the description and the query language are from the standardized XML family of technologies.

In comparison to XQuery [89], the designated information retrieval language for XML, XSLT has the advantage of being in a more stable state in terms of standardization and tool support. Further, XSLT is also useful for other purposes than information retrieval, e.g. for code generation. Thus, queries are actually XSLT transformations that are executed on the CODES descriptions. This allows specifying queries with a granularity down to the most basic elements of the descriptions. The Query Service's interface is composed of the following operations:

- Retrieve a *list of all available descriptions*. This operation is mainly necessary to allow the querying party to retrieve the input needed for the next operation. The result is a list of names of descriptions. In the prototype implementation, these names are actually the file names of the descriptions. The operation does not require any input. It returns an XML structure listing all descriptions.

- Perform a *transformation of a single description*. This operation requires two input parameters: the name of the description to be used in the transformation, and the transformation itself as a string. The operation returns the result of the transformation.

- Perform a *transformation on all descriptions*. This operation takes the transformation to be executed as the only input parameter. Again, the operation returns the result of the transformation. For practical reasons, the operation exists in two variations in the prototype implementation. The main variant combines all descriptions into a (temporary internal) system representation, and executes the transformation on the whole system. The second variant executes the transformation on each description, one at a time. The advantage of this variant is that in case of a single unprocessable description, the others are still processed ok. Using the other variant, the whole process fails, and no usable result is returned.

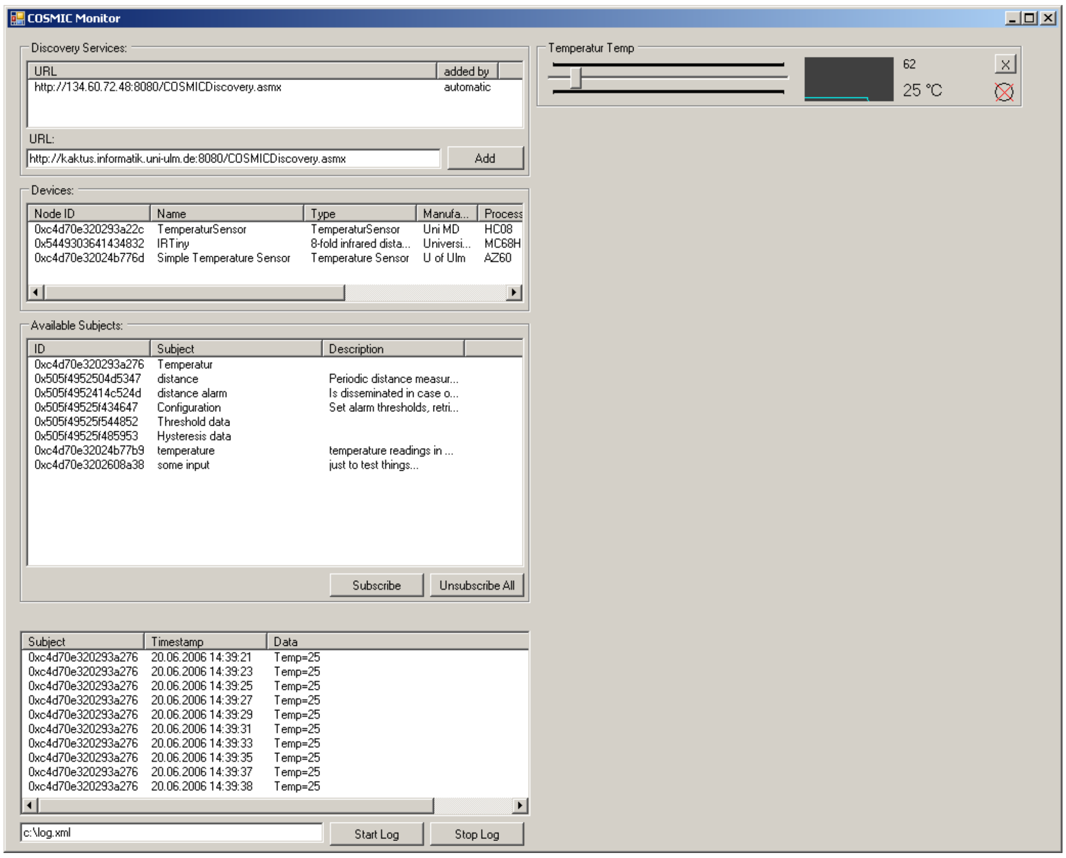
The Query Service needs access to the CODES descriptions. It must therefore retrieve all descriptions, and the corresponding parameter information. The parameter information is then integrated into the descriptions, leaving the description document as the single and complete source of information about the available components.

As the Query Service is a Web service and Web services are only executed on demand, a scheme that collects the CODES descriptions independent of the Query Service has been devised: In current setups, it is running on all gateway nodes that connect a CAN bus to the TCP/IP network. Upon node initialization, the gateway software retrieves the descriptions of all attached nodes and stores them to a description repository. The Query Service then has access to these descriptions.

7.3.4 COSMICMonitor

Logging is not at the heart of COSMIC. Yet, in practice, it plays an important role when developing and debugging systems. In distributed systems, logging can be a hard task [132]. It seldom is possible to perform logging of all communication of a distributed system on one central node. Direct communication among two components addressing each other cannot easily be logged on a third machine. Often, invasive logging techniques are necessary. Of course, this will influence the performance of the system, and it may even influence the behavior of a system if e.g. timing is affected too much.

COSMIC's communication scheme allows non–invasive logging. The logging node is simply another consumer in the system. It subscribes to all events of interest. The events will then be delivered, no matter where they



Discovery Services:

URL	added by
http://134.60.72.48:8080/COSMICDiscovery.asmx	automatic

URL: http://kaktus.informatik.uni-ulm.de:8080/COSMICDiscovery.asmx

Devices:

Node ID	Name	Type	Manufa...	Process
0xc4d70e320293a22c	TemperaturSensor	TemperaturSensor	Uni MD	HC08
0x5449303641434832	IRTiny	8-fold infrared dista...	Universi...	MC68H
0xc4d70e32024b776d	Simple Temperature Sensor	Temperature Sensor	U of Ulm	AZ60

Available Subjects:

ID	Subject	Description
0xc4d70e320293a276	Temperatur	
0x505f4952504d5347	distance	Periodic distance measur...
0x505f4952414c524d	distance alarm	Is disseminated in case o...
0x505f49525f434647	Configuration	Set alarm thresholds, retri...
0x505f49525f544852	Threshold data	
0x505f49525f485953	Hysteresis data	
0xc4d70e32024b77b9	temperature	temperature readings in ...
0xc4d70e3202608a38	some input	just to test things...

Subject	Timestamp	Data
0xc4d70e320293a276	20.06.2006 14:39:21	Temp=25
0xc4d70e320293a276	20.06.2006 14:39:23	Temp=25
0xc4d70e320293a276	20.06.2006 14:39:25	Temp=25
0xc4d70e320293a276	20.06.2006 14:39:27	Temp=25
0xc4d70e320293a276	20.06.2006 14:39:29	Temp=25
0xc4d70e320293a276	20.06.2006 14:39:31	Temp=25
0xc4d70e320293a276	20.06.2006 14:39:33	Temp=25
0xc4d70e320293a276	20.06.2006 14:39:35	Temp=25
0xc4d70e320293a276	20.06.2006 14:39:37	Temp=25
0xc4d70e320293a276	20.06.2006 14:39:38	Temp=25

c:\log.xml

Figure 7.9: Generic COSMIC monitoring and logging application

where generated, or where they are consumed. Thus logging conceptually does not introduce any overhead. Depending on the implementation, minimal overhead may be necessary, when using COSMIC over TCP/IP, e.g. an extra connection must be maintained. For broadcast media, such as the CAN bus, logging can actually be done without any additional overhead.

Because logging uses COSMIC's normal communication features, it is completely transparent for the system. Developers need not worry about including support for logging.

The logger application must know about all events that it should log. There is no special "catch–all" functionality in the COSMIC middleware. Hence, at least all subjects used within the system must be known to the logger. This is difficult to achieve without any further support. With only COSMIC, the logger would require a list of subjects to log. This list would have to be maintained manually, a cumbersome and error–prone task. If knowledge about available subjects would be inherently present in the system, it could be beneficially put to use with the logger application. The CODES descriptions discussed in this chapter hold the necessary information. The on–line Query Service makes them available, and thus conveniently allows to log any events exchanged in the system.

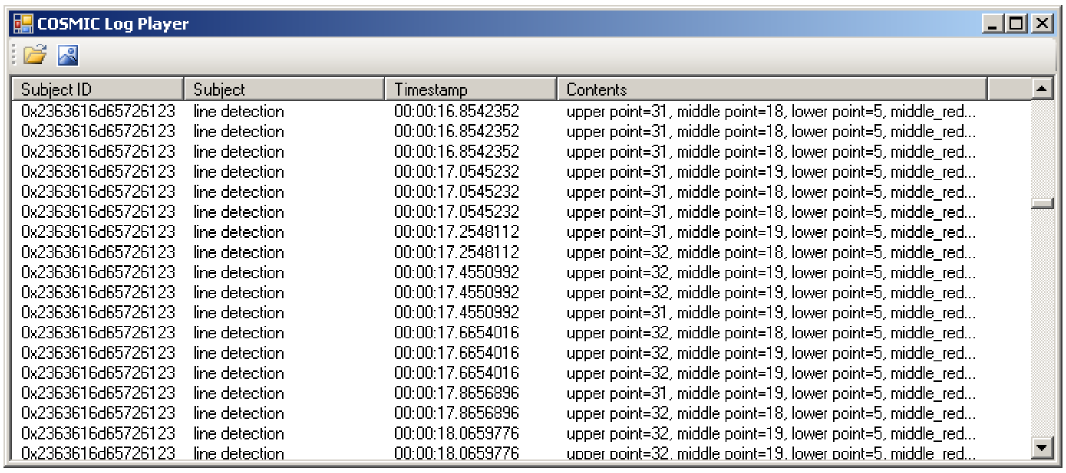
COSMICMonitor is such a generic monitoring and logging application based on COSMIC and CODES. It searches for any reachable Query Services. All known Query Services are consulted for their attached devices. The result, including details about the components, is displayed. Further, a search is performed for all subjects that any of the connected COSMIC components offer. These are also displayed. The user can then select an arbitrary subject an subscribe to it. This will display a visualization of all data fields in the subject's data structure (see right part of figure 7.9). COSMIC-Monitor also allows the logging of all subscribed events to a file. The log file is an XML file that contains the definition of all logged event types. This definition is a shortened version of the definition contained in the CODES descriptions. It does contain all necessary information to decompose and recompose an event's binary structure into/from single fields. Further, the log contains all subscribed events, each annotated with a timestamp. All events are also logged XML fragments, with the event's data in plain text:

```
<EventLog>
  <StartTime>13.09.2006 14:44:32</StartTime>
  <EventDefinition>
    <Subject>Temperatur</Subject>
    <SubjectUID>0xc4d70e320293a276</SubjectUID>
    <Description />
    <DataStructure>
      <Field>
        <Name>Temp</Name>
        <Description />
        <DataType>u_int8_t</DataType>
      </Field>
    </DataStructure>
  </EventDefinition>
  <Event>
    <SubjectUID>0xc4d70e320293a276</SubjectUID>
    <Data>
      <Field>
        <Name>Temp</Name>
        <Value>22</Value>
      </Field>
    </Data>
    <TimeStamp>2300432</TimeStamp>
  </Event>
</EventLog>
```

7.3.5 LogPlayer

The COSMIC Log Player allows to replay pre–recorded logs. This can be very useful when debugging a system. Of course, the logs can manually be modified. It is also possible to create arbitrary events to simulate future devices. A more comfortable player/simulator should be implemented in the future.

A number of features would be desirable to complement the COSMIC Log Player towards a valuable debugging tool:



Subject ID	Subject	Timestamp	Contents
0x2363616d65726123	line detection	00:00:16.8542352	upper point=31, middle point=18, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:16.8542352	upper point=31, middle point=18, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:16.8542352	upper point=31, middle point=18, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:17.0545232	upper point=31, middle point=19, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:17.0545232	upper point=31, middle point=18, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:17.0545232	upper point=31, middle point=18, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:17.2548112	upper point=31, middle point=19, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:17.2548112	upper point=32, middle point=18, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:17.4550992	upper point=32, middle point=19, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:17.4550992	upper point=32, middle point=19, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:17.4550992	upper point=31, middle point=19, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:17.6654016	upper point=32, middle point=18, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:17.6654016	upper point=32, middle point=19, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:17.6654016	upper point=32, middle point=19, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:17.8656896	upper point=31, middle point=19, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:17.8656896	upper point=32, middle point=18, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:18.0659776	upper point=32, middle point=19, lower point=5, middle_red...
0x2363616d65726123	line detection	00:00:18.0659776	upper point=32, middle point=19, lower point=5, middle_red...

Figure 7.10: COSMIC Log Player

- *Varying replay speed.* This feature may be beneficial when visualizing the system for debugging purposes. It should not be used when real components are also involved, e.g. while emulating part of a system, as timing related problems will occur.

- *Selection of start and stop points* within a loaded log. Often, logging is started before anything of interest is happening, and logging continues until after the period of interest is over. Selecting more appropriate start and end points for replay is desirable. Sometimes, even *timeline–based cutting features* known from video processing tools would be helpful.

- *Deactivation of events.* It should be possible to deactivate any of the logged event types during replay. This can be helpful when testing a new version of a component.

- *Automatic generation of events* according to a defined schedule. These events can be encoded using the event's descriptions, which is already used for log replay. The schedule according to which these events are disseminated can either be user–defined, or it can be derived from the corresponding event channel definitions. This can be very beneficial when a new device is under development. It can then already be simulated once its description exists. There is no need for a "real" prototype at this stage.

7.4 Compatibility and consistency checking with CODES descriptions

As pointed out in section 7.2, some rules that apply to CODES descriptions cannot be validated by a syntax check. A number of consistency checks is therefore done to ensure the internal consistency of each description. These checks are presented in the next section.

Further, when integrating multiple components into a system, the compatibility among these components must be ensured. CODES descriptions allow checking a number of properties to ease the integration effort. This is presented in section 7.4.2.

7.4.1 Consistency checks for validating an individual CODES description document

CODES descriptions that adhere to the rules presented in this section are considered to be valid descriptions. Still, such descriptions may not sensibly describe devices, just like a program that compiles does not necessarily do sensible things. Nevertheless having automated checks ensuring the consistency of descriptions with these rules is important.

1. `SupportedChannelTypes` must contain all employed channel types. The `SupportedChannelTypes` must obviously list all types that the device is actually using, otherwise the device cannot provide its service with the described quality.

$$SupportedChannelTypes \supseteq UsedTypes :$$

$$\forall ch \in EventChannels : ch.Type \in UsedTypes.$$

2. All `SubjectUID`s and the `NodeUID` must be unique. Using an appropriate scheme for assignment of UIDs, this can be guaranteed by design. Still, a quick check can easily be done. It is important, as descriptions can be manually created and edited.

$$\forall c_1, c_2 \in EventChannels :$$

$$c_1 \neq c_2 \Rightarrow c_1.SubjectUID \neq c_2.SubjectUID \neq NodeUID$$

3. All Events must have a Channel associated, and vice-versa. As all events are disseminated using event channels, there must be a channel definition for each event definition. Obviously, this applies to the other way round, too.

$$\forall e \in Events : \exists ch \in EventChannels :$$

$$e.SubjectUID = ch.SubjectUID$$

$$\forall ch \in EventChannels : \exists e \in Events :$$

$$ch.SubjectUID = e.SubjectUID$$

4. Each event must define all default attributes. The default attributes are:

   - The *expiration time* defines how long the event is valid, and thus useful for the system. Under no circumstances, an event is delivered after its expiration time. This attribute's physical dimension must be `seconds`, which is also checked.
   - The *periodicity* defines whether the event is generated or expected periodically or sporadically. Its value must either be `sporadic` or `periodic`. In case of sporadic events, an additional attribute `trigger` must be specified. This attribute defines under what conditions the event will be generated. The value is currently given as a plain text description.

$$EventDefaults = \{ExpirationTime, Periodicity\}$$
$$\forall e \in Events : \forall a \in EventDefaults : a \in e.Attributes$$

$$TemporalAttributes = \{ExpirationTime\}$$
$$\forall e \in Events : \forall a \in e.Attributes :$$
$$a \in TemporalAttributes \Rightarrow a.Dimension.unit = seconds$$

5. Each data field must define all default attributes. The default data field attributes are:

   - *Range Low* and *Range High* define the valid data range. In case of event production, the values disseminated will never leave these bounds. In case of event consumption, incoming data is expected to always be within these bounds. The behavior when consuming data outside this range is undefined. Components should provide protection against invalid input, though.
   - *Resolution* specifies the resolution with which values are generated or expected.
   - *Precision* defines how precise the data is, or needs to be in case of consumption.

$$FieldDefaults = \{RangeLow, RangeHigh, Resolution, Precision\}$$
$$\forall e \in Events : \forall f \in e.Fields : \forall a \in FieldDefaults : a \in f.Attributes$$

6. Each field's default attribute must have the same dimension associated as the field's contents.

$$FieldDefaults = \{RangeLow, RangeHigh, Resolution, Precision\}$$
$$\forall e \in Events : \forall f \in e.Fields : \forall a \in f.Attributes :$$
$$a \in FieldDefaults \Rightarrow f.Dimension = a.Dimension$$

7. Each hard real–time event channel must define all default attributes. The default attributes are:

   - *Period* defines the period with which periodic events are produced or expected, respectively. For aperiodic events, this attribute specifies how often a reserved slot needs to be provided. This value is derived from the event's expiration time. The period's physical dimension must be given as seconds.
   - *Omission degree* specifies the allowable probability that an event cannot be delivered on time to all consumers.

$$HRTDefaults = \{Period, OmissionDegree\}$$

$$\forall ec \in EventChannels : ec.ChannelType = HRT :$$
$$\forall a \in HRTDefaults : a \in ec.Attributes$$

7.4.2 Compatibility Tests for multiple CODES description documents

To ease the integration process of components into larger systems, a number of compatibility checks can be performed based on CODES descriptions.

1. All NodeUIDs must be unique. Again, this is guaranteed by the design of the assignment strategy.

$$\forall n_1, n_2 \in Nodes :$$
$$n_1 \neq n_2 \Rightarrow n_1.NodeUID \neq n_2.NodeUID$$

2. When constructing a system, one of the most important properties is *completeness*. For COSMIC systems in particular, this means completeness in terms of (necessary) event production. All types of events that are consumed in the system must also be produced somewhere.

   The system is composed of n components, each with its own CODES c. The set of CODES is:

$$\mathcal{C} = \{c_i\}, i = 1, ..., n$$

   The system-wide set of event channels (EC stands for Event Channels) that are consuming on at least one node is:

$$c.EC^{consuming} = \{c.EC_i : c.EC.direction = consuming\}$$
$$i = 1, ..., |c.EC|$$

   The corresponding set of Subject UIDs:

$$c.SubjectUIDs^{consuming} = \left\{c.EC_i^{consuming}.SubjectUID\right\}$$

$$i = 1, ..., \left|c.EC^{consuming}\right|$$

The set of consumed Subject UIDs:

$$Consumed = \left\{c_i.SubjectUIDs^{consuming}\right\} i = 1, ..., n$$

Correspondingly, the set of produced Subject UIDs:

$$Produced = \left\{c_i.SubjectUIDs^{producing}\right\} i = 1, ..., n$$

The system can only work, if $Produced \supseteq Consumed$.

3. All components must agree on an event's data structure for any matching SubjectUIDs, i.e. all components must share the same knowledge about the respective event types. This includes matching data structures, and matching data types and byte order for each field.

$$\forall c_1, c_2 \in \mathcal{C} : \forall e_1 \in c_1.Events, \forall e_2 \in c_2.Events :$$

$$e_1.SubjectUID = e_2.SubjectUID \Rightarrow$$
$$e_1.DataStructure = e_2.DataStructure$$

4. The expiration time of all produced events must be at least as long as on the consumer. In other words, the consumer specifies what a producer must deliver minimally.

$$\forall c_1, c_2 \in \mathcal{C} : \forall e_1 \in c_1.Events, \forall e_2 \in c_2.Events :$$

$$e_1.Direction = producing \wedge e_2.Direction = consuming \Rightarrow$$
$$e_1.ExpirationTime \geq e_2.ExpirationTime$$

5. All components must agree on the periodicity of each individual event type.

$$\forall c_1, c_2 \in \mathcal{C} : \forall e_1 \in c_1.Events, \forall e_2 \in c_2.Events :$$

$$e_1.SubjectUID = e_2.SubjectUID \Rightarrow$$
$$e_1.Periodicity = e_2.Periodicity$$

6. The dissemination period for all periodic events must match the consumption period of each event type. If the middleware implementation supports temporal buffering, the dissemination periods do not need to match. The dissemination period on the producer must be shorter than on the consumer, though. The middleware local to the consumer can then buffer the events, and deliver them when the consumer expects. This also leads to some events being discarded over time. It must be ensured that this will not influence the system. Due to these complications, the current implementation requires the dissemination periods to match.

$$\forall c_1, c_2 \in \mathcal{C} : \forall e_1 \in c_1.Events, \forall e_2 \in c_2.Events :$$

$$e_1.SubjectUID = e_2.SubjectUID \wedge e_1.Periodicity = periodic \wedge$$

$$e_1.Direction = producing \wedge e_2.Direction = consuming \Rightarrow$$

$$e_1.Period = (\leq) e_2.Period$$

7. For each event type, each data field must have compatible attributes, i.e. the attributes mentioned in the preceding section must be compatible on all communication partners. In particular, the following properties must be met:

   - The *data range* must match for all partners.
   - The resolution on the producer must be better than on the consumer.
   - The precision on the producer must be better than the consumer expects.

$$\forall c_1, c_2 \in \mathcal{C} : \forall e_1 \in c_1.Events, \forall e_2 \in c_2.Events :$$

$$e_1.SubjectUID = e_2.SubjectUID \wedge$$

$$e_1.Direction = producing \wedge e_2.Direction = consuming \Rightarrow$$

$$f_{1,i} \in e_1.Fields, f_{2,i} \in e_2.Fields, i = 1, ..., |e_1.Fields| :$$

$$f_{1,i}.Range = f_{2,i}.Range \wedge$$

$$f_{1,i}.Resolution \leq f_{2,i}.Resolution \wedge$$

$$f_{1,i}.Precision \leq f_{2,i}.Precision$$

7.5 Generating CAN network schedules from CODES descriptions

COSMIC's communication scheme in the time domain is presented in section 6.2.6.

When creating schedules for HRT capable systems, the waiting time at the start of a reserved slot, and the minimum gap between two reserved slots are determined by the parameters of the CAN bus setup. Measurements are presented in [73] and [69].

The length of a reserved slot is determined by two factors: the CAN bus setup, i.e. the bit rate, and by the allowable omission degree, which translates to a certain number of possible retransmissions.

The length of the communication round, and the respective start times of the reserved slots are derived from the periods of the HRT events.

A trivial approach towards schedule generation would be to set the length of the communication round to the least common multiple of all specified periods. The individual reservations can then be scheduled according to the event's periods starting at the beginning. Of course, this will only work, if enough slots can be fitted in the round.

Optimization is desirable, and may even be necessary, as the cycle length may otherwise explode. To the least, this poses technical difficulties on small nodes.

An integrated solution for writing these schedules to the respective target nodes is necessary to complement the existing configuration management facilities. A favorable solution would use the parametrization mechanism to modify the necessary parameters, i.e. the schedules would be represented as event channel parameters in the device description.

7.6 Assisting black–box testing

Based on the CODES descriptions, a number of black–box tests for individual components can be derived. As most COSMIC components interact with the physical environment, completely automated tests currently cannot be performed. This would require a proper description of the physical interaction, and corresponding tools — both hard– and software. However, it is possible to assist the tester considerably during the following tasks:

- Checking sporadic events for their generation, In case of events that should be generated depending on an occurrence in the physical world. The testing tool will display the event's `trigger` attribute. Once the tester stimulates the component accordingly, the generation of the respective event can be checked.

- Checking produced events for compliance with their specified attributes, specifically data range, resolution, and precision. For data measurements, checking the data range requires changing the physical environment to produce the respective measurements, e.g. heating a temperature sensor to the upper bound of the specified range. The resulting events can then be checked to correspond to the specified data ranges. Testing a device beyond its specified bounds is required to check whether the produced events will keep to the specified range. Such tests may potentially harm the device. Depending on the nature of the device, destructive testing may not be an option.

  The resolution of the values in the generated events can only be monitored for violations of the specification. Finding no violation during the tests cannot be seen as a guarantee that the device will never violate the specification.

  Testing the precision requires a test setup with known and sufficient precision to carry out the tests. Again, the sensors are stimulated, and the corresponding output is checked.

- Checking consumed events for compliance with their specified data range. Correct operation is often critical at the limits of the specified data range. Consequently, the component can be fed with events containing such values, and the resulting behavior (as given in the respective event's description) can be checked.

  In terms of resolution and precision, it can only be checked whether the component works as specified when fed with conforming events. As the behavior is undefined when the consumed events do not conform to the specification, no universally useful conclusions can be drawn from such tests.

- Checking timing behavior for produced events. Specifically periodic events can be monitored to occur exactly as specified. For sporadic events, the timing specification can only be checked if there is the possibility to trigger the event automatically.

- Checking the production of events listed in any `WillTrigger` list. This can be done by producing the events having a `WillTrigger` list, and waiting for these events to be produced.

While it can easily be argued that these checks can be deduced automatically from CODES descriptions, the development of a fully grown, flexible testing framework is not part of this thesis.

7.7 Building the basis for dynamic cooperation

Truly autonomous dynamic cooperation of components requires that the involved components can communicate, that they are aware of each other, and that they know how to make use of each others functionality. In COSMIC scenarios, the technical details of the communication are covered by the COSMIC middleware. Awareness of components is raised by the discovery and query service facilities introduced with the CODES approach. The knowledge, which other components are currently useful to a component, and how these can be used, must be derived from these components' descriptions.

The CODES descriptions, as presented in this thesis, provide a wealth of detailed information about information that a component is able to share. The semantic behind this information is currently encoded using unique identifiers and names for subjects and data fields. The meaning of these names must currently be known a priori.

Semantic inference through ontologies can form the basis for a scheme for semantic matchmaking [92], [2]. Requiring a single common ontology for all components is too restricting, as it would require a central authority to maintain said ontology. Also, either all components need to carry the complete ontology, or access to the central instance must be available. The former will sooner or later lead to inconsistencies, as the common ontology is updated and embedded in new components, while older components still use the old version. Access to a central entity does not fit a distributed system view. This is analogous to having the device descriptions stored on the device itself instead of keeping them on a central server.

[102] presents a scheme that allows the autonomous integration of multiple partial ontologies, together with a local ontology, into a single consistent view.

This leads to the idea that each component carries its own ontology. The names used in the CODES descriptions are complemented by references to the respective concepts within the local ontology. Both the CODES description, and the ontology are available for other components, either directly or via the query service. Thus, the use of semantic inference will become possible.

7.8 Personal experience

As an example, a simple temperature sensor component was developed. First, it was developed using nothing but the COSMIC middleware API. In the second run, it was developed using the CODES approach. The component is based on the Tiny platform, utilizing an HC908AZ60A micro controller. It has a simple KTY–10 PTC sensor attached to it. The voltage across this sensor must be measured, linearized and corrected. Measurement is done using the micro controller's internal 8 bit A/D converter. Linearization and correction is performed using a simple lookup table for ease of implementation. The temperature is measured periodically, and broadcast using COSMIC events. The events contain the measured temperature in degrees Centigrade. Overall, this task is of the simplest application class. The details of how the A/D converter needs to be set up and used, the creation of the lookup table, and the conversion into degrees Centigrade were done manually. Usage of the A/D converter is assisted by the Tiny HAL. The creation of the lookup table was realized using a Perl script (which had to be written by hand). The conversion function from A/D converter values to values in degrees Centigrade was directly mapped and included in the lookup table. The availability of a beneficial HAL is of course dependent on the platform used. In this case, the HAL is part of the tool chain developed for this work, yet it does not resemble a central entity in the development process as described in this work. The simulation of the temperature sensor, and the processing of the measurement values is the traditional domain of tools like Matlab. What these tools lack, is the integration with a specific hardware and software platform. In contrast, the CODES approach provides automatic integration with the COSMIC middleware. The code generated is mainly concerned with the definition of the required data structure, callback functions, and task skeletons. The time necessary to devise the necessary information as described here was about four hours. The time needed for creation of the event's data structure, the relevant casts, and measurement and publication tasks took more than an hour. Debugging the hand–written C code, and writing a small PC application displaying the measured temperature took another three hours. Debugging and writing the displaying application was an incremental process, so the time–span cannot be subdivided. In other words, about half the time necessary for developing such a component was eaten up by writing and debugging exactly the code that the CODES tool chain creates automatically from the component description. Writing the component's description using CODES Creator takes about half an hour. The creation of the code skeletons is a matter of seconds. CODES Monitor provides a generic display feature, i.e. there is no need to write an extra display application, which is otherwise needed for debugging purposes. The first part, the details of the A/D converter, and the necessary lookup table needs to be done in both cases. For the house–keeping work, a significant reduction in time necessary to produce a component is visible. Further there are all the extra benefits of CODES available, as described in this chapter. Once the component is produced, it needs to be thoroughly tested before it can be released. Again, CODES is able to help significantly by providing the correct information about the device–under–test at any point in time, as the device itself carries all the information. A supportive black–box testing environment can be based on CODES, as described in section 7.6.

Chapter 8

Conclusion

The main problems described specifically in section 3.2 are well–known: insufficient documentation, misunderstandings due to imprecise specification, and unwary behavior of developers, combined with the ever present resource constraints on small embedded systems. To overcome these problems, the CODES approach supports the component developer via a description language and a tool chain. The description language meets the requirements discussed in section 2.3: It constitutes a formal language that is easy to process, yet readable at least for domain experts. The level of detail present in the descriptions is beyond a simple interface specification. Being based on XML, the language also provides the possibility to be extended in a backward compatible way. The language covers all COSMIC abstractions, thus it can be considered to be complete. Future investigation will show which other facets of *completeness* can be met. In combination with the tool–chain, the description mechanism is capable of supporting the complete usage spectrum described in section 2.3.8. Specifically, the tool chain ensures that many common development errors are avoided, such as the wrong declaration of an event data structure, or the casting from the event data structure to the generic message structure used at the API level. This is achieved via automatic creation of relevant portions of a component's code.

For the system development process, the integration of components into larger systems is supported by providing compatibility checks, and the possibility to create network schedules for the dissemination of hard real–time events.

In conclusion, it can be stated that the development of COSMIC components has been eased and streamlined. All higher level benefits, such as integration support, and the query service are completely new to the COSMIC environment. Also, having the CODES description readily available as documentation ensures that the designer and the developer of the component will not completely forget to document it properly.

Small 8 bit and 16 bit components benefit mostly from the development and integration support offered by the CODES tool chain. Component development is raised to a higher, more appropriate and comfortable level. Many common programming errors are avoided by generating parts of the component's code. The self–describing features are helpful in maintenance situations. Of course, this not only applies to 8 bit components, but to components of all varieties. 32 bit components further benefit from the query facilities offered. They are powerful enough to process XML and e.g. are able to dynamically interpret and use incoming events with previously unknown structures. Given enough memory, they would also be able to incorporate the proposed semantic web technology to enable true dynamic interaction in the long run.

The CODES descriptions link each data field to a specific meaning. Overlaid usage, such as e.g. possible with the LIN signal definition, is not possible. The example given in section 5.6 illustrates in–band error signaling. Such a combined definition is not possible with CODES. The particular example would be modeled using special error events in COSMIC.

Generating the complete application code from CODES descriptions is not possible. To achieve this, the CODES language would have to be enhanced greatly to become a universal modeling language similar to UML. Specifically, elements to sufficiently describe the idiosyncrasies of hardware would have to be included. A bloated language would be the result. In contrast, CODES and COSMIC aim at hiding said idiosyncrasies at a component's outside interface. Reintroducing them in the modeling language would at least be awkward.

8.1 Future work

The work presented in this thesis provides a starting point for further research. Various directions of research promise to beneficial. Specifically, the following areas should be covered:

The long–term goal of dynamic cooperation of components deserves devotion. As outlined in section 7.7, the integration of semantic web technology appears to provide a promising starting point for this research.

Another interesting research topic is the recursive composition of descriptions. COSMIC components can be composed recursively. The same should apply to their descriptions. This requires methods to combine the sub–components' descriptions into a single document in a consistent manner. It is intuitively clear that a new *GeneralInformation* section must be derived or generated. The *EventDefinitions* and *EventChannelDefinitions* can be combined in principle. However, care must be taken for events that one sub–component generates, and another one consumes. It is not clear whether these events and their respective event channel definitions should appear in the enclosing system's description. While it can be argued that such events are of no importance to the outside world since they are only used internally in most cases, there is no guarantee that this is always the case.

It is important to gain more experience using the approach, since it directly aims at supporting the component life cycle. The real–world usage of

CODES will prove the approach's advantages, and it will also uncover any unknown weaknesses. Real–world evaluation will also allow to determine whether the current set of default attributes for events, data fields and event channels is suitable. If any attributes are missing, it should be rather easy to find them through experience.

To achieve sufficient real–world experience, it is important to refine and extend the existing tool–chain in order to enhance the benefits for the users. The prototype implementations developed during this work, specifically the CODESCreator, the COSMICMonitor, and the LogPlayer will benefit from refinement. While they currently provide basic functionality, they lack a comfortable, snappy look and feel expected by users nowadays. Additional beneficial functionality for each application is briefly outlined in the respective sections. Code and schedule generation should also be revised. The currently existing solutions are of a prototypic nature, and should be revised before they can be promptly used in every situation. An important extension to the current tool–chain is a configuration management facility that enables users to take full advantage of the parameterization facilities of the CODES approach. Only a powerful configuration management application will allow a user to painlessly configure individual nodes to his needs when composing a system. A second important addition to the tool–chain is support for black–box testing, as described in section 7.6.

Once enough experience is gained, the concept, specifically the description language can be advanced to support the newly gained insights. In particular, further automation in the areas of compatibility checking and code generation are of interest.

Currently, the CODES approach is limited to the COSMIC world. Generalizing CODES to support a broader spectrum of underlying technologies will provide an understanding how more universal description schemes should be constructed.

Abbreviations

ADC	Analog to Digital Converter
ANL	Abstract Network Layer
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASK	Amplitude Shit Keying
AUTOSAR	Automotive Open System Architecture
BCD	Binary Coded Decimal
CAN	Controller Area Network
CCD	Cluster Configuration Description
CLC	component life cycle
CODES	COSMIC embedded device specification
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CORTEX	Cooperating real–time sentient objects: architecture and experimental evaluation
COSMIC	Cooperating smart devices
CPU	Central Processing Unit
CSMA/CR	Carrier Sense Mulriple Access with Collission Resolution
DCF	Device Configuration File
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name Service
DOM	Document Object Model
DSL	Domain–specific language
DTD	Document Type Definition
DVD	Digital Versatile Disc
DVR	Digital Video Recorder
ECB	Event Channel Broker
ECH	Event Channel Handler
EDS	Electronic Data Sheet
EEPROM	Electrically erasable programmable read only memory
FSK	Frequency Shift Keying
FXPP	Flexible XML Processing Profile
GEAR	Generic Event Architecture
GENA	Generic Event Notification Architecture
GPS	Global Positioning System
HAL	Hardware Abstraction Layer
HRT	Hard real–time

HRTC	Hard real–time channel
HTML	Hypertext markup language
HTTP	Hypertext Transport Protocol
IDE	Integrated Development Environment
IDL	interface definition language
IEEE	Institute of Electrical and Electronics Engineers
IFS	Interface File System
IP	Internet Protocol
IVR	Interactive voice response
LAN	Local Area Network
LDF	LIN Description File
LIN	Local Interconnect
LSB	Least Significant Bit
MDA	Model–driven architecture
MEMS	Micro electro–mechanical system
MIME	Multipurpose Internet Mail Extensions
MOF	Meta Object Facility
NATO	North Atlantic Treaty Organisation
NCAP	Network Capable Application Processor
NCF	Node Capability File
NDA	Non–disclosure agreement
NIC	Network Interface Controller
NRT	Non real–time
NRTC	Non real–time channel
NTC	Negative Temperature Coefficient
OMG	Object Management Group
OO	object orientation
OUI	Organizationally Unique Identifier
OWL	Web Ontology Language
PC	Personal Computer
PDA	Personal Digital Assistant
PDF	Portable Document Format
PDO	Process Data object
PIM	Platform Independent Model
PSM	Platform Specific Model
PTC	Positive Temperature Coefficient
RAM	Random Access Memory
ROM	Read Only Memory
RFC	Request For Comments
RMI	Remote Method Invocation
RODL	Round Descriptor List
ROSE	Round Sequence
RTP	Real–time Transport Protocol
SAWSDL	Semantic Annotations for WSDL
SDO	Service Data Object
SGML	Structured General Markup Language

SI	Système international d'unités
SOAP	Simple Object Access Protocol
SOM	System Object model
SQL	Structured Query Language
SRT	Soft real–time
SRTC	Soft real–time channel
SSDP	Simple Service Discovery Protocol
STD	Smart Transducer Description
STIM	Smart Transducer Interface Module
TCP	Transmission Control Protocol
TDMA	Time division multiple access
TEDS	Transducer Electronic Data Sheet
TQFP	Thin Quad Flat Pack
TTP	Time–Triggered Protocol
TV	Television
UART	Universal asynchronous receiver/transmitter
UDDI	Universal Description Discovery & Integration
UDN	Unique Device Name
UDP	User Datagram Protocol
UHF	Ultra high frequency
UID	Unique Identifier
UML	Unified Modeling Language
UNSPSC	United Nations Standard Products and Services Code
UPnP	Universal Plug and Play
URI	Uniform Resource Indicator
URL	Uniform Resource Locater
UUID	Universally Unique Identifier
VCR	Video Cassette Recorder
VM	Virtual Machine
W3C	World Wide Web Consortium
WAN	Wide Area Network
WBXML	Wap binary XML
WSDL	Web Service Description Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema Definition
XSL	Extensible Stylesheet Language
XSLT	XSL Transformations

Bibliography

[1] 3Soft. ProOSEK 4.0. http://www.dreisoft.de/.

[2] K. Aberer, P. Cudré-Mauroux, A.M. Ouksel, T. Catarci, M.S. Hacid, A. Illarramendi, V. Kashyap, M. Meccella, E. Mena, E.J. Neuhold, O. De Troyer, T. Risse, M. Scannapieco, F. Saltor, L. de Santis, S. Spaccapietra, S. Staab, and R. Studer. Emergent Semantics Principles and Issues, 2004.

[3] Sinan Si Alhir. *UML in a Nutshell*. O'Reilly, 1998.

[4] S. Antoy, P. Forcheri, and M. T. Molfino. Specification–based code generation. In *Proceedings of the Twenty–Third Annual Hawaii International Conference on System Sciences*, pages 165–173 vol. 2, January 1990.

[5] Sergio Antoy and Dick Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, January 2000.

[6] *ATtiny15L data sheet*. http://www.atmel.com, June 2002.

[7] *Hints and Tips for Hard– and Software Developments with MARC4 Microcontrollers*. http://www.atmel.com, April 2004.

[8] *ATAM862-x data sheet*. http://www.atmel.com, May 2006.

[9] Tom Bellwood, Steve Capell, Luc Clement, John Colgrave, Matthew J. Dovey, Daniel Feygin, Andrew Hately, Rob Kochman, Paul Macias, Mirek Novotny, Massimo Paolucci, Claus von Riegen, Tony Rogers, Katia Sycara, Pete Wenzel, and Zhe Wu. UDDI Version 3.0.2. http://uddi.org/pubs/uddi_v3.htm, October 2004.

[10] Andrew Black and Jens Palsberg. Foundations of object–oriented languages. *ACM SIGPLAN Notices*, 29(3):3–11, 1994.

[11] Controller Area Network protocol specification, Version 2.0, 1991.

[12] Giorgio C. Buttazzo. *Hard Real–Time Computing Systems — Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[13] Antonio Carzaniga. Architectures for an Event Notification Service Scalable to Wide–area Networks, 1998.

[14] C. Castelluccia, W. Dabbous, and S. Oapos Malley. Generating efficient protocol code from an abstract specification. *IEEE/ACM Transactions on Networking*, 5(4):514–524, August 1997.

[15] Erik Christensen, Fracisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, March 2001.

[16] CiA Draft Standard 201–207 Version 1.1. CAN Application Layer for industrial applications, February 1996.

[17] CiA Draft Standard 301. CANopen Application Layer and Communication Profile, February 2002.

[18] CiA Draft Recommendation 303–2. CANopen Representation of SI units and prefixes, December 2004.

[19] CiA Draft Standard 306. Electronic data sheet specification for CANopen, January 2005.

[20] CiA Draft Standard 401. CANopen Device Profile for Generic I/O Modules, May 2002.

[21] CiA Draft Standard 102 Version 2.0. CAN Physical Layer for Industrial Applications, April 1994.

[22] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[23] LIN Consortium. LIN Configuration Language Specification, Revision 2.0, September 2003.

[24] LIN Consortium. LIN Node Capability Language Specification, Revision 2.0, September 2003.

[25] LIN Consortium. LIN Specification Package, Revision 2.0, September 2003.

[26] CORTEX Annex I — Description of Work, October 2000.

[27] CORTEX D13 — Evaluation Report, September 2004.

[28] CORTEX D14 — Final Report, December 2004.

[29] CORTEX D2 — Preliminary Definition of CORTEX Programming Model, March 2002.

[30] CORTEX D3 — Preliminary Definition of the Interaction Model, March 2002.

[31] CORTEX D6 — The CORTEX Programming Model, April 2003.

[32] CORTEX D7 — Proof–of–concept prototypes, April 2003.

[33] CORTEX D9 — Final Definition of CORTEX Interaction Model, November 2003.

[34] *C Cross Compiler for Motorola MC68HC08*, 1997.

[35] I. Crnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan. Building Reliable Component–Based Software Systems, 2002.

[36] D. J. Dailey, S. Maclean, F. W. Cathey, and D. Meyers. A self–describing data transfer model for ITS applications. *IEEE Transactions on Intelligent Transportation Systems*, 3(4):293–300, December 2002.

[37] Steffen Deter and Karsten Sohr. Pini — A Jini–Like Plug&Play Technology for the KVM/CLDC. In T. Böhme and H. Unger, editors, *Innovative Internet Computing Systems: International Workshop IICS 2001 Ilmenau, Germany*, June 2001.

[38] DIN 44300. Informationsverarbeitung: Begriffe, 1985.

[39] Bruce Powell Douglass. *Real–Time UML: Developing Efficient Objects for Embedded Systems*. Addison Wesley, second edition edition, 2000.

[40] Bruce Powell Douglass. *Real–Time UML: Advances in the UML for Real–Time Systems*. Addison Wesley, third edition edition, 2004.

[41] Keith Edwards. *Core Jini*. Prentice Hall, 2nd edition, 2001.

[42] S. Eisenbach, C. Sadler, and V. Jurisic. Feeling the way through DLL Hell. In *Proceedings of the First Workshop on Unanticipated Software Evolution*, 2002.

[43] Wilfried Elmenreich. Kostengünstig vernetzen mit TTP/A. *Markt&Technik – Wochenzeitung für Elektronik und Informationstechnik*, 38/2000:42–44, Sep. 2000.

[44] Wilfried Elmenreich, Günther Bauer, and Hermann Kopetz. The Time-Triggered Paradigm. In *Proceedings of the Workshop on Time-Triggered and Real-Time Communication*, Manno, Switzerland, Dec. 2003.

[45] Wilfried Elmenreich, Wolfgang Haidinger, Raimund Kirner, Thomas Losert, Roman Obermaisser, and Christian Trödhandl. TTP/A Smart Transducer Programming — A Beginner's Guide. Research Report 33/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[46] Wilfried Elmenreich and Stefan Krywult. A Comparison of Fieldbus Protocols LIN 1.3, LIN 2.0, and TTP/A. In *Proc. 10th IEEE Conference on Emerging Technologies and Factory Automation, ETFA 2005*, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, Sep. 2005.

[47] Wilfried Elmenreich, Christian Paukovits, and Stefan Pitzek. Automatic generation of schedules for time-triggered embedded transducer networks. *10th IEEE International Conference on Emerging Technology and Factory Automation (ETFA05)*, I:535–541, Sep. 2005.

[48] Wilfried Elmenreich, Stefan Pitzek, and Martin Schlager. Modeling distributed embedded applications on an interface file system. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 175–182, May 2004.

[49] Hermann Kopetz et al. Specification of the TTP/A–Protocol V2.00. Technical report, University of Technology, Vienna, September 2002.

[50] Adrian Fitzpatrick, Gregory Biegel, Slobhan Clarke, and Vinny Cahill. Towards a sentient object model, April 2003.

[51] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. *UML Profile for Framework Architectures*. Addison Wesley, 2001.

[52] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison Wesley, third edition edition, 2003.

[53] Yaron Y. Goland, Ting Cai, Paul Leach, Ye Gu, and Shivaun Albright. Simple Service Discovery Protocol/1.0, 1999.

[54] Wolfgang Haidinger and Robert Huber. Generation and Analysis of the Codes for TTP/A Fireworks Bytes. Research Report 5/2000, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2000.

[55] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state–based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[56] Gerard J. Holzmann and Joanna Patti. Validating SDL Specifications: an Experiment. In *Proceedings of the 9th International Workshop on Protocol Specification, Testing and Verification*, June 1989.

[57] John E. Hopcroft and Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison–Wesley, 1994.

[58] I. Horrocks, P. Patel-Schneider, and F. van Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[59] Polly Huang, Vincent Lenders, Philipp Minnig, and Mario Widmer. Jini for ubiquitous devices. Technical report, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology — Zürich, Jun 2002.

[60] IEEE Std 1451.1-1999, Standard for a Smart Transducer Interface for Sensors and Actuators — Network Capable Application Processor (NCAP) Information Model, June 1999.

[61] IEEE Std 1451.2-1997, Standard for a Smart Transducer Interface for Sensors and Actuators — Transducer to Microprocessor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats, September 1997.

[62] 1451.3-2003 Standard for a Smart Transducer Interface for Sensors and Actuators — Digital Communication and Transducer Electronic Data Sheet (TEDS) Formats for Distributed Multidrop Systems, 2003.

[63] 1451.4-2004 IEEE Standard for a Smart Transducer Interface for Sensors and Actuators — Mixed–Mode Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats, 2004.

[64] Infineon. Dave. `http://www.infineon.com/DAvE/`.

[65] Infineon. AP2922 'C' CAN Driver Routines for the C166 Family. `http://www.infineon.com/`, 1997.

[66] Intel. Intel Authoring Tools for UPnP Technologies. `http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/upnp/overview/index.htm`, 2003.

[67] *Linux SDK for UPnP Devices 1.2.1 (libupnp)*. `http://upnp.sourceforge.net/`, February 2003.

[68] P. Jalote. Testing the completeness of specifications. *IEEE Transactions on Software Engineering*, 15(5):526–531, May 1989.

[69] Carlos Mitidieri Jörg Kaiser, Cristiano Brudna. COSMIC: a real-time event-based middleware for the CAN–bus. *Journal of Systems and Software Special Issue: Parallel and distributed real–time systems*, 77(1):27–36, July 2005.

[70] Michael Mock Jörg Kaiser. Implementing the real-time publisher/subscriber model on the controller area netwrk (CAN). In *2nd International Symposium on Object Oriented Distributed Real-Time Computing Systems, San Malo, France*, May 1999.

[71] Jörg Kaiser. An Event Model for the Predictable Interaction of Smart Devices. In *Workshop on Dependable Embedded Systems, in conjunction with the IEEE 22nd International Symposium on Reliable Distributed Systems, Florence Italy*, October 2003.

[72] Jörg Kaiser, Cristiano Brudna, and Carlos Mitidieri. A real–time event channel model for the CAN–bus. In *Proceedings of the 11th Annual Workshop on Parallel and Distributed Real–Time Systems*, 2003.

[73] Jörg Kaiser, Cristiano Brudna, and Carlos Mitidieri. Implementing real-time event channels on CAN–bus. In *Proceedings of the IEEE Workshop on Factory Communication Systems 2004 (WFCS'2004)*, September 2004.

[74] Jörg Kaiser and Carlos Mitidieri. Attribute–based filtering for embedded systems. In *Proceedings of the 2nd International workshop on Distributed event–based systems*, 2003.

[75] Jörg Kaiser and Hubert Piontek. CODES: Supporting the development process in a publish/subscribe system. In *Fourth Workshop on Intelligent Solutions in Embedded Systems (WISES 2006)*, June 2006.

[76] Jörg Kaiser and Hubert Piontek. Self–describing components in cooperating systems. In *Proceedings of the 3rd MiNEMA Workshop*, February 2006.

[77] Michael Kay. *XPath 2.0, Programmer's Reference*. Wiley Publishing, Inc., 2004.

[78] Michael Kay. *XSLT 2.0 3rd edition, Programmer's Reference*. Wiley Publishing, Inc., 2004.

[79] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison Wesley, 2003.

[80] W. H. Ko and C. D. Fung. VLSI and intelligent transducers. *NASA STI/Recon Technical Report A*, 83:12810–+, 1982.

[81] Hermann Kopetz. *Real–Time Systems — Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[82] Hermann Kopetz. Software engineering for real–time: a roadmap. In *Proceedings of the 22nd International Conference on Future of Software Engineering (FoSE) at ICSE 2000, Limerick, Ireland*, June 2000.

[83] Hermann Kopetz, Michael Holzmann, and Wilfried Elmenreich. A low–cost time–triggered sensor network: TTP/A. Technical report, Institut für Technische Informatik, TU Wien, July 1999.

[84] Hermann Kopetz, Michael Holzmann, and Wilfried Elmenreich. A universal smart transducer interface: TTP/A. In *3rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2000)*, Mar. 2000.

[85] Stefan Krywult and Christian Steiner. TTP/A gateway. Research Report 37/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.

[86] Kang Lee. A synopsis of the IEEE P1451 standards for smart transducer communication, 1999.

[87] Kang Lee. IEEE 1451. a standard in support of smart transducer networking. In *IEEE Instrumentation and Measurement Technology Conference, Baltimore*, May 2000.

[88] Kang Lee and Richard D. Schneeman. A standardized approach for transducer interfacing: Implementing IEEE–P1451 smart transducer interface draft standards.

[89] Wolfgang Lehner and Harald Schöning. *XQuery — Grundlagen und fortgeschrittene Methoden*. dpunkt.verlag, 2004.

[90] Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 153–164, 2000.

[91] Thomas Losert. *Extending CORBA for Hard-Real Time Systems*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2005.

[92] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. OWL–S: Semantic markup for web services, 2004.

[93] M.D. McIlroy. Mass produced software, October 1968.

[94] T. Meijler and O. Nierstrasz. Beyond objects: Components, November 1997.

[95] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model–Driven Architecture*. Addison Wesley, 2002.

[96] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model–Driven Architecture*. Addison Wesley, 2004.

[97] Jack Weast Michael Jeronimo. *UPnP Design by Example*. Intel Press, 2003.

[98] Microsoft. Understanding Universal Plug and Play. `http://www.upnp.org`, June 2000.

[99] Microsoft. Microsoft Platform SDK for Windows 2003 SP1. `http://msdn.microsoft.com/platformsdk`, May 2005.

[100] Holger Mönnich. Kompakte Repräsentation der Eigenschaften von intelligenten Hard– & Softwarekomponenten und die Realisierung eines geeigneten Discovery Serice. Diplomarbeit, Universität Ulm, August 2004.

[101] Eric J. Naiburg and Robert A. Maksimchuk. *UML for Database Design*. Addison Wesley, 2001.

[102] Andronikos Nedos. Distributed ontologies and semantic service discovery in mobile ad hoc networks. In *Proceedings of the 3rd MiNEMA Workshop*, February 2006.

[103] OMG. Smart Transducers Interface V1.0. Available Specification document number formal/2003-01-01, Object Management Group, Needham, MA, U.S.A., Jan 2003. available at `http://doc.omg.org/formal/2003-01-01`.

[104] C. Pfister and C. Szyperski. Workshop on Component–Oriented Programming (WCOP'96), Summary, 1996.

[105] Hubert Piontek and Jörg Kaiser. Self–describing devices in COSMIC. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, September 2005.

[106] Hubert Piontek, Matthias Seyffer, and Jörg Kaiser. Improving the accuracy of ultrasound–based localisation systems. In *Proceedings of the First International Workshop on Location– and Context–Awareness (LoCA)*, 2005.

[107] Stefan Pitzek. Description mechanisms supporting the configuration and management of TTP/A fieldbus systems. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2002.

[108] Stefan Pitzek and Wilfried Elmenreich. Configuration and management of a real-time smart transducer network. In *Proceedings of the 9th IEEE International Conference on Emerging Technologies and Factory Automation*, volume 1, pages 407–414, Lisbon, Portugal, September 2003.

[109] Wolfgang Pree. Component-based software development — a new paradigm in software engineering? *Software — Concepts and Tools*, pages 169–174, 1997.

[110] Alan Presser, Gary Langille, Gerrie Shults, John Ritchie, Mark Walker, Nelson Kidd, Changhyun Kim, Sungjoon Ahn, Dennis Bushmitch, Minobu Abe, Masatomo Hori, Matthew Ma, Jack Unverferth, Wim Bronnenberg, Geert Knapen, John Gildred, Elias Kesh, Russell Berkoff, Irene Shen, Norifumi Kikkawa, Jonathan Tourzan, and Yasuhiro Morioka. ScheduledRecording:1 Service Template Version 1.01, May 2006.

[111] Alan Presser, Gary Langille, Gerrie Shults, John Ritchie, Mark Walker, Changhyun Kim, Sungjoon Ahn, Masatomo Hori, Matthew Ma, Jack Unverferth, Wim Bronnenberg, Geert Knapen, Russell Berkoff, Irene Shen, Norifumi Kikkawa, Jonathan Tourzan, and Yasuhiro Morioka. MediaServer:2 Device Template Version 1.01, May 2006.

[112] Alan Presser, Gary Langille, Gerrie Shults, John Ritchie, Mark Walker, Changhyun Kim, Sungjoon Ahn, Masatomo Hori, Matthew Ma, Jack Unverferth, Wim Bronnenberg, Geert Knapen, Russell Berkoff, Irene Shen, Norifumi Kikkawa, Jonathan Tourzan, and Yasuhiro Morioka. MediaRenderer:2 Device Template Version 1.01, May 2006.

[113] Alan Presser, Gary Langille, Gerrie Shults, John Ritchie, Mark Walker, Changhyun Kim, Sungjoon Ahn, Masatomo Hori, Matthew Ma, Jack Unverferth, Wim Bronnenberg, Geert Knapen, Russell Berkoff, Irene Shen, Norifumi Kikkawa, Jonathan Tourzan, and Yasuhiro Morioka. ConnectionManager:2 Service Template Version 1.01, May 2006.

[114] Alan Presser, Gary Langille, Gerrie Shults, John Ritchie, Mark Walker, Changhyun Kim, Sungjoon Ahn, Masatomo Hori, Matthew Ma, Jack Unverferth, Wim Bronnenberg, Geert Knapen, Russell Berkoff, Irene Shen, Norifumi Kikkawa, Jonathan Tourzan, and Yasuhiro Morioka. ContentDirectory:2 Service Template Version 1.01, May 2006.

[115] Alan Presser, Gary Langille, Gerrie Shults, John Ritchie, Mark Walker, Changhyun Kim, Sungjoon Ahn, Masatomo Hori, Matthew Ma, Jack Unverferth, Wim Bronnenberg, Geert Knapen, Russell Berkoff, Irene Shen, Norifumi Kikkawa, Jonathan Tourzan, and Yasuhiro Morioka. RenderingControl:2 Service Template Version 1.01, May 2006.

[116] Alan Presser, Gary Langille, Gerrie Shults, John Ritchie, Mark Walker, Changhyun Kim, Sungjoon Ahn, Masatomo Hori, Matthew Ma, Jack Unverferth, Wim Bronnenberg, Geert Knapen, Russell Berkoff, Irene Shen, Norifumi Kikkawa, Jonathan Tourzan, and Yasuhiro Morioka. AVTransport:2 Service Template Version 1.01, May 2006.

[117] Kilian Rall. Entwicklung einer Echtzeitexekutive. Technical report, Department of Computer Structures, University of Ulm, January 2005.

[118] John Ritchie and Thomas Kuehnel. UPnP AV Architecture v 1.0, June 2002.

[119] James Rumbaugh, Ivar Jacobsen, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison–Wesley, 1998.

[120] Rick Schneeman and Kang Lee. Multi–network access to IEEE P1451 smart sensor information using world wide web technology, 1997.

[121] Uwe Schöning. *Theoretische Informatik — kurzgefaßt, 3. Auflage*. Spektrum, Akademischer Verlag, 1997.

[122] Siemens. Siemens UPnP Stack. http://www.plug-n-play-technologies.com/whitepapers.htm, 2001.

[123] J. Will Specks and Antal Rajnák. LIN — protocol, development tools, and software interfaces for local interconnect networks in vehicles. In *9th International Conference on Electronic Systems for Vehicles*, October 2000.

[124] J. M. Spivey. *The Z Notation: A Reference Manual*, 2nd edition, 1998.

[125] Goerge W. Stimson. *Introduction to Airborne Radar, 2nd edition*. Scitech Publishing, 2001.

[126] Bjarne Stroustrup. *The C++ Programming Language. Special Edition.* Addison–Wesley Longman, 2000.

[127] Sun Microsystems. *Javadoc 5.0 Tool*, 31 August 2006.

[128] C. Szyperski. Components vs. objects vs. component objects. In *Proceedings of OOP 1999*, 1999.

[129] C. Szyperski. Components and architecture. *Dr. Dobbs Journal*, June 2001.

[130] C. Szyperski. Point, counterpoint. *Dr. Dobbs Journal*, June 2001.

[131] UPnP Forum. *UPnP Device Architecture 1.0*, December 2003.

[132] Hans Utz. Advanced software concepts and technologies for autonomous mobile robots., 2005.

[133] Eric van der Vlist. *XML Schema*. O'Reilly, 2002.

[134] P. Verissimo, V. Cahill, A. Casimiro, K. Cheverst, A. Friday, and J. Kaiser. CORTEX: Towards supporting autonomous and cooperating sentient entities. In *Proceedings of the European Wireless Conference 2002*, 2002.

[135] P. Verissimo and A. Casimiro. Event–driven support of real–time sentient objects. In *Proceedings of the Eighth IEEE International Workshop on Object–oriented Real–time Dependable Systems (WORDS2003)*, January 2003.

[136] Paulo Veríssimo, Jörg Kaiser, and Antonio Casimiro. An architecture to support interaction via generic events. In *24th IEEE Real–time Systems Symposium, Work in Progress Proceedings*, Cancun, Mexico, December 2003.

[137] Markus Völter. Im Fokus — Sprachen, Modelle und Fabriken in der Softwareentwicklung. *iX*, 10, 2006.

[138] OWL web ontology language overview. http://www.w3.org/TR/owl-features/, 2004.

[139] W3C Namespaces in XML. http://www.w3.org/TR/REC-xml-names, January 1999.

[140] W3C Extensible Markup Language (XML) 1.0 (Third Edition). http://www.w3.org/TR/REC-xml, February 2004.

[141] W3C XML Schema Part 1: Structures Second Edition. http://www.w3.org/TR/xmlschema-1, October 2004.

[142] W3C XML Schema Part 2: Datatypes Second Edition. http://www.w3.org/TR/xmlschema-2, October 2004.

[143] W3C XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20, June 2006.

[144] W3C XSL Transformations (XSLT) Version 2.0. http://www.w3.org/TR/xslt20, June 2006.

[145] J. Waldo. *The Jini Specifications. 2nd edition*. Addison Wesley, 2000.

[146] Richard W. Wall and A. Ekpruke. Developing an IEEE 1451.2 compliant sensor for real–time distributed measurement and control in an autonomous log skidder. In *Proceedings of the 29th Annual Conference of the IEEE Industrial Electronics Society, IECON'03*, pages 2482–2487, November 2003.

[147] Tobias Wenzel. Infineon Microcontrollers ApNote AP2925, CAN Baudrate Detection with Infineon CAN devices, 1999.

[148] Jennifer Widom. Data management for XML. *IEEE Data Engineering Bulletin, Special Issue on XML*, 22(3), September 1999.

[149] Wikipedia. Software componentry. http://en.wikipedia.org/wiki/Software_componentry, 13 May 2006.

[150] Wikipedia. International system of units. http://en.wikipedia.org/wiki/International_System_of_Units, 15 December 2006.

[151] Wikipedia. Ethernet. http://en.wikipedia.org/wiki/Ethernet, 25 August 2006.

[152] Wikipedia. Universally unique identifier. http://en.wikipedia.org/wiki/UUID, 25 August 2006.

[153] Wikipedia. Bit rot. http://en.wikipedia.org/wiki/Bit_rot, 4 December 2006.

[154] Wikipedia. DLL hell. http://en.wikipedia.org/wiki/DLL_hell, 5 December 2006.

[155] Stan P. Woods, Janusz Bryzek, Steven Chen, Jeff Crammer, Edwin Vivian El-Karch, Mike Geipel, Fernando Gen-Kuong, John Houldswoth, Norm LeComte, Kang Lee, Michael F. Mattes, and David E. Rasmussen. IEEE–P1451.2 smart transducer interface module, 1996.

[156] Web services description language (WSDL) version 2.0 part 1: Core language. http://www.w3.org/TR/wsdl20, March 2006.

[157] Web services description language (WSDL) version 2.0 part 2: Adjuncts. http//www.w3.org/TR/wsdl20-adjuncts, March 2006.

[158] Semantic annotations for WSDL (first public working draft). http://www.w3.org/TR/2006/WD-sawsdl-20060630/, June 2006.

Index

Danksagung

Für die Erstellung dieser Arbeit spielten einige Personen eine wichtige Rolle. Als erstes danke ich meinem Doktorvater, Jörg Kaiser, der immer ein offenes Ohr für Ideen hatte und sie gerne ausführlich mit mir diskutiert hat. Seine konstruktive Kritik, gepaart mit seiner Ruhe und Geduld, hat wesentlich zur erfolgreichen Fertigstellung dieser Arbeit beigetragen. Ebenso danke ich den anderen Gutachtern, Franz J. Hauck und Wolfgang Schröder-Preikschat.

Dank gilt auch meinen (ehemaligen) Kollegen des Instituts für Eingebettete Systeme/Echtzeitsysteme, ehemals Abteilung Rechnerstrukturen (Cristiano Brudna, Klaus Espenlaub, Christian Heinlein, Changling Liu, Gisela Menger und Carlos Mitidieri). Sie haben eine stets sehr angenehme Arbeitsumgebung geschaffen, in der immer Raum für wissenschaftliche (und andere) Diskussionen war. Weiterhin gilt mein Dank der Institutsleitung, zunächst Les Keedy, danach kommissarisch Herr von Henke, für die jederzeit gewährte Unterstützung.

Gisela Menger, Andreas Schmied und James Mason haben diese Arbeit freiwillig korrekturgelesen.

Von besonderer Wichtigkeit war auch die Unterstützung meiner Verlobten Anja Wurst. Sie hat mir den nötigen Freiraum für diese Arbeit eingeräumt und mich stets bei der Arbeit ermutigt.

Acknowledgements

A number of people played an important role in the creation of this thesis. First, I thank my adviser, Jörg Kaiser, who was always open for ideas, and discussed any of them with me. His constructive criticism, paired with his calmness and patience, significantly contributed to the successfull creation of this thesis. Also, I thank my other examiners, Franz J. Hauck and Wolfgang Schröder-Preikschat.

I also thank my (former) colleagues from the institute of embedded systems/real–time systems, the former department of computer structures (Cristiano Brudna, Klaus Espenlaub, Christian Heinlein, Changling Liu, Gisela Menger and Carlos Mitidieri). They always provided a very comforting work environment with room for scientific (and other) discussions. My thank goes to the head of the institute, first Les Keedy, later on provisionally professor von Henke, for their continuously granted support.

Gisela Menger, Andreas Schmied and James Mason have proofread this thesis voluntarily.

The support of my fiancée Anja Wurst was of special importance to me. She granted me the necessary freedom for this thesis and always encouraged me during my work.