Communication and Collaboration in Heterogeneous Teams of Soccer Robots

Philipp A. Baer and Roland Reichle
University of Kassel
Germany

1. Introduction

The RoboCup tournaments foster research in the area of autonomous robotics and cooperative behaviour. Recently, modifications to the rules were adopted promoting further developments towards a typical human playing ground. Some simplifications such as constant lighting were dropped and further modifications will follow in the next years. Regarding team cooperation and coordination, the most important change in 2007 is the enlargement of the playing field. The maximum number of players in a team has been increased to 6; for the long time goal the number of players will approach 11.
For research groups it may be difficult to keep up with the enlargement of team sizes, for newcomers it even constitutes a virtually infeasible financial effort. This is why so-called mixed teams gain a lot of popularity. Here, two or more research groups pool their resources together to provide a joint, more powerful team (Nardi et al., 1999; Castelpietra et al., 2000). This implies that different hardware and software systems have to communicate and collaborate. A number of problems have to be faced which arise from the heterogeneity of the systems involved. Among other things, the interpretation of different representations, the fusion of information to a consistent world view, and the realization of team-play strategies on the different platforms are predominant questions.
In order to cope with these challenges, we have adopted a model-driven software development approach. Below we introduce our development environment for communication infrastructures. Afterwards, we summarize our research activities towards a model-driven development approach for modelling cooperative behaviour in teams of autonomous soccer robots. A detailed example describes the creation of a software infrastructure for a mixed-team of soccer robots. It illustrates the benefits of our development environment and highlights our contribution. We conclude with a presentation of our vision for further developments.

2. Problem Description

When RoboCup was announced in 1995, it was a research challenge to build autonomous mobile robots (AMRs) that were able to find the ball and the goals, to avoid collisions with other players, to estimate their position on the field, and to score goals. Nowadays, a large

Open Access Database www.i-techonline.com

number of approaches are available for solving these problems. The research focus therefore shifted towards creating teams of robots that cooperatively play soccer.
To realize a team-play, robots must be able to exchange information with their team-mates, interpret the exchanged data, and fuse the information to a consistent world view, as already outlined in the introduction. This is the basis for coming to an agreement about the current situation on the field and coordinating the cooperative behaviour of the team. Nowadays, almost every team in the RoboCup middle-size league implements some kind of team-play, which in most cases is tailored to the capabilities and the needs of the underlying robotic software framework. Due to the lack of standard software and because of the variety of different software frameworks, heterogeneity issues play a decisive role when forming a mixed-team.

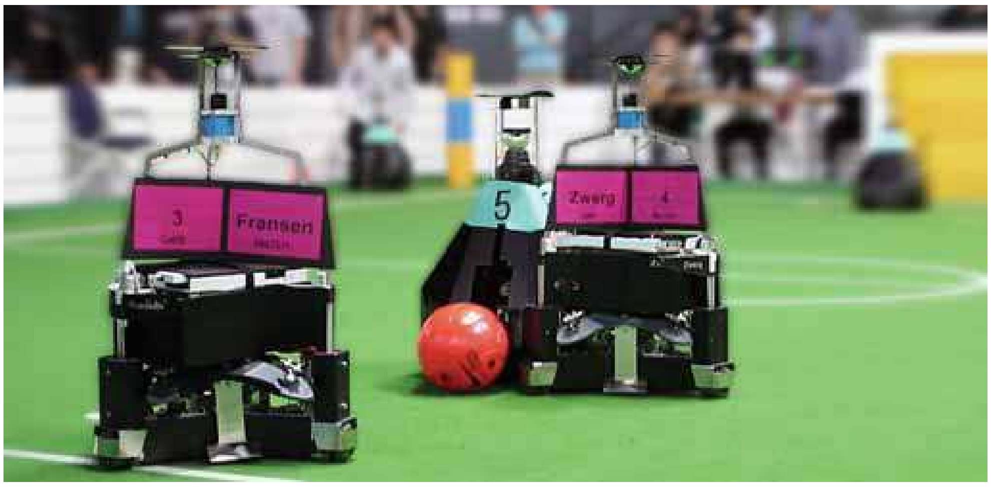


Fig. 1. Carpe Noctem fighting against another team

During the RoboCup World Championships 2006 in Bremen, Germany, the teams Carpe Noctem from Kassel University – shown in Fig. 1 – and the Ulm Sparrows from Ulm University formed a mixed-team. The Ulm Sparrows use Miro (Utz et al., 2002), a middleware framework that is implemented in C++ and heavily relies on CORBA. Greater parts of the Carpe Noctem software framework are realized in C# using the Mono (http://www.mono-project.org/) framework. To set up team cooperation a suitable communication infrastructure had to be established first. The Ulm Sparrows relied on an IP multicast-based group communication scheme over which the SharedBelief (Utz et al., 2004; Isik et al., 2007) data structure was exchanged. In order to talk to the other team, Carpe Noctem thus needed to provide a corresponding implementation along with suitable data conversion techniques. The implementation as such was a quite time consuming task.
For real interoperability it is not sufficient to only communicate data, they also have to be interpreted with regard to their semantics and representations. Teams have to either agree on a standard representation which includes common measurement units and coordinate systems or provide appropriate data conversion routines. However, recent discussions in the RoboCup community show that it is quite difficult to reach an agreement on a standard

representation, as almost any team provides a self-defined representation scheme. Fortunately, the measurements units and the coordinate systems used by the Ulm Sparrows and Carpe Noctem were quite similar.

To be able to agree on the current situation on the field and to realize dynamic role assignment, Carpe Noctem provided a cooperative world model named *SharedWorld*. It fused the exchanged information to a consistent world view and provided realizations of different team-play strategies which were used as a basis for the role assignment. *SharedWorld* was also capable of calculating the ball position by taking into account the trustworthiness and impreciseness of observations, gathered from the different robots. For the Ulm Sparrows a module with nearly the same functionality was created because decisions had to be taken consistently among all players. A re-implementation of *SharedWorld* was necessary because of the incompatible software frameworks of the two teams.

This example makes it quite obvious that the realization of cooperative behaviour in heterogeneous teams of robots is a challenging and time consuming task. This problem has even more effect in groups consisting of more than two teams. Therefore, methods and development support is needed to ease the realization of communication and collaboration in heterogeneous teams of soccer robots.

3. Our Contribution

The previous section showed that the realization of team-play strategies in heterogeneous teams of soccer robots is a quite time consuming task. In order to reduce the development effort we present SPICA, a development framework for communication and collaboration infrastructures in teams of AMRs. SPICA assists in integrating software systems realized in different programming languages and developed for different platforms in heterogeneous distributed environments. It further provides patterns for data and sensor fusion and facilitates the development of cooperative behaviour in groups of AMRs.

To be able to cope with heterogeneity issues, we have adopted a model-driven development approach for SPICA. It supports the specification of communication and collaboration infrastructures of AMRs at an abstract and platform-independent level. Models are then automatically transformed to platform-specific source code which can easily be integrated into existing software frameworks. The modelling support is based on the SPICA modelling language which consists of several domain-specific sublanguages tailored to the different aspects of communication and collaboration infrastructures. The total of all sublanguages form the SPICA modelling language, also referred to as the *Abstract Architecture Specification* (AAS) language.

With the *Message Description Language* (MDL) a developer may specify the structure of network messages in an efficient and platform-independent manner, similar to ASN.1. Communication among AMRs is mostly event-based, so we decided to apply concepts of *message-oriented middleware* (MOM) architectures as they turned out to be most appropriate.

The *Data flow Description Language* (DFDL) supports the specification of communication infrastructures in terms of modules and the data flow between them. Module stubs are created from the specifications which are basically adapters to the underlying communication infrastructure. The DFDL also facilitates the specification of the data management behaviour. For this purpose, it provides so-called *Data Management Containers*

(DMCs) which are data structures used for managing incoming and outgoing data. DMCs further build the foundation of the general purpose *Data-Analysis Description Language* (DADL). It provides modelling support for filters that operate directly on the contents of the DMCs. DADL comprises a Matlab-like syntax allowing calculations on the exchanged data to be specified in a platform-independent manner. Examples are the calculation of a robot's role or the agreed ball position. In addition, the DADL also provides some predefined filter patterns to fuse the exchanged data to a consistent world view. The integration of other services such as data encryption or authentication is possible as well. Apart from these sublanguages we employ the concept of ontologies. They allow us to establish a common understanding of the involved semantic concepts and different representations and therefore help to realize automatic conversion of data representations.
With the help of the tools provided by the SPICA development environment, the resulting *platform-independent models* (PIMs) of the communication and collaboration infrastructure can be transformed into platform-specific implementations in C#, C++, and Java. Our template-based approach allows for easy integration of further programming languages.
The model-driven development approach proved to significantly reduce the development effort for the realization of communication and collaboration infrastructures for heterogeneous teams of AMRs. A suitable communication and collaboration infrastructure has to be developed only once by specifying the desired functionality in a platform-independent manner. The corresponding platform-specific implementations are generated automatically and can be integrated into new or existing software frameworks very easily. In addition, the SPICA development framework also completely relieves the developers from the burden of dealing with encoding and decoding issues, heterogeneous data representations, and synchronization issues. In this aspect, the SPICA-based implementations are comparable to Remote Procedure Call-based (RPC) solutions. The main difference here is that generated implementations are tailored to the characteristics of event-driven AMR group communication. Using SPICA, the developers furthermore do not have to deal with the time-consuming implementation of data fusion and analysis schemes for each of the involved platforms.
In the following, we will introduce the SPICA development environment in more detail along with its modelling language and associated capabilities. Afterwards, an elaborate example will outline the steps required to specify a communication and collaboration infrastructure between two different groups of AMRs.

4. The Spica Approach

The concept of the SPICA development environment was first published in 2007 (Baer et al., 2007). The first generation of SPICA was capable of generating message structures, the second generation added support for modelling data flow. The third generation we outline here, brings major language cleanups, enhancements, and new features such as dynamic module binding, semantic annotation, and automatic data conversion.
Dynamic module binding relies on a service discovery engine embedded into the generated implementation. The creation of channels is based on the availability of resources. It is also possible to create static channels which do not require the service discovery engine.
Automatic data conversion relies on the semantic annotation of the specified data structures. Here, a common understanding of the semantics of data structures and the relations

between them may be established using an ontology specification. Our framework also foresees the integration of ontologies provided by third parties.
For several reasons we decided to develop a textual domain-specific modelling language instead of using existing general purpose ones. First of all, the development of a novel *domain-specific language* (DSL) enables us to provide a very compact modelling notation, reduced to the needs and tailored to the semantics of our modelling domain. The SPICA sublanguages cover these areas where more specific modelling support is required. They are designed in such a way that they combine to the overall SPICA modelling language in a consistent fashion. This is not only advantageous for the model transformation process but also for the developer, who does not have to deal with different semantics of different description languages. A textual notation is furthermore sufficient for most modelling tasks. If designed with simplicity in mind, it is often even more convenient to use than graphical notations and it may allow for rapid prototyping.

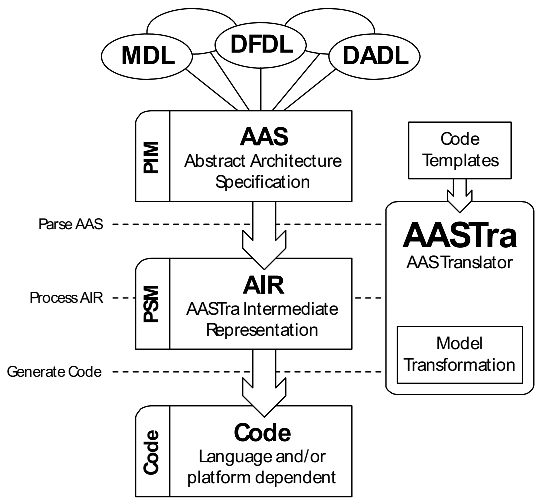


Fig. 2. Internal workflow of the SPICA development environment

DSLs, of course, require us to create tools that interpret and transform models into concrete implementations. The corresponding tool developed for SPICA is the *AAS Transformator* (AASTra), covering the whole process from interpreting a model down to generating concrete source code. It follows a three-layered approach as shown in Fig. 2, representing the reduction of abstraction from the topmost down to the lowest level of modelling. The SPICA modelling language resides on the topmost layer named AAS; it represents the PIM in the context of Model-driven Development (MDD). On the second layer, an in-memory intermediate representation of the AAS models is generated which is referred to as the *AASTra Intermediate Representation* (AIR). Processing is carried out in two steps: First, the model is parsed and references are resolved. Afterwards, the model is processed and transformed into a representation more suitable for the final code transformation. Here, the consistency of the model is checked and the required non-trivial transformations are performed. Language compilers follow a very similar approach when transforming a language specification to assembler or executable binaries.

The actual code transformation is performed on the third layer. Here, a template engine keeps the transformation process very flexible and customizable. Templates can access to the AIR directly. Reoccurring patterns are encapsulated in smaller sub-templates. To reduce redundancy in the code, frequently used functionality is relocated to libraries.
We will now introduce the SPICA modelling language along with all its sublanguages, starting with a description of common language features.

4.1 Common Model Semantics

4.1.1 Blocks

Related statements are grouped together in logical blocks. Each such block has a type and a name. The body of the block is enclosed in curly braces. The example below outlines the general structure.

```
<type> <name> [<inheritance spec>] [<annotations>] {
       <body>
}
```

The type of such a block is given by one of the predefined keywords `header`, `message`, `container`, `coord`, or `module`. The first four keywords are belonging to MDL while the remaining one is part of DFDL. A type is followed by a name, an inheritance specification, and additional annotations. The optional inheritance specification is available in the MDL only. Annotations are lists of key-value pairs enclosed in square brackets; values may be omitted. They parameterize the respective elements and thus influence the code generation process. Annotations are supported in every sublanguage but are optional as well.

4.1.2 Semantic Model

In a heterogeneous distributed environment, where a-priori unknown systems have to communicate and interpret the exchanged data, it is essential to establish a common understanding of their semantics and representations. Therefore, the SPICA AAS language supports semantic annotation of data structures. The semantic model provides two types of classes: concepts and representations. A concept defines the conceptual appearance of an element while a representation defines its concrete representation. A base ontology defines fundamental concepts like ball or player, but also coordinate systems and representations such as physical units. It thus builds the foundation for an automatic conversion of representations, a conversion from mm to m, for instance. Due to the availability of coordinate system specifications defined with regard to a reference system also non-trivial operations like converting the representation from one coordinate system to another can be provided automatically. For even more complex tasks we also allow the definition of custom conversion methods in a way similar to (Strang et al., 2003).
Semantic descriptions provided by third-parties may be used as well. The base ontology and third-party additions are managed by a simple, distributed storage system which supports retrieval or insertion of ontology classes in a lightweight manner.
Two semantic annotations are created for all blocks in a SPICA model automatically if not specified explicitly: `concept` specifies the ontology class name while `rep` specifies the representation of the element. To reference a block type, its concept, or representation, the

`reftype`, `refconcept`, and `refrep` annotations are provided. The type reference replaces the other two. References to coordinate system are modelled using the `refcoord` annotation.
Ontology classes such as concepts or representations are referenced using URNs. Unique URNs are assigned to elements implicitly, providing a reference name to their concept and representation. In order to allow for a compact specification the developer can use abbreviations to URN in terms of prefixes. The default prefix # references the SPICA URN namespace `urn:spica`.

4.1.3 Variants

Block variant identifiers have been introduced because several blocks with the same name may be defined. Variant identifiers are arbitrary strings representing the target AMR platform. They are appended to the block name, wrapped into angle brackets. If there is more than one target platform, it is possible to specify multiple variant identifiers delimited by colons.
The existence of variant identifiers changes the automatic generation of concept and representation URNs for blocks. The following concept and representation URNs are created by default if no variant names are given:

```
urn:spica:<type>:concept:<name>
urn:spica:<type>:rep:<name>
```

If a variant name is provided, the URNs read as follows:

```
urn:spica:<variant name>:<type>:concept:<name>
urn:spica:<variant name>:<type>:rep:<name>
```

4.2 Message Specification

As already outlined above, the concept of message-oriented communication is well suited for AMRs. This has several reasons: The network infrastructure of mobile autonomous system resembles the characteristics of mobile ad-hoc networks, so communication links are likely to exist only for a limited period of time. This is why a message-oriented and connection-less communication scheme has clear advantages over a connection-oriented one. Another reason stems from the communication behaviour of AMRs. As environmental monitoring and sensing are known to be mostly event-driven, message-oriented communication here directly reflects their characteristics. Messages may further get lost during transmission where a dependency to previous messages may average the usefulness of information.
Based on these observations we created the MDL as a sublanguage of the SPICA modelling language. It is used to specify messages and containers for the SPICA communication infrastructure. The modelling concept is closely related to structure definitions in programming languages such as C, but offers more advanced features like single inheritance, dynamic arrays, and strings. It provides a set of commonly used primitive types. Complex types are containers in the SPICA context, which are made up of primitive types or other containers again. A customizable serialization and de-serialization interface allows arbitrary message encodings to be used. Support for automatic conversion of

message values is supported by augmenting the model with references to semantic concepts and representations.
The objectives of ASN.1 and MDL are quite similar. However, ASN.1 lacks support for some fundamental techniques required by SPICA: It does not provide, for example, the mandatory concept of semantic annotations. In contrast, a custom modelling language like MDL may be designed in such a way that all required functionality is covered in a lean fashion. The specification support of MDL is sufficient for SPICA and we can easily extend it to our specific requirements.
We will now describe the modelling entities of MDL in more detail, covering the definition of message headers and containers. Messages, as they represent a specialization and aggregation of these concepts respectively, are introduced afterwards.

4.2.1 Headers

A header specification represents a special form of container used only for structuring the fields of the message header. These fields are used to control the way a message is handled and processed. There is only one mandatory field in SPICA: A special field holds the type identifier of the message as messages are strongly typed. A unique identifier is generated automatically using a suitable hash function. Other, mostly optional control fields in the header include the endian flag or the message identifier. Message headers may have different variants and be derived from each other. Headers may not be instantiated. Every message must be derived from exactly one header. From this point of view, a header represents an abstract class in the context of object-oriented programming.
The example below depicts the layout of a header specification using a minimal header with only one field identifying the type. A reference to the corresponding concept is required here to establish the meaning of the field.

```
header MessageBase {
        uint16 type [refconcept=#message:concept:type];
}
```

As outlined above, default context and representation URNs are assigned automatically. For this example they read `urn:spica:header:concept:MessageBase` and `urn:spica:header:rep:MessageBase` respectively.

4.2.2 Containers

Container specifications create composite types consisting of zero or more primitive or composite types. Containers may be derived through single inheritance from other containers. Compared to ordinary structures in C, instances of containers have the additional capability of being able to serialize and de-serialize themselves. Besides, they are not bound to the SPICA communication infrastructure only, but can be used as general purpose data structures as well.
A container is similar to a header but differs in one point: headers are only used as supertypes for messages while containers may only be used as a part of the message body. The example below shows a simple container. It defines the field `double d` referencing a semantic concept as well as a representation. A unique type identifier based on the

representation URN is assigned to each container implicitly. The example further defines the container as being a variant of type `cn`.

```
container Distance<cn> {
        double d [refconcept=#coord:distance, refrep=#rep:units:mm];
}
```

The concept URN reads `urn:spica:cn:container:concept:MessageBase`; its representation counterpart `urn:spica:cn:container:rep:MessageBase`. Considering the referenced concept and the representation, the value of `d` is a distance measured in mm.

4.2.3 Messages

One header and zero or more containers make up a message. The specification of a message differs from that of headers or containers in two ways. First, the topmost message in the inheritance hierarchy must be derived from exactly one header definition, providing all the required header fields. A container contains an implicitly defined type identifier which, however, is not represented as a field. A header is thus not required for a container. The second point in which a message differs from a container is that no primitive types may be added to the payload of a message; only containers are allowed. The example below outlines the definition of a message.

```
message DistanceMessage : MessageBase {
        Distance<cn> dist;
}
```

4.2.4 Coordinate Systems

In the area of AMRs, many containers are most likely to be used to store the position of some objects or observations in terms of their coordinates. In order to facilitate a correct interpretation of the fields of the corresponding container, the MDL also includes specification means to define coordinate systems. We include this modelling support into the MDL as it can be seen as additional semantic description of the containers – a container can reference a certain coordinate system. The specification of coordinate systems with regard to reference systems retains the freedom of choosing your own coordinate system and allow for automatic conversion between different ones.

In our modelling approach we currently support three basic types of coordinate systems – `Cartesian2D`, `Cartesian3D`, and `Polar2D` – and two different views – ego and allo. Egocentric coordinates resemble the egocentric view of the robot, whereas allocentric coordinates resemble the view of an external observer of the field. For the different types of coordinate systems we have some predefined concepts like an x-coordinate (`#coord:xcoord`) or a distance (`#coord:distance`) indicating the distance of the object from the pole of an polar coordinate systems. For a correct interpretation of container fields they have to refer to such a predefined concept.

In order to define a new coordinate system, the new origin, the axes (or a zero-ray for polar systems), and the view have to be specified. Some example specifications are shown in section 5.

4.3 Data flow Specification

One intention of the SPICA modelling language is to describe the communication behaviour in groups of AMRs in an abstract and platform-independent fashion. AMRs are basically hardware agents that are asked to accomplish a job or mission, similar to software agents. Just as software agents, AMRs can be regarded as modules which are mostly independent from each other. They further initiate mutual communication to exchange information and to collaborate. This is why the Data flow Definition Language (DFDL) follows an inherently modularized approach. Each AMR may be made up of several modules that communicate with each other or other AMRs. Such a scenario can easily be modelled given a modular software architecture where modules are connected via network links. There is, in fact, no difference between local and remote modules. For local communication, however, more suitable communication schemes might be chosen whereas communication between robots should be based on proven network communication schemes.

SPICA now introduces specification means for modelling data exchange between modules in an abstract manner. Modules are the main modelling entity here. For each module the requested as well as the offered message types have to be specified. At least one of the two options must be present; the module otherwise exhibits no functionality. For each communication direction – i.e. incoming and outgoing –DMCs are responsible for the management of messages and containers. They are used by the communication engine and by filters for passing data. Finally, the message transmission schemes have to be specified. They represent specific communication techniques tailored to the communication behaviour of AMRs. The block layout below outlines the basic structure of a module specification.

```
module Communication {
   offer { ... }
   request { ... }
   export { ... }
}
```

The offer, request, and export blocks will be introduced below in more detail. We will first start with the offer and request blocks that describe the basic communication structure of a module. The DMCs and transmission schemes are outlined thereafter. This section closes with the presentation of the description of filters incorporating the DADL sublanguage.

4.3.1 Message offers and requests

Let us now have a look at the specification of the most important parts of the module model. Offering and requesting messages is a fundamental functionality of MOM-based communication systems. In the DFDL model, *offer blocks* provide the information about offered, i.e. outgoing messages whereas *request blocks* deal with the reception of messages.

Every message that is provided by a module has to be listed in an offer block using the `message` directive. Along with the name of the message the specification of a transmission scheme is mandatory as it determines in which way the given message is handled. Apart from that, DMCs are required as input buffers and as temporary data storage for filters. The example below outlines the structure of an offer block.

```
offer {
   message DistanceMessage scheme ...;
   dmc ...;
}
```

It has to be noted here that arbitrarily many **`message`** and **`dmc`** statements may be listed. The **`scheme`** keyword defines the transmission scheme to be applied here. A request block is specified in exactly the same way except that it is not mandatory to specify a transmission scheme.
The relations between messages and DMCs are not explicitly modelled in the above example. It is, however, established automatically during model transformation. The next subsection will show how this can be accomplished.

4.3.2 Data Management Containers

Data Management Containers (DMCs) have been mentioned earlier already. They resemble data management structures for messages or containers in SPICA. They also perform basic synchronization tasks. The most important characteristic of the DMCs is the fact that each DMC is responsible for a specific semantic concept and a respective realization. This is where the relations between messages and DMCs are identified automatically.
In request blocks the identification of relations even goes one step beyond: For each incoming message not only the message type but also the types of the enclosed containers are checked. If the semantic type of a message container conforms to the referenced semantic type of a DMC, it is added to this DMC automatically. The representation of the container is further adapted to the DMC's representation if required. This way, further processing on the incoming data is possible in a very efficient manner. Irrelevant information is further discarded without manual intervention.
DMCs are implemented as linear lists with characteristics specific to the application domain: message passing. Queues and ringbuffers are more elaborate instances of linear lists and well-known examples of data structures in this context. All DMCs exhibit a consistent interface through which elements can be added, accessed, or retrieved. The semantic of these operations depends on the actual parameterisation, though. The retrieval operation, for example, may change the number of elements in the DMC, i.e. remove the element in question, or leave it alone.
The following DMCs with the stated characteristics are available in SPICA so far. More may be added if required. The size (`size`) and the management scheme (`scheme`) of a DMC may be changed using the appropriate annotations.

list: A list implements the semantics of an ordered list using a fixed-size buffer space. If the buffer is full, no new elements may be added. Elements have to be removed explicitly. The management scheme defaults to FIFO.

queue: A queue implements the semantics of an ordered list using a fixed-size buffer space. If the buffer is full, no new elements may be added. Elements are removed on retrieval except when using indexers. The management scheme defaults to FIFO.

ringbuffer: A ringbuffer implements the semantics of an ordered, circular list using a fixed-size buffer space as if it were connected end-to-end. If the buffer is full, the oldest element is overwritten if a new one is added. The management scheme defaults to LIFO.

The generic annotations `refconcept`, `refrep`, and `reftype` as introduced earlier are supported by every DMC. They are required to specify the element type.

Arrays of DMCs are supported as well. The array semantic is, however, not quite as expected: A DMC array is managed in such a way that each array element is uniquely assigned to one specific system that attends the communication. Every array thus has the same number of elements in the array, each of which corresponds to the same system. The DMC of the local system is also contained in the DMC array and can be retrieved using a dedicated operation on the DMC.

In order to complete the **`dmc`** statement in the example above, we will present a possible parameterization below. We will assume that a ringbuffer with only one element is used which accepts elements of the type `DistanceMessage`:

```
dmc ringbuffer dist [type=DistanceMessage, size=1];
```

DMCs have been defined only in the context of the model so far. It is very likely that more than one DMC is used and only a subset of these need to be accessible from userspace. The DFDL provides the **`export`** block for this purpose. DMCs that have to be visible from userspace only have to be added to the export block. The example below illustrates this.

```
export { dist; }
```

4.3.3 Transmission Schemes

Transmission schemes fulfil another very important task especially for offering messages. Data transmission in groups of AMRs is assumed to be very dynamic. Locations of modules local to a system are normally not subject to change but the location of modules on remote systems: Robots may join or leave a group spontaneously; other types of systems may appear and disappear in the same way.

This is why SPICA introduces the concept of transmission schemes for establishing communication links. In contrast to ordinary socket-based communication establishment, these schemes exhibit a special behaviour which is tailored to the dynamic communication behaviour of AMRs. For local modules, a static scheme that does not change its endpoints is sufficient as it can do without the overhead for dynamic binding. For ad-hoc communication with remote systems, however, heartbeat techniques are tried and tested. This is especially true for unreliable environments. Three schemes are outlined below which implement the requested characteristics of static and dynamic binding. It has to be noted here that only messages may be sent. Containers have to be wrapped in messages for this purpose.

static: The static transmission scheme is a one-to-one communication scheme which supports local communication characteristics. Basically, two modules are bound together statically. The location of communication partner must not be subject to change. It is not possible to change the communication endpoints during runtime. The `module` annotation is used by the sender to reference a destination module.

announce: The announce transmission scheme is a one-to-many communication scheme. The sender announces the availability of a data source to which one or more interested receivers can listen to. This scheme implements the heartbeat technique: The sender provides an alive-signal which contains the respective endpoints of the data source. This information is used by interested receivers to listen to the data source. The `heartbeat` annotation is used by the sender to specify the interval for the alive-signal.

request: The request transmission scheme is a one-to-many communication scheme. It is similar to announce but implements a variant of a publish-subscribe protocol. The sender again announces the availability of a data source but without immediately starting the transmission: It is triggered once at least one receiver is available. Receivers have to emit subscription heartbeats that inform the sender of the availability of an interested party. It depends on the endpoints provided by the receivers and on the number of receivers where and how the sender directs the data to. The `heartbeat` annotation is used to specify the interval for the alive signal for the sender as well as the receiver.

There are some annotations that are available for every transmission scheme. They basically resemble event processing capabilities: Transmission schemes must be either able to send messages periodically or after some events occurred. These annotations may be used both at the same time. The `interval` annotation is used to specify if a transmission should be triggered periodically. With the `on` annotation, the transmission scheme reacts on the events given in the annotation value.

In order to complete the **`scheme`** statement in the example above, we will present a possible parameterization below. We will assume that the announce scheme is used which sends messages with 30 Hz:

```
message DistanceMessage scheme announce [interval=33ms];
```

4.3.4 Filters

In the previous sections we introduced the MDL and the DFDL which allow us to realize communication infrastructures for heterogeneous groups of AMRs. The modules defined in DFDL exchange messages, adjust the representation of received data if required, and store the information into appropriate DMCs automatically. However, in order to also facilitate the development of a collaboration infrastructure in terms of a cooperative world model, we have to go beyond this: it must be possible to interpret the exchanged data and perform arbitrary calculations on them. For this purpose, we introduce the Data-Analysis Description Language (DADL). The design of the language and its modelling elements is based on the following observations.

The realization of a cooperative world model first requires data to be collected from the different robots in the group, which must then be combined to a consistent world view. Here, the impreciseness of the observations – which is mainly due to the physical limitations of the sensors used – has to be taken into account. For this purpose approaches for probabilistic state estimation are commonly applied, such as e.g. Bayesian Filtering (Aström, 1965; Fox et al., 1999; Thrun et al., 2000) and derivates like Kalman-Filtering (Kalman, 1960) or Particle Filters (Handshin & Mayne, 1969; Akashi & Kumamoto, 1977). In order to deal with uncertainty a number of different approaches are available: Dempster-Shafer theory (Dempster, 1968; Shafer, 1967), Bayesian Inference (Pearl & Kaufmann, 1988), and Fuzzy sets (Zadeh, 1978), to mention only some. In most cases, however, the implementation of these approaches is non-trivial and time-consuming. Therefore, we tried to identify frequently used patterns for which appropriate predefined filters are included in the DADL.

Especially in the area of Computer Vision but also in robotics, Matlab (http://www.mathworks.com/) is widely used for rapid prototyping. Matlab is a numerical computing environment and programming language, designed to efficiently deal with

matrices and operations on them. When dealing with arrays or lists it provides a very compact syntax and elaborate indexing methods. Therefore, we decided to adopt a Matlab-like syntax for specifying the algorithms needed to perform calculations on the exchanged data.
The main modelling element in DADL is a filter block because all calculations on exchanged data are considered as a form of filtering operation in SPICA. The basic structure of a filter block depends on whether a predefined filter pattern or a custom filter is required. We will first discuss the predefined filter patterns. Afterwards the custom filters are introduced.
The structure of a predefined filter patterns is given below.

```
filter <name> {
        <spec for the result DMCs>
        predefined <filter type> <annotations> {
                <additional input specs>
        }
}
```

A filter specification starts with a list of DMCs that store the results of the calculations. They are defined in exactly the same way as the DMCs for offer or request blocks in a module. They are furthermore globally accessible, can be referenced throughout the module and in other filters. So, it is possible to specify whole filter chains.
The actual specification for predefined patterns only includes the filter type and associated annotations. The annotations contain basic parameters for the filter and specify when the filter is triggered, similar to the annotations of the transmission schemes described above. The input for the predefined filter is specified in its body. It has to be noted here that it depends on the type of predefined filter which DMCs have to be provided for input and output. However, the SPICA framework is able to investigate the container for fields that correspond with the expected concepts and to associate them to the inputs and outputs of the filter.
Currently three predefined patterns are available to be used for data-fusion: *Kalman-Filter* (KF) (Kalman, 1960), *Multi-Hypothesis-Kalman-Filter* (MHKF) (Reid, 1979), and *Simple-Multi-Hypothesis-Estimation* (SMHE).
A KF is an approach for probabilistic state estimation that can be used to fuse data that represent observations on the field in terms of their Cartesian coordinates. It can also be utilized for object tracking and velocity estimation. However, it should be guaranteed that all the data to be fused correspond to the same object, i.e. the KF is not able to deal with false positives. The KF pattern can be parameterized in several ways. For example, it can be configured to realize a linear or a non-linear KF. In addition, it can be specified if the velocity of the observed objects should be taken into account and estimated and if the filtering should be iterative or non-iterative.
MHKF is an extension of the KF that is able to deal with false positives, i.e. not all observations must belong to the same object. For this purpose, the observations are forming a set of hypothesis for the object state and for each hypothesis a separate KF is applied. The pattern can be configured in the same way as the KF pattern.
SMHE is a simplification of the MHKF used in the Carpe Noctem software framework. It focuses on the multi-hypotheses tracking and avoids the complexity of MHKF through applying some heuristics. The pattern can be parameterized for an iterative or a non-iterative filtering.

We also intend to provide some predefined patterns to estimate situations based on the evidence for hypotheses. Here, Bayesian Inference and Dempster-Shafer theory will be applied. However, work on this is in a very preliminary state and therefore detailed descriptions are omitted here.
Custom filters are specified in a filter block as shown below:

```
filter <name> {
        <spec for the result DMCs>
        call <annotations> {
                <filter body>
        }
}
```

As with the predefined filters, the block for a custom filter starts with the specification of the result DMCs. In contrast to predefined filters, no special rules have to be followed here; the most appropriate DMCs can be chosen to store the results of the calculations. Annotations are used here as well to specify the way a filter is triggered. The filter body is a collection of statements in the Matlab-like DADL syntax, supporting arbitrary calculations on the DMCs.
From the Matlab programming language we have adopted – among other things – the following concepts:

- implicit typing
- control statements like for-loops, while-loops and if-conditions
- basic arithmetic and logical operations
- definition of matrices and vectors
- matrix operations
- array and matrix indexing methods
- a basic set of functions like `min`, `max`, `cos`, `sin`, `tan`, `atan`, `atan2`, `sqrt`, that accept also matrices as input values, perform the calculations per element, and may return matrices as well.

In addition to these concepts, MDL containers and DMCs are seamlessly integrated into the DADL language. DMCs can be indexed just like arrays in Matlab. We also allow calling methods on DMCs as, for example, to create or delete an element. The containers are accessed in the same way as structures in Matlab or in other common programming languages. Due to space limitations, we do not present the whole specification support provided by the DADL for the definition of custom filters here. In section 5, however, two custom filters are presented.

5. Evaluation and Results

In this section, we demonstrate the applicability of our approach. We show how the SPICA development environment was used to realize communication and collaboration in a mixed-team of soccer robots involving two heterogeneous platforms. For this purpose, we return to the scenario that was already outlined in section 2: A mixed-team formed by the Ulm Sparrows (referred to as US) – and Carpe Noctem (referred to as CN).
In our example, SPICA is used to establish a communication infrastructure bridging the gap between the two software frameworks. Afterwards, we show how SPICA facilitates the

development of a cooperative world model which is used to agree on a common ball position within the team and to realize a basic role assignment of the robots.
The cooperative world model can be realized either in a centralized or decentralized way. In a centralized scenario, the robots would share information with only one leading robot acting as hub or data sink. In the decentralized approach, AMR exchange information directly with all their team-mates in a peer-to-peer fashion. There is no designated data sink to which all data is sent to, but all systems process the provided information on their own. Each robot can so decide which piece of information is important and should be further processed.
We decided to go for the decentralized approach, as during the last RoboCup tournaments it has shown to be better suited and less error-prone. This is mostly because robot systems are likely to crash once or several times during a match due to hardware or software failures. Therefore, a team with one leading robot that coordinates the cooperative behaviour should be avoided. Besides, according to our experiences the network infrastructure at RoboCup tournaments is quite unreliable, bandwidth is scarce and the network is sometimes not available at all. For these reasons, we base our communication infrastructure on IP multicast, as it facilitates our decentralized approach and, as a connection less communication scheme, it is also not affected by an unreliable network with regard to blocking issues.

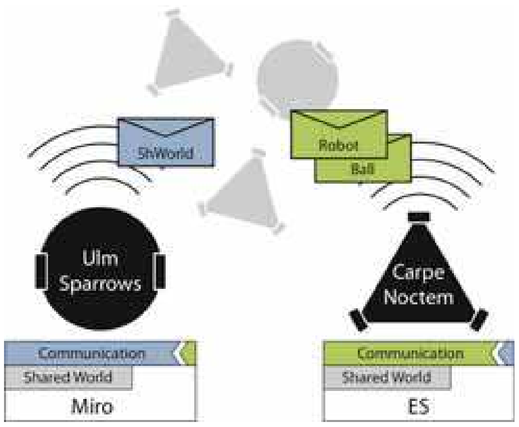


Fig. 3. System architecture of the mixed team “Ulm Sparrows and Carpe Noctem”

Fig. 3 provides a basic overview of the system architecture. Each of the robots of the Ulm Sparrows, depicted by a circle, and of Carpe Noctem, depicted by a triangle, runs an instance of the corresponding software framework which integrates two SPICA-generated modules, namely `Communication` and `SharedWorld`. The `Communication` module of the Ulm Sparrows is responsible for sending a `SharedWorld` message to the multicast group, containing the detected ball position and the robot’s position on the field. In contrast, the `Communication` module of Carpe Noctem provides a separate `BallMessage` for the ball position and a `RobotPosMessage` for the robot’s position on the field. Here, it is assumed that it fits more into the existing framework to send one combined message or two separated messages, respectively. The `SharedWorld` modules of the Ulm Sparrows and

Carpe Noctem have identical functionality: receiving the messages from all the team-mates, fusing the ball information to an agreed ball position and calculating a basic role assignment. Therefore, this module has to be specified only once and is generated for the two different target platforms: C++ for the Ulm Sparrows and C# for Carpe Noctem.
In the following paragraphs, we show how this architecture can be specified and realized with the help of the SPICA development environment. First, we start with the definition of messages, containers and the coordinate systems used by these two teams.

```
container BallPosition<cn> [refconcept=#concept:BallPosition,
                                  refcoord=#cn:coord:Cartesian2D] {
   double x [refconcept=#coord:xcoord, refrep=#rep:units:mm];
   double y [concept=#coord:ycoord, refrep=#rep:units:mm];
   double probability [refconcept=#refconcept:propability,
                           refrep=#rep:units:pct01];
   cov(x, y) cov;
}
container AlloBallPosition<cn> : BallPosition<cn>
                           [refcoord=#cn:coord:AlloCartesian2D];
container BallPosition<us> [refconcept=#concept:BallPosition,
                                  refcoord=#us:coord:Polar] {
   double d [refconcept=#coord:distance, refrep=#rep:units:mm];
   double alpha [refconcept=#coord:angle, refrep=#rep:units:deg];
   cov(d, alpha) cov;
}
container RobotPosition<cn,us> [refconcept=#concept:RobotPosition,
                                  refcoord=#coord:Cartesian2D] {
   double x [refconcept=#coord:xcoord, refrep=#rep:units:mm];
   double y [refconcept=#coord:ycoord, refrep=#rep:units:mm];
   double heading [refconcept=#coord:heading, refrep=#rep:units:deg];
   cov(x,y,heading) cov;
}
```

Listing 1. Definition of the containers

Listing 1 shows the specification of the containers that have to be defined for our communication and collaboration infrastructure. The listing includes two variants of the `BallPosition` container, one for Carpe Noctem and one for the Ulm Sparrows. Both containers refer to the concept `#concept:BallPosition`, but different representations corresponding to different coordinate systems are used. Carpe Noctem represents the position of the ball in an egocentric Cartesian coordinate system (`#cn:coord:Cartesian2D`), whereas the Ulm Sparrows use an egocentric polar coordinate system (`#us:coord:Polar`). For both containers the covariance matrices, representing the uncertainty of the observations with regard to the corresponding coordinate systems, are defined. In addition, the `BallPosition` container of Carpe Noctem also includes a field indicating the probability of the observation to be correct. The `AlloBallPosition` container of Carpe Noctem has the same fields as the Carpe Noctem `BallPosition` container, however refers to an allocentric coordinate system (`#coord:cn:AlloCartesian2D`). Besides, the `RobotPosition` container definition is identical to Carpe Noctem and the Ulm Sparrows. It is noteworthy here that the semantic annotation of containers can be considered as just a way of commenting, but of course with some guidelines.

```
coord Cartesian2D<cn> [refconcept=#coord:Cartesian2D] {
   origin = (0.0, 0.0) [refconcept=#coord:origin, rep=#rep:units:mm];
   xAxis = (1.0, 0.0) [refconcept=#coord:xaxis, rep=#rep:none];
   yAxis = (0.0, 1.0) [refconcept=#coord:yaxis, rep=#rep:none];
   view = ego [refconcept=#coord:coordView, rep=#rep:coord:coordView];
}
coord AlloCartesian2D<cn> [refconcept=#coord:Cartesian2D] {
   origin = (0.0, 0.0) [refconcept=#coord:origin, rep=#rep:units:mm];
   xAxis = (1.0, 0.0) [refconcept=#coord:xaxis, rep=#rep:none];
   yAxis = (0.0, 1.0) [refconcept=#coord:yaxis, rep=#rep:none];
   view = allo [refconcept=#coord:coordView, rep=#rep:coord:coordView];
}
coord Polar<us> [refconcept=#coord:Polar2D] {
   pole = (0.0, 0.0) [refconcept=#coord:pole, rep=#rep:units:mm];
   zero_ray = (1.0, 0.0) [refconcept=#coord:zero_ray, rep=#rep:none];
   view = ego [refconcept=#coord:coordView, rep=#rep:coord:coordView];
}
```

Listing 2. Definition of the coordinate systems

Listing 2 shows the specification of the coordinate systems used by Carpe Noctem and the Ulm Sparrows. As already mentioned above, the coordinate systems are described by providing values for predefined concepts like the view, the axis and the origin (`pole`) of the coordinate system with regard to the reference systems.

```
message SharedWorldMessage<us> : MessageBase {
   BallPosition<us> ballPos;
   RobotPosition<us> robotPos;
}
message BallMessage<cn> : MessageBase {
   BallPosition<cn> ballPos;
}
message RobotPosMessage<cn> : MessageBase {
   RobotPosition<cn> robotPos;
}
```

Listing 3. Definition of the messages

With the help of these specifications the framework is able to provide the appropriate transformations between the representations in different coordinate systems automatically. Here, the development environment also deals with automatic conversions between the measurement units as well as with automatic transformation of the covariance matrices. As now the specification of all needed containers and of the coordinate systems are available, the corresponding messages can be defined as shown in Listing 3.
As depicted in the overview of the architecture, we have to define a `SharedWorldMessage` for the Ulm Sparrows that includes a `BallPosition` and a `RobotPosition`. The communication module of Carpe Noctem provides the same kind of information but in two separate messages: `BallMessage` and `RobotPosMessage`.

```
module Communication<us> {
   offer {
      dmc ringbuffer sharedWorld [type=SharedWorldMessage<us>, size=1];
      message SharedWorldMessage<us> scheme announce
                                   [on=sharedWorld.new"];
   }
   export { sharedWorld; }
}
module Communication<cn> {
   offer {
      dmc ringbuffer ballMessage     [type=BallMessage<cn>, size=1];
      dmc ringbuffer robotPosMessage [type=RobotPosMessage<cn>, size=1];
      message BallMessage<cn> scheme announce [period=100ms];
      message RobotPosMessage<cn> scheme announce [period=100ms];
   }
   export {
      ballMessage;
      robotPosMessage;
   }
}
```

Listing 4. Definition of the *Communication* modules

Now we can start with the definition of the modules of our architecture. The specifications of the *Communication* modules for the two teams are shown in Listing 4. The Communication module of the Ulm Sparrows offers a `SharedWorldMessage` using the announce scheme, when a new message is placed into the ringbuffer. In contrast, the *Communication* module of Carpe Noctem is defined to offer the two messages, `BallMessage` and `RobotPosMessage`, each of which is sent with 10Hz. In order to allow other parts of the software framework to access the message DMCs, they are included into the export block in both modules.

```
module SharedWorld<cn,us> {
   request {
      message SharedWorldMessage<us>;
      message RobotPosMessage<cn>;
      message BallMessage<cn>;
      dmc ringbuffer[] balls [concept=#concept:BallPosition,
               rep=#cn:container:rep:BallPosition, size=10, ttl=5s];
      dmc ringbuffer[] robots [concept=#concept:RobotPosition,
               rep=#cn:container:rep:RobotPosition, size=10, ttl=5s];
   }
   filter ego2Allo { ... }
   filter calculateSharedBall { ... }
   filter calculateRoleAssignment { ... }
   export { ... }
}
```

Listing 5. Definition of the *SharedWorld* module

As the `SharedWorld` module provides a more complex functionality – receiving messages, fusing the exchanged data and calculating a role assignment – we show the corresponding

specification in several steps. Listing 5 provides a general overview of the description and focuses on the request block for receiving messages.
The module is defined to receive the messages `SharedWorldMessage` from Ulm Sparrows robots, and the `RobotPosMessage` and the `BallMessage` from Carpe Noctem robots. DMC arrays are specified to be filled with the `BallPosition` (`#concept:BallPosition`) and `RobotPositions` (`#concept:RobotPosition`) containers for each of the robots in the team. The corresponding information is automatically extracted from the messages named above.
In order to realize the functionality of calculating the agreed ball position and a basic role assignment, we have to define filters that are able to process the data collected in the DMCs or more precisely DMC arrays. The `BallPosition` containers are represented with regard to an egocentric coordinate system, so we first have to transform the data into an allocentric representation. A common allocentric view is a prerequisite to fuse data about observations collected by a group of robots. This is the purpose of the filter `ego2Allo`. The corresponding specification is shown in Listing 6.

```
filter ego2Allo {
   dmc ringbuffer[] alloBalls [concept=#concept:BallPosition,
                               rep=#cn:container:rep:AlloBallPosition,
                               size=10];
   call [on=balls.new]{
      index = balls.changedIndex();
      heading = robots(index).last.heading;
      alloX = robots(index).last.x;
      alloY = robots(index).last.y;

      alloX = alloX + cos(heading)*balls(index).last.x -
                               sin(heading)*balls(index).last.y;
      alloY = alloY + sin(heading)*balls(index).last.x +
                               cos(heading)*balls(index).last.y;
      alloBalls(index).new;
      alloBalls(index).last.x = alloX;
      alloBalls(index).last.y = alloY;

      //Calculate and assign transformed covariance matrix
      alloBalls(index).last.cov = ... ;
   }
}
```

Listing 6. A filter performing the transformation from egocentric to allocentric view

First, the filter defines a ringbuffer array named `alloBalls` that stores the allocentric ball positions (`#concept:BallPosition`) in the representation `AlloBallPosition` (`#cn:container:rep:AlloBallPosition`) for each of the robots in the team. Afterwards, the body of the filter is defined. It is called every time a new ball position is available. The filter fetches the index of the corresponding ball DMC in the array which represents the number of the respective player in the team. Then variables are declared and initialized with the robot's last position on the field. To this position the egocentric ball position rotated by the heading of the player on the field is added. Now a new container is added to the array and initialized with the coordinates of the allocentric ball position calculated above. At the end of the filter body, the field for the covariance matrix is assigned. Due to space limitation the corresponding transformation was left out. However, as just a rotation of the matrix is required, the transformation can be specified in basically the same way as the rotation of the egocentric ball coordinates.

```
filter calculateSharedBall {
   dmc list sharedBall [concept=#concept:BallPosition,
                           rep=#cn:container:rep:BallPosition];
   predefined MHKF [period=100ms, iterative, linear, staticObject] {
      input = alloBalls(:).last;
   }
}
```

Listing 7. An MHKF for the agreed ball position

All ball positions are now available in a common allocentric view, thus we can apply a Multi-Hypothesis-Kalman-Filtering (MHKF) to fuse the information and keep track of different hypothesis of the ball position on the field. In our case, the agreed position can be determined as the position hypothesis with the highest probability. Listing 7 illustrates how the corresponding MHKF can be included into the specification.

```
filter calculateRoleAssignment {
   dmc ringbuffer[] roles [concept=#concept:Role, rep=#rep:string,
                                   size=1, init="None"];
   call calculateRoleAssignment [period=100ms]{
      [maxProb, maxIndex] = max(sharedBall.probability);
      teamBall = sharedBall(maxIndex);
      ballDistances = sqrt((robots(:).last.x - teamBall.x).^2 +
                           (robots(:).last.y - teamBall.x).^2);
      [minBallDist, AttackerIndex] = min(ballDistances);
      roles(AttackerIndex) = "Attacker";

      minXPos = 20000.0;
      minIndex = -1;

      for i = 1:teamsize()
         if(roles(i) == "None" && robots(i).last.x < minXPos)
            minXPos = robots(i).last.x;
            minIndex = i;
         end;
      end;

      roles(minIndex) = "Defender";

      for i = 1:teamsize()
         if(roles(i) == "None")
            roles(i) = "Supporter";
         end;
      end;
   }
}
```

Listing 8. A filter for calculating the role assignment

The filter calculateSharedBall uses a list named sharedBall to store the hypothesis for the ball positions returned by the MHKF. The probability of the hypothesis is automatically assigned to the corresponding field in the container which is determined by the associated concept #concept:probability. The MHKF is iteratively applied with 10Hz. The options linear and staticObject indicate that a linear Kalman-Filter is used and that no velocity of the object should be estimated and the velocity is not considered when applying the motion model. Of course, the MHKF also has to know which data it has to work on. The input of the Kalman-Filter is given as the last observed allocentric ball positions of all the robots in the team.
Listing 8 shows the specification of the filter calculateRoleAssignment. As for all filters, the definition starts with creating a DMC or DMC array to store the results of the filtering. Here a ringbuffer array named roles is defined to store the roles of all the robots (#concept:Role). Each array element is a ringbuffer that contains exactly one element of type string that is initialized with the string "None". At the beginning of the filter body the index of the shared ball hypothesis with the highest probability is calculated using the max-function. This index is used to store the corresponding shared ball hypothesis in teamBall. Afterwards, the distances of the robots to the team ball are calculated and the index of the player nearest to the ball is determined. The following line associates the role "Attacker" with it. Next, we determine the player which is nearest to the own base line – indicated by the minimal x-coordinate of the corresponding robot position – and has no role associated yet. This is achieved by using a for-loop and an appropriate if-condition. Afterwards, the resulting player gets the role "Defender". All remaining players are associated with the role "Supporter", which is also done using a for-loop.
Now the specification of the SharedWorld module is nearly complete. Only the DMCs sharedBall and roles have to be exported, in order to make them available for access from other parts of the underlying software framework (not shown here).
With the help of all theses specifications, the SPICA development environment is able to generate source code for the whole communication and collaboration infrastructure. For this purpose, the AASTra tool has to be told about the target platform the modules and data structures should be generated for, and the desired communication scheme (IP multicast) has to be configured. After the transformation, the resulting modules and classes can easily be integrated into the underlying communication frameworks. Only an instance of the generated module has to be created as a singleton and the DMCs can be accessed by the generated API.
Listing 9 illustrates the corresponding source code fragments for integration into the Carpe Noctem framework in C#. First, a callback method GetSharedBallHypotheses is defined. It is called when the sharedBall DMC of the SharedWorld model has been changed. This is the case, every time the MHKF has finished an iteration. In this simple example, the method just writes all hypotheses to the console. Afterwards, an instance of the SharedWorld module is created and the callback is added as a delegate to the Changed-event of the SharedBall property for the respective DMC. The following line shows how the role of the current robot can be accessed (MyBuffer returns the DMC of the current robot from the DMC array). The rest of the source code fragment creates an instance of the Communication module, creates a new BallMessage, initializes its fields, and adds it to the corresponding DMC. The transmision of the message is handled by the Communication module as specified above.

```
using Spica.Modules;
using Spica.Messages;

...

protected void GetSharedBallHypotheses(Module m) {
   SharedWorld sw = (SharedWorld)m;

   Console.WriteLine("SharedBall Hypotheses: {0}",
      sw.SharedBall.ToString());
}

SharedWorld sw = SharedWorld.GetInstance();

sw.SharedBall.Changed += GetSharedBallHypotheses;

Console.WriteLine("Own Role: {0}", sw.Roles.MyBuffer.Last);

Communication c = Communication.GetInstance();

BallMessage bm = new BallMessage();

bm.BallPos.X = 1000.0;
bm.BallPos.Y = 1000.0;
bm.BallPos.Certainty = 1.0;

c.BallMessage.Add(bm);
```

Listing 9. C# fragment showing the integration of SPICA-generated code

6. Related Work

Research on robot software architectures in the past mostly focused on middleware frameworks for autonomous robot development. Abstraction layers in this approach simplify access to robotic hardware and make it more convenient to use. By adding abstract interface definitions and APIs, modular programming is promoted.

Our approach shifts the focus right to the development process, a conceptually even more abstract level. We address the way systems have to be developed and the question what has to be implemented. The goal is to make the development process and the implementation more platform-independent, enabling the developer to focus on the actual functionality rather than bothering with characteristics of the platform. Our development environment for robotic software neither has hard dependencies on hard- or software architectures nor on operating systems. We provide modelling facilities that are focused on the respective program domain such as multi-party interaction or distributed sensor fusion. By combining ideas from the model-driven development movement with lessons learned from the development of middleware frameworks, a powerful development tool chain is provided.

As robotic systems are normally quite reactive and the system configuration is likely to be modified during the development process, one key requirement is the ability to incorporate new or existing components into the given software architecture. Furthermore, especially AMRs have to be able to use heterogeneous hardware devices, cope with physical variability in measurement, and bypass architectural mismatches. Several approaches have been proposed in the last years which try to provide suitable solutions or address similar problems in other related areas. A range of solution and projects is outlined below.

6.1 Middleware Frameworks

Several middleware frameworks utilize the concept of abstraction layers to ease the development of robotic software in a heterogeneous environment.

MARIE (Mobile and Autonomous Robotics Integration Environment) (Côté et al., 2006) is a middleware framework for robots that targets the development and integration of software components. It provides the Mediator Interoperability Layer (MIL), a design pattern that offers a common interaction language for components in the system. MARIE itself is written in C++ for UNIX environments. CLARAty (Coupled Layer Architecture for Robotic Autonomy) (Volpe et al., 2001) is an object-oriented framework for robotic systems which focuses reusability and integration of algorithms and components. It basically reduces the software hierarchy to two layers, a decision and an execution layer; realizations of functional requirements can be integrated into the decision layer while the execution layer is not affected. Another object-oriented framework for robotic applications is MIRO (Middleware for Robots) (Utz et al., 2002). It provides abstraction from system-specific implementations and is based on the ACE/TAO (Schmidt et al., 1997) framework. A device layer features hardware abstraction and takes care of the operating system integration. A communication layer offers services required in distributed systems. A Service Layer finally provides abstractions for sensors and actuators by decoupling the device interfaces from the driver implementations.

Similar to MIRO where skeletons for sensors and actors can be described using an Interface Definition Language (IDL), the Reconfigurable Context-Sensitive Middleware (RCSM) (Yau et al., 2002) uses a newly defined IDL to specify context requirements. It is a middleware framework supporting the development of context-aware applications focusing on spontaneous interactions. Application skeletons are generated from the IDL specifications which interact with the RCSM Object Request Broker (R-ORB), the context management processor in RCSM.

The Pervasive Autonomic Context-aware Environments (PACE) (Henricksen et al., 2005) middleware provides tools for validating context models, generating stubs for different languages, and accessing context from different programming languages and platforms. It provides a context management system (CMS) with a distributed set of content management repositories. The queries to the CMS can be placed using RMI or automatically generated stubs, for example.

In contrast to the approaches outlined above, SPICA is no middleware framework but a development environment aiming at platform-independent specifications and automatic code generation. Therefore, we address a conceptually different level. Besides, its flexible code generation system easily adapts to new target languages and we focus on a convenient modelling and on lean generated code.

AMQP (Advanced Message Queuing Protocol) (http://www.amqp.org/) is an open standard messaging middleware. It was developed first off to meet the needs of investment banks, employing a network-friendly, binary protocol. Similar to the DMCs used in SPICA, AMQP provides queues to accomplish a store-and-forward semantic. Message routing and delivery is due to centralized message broker systems.

The implementation generated by SPICA exhibits MOM characteristics, as well. In contrast to AMQP, however, SPICA also supports decentralized peer-to-peer techniques.

6.2 Development Environments

A quite different approach is followed by the Microsoft Robotic Studio (http://msdn.microsoft.com/robotics/). It is a development environment for robotics that targets different robot platforms. It builds on the .NET framework and offers a runtime as well as a powerful simulation environment. Besides the programming languages available in .NET, a so-called Visual Programming Language may be used for development of robotic software. Therefore, it can be considered as a model-driven software development approach. CoSMIC (Component Synthesis using Model-Integrated Computing) (Gokhale et al., 2003; Balasubramanian et al., 2005) is another development environment which follows the paradigm of MDD. It is a collection of domain-specific modelling languages and generative tools for the development, configuration, deployment, and validation of distributed component-based real-time systems.
Both approaches are similar to SPICA. The Microsoft robotics studio targets rapid development of robot control software but focuses more on prototyping than on efficient and domain-adapted solutions. CoSMIC is a complex, model-driven approach that follows very similar goals. In contrast, our approach aims to be lightweight and allows for rapid development and easy integration.

6.3 Context Management Systems

In the area of context-aware computing applications and middleware services use information about their execution environment to adapt their functional and non-functional behaviour for appropriate quality of service in every situation. For this purpose, context management systems are required which collect context information and make them accessible for adaptation reasoning. However, in a pervasive computing environment it is very likely that context information originate from heterogeneous sources. Therefore, many research activities addressing the development of context management systems also focus on heterogeneity issues. Examples are RCSM and PACE already mentioned above, but also the Context Toolkit (Salber et al., 1999), CoCo (Buchholz et al.) and CoBrA (Chen et al., 2003). While RCSM and PACE aim at providing an infrastructure to integrate heterogeneous context providers, the Context Toolkit, CoCo, and CoBrA focus on the interpretation of context information from heterogeneous sources. In particular, CoCo and CoBrA are related to our approach as they claim the necessity of using ontologies to establish a common understanding of the semantics of context information and their representations. However, here it has to be distinguished between approaches using ontologies for runtime reasoning and for code generation purposes as in our case. Our approach also has many similarities to the Context Ontology Language (CoOl) already mentioned above, as they also deal with different representations and define operations to convert between them. However, in our approach the operations for conversion are not defined explicitly, but we aim at automatically deriving the conversion methods from the definition of coordinate systems and references to measurement units.
In general, the development of a cooperative world model has many similarities to the development of context management systems. Here too, information about the current

environment, like the ball position, player position etc., has to be collected and calculations have to be performed on them. In the area of context aware computing this is referred to as context reasoning. There are also some approaches providing development support and patterns for context reasoning. An example is the work done by Chen et al. (Chen et al., 2004). They propose the use of Context Fusion Networks (CFNs) to provide data fusion services with regard to the aggregation and interpretation of sensor data to context-aware applications.

7. Conclusion and Future Work

Because of the lack of standard software, which prompts every RoboCup team to develop its own software framework, heterogeneity issues play a decisive role. They cause several problems when establishing a mixed-team of soccer robots involving different hardware and software platforms.

In order to cope with theses issues, we presented SPICA, a development environment for communication and collaboration infrastructures for heterogeneous teams of soccer robots. In SPICA, we have adopted a model-driven development approach which is naturally very appropriate to cope with heterogeneity. One of its basic paradigms is the platform-independent specification of software allowing automatic generation of source code for different platforms. Accordingly, SPICA provides a modelling language and tools facilitating the specification of communication and collaboration infrastructures as well as the automatic transformation of the resulting models into source code.

The SPICA modelling language consists of three domain-specific sublanguages, which are tailored to different aspects of the infrastructure. The MDL allows the specification of messages and containers along with their representations. The DFDL provides specification means for module stubs, the data flow between them, and for their data management capabilities. In order to allow a flexible filtering of data and to support the creation of a cooperative world model, the DADL was developed. It is a general purpose language for calculations on the exchanged data and also provides some predefined patterns for data fusion. As illustrated in a detailed example, the development effort for a team-play in heterogeneous teams of soccer robots can be reduced significantly with the help of SPICA. The generated source code can be integrated into the existing software framework very easily and with very little effort.

However, as already mentioned above, the development of SPICA is still work in progress. In particular, this is true for the DADL and the corresponding transformation support. Appropriate support for code generation is available only for a subset of the predefined data-fusion patterns at the moment and only a basic set of predefined functions is integrated into the language. In the future, we will enhance the language and the corresponding code generation tools with regard to especially these issues. We also aim at integrating support for defining functions and calling functions from external libraries. Besides, as not only the programming of a complex communication infrastructure is a challenging task, but also its configuration, we try to include support for self-configuration of the generated infrastructures into the SPICA environment.

However, we are quite confident that with the SPICA development framework one important step was made towards the realization of cooperative team organization. It is our vision that teams provide and publish descriptions of the messages and corresponding data they would like to communicate. For new mixed teams only the tactics would have to be specified then; the appropriate communication and collaboration infrastructure is generated by the SPICA development framework automatically.

8. References

Akashi, H. & Kumamoto H. (1977). Random sampling approach to state estimation in switching environments. *Automatica*, Vol. 13, pp. 429–434.

Aström, K. J. (1965). Optimal control of markov decision processes with incomplete state estimation. *Journal of Mathematical Analysis and Applications*, Vol. 10, pp. 174–205.

Baer, P. A.; Reichle, R.; Zapf, M.; Weise, T. & Geihs, K. (2007). A Generative Approach to the Development of Autonomous Robot Software. *Proceedings of the Fourth IEEE International Workshop on Engineering of Autonomic & Autonomous Systems (EASe 2007)*, pp. 43-52.

Balasubramanian, K.; Krishna, A. S.; Turkay, E.; Balasubramanian, J.; Parsons, J.; Gokhale, A. & Schmidt, D. C. (2005). Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems. *International Journal of Embedded Systems, special issue on Design and Verification of Real-Time Embedded Software.*

Buchholz, T.; Krause, M.; Linnhoff-Popien, C. & Schiffers, M. (2004). Dynamic Composition of Context Information. *Proceedings of 1st Ann. International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous 04)*, pp. 335-343.

Castelpietra, C.; Iocchi, L.; Nardi, D.; Piaggio, M.; Scalzo, A. & Sgorbissa, A. (2000). Coordination among heterogeneous robotic soccer players. *Proceedings of Intelligent Robots and Systems 2000 (IROS 2000)*, Vol. 2, pp. 1385-1390, ISBN 0-7803-6348-5, Takamatsu, Japan.

Chen, G.; Li, M. & Kotz, D. (2004). Design and implementation of a large-scale context fusion network. *Proceedings of 1st Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous)*, pp. 246–255, IEEE Computer Society.

Chen, H.; Finin, T. & Joshi, A. (2003). Using OWL in a pervasive computing broker. *Proceedings of Workshop on Ontologies in Agent Systems*, July 2003.

Côté, C.; Brosseau, Y.; Létourneau, D.; Raïevsky, C. & Michaud, F. (2006). Robotic Software Integration Using MARIE. *International Journal of Advanced Robotic Systems, Special Issue on Software Development and Integration in Robotics*, Vol. 3, No. 1, pp. 55-60.

Dempster, A. P. (1968). A generalization of Bayesian inference, Journal of the Royal Statistical Society, Vol. 30, Series B, pp. 205-247.

Fox, D.; Burgard, W. & Thrun, S. (1999). Markov localization for mobile robots in dynamic environments. *Journal of Artificial Intelligence Research*, Vol. 11, pp. 391–427.

Gokhale, A. S.; Schmidt, D. C.; Lu, T.; Natarajan, B. & Wang, N. (2003). CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Applications. *Proceedings of Middleware Workshops*, pp. 300–306.

Handschin, J. E. & Mayne, D. Q. (1969). Monte carlo techniques to estimate the conditionalexpectation in multi-stage non-linear filtering. *International Journal of Control*, Vol. 5, No. 5, pp. 547–559.

Henricksen, K.; Indulska, J.; McFadden, T. & Balasubramaniam, S. (2005). Middleware for Distributed Context-Aware Systems. *On the Move to Meaningful Internet Systems 2005*, LNCS 3760, pp. 846-863, Springer.

Isik, M.; Stulp, F.; Mayer, G. & Utz, H. (2007). Coordination without Negotiation in Teams of Heterogeneous Robots. In: *RoboCup 2006: Robot Soccer World Cup X*, Vol. 4434, ISBN 978-3-540-74023-0, to appear.

Kalman, R.E. (1960). A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, Vol. 82, Series D, pp. 35–45.

Nardi, D.; Adorni, G.; Bonarini, A.; Chella, A.; Clemente, G.; Pagello, E. & Piaggio, M. (1999). ART – Azzurra Robot Team. In: *RoboCup-99: Robot Soccer World Cup III*, LNCS 1856, pp. 15-39, Springer.

Nesnas, I. A.; Simmons, R.; Gaines, D.; Kunz, C.; Diaz-Calderon, A.; Estlin, T.; Madison, R.; Guineau, J.; McHenry, M.; Shu, I.-H. & Apfelbaum; D. (2006). CLARAty: Challenges and Steps Toward Reusable Robotic Software. *International Journal of Advanced Robotic Systems, Special Issue on Software Development and Integration in Robotics*, Vol. 3, No. 1, pp. 23-30.

Pearl, J. & Kaufmann, M. (1988). Probabilistic Reasoning in Intelligent Systems. San Mateo CA, ISBN 0-934613-73-7.

Reid, D. (1979). An algorithm for tracking multiple targets. *IEEE Transactions on Automatic Control*, Vol. 24, Nr. 6, pp. 843–854.

Salber, D.; Dey, A. K. & Abowd, G. D. (1999). The Context Toolkit: Aiding the Development of Context-Enabled Applications. *Proceedings of Conference on Human Factors in Computing Systems (CHI '99)*, pp. 434-441, Pittsburgh, PA, May 15-20, 1999.

Schmidt, D. C.; Gokhale, A.; Harrison, T. & Parulkar; G. (1997). A high-performance endsystem architecture for real-time CORBA. *IEEE Communications Magazine*, Vol. 14, No. 2.

Selic, B. (2003).The pragmatics of model-driven development. IEEE Software, Vol. 20, No. 5, pp. 19–25.

Shafer, G. (1976). A Mathematical Theory of Evidence. Princeton University Press.

Strang, T.; Linnhoff-Popien, C. & Korbinian, F. (2003). CoOL: A Context Ontology Language to enable Contextual Interoperability. *Proceedings of 4th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS 2003)*, LNCS 2893, pp. 236-247, ISBN 3-540-20529-2, Springer.

Thrun, S.; Fox, D.; Burgard, W. & Dellaert, F. (2000). Robust monte carlo localization for mobile robots. *Artificial Intelligence*, Vol. 128, No. 1-2, pp. 99–141.

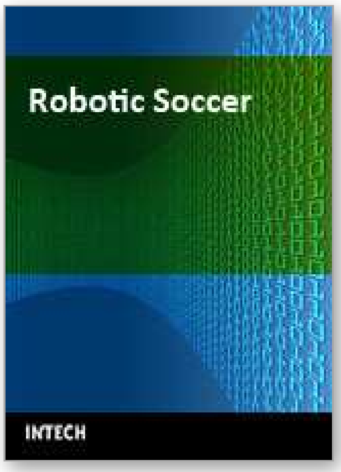
Utz, H.; Sablatnög, S.; Enderle, S. & Kraetzschmar, G. K. (2002). Miro–Middleware for Mobile Robot Applications. *IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures*, Vol. 18, No. 4, pp. 493-497.

Utz, H.; Stulp, F. & Mühlenfeld, A. (2004). Sharing Belief in Teams of Heterogeneous Robots. In: *RoboCup 2004: Robot Soccer World Cup VIII.* LNCS 3276, pp. 508-515, Springer.

Volpe, R.; Nesnas, I.; Estlin, T.; Mutz, D.; Petras, R. & Das, H. (2001). The CLARAty Architecture for Robotic Autonomy. *Proceedings of IEEE Aerospace Conference*, Big Sky, Montana.

Yau, S. S.; Karim, F.; Wang, Y.; Wang, B. & Gupta, S. K. S. (2002). Reconfigurable Context-Sensitive Middleware for Pervasive Computing. *IEEE Pervasive Computing,* Vol. 1, No. 3, pp. 33-40.

Zadeh, L. A. (1978). Fuzzy sets as a basis for a theory of possibility. Fuzzy Sets and Systems, Vol. 1, pp. 3–28.



Robotic Soccer

Edited by Pedro Lima

ISBN 978-3-902613-21-9

Hard cover, 598 pages

Publisher I-Tech Education and Publishing

Published online 01, December, 2007

Published in print edition December, 2007

Many papers in the book concern advanced research on (multi-)robot subsystems, naturally motivated by the challenges posed by robot soccer, but certainly applicable to other domains: reasoning, multi-criteria decision-making, behavior and team coordination, cooperative perception, localization, mobility systems (namely omni-directional wheeled motion, as well as quadruped and biped locomotion, all strongly developed within RoboCup), and even a couple of papers on a topic apparently solved before Soccer Robotics - color segmentation - but for which several new algorithms were introduced since the mid-nineties by researchers on the field, to solve dynamic illumination and fast color segmentation problems, among others. This book is certainly a small sample of the research activity on Soccer Robotics going on around the globe as you read it, but it surely covers a good deal of what has been done in the field recently, and as such it works as a valuable source for researchers interested in the involved subjects, whether they are currently "soccer roboticists" or not.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Philipp A. Baer and Roland Reichle (2007). Communication and Collaboration in Heterogeneous Teams of Soccer Robots, Robotic Soccer, Pedro Lima (Ed.), ISBN: 978-3-902613-21-9, InTech, Available from: http://www.intechopen.com/books/robotic_soccer/communication_and_collaboration_in_heterogeneous_teams_of_soccer_robots

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2007 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike-3.0 License, which permits use, distribution and reproduction for non-commercial purposes, provided the original is properly cited and derivative works building on this content are distributed under the same license.