

A Generative Approach to the Development of Autonomous Robot Software

Philipp A. Baer, Roland Reichle, Michael Zapf, Thomas Weise, and Kurt Geihs
Distributed Systems Group, University of Kassel
Wilhelmshöher Allee 73, 34131 Kassel, Germany
{baer,reichle,zapf,weise,geihs}@vs.uni-kassel.de

Abstract

The integration of new or existing software components into established architectures and the ability to deal with heterogeneity are key requirements for middleware and development frameworks for robotic systems. This paper presents SPICA, a software development framework for communication infrastructures of autonomous mobile robots. Utilizing the model-driven software development paradigm, communication and data flow can be defined on an abstract level. For this purpose, domain-specific languages and tools are provided that allow specification and generation of module communication infrastructures for communication between modules along with primitives for data management. The high-level platform-independent specifications are automatically transformed into low-level platform and programming language-specific source code. We illustrate the applicability of our approach with an elaborate example describing the design of a soccer robot architecture that has proven its strength during RoboCup 2006. Our experiences have revealed that SPICA is advantageous for prototyping as well as for building high performance systems.

1. Introduction

Software systems for autonomous robots consist of different kinds of software components. Sensors enable the robots to retrieve information from their environments. Actuators, in contrast, are used to interact with the environment. These components are normally located directly on the robot. They resemble device drivers which need to be tightly coupled with the software. In addition, robotic systems typically comprise components for world modeling, self localization, behavior reasoning, or path planning, which realize the artificial intelligence part of the system. These components can be located directly on the robot or realized as self-contained modules (processes) located on remote systems as well.

Modularity has shown to be advantageous for large and complex software systems and is a fundamental design principle for distributed applications. However, this is not always only an abstract design objective. Often it is a necessity implied by specific characteristics of modular robotic systems. Image processing, for example, has to be implemented mainly with efficiency in mind. This requires programming languages which introduce minimal overhead in computations like C or C++. For less computation-intensive tasks or for programs that have to be implemented in a very portable way, other languages like Java or C# may be preferable. Different programming languages or even different platforms normally imply using separate processes as well. Finally, integration of third party modules is facilitated by a modular architecture with clear and simple interfaces.

Nevertheless, modularity also involves complication to some degree: Interaction between modules has to overcome process boundaries. A very flexible and fast inter-process communication (IPC) scheme is socket communication. It allows a module to communicate with other modules on the same or remote systems. Socket interfaces are available on all common software platforms and thus facilitate portability as well. Socket communication is message or stream-based, so data exchanged between modules have to be serialized before transmission and deserialized after reception.

In this paper we present SPICA, a development framework for modular robot architectures, together with an elaborate real-life example. SPICA assists in integrating software modules realized in different programming languages and for different platforms in heterogeneous distributed environments. The focus of SPICA is on the specification and generation of communication infrastructures needed for module integration. We utilize the *Model-Driven Development* (MDD) [18] approach by providing appropriate specification means and support for model transformation. SPICA provides tools that are able to generate platform-specific implementations in Java, C++, and C# from abstract platform-independent models. Our template-based approach enables easy integration of further programming languages.

A developer defines module communication interfaces with data management capabilities as well as communication protocols tailored to mobile ad-hoc interaction schemes. Message structures are specified in a platform-independent manner similar to ASN.1. The developer models the architecture of the robotic system in terms of collaborating modules and directed communication channels. Tests have revealed that the approach is suitable for prototyping as well as for building complex high performance systems.

The remainder of the paper is organized as follows. In the next section we discuss related work facing similar challenges or using similar techniques. Section 3 presents the overall approach and introduces the tools provided in the SPICA framework. Section 4 describes the specification languages. An elaborate example in section 5 illustrates an application of SPICA: The design of a software architecture for autonomous soccer robots. The last section concludes the paper, summarizes the main contributions, and points at future work.

2. Related Work

The ability to incorporate new or existing components is one of the key requirements for robotic systems. Integration of new capabilities and the ability to deal with heterogeneity is an important issue as well. Robotic systems have to be capable of controlling heterogeneous hardware and cope with physical variability and architectural mismatches [13]. Several approaches have been proposed in the last years which try to provide suitable solutions. We will introduce some of them shortly below.

The *Mobile and Autonomous Robotics Integration Environment* (MARIE) [7] is a middleware framework for robots that targets development of new and integration of existing software components. It comprises a middleware layer to interconnect components. The main building block of MARIE is the *Mediator Interoperability Layer* (MIL), a design pattern that offers a common interaction language for components in the system. *Application Adapters* (AA) interface applications with the MIL. So-called *Communication Adapters* (CA) are communication endpoints for components with incompatible communication mechanisms and protocols. MARIE itself is written in C++ for UNIX environments. Wrapper modules exist for some established middleware and development frameworks. To integrate own or third party components, however, new wrappers have to be provided.

It is a very common approach to facilitate integration and reusability of components through abstraction layers. Issues like communication and decision making can so be separated. CLARATy (*Coupled Layer Architecture for Robotic Autonomy*) [23, 13] is a framework for robotic systems with

a focus on reusability and integration of existing algorithms and components. It is a generic object-oriented framework that decomposes the software architecture into a decision and an execution layer. Realizations of functional requirements can be integrated into the decision layer, while the execution layer is not affected.

Miro (*Middleware for Robots*) [22] also features a layered software design based on the ACE/TAO CORBA framework [16]. The device layer features hardware abstraction and takes care of the operating system integration. The communication layer offers services required in distributed systems. The Service Layer provides abstractions for sensors and actuators by decoupling the device interfaces from the driver implementations.

Microsoft Robotic Studio [10] is a development environment for robots featuring a runtime environment supporting different robot platforms and a simulation environment. It builds on WebService infrastructures.

CoSMIC (*Component Synthesis using Model-Integrated Computing*) [8, 3] follows the paradigm of MDD. It is a collection of domain-specific modeling languages and generative tools supporting development, configuration, deployment, and validation of distributed component-based real-time systems. Applications created with CoSMIC are based on the ACE/TAO CORBA framework. The implementation language is C++ other languages are not directly addressed but CORBA-enabled components can be integrated natively.

The OROCOS (*Open Robot Control Software*) [4, 5] project provides a general-purpose, real-time-oriented, modular framework for robot control. Templates help new users to create applications without in-depth knowledge of OROCOS. Components can be interfaced in different ways: callbacks are one way to handle events. OROCOS is freely available and written in C++.

The architectures and development environments for robotic systems analyzed here try to foster generic interaction between modules. Most are based on CORBA or SOAP. Examples are Miro, CoSMIC, OROCOS, and Microsoft Robotics Studio. These implementations are heavyweight and introduce additional complexity which is a great disadvantage in applications where communication speed is vital. Our approach thus focuses on a simple, lean, and fast communication infrastructure with domain-specific customizable protocols tailored to ad-hoc group communication.

Most middleware architectures or frameworks are concrete implementations in specific languages for specific platforms. They lack the important ability to mediate between processes written in different programming languages and running on different platforms. SPICA not only meets this requirement, its flexible template system can furthermore easily be adapted to new target languages.

3. The SPICA Approach

A central goal of the MDD approach is to separate the design and architecture from concrete realizations. The developer should be able to design the application at an abstract level and not be confronted with platform-specific implementation details. The conceptual design realizing the functional requirements is specified in the *Platform-Independent Model* (PIM). It is then transformed to one or more *Platform-Specific Models* (PSM) which provide the base for the actual implementation.

In SPICA, the developer focuses on the overall architecture of the robotic system in terms of modules and communication channels between them. Here, the platform independent model is called *Abstract Architecture Specification* (AAS). The transformation process from the AAS to the finally resulting source code in one or multiple programming languages is outlined in Figure 1.

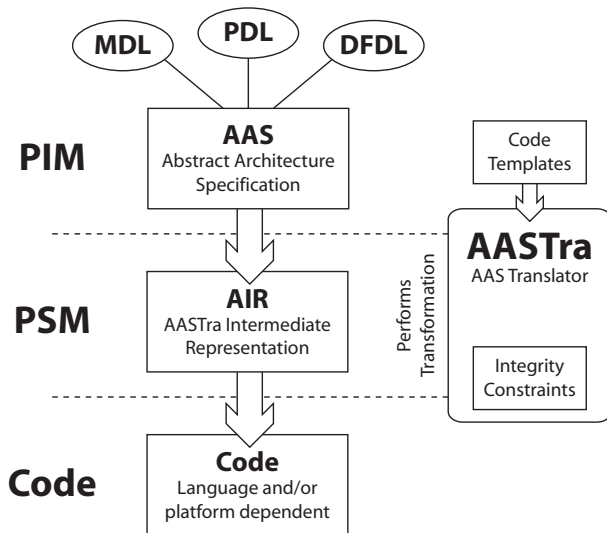


Figure 1. General development approach

The main AAS modeling entities are **modules**, **messages**, and **protocols**. Modules represent the building blocks of the architecture. Messages are data structures exchanged between the modules, while protocols are descriptions of the transmission behavior.

The AAS is defined using three domain-specific modeling languages: the *Data Flow Description Language* (DFDL), *Message Description Language* (MDL), and *Protocol Description Language* (PDL). They are tailored to the different aspects of the architecture and will be introduced in section 4. These three languages are used in order to clearly separate the orthogonal concerns of data flow, message structure and protocol state transitions.

The AAS can be considered as the overall system model, which represents the layout of the target software architec-

ture. At the moment, it is prepared in a text format but we are currently working on a graphical composer incorporating UML 2.0 [14].

The *AAS Transformator* (AASTra) is the central part of SPICA. It performs integrity checks and model transformations from the AAS down to the concrete implementation. The *AAS Intermediate Representation* (AIR), generated from the AAS model, forms a data pool for the final code transformation step and represents the PSM. AIR is a tree representation of the AAS. It is directly transformed to source code using language-specific templates. For this purpose, AASTra incorporates StringTemplate [20], a powerful template engine. The AAS specification is parsed and interpreted with the help of ANTLR [1], an advanced parser generator.

The results of the model transformation are module skeletons providing implementations of messages, protocols, and communication interfaces along with data management capabilities. After the code generation process, a basic realization of the robotic system is available which provides modules capable of communicating and handling data in the specified way.

4. AAS Languages

AAS represents the PIM layer in SPICA. Three different domain-specific languages cover the basic requirements for inter-module communication in autonomous robot systems. These languages are introduced below.

4.1. Data Flow Description Language

In robotic systems the drivers for sensors and actors may be realized in independent modules. Other functional entities may as well be realized in such a way. Here, modularity often implies a directed data flow from source to sink modules or from sensors to data pools. Therefore, we consider communication channels being unidirectional. They are sufficient for the scenario described above and may easily be extended to bidirectional channels.

Managing incoming and outgoing data is another task that modules have to deal with. They have to deliver real-time characteristics without caching messages or integrate data over time, for instance. Real-time capabilities are especially important for message forwarding where each hop introduces latencies and processing overhead. Our experiences have shown that only a small set of basic data management primitives is required to cover issues as sketched above.

The *Data Flow Definition Language* (DFDL) comprises three main conceptual entities: **Modules**, **unidirectional connections** and, **data management** primitives as mentioned above. These entities are outlined below.

4.2. Module Model

A module consists of the name, target language, target platform, and the module type. The module type indicates where the module will be executed, locally or remote. It actually specifies how connections are handled with regard to the implementation, e.g. to optimize communication schemes if only system-local communication is required. The general structure of a module specification is shown below in figure 2. It also provides an overview of the connection model described next.

4.2.1 Connection Model

Connections are unidirectional communication channels established between exactly two module types. This simplifies the specification of the data management behavior since sender and receiver are clearly distinguished.

A connection specification consists of a start point (**Source**) and an end point (**Destination**), i.e. the names of the modules that should be connected. For each connection a transport protocol (**T-Protocol**) must be specified. This is a socket-level protocol natively supported by the underlying system. Our model currently supports UDP in different occurrences (unicast, broadcast, and multicast), TCP, and UNIX domain sockets. Optionally, a custom protocol (**C-Protocol**) can be specified. C-Protocols are specified using the Protocol Definition Language (PDL) introduced below.

For data management three basic types of data management schemes are introduced. Callbacks implement a kind of pass-through semantic, which is required for real-time message processing, for instance. Ringbuffers provide a fixed buffer size. Old messages are overwritten if the buffer is full. Finally, queues implement a variable-sized message buffer for reliable message handling. These basic schemes exhibit special characteristics for senders and receivers, depending on their needs.

The data management is not only responsible for data caching and relaying but also handles asynchrony. Asynchronous event handling is challenging since it may lead to race conditions. In order to prevent this, the data management synchronizes events. This has some impact on real-time processing but is negligible compared to transmission latency.

The logical GROUP module acts as a hub for messages communicated between different instances of a System Model. It furthermore facilitates communication with third party modules. Multiple GROUP modules can be used by a System Model, each of which constitutes a confined hub. If possible, a message of a given type is exclusively forwarded to systems that accept it. If no such system is available the message is dropped.

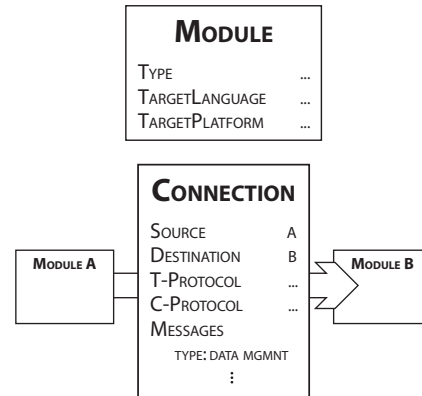


Figure 2. Module and connection models

The concept of a logical GROUP module indicates a transition between two or more instances of the System Model. It avoids ambivalences when different instances of one module communicate with each other.

4.2.2 Protocol Description Language

Ad-hoc networking often implies limitations in bandwidth, reliability, or quality. This is especially true for mobile robot scenarios with ad-hoc characteristics which render many established protocols unusable. Furthermore security implications are omnipresent. For mobile robots that communicate, message replay can affect the behavior rather seriously, for example. Thus, protocols tailored to the environment and aware of these limitations are eligible.

Over the last years we have analyzed the situation of communication between mobile robots [2] in RoboCup tournaments. The Transmission Control Protocol (TCP), for example, is accepted and wide spread. Nevertheless, it was created for networks with low error rates and is thus not suited well here. Solutions for unreliable ad-hoc networks are available but the precise needs of reactive, autonomous, mobile systems, like real time behavior, reliable transmission, and resistance against attacks, are often not met. Different technologies have to be incorporated into a protocol specifically tailored to a given scenario that meets the requirements outlined above.

In general, protocols cannot be specified using static schemes. The activity strongly depends on the previous and the current states. They are, thus, commonly specified using *Finite State Machines* (FSM). This is where our approach takes up.

The *Protocol Definition Language* (PDL) supports modeling of protocols using an *Extended Finite State Machine* (EFSM). The protocol model consists of states which can be switched using transitions and shared context information. This is required to keep track of message identifiers

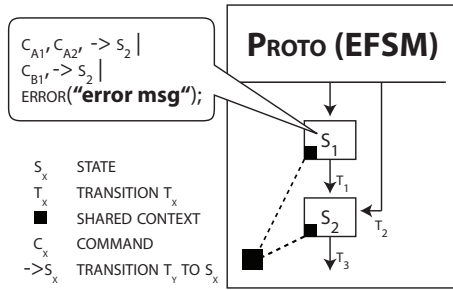


Figure 3. PDL model

or state information, for example. Each state represents one functional part of the protocol and defines actions to be performed on the messages. For the specification of actions we have defined a domain-specific language. It incorporates principles found in logical programming or network packet filtering which simplify specification substantially. The grammar is given in EBNF [9] in listing 1.

```
Name ::= [a-zA-z][a-zA-Z0-9]*
Value ::= ?variable, function, string, etc?
String ::= '"' [^"]* '"'
StateAction ::= (Rule ";")*
Rule ::= Command (
  [ ", " | "|" ] ( Command | "(" Rule ")" )
)*
Command = CheckType | Trans | Assign |
  Error | Call
CheckType ::= Name "is" Name
Trans ::= "->" Name
Assign ::= Name "=" Value
Call ::= ( "send" | "receive" |
  "retrieve" | "return" ) Value
Error ::= "Error(" String ")"
```

Listing 1. PDL language grammar

The state action is made up of one or more rules, each rule consists of several statements. Statements are logically connected. Once a statement fails, all preceding AND connected statements are rolled back. This is known as backtracking. If one or more OR operations are available, they are executed one by one in the same manner. A colon represents an AND connection, the pipe character an OR connection. A semi-colon marks the end of a rule. Since each statement implicitly acts as a conditional no designated conditional statements are available.

Figure 3 outlines the modeling approach. A translation into SDL or Message Sequence Charts (MSC) is possible.

4.2.3 Message Description Language

Messages that are exchanged in distributed and heterogeneous software environments must be interpreted consistently on each platform and in every programming lan-

guage. The *Message Description Language* (MDL) we introduce in this section allows specifying such messages. The criteria for the design of MDL and its requirements are mainly based on the following observations.

- Transmission bandwidth and quality are limited in mobile robot scenarios, thus an optimization in message size is desirable.
- A common data structure layout is required for different target platforms. The *Abstract Syntax Notation One* (ASN.1) [21] follows the same approach.
- To retain backward compatibility with modified message structures, some version management is highly desirable.

The MDL partially follows the well-known concept represented by *Interface Definition Languages* (IDL). IDL dialects are commonly used by middleware architecture such as CORBA or DCOM [19] to describe remote interfaces.

MDL differs from ASN.1 and IDL in two points. It allows customizing the automatic code generation process and supports generation of message implementations that are version aware. The encoding scheme used by the message implementations can be customized or replaced

An MDL specification basically contains structure definitions that represent the layout of messages. These structures may contain other structures; namespaces are used for logical separation. The generic structure of MDL is shown in listing 2. The grammar is again specified using EBNF.

```
Name ::= [a-zA-z][a-zA-Z0-9]*
String ::= '"' [^"]* '"'
Modifier ::= 'ident' | 'field'
TagName ::= Name
TypeName ::= Name
TagAssign ::= TagName '=' String
Namespace ::= 'namespace' Name
  '{' [ Namespace | Struct ] '}'
Struct ::= 'struct' Name
  ( 'extends' Name )?
  ( '[' TagAssign ( "," TagAssign )* ']' )?
  "{" Field+ "}"
Field ::= Modifier TypeName Name
  ( '[' TagName ']' )? ";"
```

Listing 2. MDL grammar

Structures are containers for the message fields, each of which is either a primitive type or another structure, i.e. a composite type. Each field can be attributed either to the message header or to the message body. For this they have to be marked with the **ident** or **field** modifiers. Single inheritance of message structures is supported; derived structures contain all fields of the parent.

Versioning of structures is explicitly supported in MDL. Each version may contain new fields or change the alignment of fields, but only the body of a structure may be modified. The header must be left untouched.

Other characteristics of MDL are *tags* and *explicitly typed fields*. Tags are assigned to message fields. Each tagged field is then directly accessible in the code generation template. Furthermore, tagged fields can be assigned default values. For explicitly typed fields, the identifier of the type is encoded along with the value. This is useful for lists of similar structures which have to be encoded. Only composite types can be encoded explicitly since primitives do not have unique identifiers. This is one noticeable difference to ASN.1. An example of an MDL specification is shown in section 5.

5 Example

In this section, we will present an elaborate example that demonstrates the power of the SPICA framework. We have developed a software architecture for a team of autonomous soccer robots [6] that participated in the RoboCup [15] World Championships 2006 that took place in Bremen, Germany.

RoboCup is an international joint project attempting to foster research in robotics, artificial intelligence, and related fields. Soccer is a well understood highly dynamic game where a wide range of technologies can be integrated and examined. The robots are completely autonomous, i.e. have all the necessary sensors and control devices on board and must navigate autonomously without external contact. Figure 4 shows two of our robots fighting against one opponent.



Figure 4. The Carpe Noctem robots fighting against the robot of another team.

Our participation was a joint effort of two universities that formed a mixed team: the University of Kassel and the University of Ulm, both from Germany. Such teams always face the challenge to support communication between

two or more different hardware and software platforms. In our case, we had to establish communication with a robot software system based on Miro [22] Even modules within our own architecture are implemented in different programming languages because of task-specific needs. The image processing module, for example, must deliver high performance and is thus implemented in C++.

The required communication infrastructure of the robot software architecture was developed using SPICA. As SPICA has evolved since the World Championships, we will describe the architecture using the techniques we introduced above.

5.1. Architecture Design

For the design of our robotic system we follow a top-down approach. We start with the definition of the overall architecture by identifying its constituting modules and the data flow between them. The necessary message structures and custom protocols are specified thereafter. Figure 5 outlines the architecture layout.

5.1.1 Modules

The architecture mainly consists of six modules and a logical GROUP module. The function of these modules is outlined below.

The Vision module is responsible for image processing with focus on feature detection. Since an omni-directional camera is the main sensor of a robot the Vision constitutes the first and most important sensor module. World modeling, object tracking, behavior reasoning, and role assignment capabilities are concentrated in the Base. The Motion module interfaces the motor controller and provides odometry feedback. For communication with teammates and modules located on remote systems we introduce the Communication module. The Control Client/Viewer serves visualization and remote control purposes. The Joystick module offers a joystick interface for debugging and testing. The GROUP module also facilitates the communication between teammates.

Vision, Base, Motion, and Communication modules are located on the robot, Control Client/Viewer and Joystick can be executed on arbitrary systems. For performance reasons the Vision is implemented in C++. Java is chosen for the Control Client/Viewer because it is well suited for graphical client programs. All the remaining modules are implemented in C#. Since our robots are Linux-powered, we stick to the Mono .NET framework [12].

```
module Vision {
  Type = local;
  TargetLanguage = C++;
  TargetPlatform = Linux;
}
```

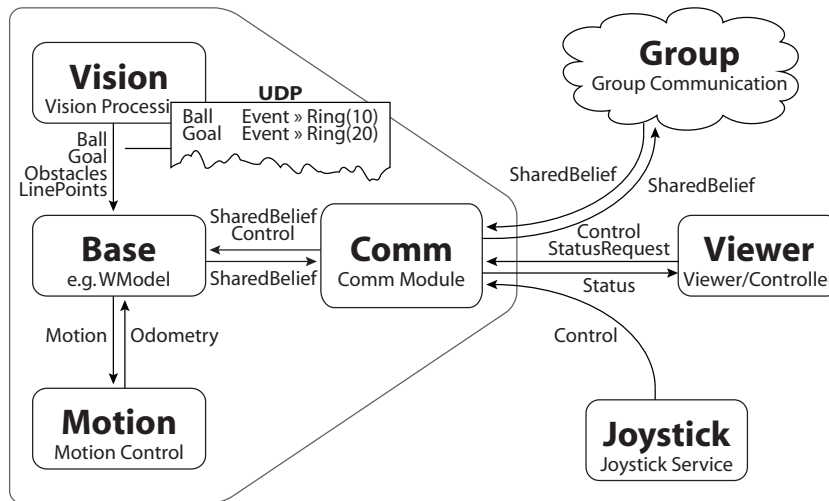


Figure 5. Data flow within the robotic software architecture.

```

module ControlViewer {
  Type = remote;
  TargetLanguage = Java;
  TargetPlatform = Win32;
}

```

The module definition for the Vision points out that it is executed on the robot and written in C++. The Control Client/Viewer may be run remotely and is written in Java for Win32. The platform is important here, because it could be possible that JNI be required.

```

connection Vision_Base {
  Source = Vision;
  Destination = Base;
  T-Protocol = UDP { Port = 11001; }
  C-Protocol = None;
  Messages = {
    (BallMessage : Event->Ringbuffer(10)),
    (GoalMessage : Event->Ringbuffer(20)),
    (ObstacleMessage : Event->Ringbuffer(10)),
    (LinePointMessage : Event->Ringbuffer(1))
  }
}

```

The Vision and the Base modules communicate via UDP without a custom protocol. All messages are sent immediately using the event-based data management scheme and stored in a non-blocking ring buffer. This is used by the Base to integrate data over time. Size 20 indicates a longer integration time, i.e. 20 messages. Size 1 indicates that only the most current data are relevant.

```

connection Communication_Group {
  Source = Communication;
  Destination = GROUP;
  T-Protocol = UDP/MCAST { Port = 11101; }
  C-Protocol = AuthAndCompress;
  Messages = {
    (SharedBelief : Event->Ringbuffer(10))
  }
}

```

The outgoing connection of the Communication module shows the application of the logical GROUP module. Communication emits only SharedBelief messages using UDP/Multicast as the transport protocol. It is not aware of the destination. The GROUP module indicates a transition between system instances. Both, SPICA modules and unknown third-party modules are potential receivers which is mandatory for communication in a mixed team. Furthermore, the number of receiving modules is not limited.

This channel uses the AuthAndCompress protocol which is introduced later. SharedBelief messages are sent immediately and inserted into a ring buffer on the receiver side. This is, however, only guaranteed for SPICA modules.

```

connection Communication_ControlViewer {
  Source = Communication;
  Destination = ControlViewer;
  T-Protocol = UDP { Port = 11008; }
  C-Protocol = None;
  Messages = {
    (SharedBelief : Event->Event),
    (Status : Synchronized(1)->Event),
    (Status : Event->Event)
  }
}

```

There is a second channel originating from the Communication module. This time it ends at the Control Client/Viewer. They exchange two types of messages via UDP, SharedBelief and Status messages. All messages are received asynchronously. The Status message is sent once every second (Synchronized(1)->Event) and it can be polled (Event->Event).

5.1.2 Data Structures

The specification of some messages shown in the data-flow models are given below. We will start with the most generic message from which all subsequent messages are derived.

```
struct BaseMessage [Type=0] {
  ident endianness;
  ident int8 type [Type];
}
```

BaseMessage merely contains only two fields: a field that describes the endian encoding of the fields and a field that identifies the message's type. All other messages are derived from BaseMessage. The BaseMessage is assigned type id 0.

```
struct BallMessage
  extends BaseMessage [Type=10]
{
  field double x;
  field double y;
  field double certainty;
}
```

A BallMessage is sent from the Vision to the Base module in the data flow specification above. It contains the computed coordinates of a ball. A certainty value indicates the probability of the detected object being the ball.

```
struct SharedBelief
  extends BaseMessage [Type=15]
{
  field BallMessage ball;
  field GoalMessage goal;
  field OpponentsMessage opponents;
  field (BaseMessage)[] addObservations;
}
```

For information exchange with other team members, SharedBelief messages are multicast by the Communication module. SharedBelief is a container holding several other messages. This avoids jamming the network caused by a great number of small packets. By default a BallMessage, a GoalMessage, and an OpponentsMessage are included. An arbitrary number of messages derived from BaseMessage can be attached for additional observations.

5.1.3 Protocol Definition

In order to guarantee that messages are only exchanged between teammates and to minimize network traffic we used the custom protocol AuthAndCompress already mentioned in the data flow model. The PDL specification below describes the protocol. It first authenticates then compresses outgoing and decompresses then verifies incoming messages.

The PDL specification is made up of the shared context information, transition handlers, and states. The shared context block contains a message and an address variable as well as two variables holding public key and one holding private key data for message authentication. These variables are accessible in all states.

```
protocol AuthAndCompress {
  transition on send outgoing;
  transition on receive incoming;
  context {
    Message _Msg;
    Address _Remote;
    Message _TMsg;
    Address _TRemote;
    PublicKey _pk;
    PublicKey _mypk;
    SecretKey _mysk;
  }
  incoming {
    receive [_Msg, _Remote] |
    Error("Error receiving message!");
    -> handle_incoming;
  }
  outgoing {
    retrieve [_Msg, _Remote],
    (
      AuthH ah,
      _mysk != null,
      ah.sig = auth(_Msg, _mysk),
      ah.payload = _Msg,
      (
        CompH ch,
        ch.payload = compress(ah) |
        Error("Unable to compress!");
      ) |
      Error("Unable to authenticate!");
    );
    send [ch, _Remote];
  }
  handle_incoming {
    _Msg is AuthReqPK,
    -> handle_pk_req |
    _Msg is CompH, (
      Msg = decompress(_Msg.payload) |
      Error("Unable to decompress")
    ),
    -> handle_verify |
    Error("Invalid message");
  }
  handle_verify {
    _Msg is AuthH |
    Error("Invalid message");
    _pk != null |
    -> handle_get_pk;
    verify(_Msg.payload) |
    Error("Unable to verify!");
    return [_Msg, _Remote];
  }
  handle_get_pk {
    AuthReqPKH apk,
    send [apk, _Remote],
    receive [_TMsg, _TRemote] timeout 3000 |
    Error("Unable to receive Public Key");
    _TMsg is AuthSendPK,
    _pk = _TMsg.pk,
    -> handle_verify;
  }
  handle_pk_req {
    AuthSendPKH apk,
    apk.pk = _mypk,
    send [apk, _Remote];
  }
}
```

Below the shared context two transition handlers are defined. The first activates the outgoing state if a send transition is received. The second activates the incoming state if a receive transition is received. They mimic the behavior of send and receive functions. The remaining definitions model states of the EFSM. They are described below shortly.

incoming The incoming state first tries to receive a message blocking infinitely long. If a message is received, a transition to `handle_incoming` is initiated.

outgoing The shared state variables `.Msg` and `.Remote` are first initialized using the `retrieve` call. Afterwards, an authentication header `AuthH` is created to which the signature of the message along with the message itself is attached. Finally, a compression header `CompH` is created with the compressed `AuthH` header attached. Authentication fails if no secret key is available.

handle_incoming If an `AuthReqPK` message is received, a transition to `handle_pk_req` is initiated. This message represents an inquiry from a remote system for the public key.

If a `CompH` message is received, the payload is decompressed and a transition to `handle_verify` is triggered. An error is returned in case of some other message.

handle_verify If an `AuthH` message is received and a public key is available, message verification is performed. If no error occurred, the message is returned to the system.

An error is returned if some other message was received. If no public key is available, a transition to `handle_get_pk` is triggered in order to retrieve the public key from the sender.

handle_get_pk This state is responsible for requesting the public key from a remote system. In order to do that a new `AuthReqPKH` message is sent. The protocol then waits for an answer for three seconds. If no answer is received, an error is returned. Otherwise, the public key is stored in the shared context.

handle_pk_req To serve a public key request, an `AuthSendPKH` message with the public key attached is sent to the requesting system.

Two default transitions are handled by this protocol, **send** and **receive**. Please note that if no public key is available, it is requested from the sender of the message. In this stage the protocol will block and no other messages may be in transition during that time. This approach is very basic and very insecure. A complete specification would, however, go beyond this paper's scope. The specification of the MDL messages `AuthReqPK`, `AuthSendPKH`, `AuthH`, and `CompH` is omitted.

6. Conclusions and Future Work

In this paper we presented the model-driven development framework SPICA. It is developed for building modular software architectures for autonomous robots. We proposed abstract modeling languages addressing three different domains. The Data Flow Description Language (DFDL) is capable of specifying modules and connections between modules. The Protocol Description Language (PDL) provides constructs for modeling domain-specific protocols. Finally, the Message Description Language (MDL) is a modeling language for message structures. The specifications are mapped to concrete implementations and translated to efficient and lean source code. For this purpose we introduced AASTra, a generic translation tool for model-driven development.

Established frameworks or middleware systems for mobile robots are mostly limited to one programming language or platform. The model-driven development paradigm followed by SPICA proved to be very powerful; communication interfaces, module skeletons, and messages structures can be generated automatically. Reoccurring code patterns can be integrated in a modular fashion. The efficiency of the architecture is improved by generating lean native implementations. We developed a modular software architecture for autonomous soccer robots using SPICA.

In order to improve and enhance SPICA, we will investigate the requirements of the DFDL and PDL with regard to new communication schemes. Besides, more flexible primitives for data management are planned. The PDL will be extended and generalized to support a wider range of protocols. The most important extension to PDL will be modeling support for cryptographic protocols. For this purpose, we have evaluated several languages for cryptographic protocol design such as e.g. CAPSL (Common Authentication Protocol Specification Language), MuCAPSL (Multicast CAPSL) [11], and LaCodA (Language for Source Code Generation and Analysis) [17]. We will investigate which of the concepts introduced by these languages can be incorporated in the PDL.

Graphical modeling tools for the languages and for the AASTra target template library are currently being developed, adding UML 2.0 support. We will conduct further research mainly in the model-driven development of robot software architectures. The main focus regarding communication will be on interoperability and security. Research will be conducted in modeling functional aspects of robot architectures as well. This includes fundamentals in data handling, sensor data fusion, and behavioral control.

References

- [1] ANTLR Parser Generator. <http://antlr.org/>.
- [2] P. A. Baer. Group authentication and encryption in distributed environments. In *I. Kryptotag – Workshop über Kryptographie*. Katholieke Universiteit Leuven, Technical Report ESAT-COSIC, 2004-CW-1, 2004.
- [3] K. Balasubramanian, A. S. Krishna, E. Turkey, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt. Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems. *International Journal of Embedded Systems, special issue on Design and Verification of Real-Time Embedded Software*, April 2005.
- [4] H. Bruyninckx. Open Robot Control Software: the OROCOS project. In *ICRA*, pages 2523–2528. IEEE, 2001.
- [5] H. Bruyninckx, P. Soetens, and B. Koninckx. The real-time motion control core of the Orocos project. In *ICRA*, pages 2766–2771. IEEE, 2003.
- [6] Carpe Noctem Robocup Team Website. <http://carpenoctem.das-lab.net/>.
- [7] C. Côté, Y. Brosseau, D. Létourneau, C. Raïevsky, and F. Michaud. Robotic Software Integration Using MARIE. *International Journal of Advanced Robotic Systems, Special Issue on Software Development and Integration in Robotics*, 3(1):055–060, March 2006.
- [8] A. S. Gokhale, D. C. Schmidt, T. Lu, B. Natarajan, and N. Wang. CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Applications. In *Middleware Workshops*, pages 300–306. PUC-Rio, 2003.
- [9] International Organization for Standardization. *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF*. International Organization for Standardization, Geneva, Switzerland, 1996.
- [10] Microsoft Robotics Studio Website. <http://msdn.microsoft.com/robotics/>.
- [11] J. Millen and G. Denker. CAPSL and mucapsl. *Journal of Telecommunications and Information Technology*, pages 16–27, 2002.
- [12] Mono Project .NET Framework Website. <http://www.mono-project.com/>.
- [13] I. A. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I.-H. Shu, and D. Apfelbaum. CLARATy: Challenges and Steps Toward Reusable Robotic Software. *International Journal of Advanced Robotic Systems, Special Issue on Software Development and Integration in Robotics*, 3(1):023–030, March 2006.
- [14] Object Management Group UML Website. <http://www.uml.org/>.
- [15] Official RoboCup Foundation Website. <http://www.robocup.org/>.
- [16] D. C. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar. A high-performance endsystem architecture for real-time CORBA. *IEEE Comm. Magazine*, 14(2), 1997.
- [17] N. Schmoigl. *Design und Implementation eines kryptographischen Compilers*. University of Mannheim, 2005.
- [18] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.
- [19] R. Sessions. *COM and DCOM: Microsoft’s vision for distributed objects*. John Wiley & Sons, Inc., New York, NY, USA, 1998.
- [20] StringTemplate Template Engine. <http://www.stringtemplate.org/>.
- [21] I. T. Union. Information Technology–Abstract Syntax Notation One (ASN.1): Specification of Basic Notation. *ITU-T Recommendation X.680*, July 2002.
- [22] H. Utz, S. Sablatnög, S. Enderle, and G. K. Kraetzschmar. Miro–Middleware for Mobile Robot Applications. *IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures*, 18(4):493–497, August 2002.
- [23] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARATy Architecture for Robotic Autonomy. In *Proceedings of the 2001 IEEE Aerospace Conference, Big Sky, Montana*, March 2001.

```

@inproceedings{BRZWG2007AR,
author      = {Philipp Andreas Baer and Roland Reichle and Michael Zapf and Thomas Weise and Kurt Geihs},
title       = {A Generative Approach to the Development of Autonomous Robot Software},
booktitle   = {Proceedings of 4th IEEE Workshop on Engineering of Autonomous and
                Autonomous Systems (EASe 2007)},
publisher   = {IEEE},
year        = {2007},
month       = mar,
type        = {Research Talk Paper},
affiliation = {University of Kassel},
location    = {Tucson, AZ U.S.A},
note        = {The work is online available at
                http://www.it-weise.de/documents/index.html\#BRZWG2007AR.\
                The publication can be downloaded at
                http://www.it-weise.de/documents/files/BRZWG2007AR.pdf.\
                Contact Thomas Weise at tweise@gmx.de or http://www.it-weise.de/},
abstract     = {The integration of new or existing software components into established
                architectures and the ability to deal with heterogeneity are key
                requirements for middleware and development frameworks for robotic
                systems. This paper presents SPICA, a software development framework
                for communication infrastructures of autonomous mobile robots.
                Utilizing the model-driven software development paradigm, communication
                and data flow can be defined on an abstract level. For this purpose,
                domain-specific languages and tools are provided that allow
                specification and generation of module communication infrastructures
                for communication between modules along with primitives for data
                management. The high-level platform-independent specifications are
                automatically transformed into low-level platform and programming
                language-specific source code. We illustrate the applicability of our
                approach with an elaborate example describing the design of a soccer
                robot architecture that has proven its strength during RoboCup 2006.
                Our experiences have revealed that SPICA is advantageous for
                prototyping as well as for building high performance systems.},
contents     = {* Introduction\
                * Related Work\
                * The SPICA Approach\
                * AAS Languages\
                * Example\
                * Conclusions and Future Work},
keywords     = {Software Engineering, Autonomous Robots, RoboCup, CarpeNoctem, C++,
                Java, C\#, Code Generation, Middleware, Communication},
language     = {en},
url          = {http://www.it-weise.de/documents/index.html\#BRZWG2007AR}
}

```