A Meta-Model and Toolchain for Improved Interoperability of Robotic Frameworks

Johannes Wienke, Arne Nordmann, and Sebastian Wrede

Research Institute for Cognition and Robotics, Bielefeld University, Germany

Abstract. The emerging availability of high-quality software repositories for robotics promises to speed up the construction process of robotic systems through systematic reuse of software components. However, to reuse components without modification, compatibility at the interface level needs to be created, which is particularly hard if components were implemented in different robotic frameworks. In this paper we propose an approach using model-based techniques for improving component reusability. We specifically address data type compatibility in a structured way through the development of a generic meta-model capable of representing data types from different frameworks and their relations. Based on this model a code generator emits serialization code which makes it possible to seamlessly reuse the existing data types of different frameworks. The application of this approach is exemplified by connecting the YARP-based iCub simulation with a component architecture using a current robotics middleware. Based on our experiences we describe requirements on robotics frameworks to further increase the level of interoperability between available components.

1 Introduction

In order to increase the usefulness of robots, they need to be equipped with a multitude of capabilities, which need to be reasonably combined into a working system. While many capabilities have already been implemented on different robots, their integration into a single system is still an open problem. A successful integration relies on an appropriate design of the functional architecture but often also more technical aspects slow down and complicate the development of integrated systems. As such, a major practical issue is the diversity of existing development frameworks like ROS [1], YARP [2], OROCOS [3] or OpenRTM-aist [4]. Even though most of the recent frameworks use a component-based approach and hence are composed of generally reusable building blocks, the created components can only be reused easily within the framework they have been developed for. This is caused by the lack of interoperation features in most frameworks. Recent development efforts tried to approach this problem, in particular by equipping frameworks with exchangeable transport layers, e.g. as done in OpenRTM-aist [5] with a ROS transport. Also YARP and OROCOS now have ROS transports. While this is a step towards interoperability, several issues still remain. Besides being able to use another framework's protocol, full interoperability on the transport level requires using foreign nameservices and introspection mechanisms. Otherwise, components imported from a foreign framework expose restrictions in their usability compared to native components and developers need to add specific exceptions for these components. On a more conceptual level, different transport semantics can prevent interoperability, e.g. if remote procedure call-based interfaces conflict with event-based communication. Another issue is the incompatibility of data types between different frameworks. Many frameworks developed type libraries with comparable semantics but using different approaches for Interface Definition Languages (IDLs), serialization schemes, and client APIs. This effectively prevents the reuse of components across frameworks. Even if transport-level communication is established, either (de-) serialization would fail or client components would be unable to deal with foreign data types. This commonly results in exceptions in the architecture like adapter and bridge components. Such solutions introduce inefficiencies and require manual work. Hence, a more structured approach to deal with data type incompatibility is required to fully make use the achieved transport-level interoperation support. Such an approach has to be easy maintainable and needs to be efficient to preserve the reactivity of the robot system.

In this paper we introduce an approach, termed Rosetta Stone, to deal with incompatible data types. It employs a generic meta-model for data representation and combines it with code generation. Section 2 starts with a discussion of how the interoperability issue has been addressed in other frameworks so far. Afterwards, we outline the conceptual ideas of our approach in Section 3. Based on a use case described in Section 4 our implementation will be explained in Section 5 and gained experiences and results of the application in the use case are detailed in Section 6. Finally, we conclude in Section 7 including a set of requirements which should be fulfilled by future frameworks to support data type compatibility.

2 Related Work

Interoperability between software components implemented in different robotics frameworks is usually achieved by applying classical software design patterns such as bridge, wrapper or protocol translator [6]. While it is reasonable to apply these techniques to specific interoperability problems, it imposes significant challenges for the software design of larger robotic systems if applied in the general case. For instance, using a dedicated bridge introduces performance penalties due to the additional reading, (de-) serialization and writing of data from and to connectors which are bound to the different frameworks. Furthermore, bridges are often specific to single data types and need to be kept up-to-date with the target types of both frameworks. As a consequence, bridges or wrappers are constantly out of date [5]. Instead, a more native level of interoperation between different frameworks is highly beneficial to achieve efficient solutions. In the following, we analyze different frameworks for features they provide towards native interoperability. As the representation of data is the key focus of this paper, we specifically examine these issues.

Recently, the Robot Operating System (ROS, [1]) has gained a lot of attention in the robotics community. It combines a middleware layer for communication with an extensive component collection, especially for mobile robots. Communication is statically typed and ROS comes with a collection of types based on a custom IDL. A compiler generates programming language types from these descriptions as well as framework-specific serialization code. While the transport layer in ROS can be exchanged and different implementations exist, the serialization code cannot be varied by the user, hence limiting interoperation with other frameworks.

A second widely used framework in robotics is OROCOS [3]. It has exchangeable transport implementations, e.g. CORBA, mqueue as well as a ROS transport. Additionally, different data representations can be used, called *typekits*, which are exchangeable through a plugin system. The traditional typekit is based on user-level C++ class definitions and a compiler builds serialization code for them by parsing the C++ code. Recently, support to interoperate with ROS has been added through the respective transport and a new typekit which has to be used by client components instead of the traditional one. This typekit is based on the ROS IDL to provide compatibility with ROS and the remaining OROCOS transport implementations cannot be used with these types. While transports and typekits are exchangeable, their choices are coupled and modifications to client components are required.

Another framework with exchangeable transports is OpenRTM-aist [5]. It provides features for hard real-time control and originally uses CORBA for data exchange between its components. Data types are defined through the CORBA IDL and in the case of the ROS transport, a duplicated definition of the respective type using the ROS IDL is required in order to provide the required serialization code to the ROS transport. The reuse of existing data types in both frameworks is not discussed so far and no mapping between them is explained in publications.

YARP [2] is a framework mainly used for the iCub robot. In contrast to the aforementioned frameworks, communication in YARP is dynamically typed, employing a custom data representation (*bottles*) and serialization approach. The framework has exchangeable transports and a partial ROS implementation is available. To combine the dynamic typing of YARP with the static ROS messages, the ROS transport comes with a specific compiler which generates YARP-serializable classes from the ROS IDL definitions, which effectively replaces the existing type system visible to the client components and hence requires modifications to them.

Summing up, while most recent frameworks provide ways to connect with other frameworks on a transport level, the way how data types are handled in these cases is not clearly structured. Often, special data types need to be chosen based on the transport decision because for most frameworks there is a strong connection between the transport, the applied serialization mechanism, and the exposed user-level data type API. This restricts the possible uses cases of the interoperation features and increases the development overhead when using such features. The issue of how to deal with semantically compatible but differently expressed data types, i.e. how to map between them, is completely neglected so far.

3 The Rosetta Stone Approach

In order to provide native data-level interoperability between different robotic frameworks we have developed a generic approach to mediate between the different technologies, which will be described on a conceptual level in the following paragraphs. The approach aims at a native integration inside the frameworks for high efficiency without the need for manual development of bridge components or a dependency on foreign frameworks. We assume that each of the potentially relevant robotic frameworks uses a structured and consistent approach of producing and consuming serializable data to be sent over a network connection. This means that a unified API for data holder classes and consistent serialization schemes for transmitted data exist, e.g. based on an IDL, so that at least inside each framework a level of generalization with respect to the data access is possible. Once this assumption is fulfilled, data communication in robotic frameworks fundamentally varies in four aspects: a) *the programming language used to produce or consume data*, b) *the API of the used data holders in this language*, c) *the (de-) serialization scheme applied to the data found in the data holders*, and d) *the transport mechanism used to communicate the serialized data*.

Our approach is based on a *meta-model* which describes data from the various robotic frameworks in an abstract and unified way, but including the aforementioned variables. Once the meta-model is populated with data types from different robotic frameworks it generates serialization code. The generated code uses the serialization scheme of one framework (A) on the network level and data types from another one (B). Assuming that framework B has exchangeable transports and serialization mechanisms, client applications written against this framework can connect with framework A by plugging in the generated serialization code and the transport for framework A. As a result, these applications can continue to use their native data types and no modifications to them are required. By using generated serialization code we can reach a high performance without requiring a dynamic use of the meta-model at runtime and without a dependency on the foreign framework's libraries (and hence increased compilation efforts) to import their serialization implementation. Moreover, no unnecessary serialization or conversion step is required as in the case of bridges. Finally, the mapping of different data types becomes a configuration aspect of the robotic system instead of being hard-coded.

3.1 Data Representation Features

A first question that arises for the proposed approach is whether it is possible to treat data from all recent frameworks in a common way, and if so, which features for data representation are required in the meta-model. For this purpose we analyzed the available representation features of common IDL-based serialization mechanisms (and hence statically typed) as well as the ones of 3 robotics-relevant dynamically typed systems[1]. The results of this analysis can be found in Table 1, indicating that besides some variations, a common and limited feature set available in most solutions can be identified, which is feasible to represent. On this basis we constructed a meta-model which represents the majority of the found features and is hence applicable for a broad variety of data types. In its essence, the meta-model represents data types as composed structures of named fields, comparable to the structure of most IDLs and programming languages.

[1] In our analysis we focused on features for the description of data types. Additional features found in several IDLs like the possibility to describe RPC interfaces are ignored.

	Signed 64bit Integers	Unsigned 64bit Integers	Double Type	ByteBlob Type	Null Type	Enums	Constants	Variable Length Arrays	Multidimensional Arrays	Fixed Length Arrays	Maps	Optional Fields	Default Values	Unions	Message Nesting	Data Type Inheritance	Namespaces	Typedefs	Any Type
Static Typing																			
Protocol Buffers	x	x	x	x		x		x				x	x		x		x		
ROSmsg	x	x	x	x			x	x		x							x		
LCM		x	x	x			x	x	x	x							x		
Message Pack IDL	x	x	x	x		x		x			x	x	x			x			
Apache Thrift		x	x	x		x	x	x			x	x	x				x		
Apache Avro IDL		x	x	x	x	x		x	x	x	x		x	x			x		
Apache Etch		x	x	x		x	x	x	x		x					x			
OMG IDL	x	x	x	x		x		x	x	x				x			x	x	x
Dynamic Typing																			
YARP			x	x		x		x		▨	x	▨	▨	x	▨	▨	▨	▨	▨
ALValue			x	x				x		▨		▨	▨	x	▨	▨	▨	▨	▨
JSON	x	x	x		x			x		▨	x	▨	▨	x	▨	▨	▨	▨	▨

Table 1: A matrix of features commonly found in different IDLs / dynamically typed data serialization solution. Features not applicable for dynamically typed solutions are marked in gray.

Currently supported data type features are: (signed/unsigned integers with different bit sizes, floats, strings, blobs, structs, arrays, unions.

3.2 Required Mapping Capabilities

Even though data types can be represented in a common meta-model, compatible types from different robotic systems still need to form separate entities in the meta-model, because for the code generation we need to be able to distinguish between them. Moreover, field names and representations usually do not exactly match between two semantically compatible types from different frameworks. E.g., a field might be called `angle` in one framework using radians but in the second framework it is called `phi` and based on degrees. For these reasons the meta-model also needs to contain a *mapping* between types which relates different fields and converts some representational differences. While the ability to map fields of different names is an essential requirement, the question arises which further capabilities are usually required. To find out an initial set of these capabilities that copes with the majority of cases, we analyzed the mapping of messages from the ROS `common_msgs` package to and from our own data types defined in the RST library [7]. For this purpose, we first decided which data types of one framework are semantically compatible with data types from the second, to define a set of realistic mapping candidates. For each candidate we then manually decided which operation categories are required to achieve a mapping. The results can be found in Table 2. Categories were chosen to reflect semantically related operations[2], namely: *Arithmetic*: basic mathematical operations (always selected when units were not clear from the type description, e.g. rad vs. degrees); *Array reorder*: shifting of entries of

[2] For brevity we excluded the obvious category of mapping varying field names.

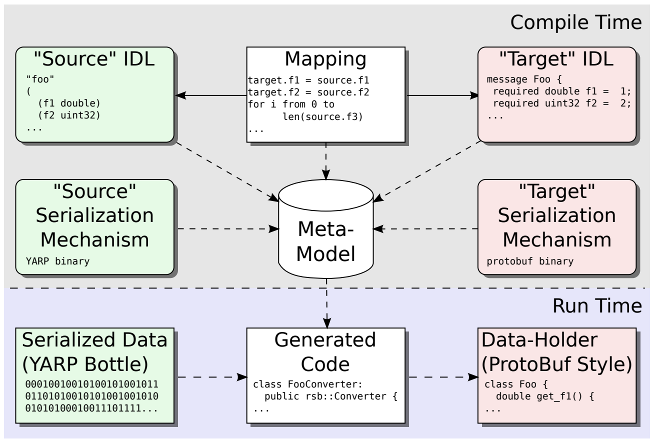
ROS	Arithmetic	Array Reorder	String Manipulation	Image Compression	Predicate Assignment	Subtype Loops	Rejection	Generators	Date Manipulation	RST
CompressedImage.msg				x	x			x?		Image.proto
Image.msg		x			x			x?		Image.proto
JointState.msg	x					x				JointPositionState.proto
JointState.msg	x									JointAngles.proto
JointState.msg	x									JointTorques.proto
JointState.msg	x					x				ProprioceptionState.proto
JointTrajectory.msg	x					x				Point2DTimeseries.proto
JointTrajectoryPoint.msg	x					x				Point2DTimestampPair.proto
KeyValue.msg			x				x			KeyValuePair.proto
Odometry.msg	x									Pose.proto
Path.msg	x					x		x		Point2DTimeseries.proto
Point.msg	x									Vec3DDouble.proto
Point.msg	x									Vec3DFloat.proto
Point.msg	x									Translation.proto
Point32.msg	x									Vec3DDouble.proto
Point32.msg	x									Vec3DFloat.proto
Point32.msg	x									Translation.proto
PointCloud.msg	x					x				PointCloud3DFloat.proto
Polygon.msg	x					x				PointCloud3DFloat.proto
Pose.msg	x									Pose.proto
Pose2D.msg	x					x				Pose.proto
Quaternion.msg	x									Rotation.proto
RegionOfInterest.msg	x									BoundingBox.proto
TimeReference.msg	x							x	x	Timestamp.proto
Transform.msg	x									Pose.proto
Vector3.msg	x									Vec3DDouble.proto
Vector3.msg	x									Vec3DFloat.proto
Vector3.msg	x									Translation.proto
Wrench.msg	x									Wrench.proto

Table 2: Operations likely to be required when converting between ROS and RST types.

any sequence-like container, e.g. to swap planes in image types; *String manipulation*: common string operations; *Image compression*: to decode compressed images; *Predicate assignment*: assignment to target type values based on logical predicates; *Subtype loops*: looping on and conversion of nested subtypes (composition); *Rejection*: abort translation in case of a dynamically detected incompatibility (e.g. unsupported image encoding); *Generators*: generation of values not found in the source type, e.g. constant value or loop variables; *Date manipulation*: conversion of date formats.

Starting with the total 65 message definitions of ROS and 65 message types in RST (by coincidence), 29 possible mapping candidates were found[3]. The results of this analysis are depicted in Table 2. For the majority of the mapping candidates arithmetic operations and the possibility to handle and convert collection-like contents (e.g. a list of floats to a list containing special `JointValue` objects) sufficed to achieve a mapping of all contents that can be represented in both messages. A notable but important exception is the mapping of image types. The definitions in both frameworks are rather

[3] In ROS messages types often exist once without a header containing timestamps and if necessary a second time including this header. We did not consider the timestamped messages for the mapping if the comparable version without timestamp was already a valid candidate.


Compile Time
"Source" IDL
Mapping
"Target" IDL
"Source" Serialization Mechanism
Meta-Model
"Target" Serialization Mechanism
Run Time
Serialized Data (YARP Bottle)
Generated Code
Data-Holder (ProtoBuf Style)

```
"foo"
(
  (f1 double)
  (f2 uint32)
...
```

```
target.f1 = source.f1
target.f2 = source.f2
for i from 0 to
      len(source.f3)
...
```

```
message Foo {
 required double f1 =  1;
 required uint32 f2 =  2;
...
```

```
YARP binary
```

```
protobuf binary
```

```
0001001001010010100101l
011010100101010010010l0
01010100010011101111...
```

```
class FooConverter:
  public rsb::Converter {
...
```

```
class Foo {
  double get_f1() {
...
```

Fig. 1: User-supplied contents of the meta-model used to create serialization code and the application of this serialization code.

complex involving different byte representations, image depths, channels etc. While the meta-data representing these different image types can be mapped with the aforementioned operations, the actual image data needs to be manipulated if no matching representation modes exist. This requires a more fine grained operation on the byte array contained in both message types.

We also analyzed a mapping from RST to data types used in YARP-based systems [2] and vice versa. Here the situation is different, as no formal message definitions exist. To find out commonly used message formats we analyzed the data continuously sent by the simulator for the iCub robot. For publisher-based data two semantically different types could be identified:

1. messages containing joint angle information as a list of doubles
2. messages for images with the following format (Lisp-like notation):

```
((VOCAB mat) (VOCAB rgb) ((INT 3) (INT 230400)
 (INT 8) (INT 320) (INT 240)) (BLOB 230400))
```

A mapping of the first message type requires arithmetic operations and potentially generators (e.g. to fill out the joint names which are not given in the bottle) and subtype loops to convert the angle list to more specific types or vice versa. For the image type the same remarks are valid as in the ROS case. Additionally, for controlling the robot, messages in the following format are sent over RPC-based channels:

```
((VOCAB set) (VOCAB poss)
 ((DOUBLE 0.0) (DOUBLE 0.0) (DOUBLE 0.0)))
```

These messages are comparable to the joint angle lists except that also YARP vocabulary items need to be generated.

Based on the aforementioned observations our meta-model contains mapping abilities for the most common operations. This is realized by defining each mapping through a Lisp-like code block with a restricted feature set that allows facilitates the static code generation.

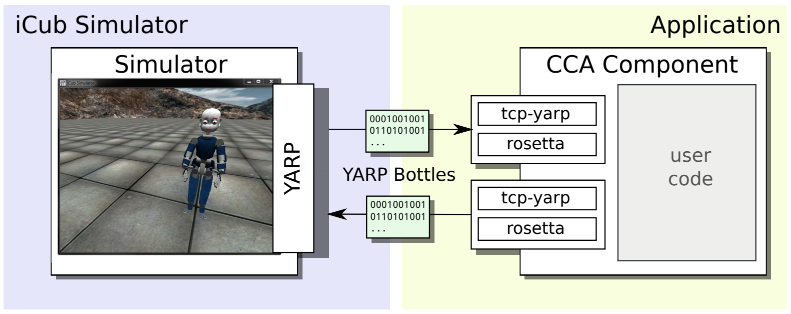


Fig. 2: Use case, connecting CCA components to the iCub simulator by using the Rosetta framework to interchange proprioceptive sensor feedback and an image stream from the iCub's internal cameras.

3.3 Application of the Meta-Model for Code Generation

For being able to generate serialization code from the meta-model two additional aspects must be contained in the model. These are the data holder APIs and the serialization schemes. The API needs to be known to provide the native interface to client applications while the serialization scheme is required to provide transport-level compatibility of data types through the correct serialization scheme. Figure 1 summarizes the required data in the meta-model (upper part, supplied by the user) and how the data can be used to generate deserialization code for binary data received from a foreign framework. Summing up, by using a generic meta-model for the unified representation of data types and their relation we are able to create code that efficiently translates serialized data from one robotic framework to the native data holder API in another framework. As a result, frameworks are connected without requiring manually written bridge components and the native interfaces in both middlewares are preserved. This prevents changes in components and increases their reusability.

In the following sections we are now going to introduce a use case where the connection of different frameworks is beneficial. Afterwards we will describe the actual implementation of the Rosetta Stone system and its application for the use case.

4 Use Case

As a an exemplary use case for the aforementioned approach, we decided to integrate the iCub simulator from the RobotCub project into our component architecture CCA (*Compliant Control Architecture* [8]), which is based on the RSB middleware [7]. The use case is driven by the AMARSi[4] project, where CCA was developed and the iCub is used as one of the main robot platforms. However, development of the robot was done in a predecessor project and diverging requirements resulted in the existence of two frameworks. To be able to conduct experiments with CCA components on the iCub simulator, the Rosetta approach allows CCA components to communicate over YARP, the middleware used by the iCub simulator. For this purpose joint angles and camera

[4] `http://amarsi-project.org`

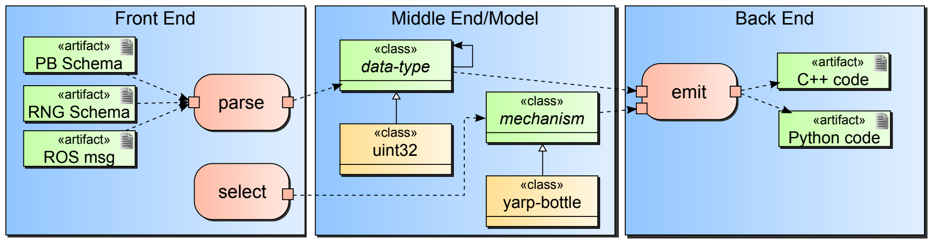


Fig. 3: Architecture overview for the Rosetta implementation.

image need to be transferred, which are two diverse but very common data types in robotics, hence reflecting a wide variety of possible applications.

In more details, the iCub simulator transmits images and joint angles as YARP bottles and CCA uses protocol buffer representation from RST for exchanging data between different processes. Therefore we need to translate between these two data representations. The anticipated result is that this is possible without changes in the implementation of the simulator or in CCA components, i.e. through configuration. Moreover, performance should not be degraded, which is specifically challenging for the image types with a significant payload size.

5 Rosetta Implementation

The Rosetta approach is realized as a compiler toolchain implemented in Common Lisp. As shown in the static view in Figure 3, the Rosetta implementation is structured as a traditional compiler with frontend, middleend and backend.

The frontend parses data type specifications expressed in existing IDL languages into meta-model elements resolving dependencies between type specifications. The current implementation supports parsing of protocol buffer, LCM and ROS IDL definitions. Similarly, mapping specifications are processed by this component. Additionally, syntactic as well as basic semantic checks (e.g., duplicate field names or unresolved forward references) are performed by the parsers.The middleend manipulates meta-model

Listing 1.1: Excerpt from the mapping specification for the iCub simulator joint angles.

```
import "rst/kinematics/JointAngles.proto"
import "bottle-structure-icub-torso-command.bottle-schema"

data-holder: rst.kinematics.JointAngles
wire-schema: yarp.icub.torso.command

unpack-rules:
  len(.angles) = 3
  .angles[0] = .angles.a0
  .angles[1] = .angles.a1
```

elements encompassing data types, mappings and serialization mechanisms. According to the desired code generation target (either data holders or serialization code) suitable intermediate representations are generated. The backend generates output based on the intermediate representation and language-specific templates for different programming languages. Currently supported are C++, Python and Common Lisp[5]. Generated code artifacts include data holder classes mimicking a native API (e.g., protocol buffer) and serialization code for different robotic frameworks (e.g., ROS or YARP).

To provide a dynamic view of the Rosetta implementation a typical sequence of processing steps in relation to the use-case introduced in Section 4 is as follows:

1. One or more IDL files are read and parsed by the frontend (in the use-case: files containing RST types `JointAngles` and `Image`, and the descriptions of the corresponding YARP bottle structures[6]) .
2. Optionally, a mapping description is read (in the use-case: a mapping for joint-angle types and a mapping for image types, see Listing 1.1).
   At this point, data types and mappings are represented in the meta-model.
3. For the given serialization mechanism and data types the transformation rules specified in the mapping are applied (in the use-case: respective mapping rules for joint-angle types and image types).
   At this point, abstract intermediate code implementing the mapping and (de-) serialization has been generated.
4. A template for the given target language is instantiated and populated (in the use-case: C++ protocol buffer API templates).
5. The template is expanded and the result written into output file(s) (in the use-case: C++ code in separate files for data-holder and serialization code).

The resulting serialization code needs to be used by the framework which attaches to a foreign one, ideally through configuration changes.

6 Gained Experiences

Our use case served as a first qualitative evaluation of how the Rosetta framework improves the compatibility between different robotic frameworks, without the need for changes or adaptations on the implementation level in one of the involved frameworks or applications. In this case we evaluated what and how much work had to be done in i) the iCub simulator, ii) Rosetta and iii) the application (CCA components).

As anticipated, the iCub simulator was left untouched for integration with the CCA application. We completely relied on the YARP bottle format and serialization for joint angles and images, as required by the simulator. For Rosetta, we had to specify the mapping between YARP bottles and the domain-types used in CCA. As YARP does not provide static types and a declarative syntax for them, we had to add such a schema description for YARP in Rosetta, in order to address fields from the YARP bottles in the mapping. Afterwards, mappings were defined between the YARP bottle format for

[5] For Common Lisp output of the middleend is directly passed to the integrated compiler.

[6] Cf. Section 6

images and joint angles, and the RST types used in CCA. Within the CCA application, the transport and the data converter for network communication had to be changed through RSB's configuration mechanism. The relevant ports of the involved CCA components had to be configured to publish over and listen to the `tcp-yarp` transport that is available as an RSB extension. Furthermore, the generated Rosetta serialization code (generated from IDL and mapping specifications) had to be registered for incoming and outgoing YARP image bottles and YARP joint angle bottles.

In summary, after specifying the Rosetta mappings between data representations and generating the serialization code, no changes were required for the simulator and necessary changes on CCA-side were limited to configuration aspects.

In addition to this qualitative evaluation we also analyzed the performance of the generated serialization code within the use case. For this purpose we compared the deserialization of a binary encoded YARP bottle containing double-precision joint angles using Rosetta and with the native YARP implementation. Rosetta deserialized to C++ RST types while YARP used the native bottle classes in the C++ API. The encoded message had a size of 200 bytes and was deserialized 1.000.000 times. On a Linux desktop computer with an Intel Xeon 8 core processor at 2.4 GHz the YARP implementation required 3.96 s *real time* whereas our own deserialization code needed 0.13 s. Test programs were compiled using GCC and the O2 optimization level. The huge performance boost can can be explained by the fact that Rosetta has an additional schema of the transmitted data (see above), while the native YARP implementation is completely dynamic. As a consequence, the Rosetta-generated deserialization code can skip many checks and decoding of several bytes that need to be deserialized by the native implementation to dynamically find out the structure of the bottle. This performance allows to receive simulation results in the use case without any performance degradation.

7 Conclusion

Nowadays, the middleware layers of most modern robotic frameworks allow a direct embedding of multiple transports acting as connectors to other frameworks. Furthermore, many of the current frameworks already employ a code generation approach where the necessary client API classes and serialization code are generated. However, still most of the frameworks lack a clear separation of concerns regarding the type representation at user-level and the resulting serialization format which is beneficial to achieve interoperability without component modification. Moreover, frameworks do not consider type mapping and transformation as an important concern.

The Rosetta approach proposed in this contribution addresses these aspects in order to achieve native interoperability between components of different robotics architectures. Based on an analysis of features commonly found in different IDLs and necessary mapping operations between a set of typical robotic data types expressed in different representations, required capabilities for a type mapping language were identified in this contribution. On this basis a meta-model for representing types and mapping functions was developed, which allows to generate several code-level artifacts for native robotic system interoperabiliy. The introduced approach is so far unique, because none of the frameworks addressed native type mapping and transformation routines in the process of generating serialization code for seamless interoperability.

Scientifically, a common representation for data types and their mappings will facilitate, e.g., analysis and comparison of different data types used in current robotic systems. Practically, the developed approach not only allows better interoperability and thus reusability of software components across robotic frameworks but also contributes to achieve integration within a single large-scale ecosystem such as ROS given the number of semantically equivalent but syntactically different robotics data types. Future work will concentrate on the development of a domain specific language which eases the specification of mappings and transformations between data types in robotic systems. Moreover, additional ways of applying the compiler architecture will be evaluated. E.g., in cases where frameworks lack interchangeable transports and serialization mechanisms Rosetta can generate serialization-to-serialization code for efficient and configuration-defined bridge components.

Acknowledgments

This work partially was funded by the European FP7 projects HUMAVIPS (grant no. 247525) and AMARSi (grant no. 248311). Many thanks to Jan Moringen for his work on the Rosetta toolchain.

References

1. Morgan Quigley et al. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.
2. Giorgio Metta and Paul Fitzpatrick. YARP: yet another robot platform. *Journal on Advanced Robotics*, 3(1):43–48, 2006.
3. Peter Soetens. *A Software Framework for Real-Time and Distributed Robot and Machine Control*. PhD thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, 2006. `http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf`.
4. Noriaki Ando, Takashi Suehiro, and Tetsuo Kotoku. A Software Platform for Component Based RT-System Development: OpenRTM-Aist. In *Simulation, Modeling, and Programming for Autonomous Robots*, Berlin, Heidelberg, 2008. Springer.
5. Geoffrey Biggs, Noriaki Ando, and Tetsuo Kotoku. Native Robot Software Framework Inter-operation. In *Simulation, Modeling, and Programming for Autonomous Robots*, Darmstadt, Germany, 2010. Springer.
6. Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. Wiley, Chichester, UK, 2007.
7. Johannes Wienke and Sebastian Wrede. A middleware for collaborative research in experimental robotics. In *IEEE/SICE International Symposium on System Integration (SII2011)*, Kyoto, Japan, 2011. IEEE, IEEE.
8. Arne Nordmann, Matthias Rolf, and Sebastian Wrede. Software Abstractions for Simulation and Control of a Continuum Robot. In *SIMPAR*, Tsukuba, Japan, 2012. accepted.