Progress in RoboComp

Marco A. Gutiérrez, A. Romero-Garcés, P. Bustos, J. Martínez

***Abstract*—During the last six years the RoboComp robotics framework has been steadily growing in the number of software components, the variety of robots supported and in new solutions to the maintenance of large robotics software repositories. In this paper we present recent advances in the formal definition of the RoboComp component model and a new set of tools based on Domain Specific Languages that have been created to simplify the whole development cycle of the components. Moreover, a new robot simulation tool has been created providing perfect integration with RoboComp and better control over experiments than current existing simulators. Finally, the paper describes a working solution to the important problem of communications middleware independence, which allows users to decide which middleware the components will be compiled with. Our solution has been validated by the integration of *Nerve*, a novel middleware for critical robotics tasks, in RoboComp.**

***Index Terms*—Robotics Framework, Software Engineering, middleware, performance, robotics**

I. Introduction

ROBOTICS software has to deal with very specic problems such as complexity, code reuse, scalability, robustness, distribution, language and platform support, or hardware independence. These problems should be addressed with appropriate software engineering techniques and should be transparent to the developer when possible. Software complexity, from the developer point of view, is an important topic because scalability decreases as complexity increases. The robotics community is already aware of this fact and has steered towards component oriented programming (CBSD) [1]. Many different approaches have raised in order to solve the robotics specific issues. RoboComp [2] is a robotics framework focused on ease of use and rapid development that is continuously evolving to cope with the demanding requirements of current robots. According to our experience, the most challenging issues that these frameworks must face now are reusability, scalability, robustness and adaptability. Although these issues are common to other application domains, Robotics incorporates additional difficulties such as real/critical time, hardware interaction, physical security conditions and human-robot interaction just to name a few ones. In this paper we present some advances in the scalability and adaptability issues. Scalability is the problem of how to manage an increasing number of static components -in repositories- and running components -in deployed networks with dozens of processes. Adaptability is the problem of maintaining alive the framework technology during many years, given the huge number of dependencies from third party providers that unavoidably end up forming part of the software.

Marco A. Gutiérrez and P. Bustos are with RoboLab at the University of Extremadura
E-mail: marcog@unex.es
A. Romero-Garcés and J. Martínez are with the University of Málaga

We have approached these problems using Domain Specific Languages (DSL) based tools and a flexible component model that can be adapted to new and unforeseen requirements. Automatically generated code, online syntax checking, automated components deployment and free generated parsers are some of the advantages that can be obtained from the use of these tools. Several DSLs have been designed to facilitate the work of developers. These languages provide a textual or graphical high-level definition of the internal structure of the components, interfaces, configuration parameters, deployment configurations and of the kinematics of the robots and the scene. The specific design of each of these languages and their role in the scalability and adaptability issues will be furtherly explained in section IV

The availability of a DSL-based definition of the elements just mentioned opened new unexpected possibilities to tackle the adaptability problem. Up to now, RoboComp has based its communications on a third party industrial-quality middleware, Ice, from ZeroC [3]. Although this choice has never been a problem so far, it is true that sticking to an specific middleware entails some risks, for example staying out of some recent technological advances. An interesting solution to this problem is to have middleware independence in the framework, meaning that the user can select which communications middleware will be used in the component. This feature adds even more flexibility to our framework in terms of components interconnection and developer abstraction, facilitating the survival of the repository across technology changes. Another strategic decision that we have taken is to create a new 3D robotics simulator. There are several reasons for starting this gigantic endeavour that are discussed later but, essentially, we needed total control over the simulator and its interfaces because we wanted to integrate it as a native component in the RoboComp architecture. As it will be sketched below, a simulator component can be used in two ways: externally, as a regular provider of synthetic reality; and internally, as an inner-model of the perceived reality. It is this last use and its role in the new cognitive architecture we are constructing what has driven us to build the RCIS simulator.

The rest of the paper is organized as follows: Section II will briefly review the most related frameworks. Section III gives a quick overview of the general features of RoboComp. Section IV explains the new DSL technology and its implementation in the life-cycle of component development. In Section V how middleware independence in RoboComp is achieved and a case of study are explained. Section VI describes the intrinsics of the new 3D simulator tool. RoboComp packaging is explained through section VII. Finally VIII will provide some conclusions along with some future lines of work.

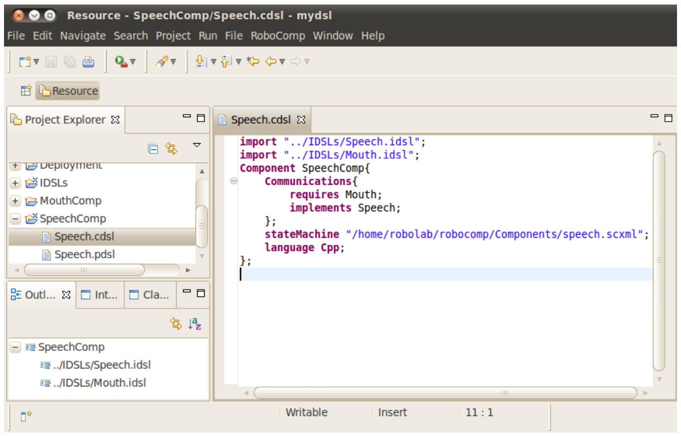
II. Related work

Several frameworks for robotics development are currently widely used. We will briefly review those most related to our line of work or that share some of our concerns on scalability and adaptability.

Robotics Operating System (ROS) [4] is the most famous and most widely used framework. It contains a wide set of reusable components most of them including state of the art solutions to main problems in robotics. Many commercial and research hardware platforms can directly run on it and use the available components making it really easy to run state of the art algorithms with very small time investment. On the other hand, although it is widely used and it has hundreds of components, this very same success limits the potential of adaptation to the challenges mentioned above. Features like automatic code generation, modifications in the component model, middleware independence or other technological updates yet to come are difficult to incorporate because they might imply critical changes in the current kernel technology. The huge inertia caused by high number of users limit the rate at which new concepts can be introduced. Moreover, many of the most interesting libraries that are being developed inside ROS are also available as standalone packages that are easily integrable by other frameworks.

SmartSoft [5] is one of the most advanced middlewares in terms of the Software Engineering technology that it incorporates. The creators of SmarSoft designed a set of communication patterns that are at the core of its component model and that were devised as a key tool for middleware independence. SmartSoft supports dynamic reconnection of components at run-time making use of the so called *wiring Patterns*. This feature allows the re-wiring of components according to the perceived context and changing requirements. This framework also includes advanced tools for Model Driven Software development as well as task level abstraction. Currently provides two reference implementations, one using CORBA [6] and another one using the Adaptive Communications Environment (ACE) [7] as the underlying communication infrastructure.

OpenRTM-aist [8] is a Japanese robotics framework designed also using a Model Driven Architecture (MDA [9]) technology. The communication middleware used by Open-RTM is omniORB, which is an open source CORBA implementation. It also supports model driven developments for the design of components and several tools are provided, such as the RtcTemplate that is the main development tool used to generate the RT-Components and RTCBuilder that is a template generator tool for new components. The RTSystemEditor is based in the Object Managment Groups (OMG) Robot Technology Component (RTC) [10] specification and is used for manipulating the components in a real-time graphical interface. Some other tools worth mentioning are the RtShell which includes the commands to interact with the components and the managers and TtcLink a tool for operating with the components in real time through a GUI. Recently they have been also working on some interconnection issues and soon the components from Open-RTM will be able to talk to ROS.



```
import "../IDSLs/Speech.idsl";
import "../IDSLs/Mouth.idsl";
Component SpeechComp{
	Communications{
		requires Mouth;
		implements Speech;
	};
	stateMachine "/home/robolab/robocomp/Components/speech.scxml";
	language Cpp;
};
```

Fig. 1: Screenshot of the RoboComp DSL Editor while modifying the CDSL model.

III. RoboComp overview

RoboComp is a model-driven, component-oriented framework built around three key elements: a component model, a communications middleware and a set of tools that facilitates the writing and maintaining of robotics code. It started in 2005 as a way to create and reuse code written by many different people and that was meant to be used in many different robots. The central idea is to define a processing and coding entity that can be created and maintained largely decoupled from the rest of the system. These units or components are full fledged processes when running and occupy its own subdirectory in the global code repository. They communicate with other components using a public interface and through an underlying communications middleware. Building on this generic idea, RoboComp is now the result of many years of further elaboration and adaptation to our everyday research and engineering activity and, nevertheless, many more improvements are in the way now due to the increasing complexity of current robots and their control and cognitive architectures. The repository holds now more than one hundred components, along with classes and tools specifically designed to improve and ease the robotics software designer experience. It covers functionalities of different robotics and artificial vision topics mainly through integration of third party libraries. More detail of the initial design of robocomp is given in [2].

IV. Domain specific languages for component oriented programming

In order to improve the scalability and adaptability of RoboComp, a major redesign has been undertaken. The complete software life-cycle of the components has been transformed by introducing DSL technology under a Model-Driven approach. In this new design every critical part of the framework is specified using a Domain Specific Language and the corresponding transformation tools generate the source code according to the detailed specification expressed in them. The user now only needs to add the specific working code in his component. Figure 1 shows the new DSL Editor. The next five Subsections describe each of the DSLs that are now part of RoboComp development model.

A. CDSL

RoboComp provides both client/server and publish/subscribe communication models. In order to establish communication, components must perform different operations. For instance, if a component needs to perform remote calls to other components, its code will have to: **a)** include the definition of the proxy classes corresponding to the interfaces it is going to connect to; **b)** read from the configuration file how to reach the remote component; **c)** create the proxy object using the previously read configuration; **d)** provide the proxy object to the classes that will be using it. Similar scenarios exist when providing new interfaces, subscribing or publishing new topics (see [2] for more details). In RoboComp, this code was automatically generated by Python scripts when the component was created for the first time. However, until the adoption of MDA-based techniques, if the connectivity of a component changed after its creation -a considerably common scenario- the code had to be manually modified. Also, successive changes to the component model required a cumbersome and error prone work on the Python scripts.

Components became difficult to maintain through all these situations mainly because their source code depends on these parameters. In order to solve these problems, we have developed a DSL to create and modify component properties. This DSL is called the Component Description Specific Language (CDSL) and allows users to create and maintain their component descriptions from a textual model. CDSL files contain the basic definition of the structure of the component and information about communication parameters such as proxies, the programming language of the component, interfaces and topics used by the components, the optional support of Qt graphical interfaces, their dependencies with external classes and libraries, and an optional SCXML [11] file path for embedding a state machine in the component.

The properties that are contemplated in CDSL are the following:

- Component name.
- Interfaces and data types defined in external IDSL files.
- Client/Server communication model: required and provided interfaces.
- Publish/Subscribe communication model: topics that the component will publish or subscribe to.
- Graphical interface support.
- State machine support.
- Dependences with external classes and libraries.
- Programming language of the component.

In order to match the CDSL development model, the structure of RoboComp components has been remodelled. The internal structure of the components has been divided in two parts: a generic and a specific part as shown in figure 2. The generic part contains the logic of interprocess communication, the general structure of the components such as the main program, thread pool, source directory structure, documentation rules and some introspection and self-monitoring capabilities. This generic functionality is implemented with abstract classes that are inherited and extended by the user-specific code. Thus, a component can be divided in two parts by a line separating the generic from the specific. The specific component code is generated by the RoboComp DSL tool only once, but the generic code is always regenerated when the CDSL is modified. Proceeding this way, the users are guaranteed that their specific code will never be deleted or modified and that, at the same time, they are able to modify any generic component property. This basic structure is one of the most important design decisions taken at this stage of RoboComp development.

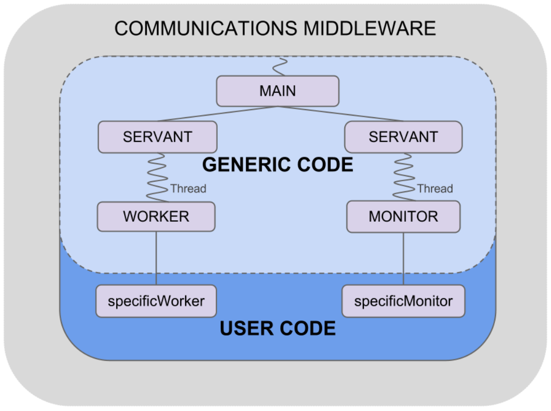


Fig. 2: New structure of RoboComp components.

B. IDSL

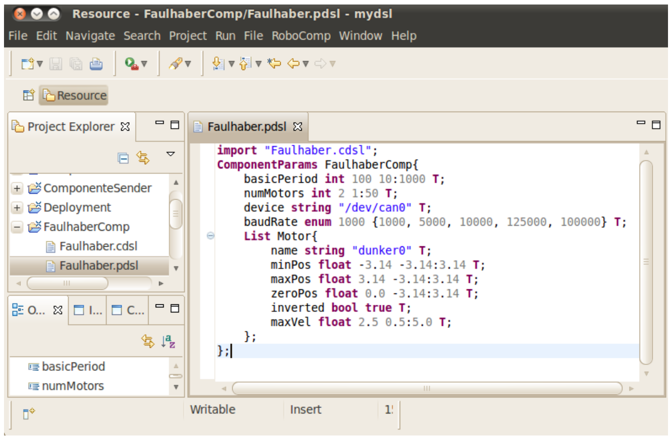
RoboComp initially used the Ice Interface Definition Language (Slice by ZeroC [3]) to define component interfaces. The problem with third party IDLs is that they create an unnecessary dependency that might be difficult to eliminate, since the data types and proxies of the middleware tend to mix with the user code. Moreover, any change in the Slice language would require modifying the RoboComp interfaces already defined. To avoid this dependency we developed the Interface Description Specific Language (IDSL). IDSL was initially designed as a subset of the Slice features, mainly data types and definition of remote calls. Now users can define their own interfaces using the IDSL language which is fully supported by the RoboComp DSL Editor. These definitions can be easily and safely translated to a target IDL when needed.

Currently, IDSL supports the following features:

- Interfaces and topics definition.
- Basic data types, such as integer and real numbers or strings.
- Enumerated types.
- Custom structures and data types such as sequences and maps.
- Exceptions.

C. PDSL

The Parameter Definition Specific Language provides a formal tool for the specification of configuration parameters that define the runtime behavior of the components. We have noted that creating configuration parameters and reading them into the code are one of the main causes of execution



```
import "Faulhaber.cdsl";
ComponentParams FaulhaberComp{
    basicPeriod int 100 10:1000 T;
    numMotors int 2 1:50 T;
    device string "/dev/can0" T;
    baudRate enum 1000 {1000, 5000, 10000, 125000, 100000} T;
    List Motor{
        name string "dunker0" T;
        minPos float -3.14 -3.14:3.14 T;
        maxPos float 3.14 -3.14:3.14 T;
        zeroPos float 0.0 -3.14:3.14 T;
        inverted bool true T;
        maxVel float 2.5 0.5:5.0 T;
    };
};
```

Fig. 3: Screenshot of the RoboComp DSL Editor while editing a PDSL file.

errors when creating components. With the PDSL, the user defines a template of configuration parameters that can have a hierarchical structure -i.e., nested lists-, explicit types, initial values and valid ranges. This template is used by the DSL tool to generate specific parsing source code structures with access control methods that are included in the generic classes of the target component. Thus, PDSL guides developers in writing the necessary configuration files in a standardized way within the framework. Besides the increased expressive power of the new method to define complex set of parameters, the most remarkable improvement is the reduction in the number of programming and execution errors due to misreading or misinterpreting the values assigned to the parameters before deploying the component. Figure 3 shows a capture of RoboComp DSL editor while editing a PDSL file that defines the parameters -nested- of a bus of motors.

D. DDSL

Components are independently executed programs that interact with each other. When using a component-oriented robotics framework, a robotic software system is composed of several interconnected components, representing a component network. These components can be executed manually, but as the number of components grows, it becomes increasingly difficult to manage them appropriately. Robots of middle complexity -e.g., mobile robots equipped with stereo heads- and high complexity robots -e.g., mobile manipulators with expressive heads- are controlled by graphs containing dozens of components running on several computers. The configuration and management of these networks of processes suggest the combination of a graphical tool and a representation language. The Deployment Description Specific Language was developed as the underlying language to make this management task easier.

With DDSL users are able to describe which components will be used, where they should be executed and which configuration to use. This makes it possible to automatically deploy RoboComp components in a certain computer or computer network. DDSL has been designed to simplify the system deployment and integration.

In order to define a component network, the following parameters have to be specified for each component:

- Component: the CDSL file path of the component to execute.
- Path to the executable file of the component.
- IP address and port.
- Path to the configuration file.

With the information in this file, all component dependences can be precomputed. Thus, the DSL editor can warn users of basic configuration errors while editing the file and prior to the actual deployment.

E. InnerModelDSL

InnerModelDSL is an XML-based Domain Specific language specifically designed for describing the kinematics of robots. InnerModelDSL files are parsed by a C++ class called *InnerModel* which is widely used by components to maintain an updated representation of the robot and to compute transformation between frames of reference. InnerModelDSL has been considerably extended in the last years. Despite it was initially designed only to define kinematic descriptions, several elements were incrementally added to increment its expressive power. Now, sensors can be specified and 3D meshes can be assigned to kinematic links providing the volumes necessary to compute collisions. Also, the scene around the robot can be also defined using volumetric primitives. As a side effect, this updated description of the scene can be used with a 3D scene graph engine -OpenSceneGraph- to visualize the robot in its environment at any time and in any component that includes the corresponding libraries, which is automatically done from the PDSL.

After parsing an InnerModelDSL file, a transformation tree is created in memory that holds all the nodes defined in the kinematic description. This tree is managed inside a class that offers an interface to manipulate and query its state it at runtime. Listing 1 shows an implementation of a tree using the innerModelDSL. The attributes (t_x, t_y, t_z) correspond to the translation vector (x, y, z) and the attributes (rx, ry, rz) correspond to the rotation angles (α, β, γ). If any of the attributes is missing InnerModel assumes they are 0. Cameras and other sensors can be included in the model using specific tags.

```
<innerModel>
  <transform id="world">
    <transform id="target" />
      <transform id="robot">
        <translation id="head" y="1002" z="-120">
          <translation id="cameraPose" x="0" y="1002">
            <camera id="camera" width="640" height="480">
          </translation>
        </translation>
        <translation id="shoulderPose" y="597" z="-23">
          <rotation id="shoulder">
            <translation id="elbowPose" z="300">
              <rotation id="elbow">
```

```
                                    <translation id="
                                        hand" z="120"
                                        />
                                </rotation>
                            </translation>
                        </rotation>
                    </translation>
                </transform>
            </transform>
        </transform>
</innerModel>
```

Listing 1: Example of a tree defined using the innerModel DSL

F. *Experiment: DSL benefits on common developers tasks*

An experiment has been conducted in order to provide empirical evidences supporting the benefits of the usage of Domain Specific Languages and the related tools that have been developed. Five of the most common repetitive tasks that developers usally perform when developing and managing robotics software have been selected-. Particularly developers wheres asked to make the following operations over existing components:

- Deploy a small component network
- Include a new library and a class in a project
- A new method on a previously developed interface
- Make a component provide a new interface
- Add a new proxy for a given interface.

Experiments have been performed twice, first using the developed DSL tools and then, without any of these tools. Two main metrics have been considered, the number of lines of code written and the time spent doing the specific task. We have tested it with twelve robotics developers with a previous basic knowledge of the RoboComp Framework and the related DSLS. Measurement data is represented as boxplots containing all measurements ranging between the first and third quartiles. The location of the median of the measurements is indicated by a red line crossing the rectangle vertically. Measurements outside the box are considered outliers and are drawn using green diamonds.

Figure 4 shows the results regarding the time spent in the experiments. It is worth mentioning that the only time taken into account was the one in which the subject was typing code, not thinking. Since users need less time to think when using DSLs (i.e., there is no need to think which code pieces should they change) this plays against the use of DSLs. In spite of this, it can be seen how using the DSL approach shorter times were achieved for all of the experiments performed. Figure 5 shows the results regarding the lines of code written while performing the experiments. As happened with time, the figure shows that, using the DSL approach, fewer lines of code were written for all of the experiments performed. This is not a surprise, since small changes in the DSLs might involve many changes in the generated code. The only experiment in which no considerable improvements were achieved (only a few seconds) was the experiment number 3. This is because the framework was already making use of CMake features for the operations involved in the experiment, so the initial number of lines to modify was already low.

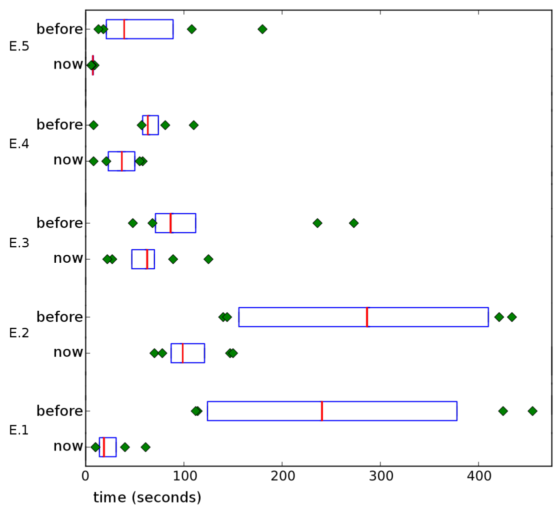


Fig. 4: Boxplots of the time spent writting for each experiment.

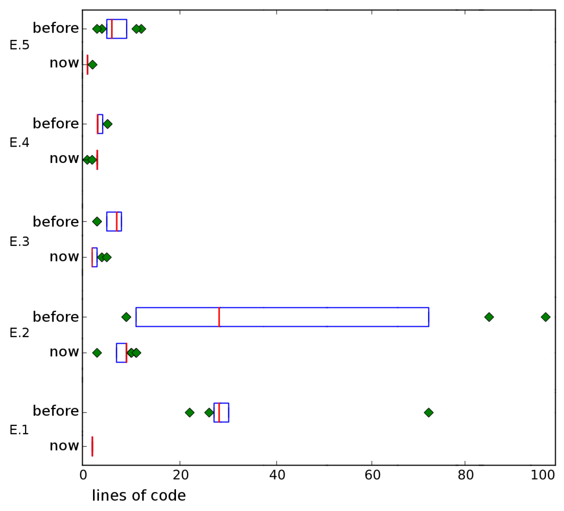


Fig. 5: Boxplots of the lines of code written for each experiment.

V. Middleware independence

One of the main objectives in the robotics field has been the creation of libraries and object-oriented frameworks for building robotics software, focusing on reusability and software evolution. In this application domain, the complex tasks performed by a robot are divided into distributed processes which communicate using existing middleware (CORBA, Ice, or DDS [12]) or ad-hoc mechanisms (such as ROS or Player [13]) for communications. These mechanisms allow developers to create networked applications in a platform and language-independent manner which also hide low-level communication details. However, if the selection of a specific middleware (or an ad-hoc implementation) has influenced the whole architecture of the distributed system (in this case the RoboComp framework), it could compromise the capabilities of the whole robotic system to adapt it to new requirements, such as real-time or quality-of-service (QoS) properties. From its inception, RoboComp components were strongly coupled with Ice, using not only its communication model (based on remote method invocations or RMI [14]) but also part of its proprietary API in the main structure of each component. Nevertheless, Ice is not suitable for real-time scenarios and different communication models and, therefore, we have designed a mechanism to make RoboComp a middleware-independent robotics framework. With this new redesign it is now possible for developers to select the appropriate communication model for exchange data among components without any binding to a specific middleware until code generation. It is worth noting that existing Ice-based components in Robocomp can be now transformed into new middleware-independent ones without changes in the user-defined business logic of the component. In order to obtain a middleware-independent framework, RoboComp's DSLs were improved with new semantics when defining interfaces, which now add three communication patterns to describe their operations:

- *command*. One to one communication. Operations without output/return values.
- *query*. One to one communication. Operations with output/return values.
- *publish*. One to many communication. Operations without output/return values.

These patterns are heavily influenced by the ones described in [15] and represent the two communication models more used in networked applications: the *client/server* (*command* and *query*) and the *publish/subscribe* (*publish*) model. *command* and *query* patterns encapsulates the Remote Method Invocation paradigm in order to perform one-way or two-way remote calls and *publish* wraps a mechanism to publish data to multiple subscribers. Two RoboComp DSLs are used to select the communication model and the operation patterns. The CDSL helps developers to select the client/server or the publish/subscribe communication model for a specific interface (selecting *request*, *implements*, *publishes* or *subscribesTo*). The IDSL has been changed in order to provide a middleware-independent interface using the communication patterns. Listing 2 shows the definition of the Speech interface using the communication patterns. When a *command* or a *publish* operation is defined, only an input parameter (a data request structure) is needed. If a *query* operation is described, the operation must have an input request parameter and an output response parameter. Data request parameters will contain all the information that will be sent or published to other components and the data response will contain the result of a *query* operation. It is worth noting that the code generated from an IDSL will depend on the middleware that will be used in the component.

```
module RoboCompSpeech{
    struct datasayin{
      string text;
      bool override;
    };
    struct dataresponse{
      bool resp;
    };
    struct dataenableSynchronizationin{
      bool value;
    };
    interface Speech{
      [query]void say(datasayin pin,out
          dataresponse pout);
      [query]void isBusy(out dataresponse p);
      [command]void enableSynchronization(
          dataenableSynchronizationin p);
    };
};
```

Listing 2: Speech.idsl

Once the component and the interface have been described, the user will select a middleware during the source code generation step for each component. Robocomp components share all the same code structure with a generic and a specific part, independently of the middleware selected. If the component implements an interface, the user only needs to implement the operations defined in the IDSL file in the SpecificWorker class of the component. On the other hand, if the component requires an interface, then the communication patterns must be used. Listing 3 depicts the code of a component that uses the communication patterns to perform the operations defined in the Speech interface.

```
1  void SpecificWorker::compute(){
2      ...
3      RoboCompData<dataresponse> resp_busy;
4      speechcomm->query<isBusy>(resp_busy);
5      if(resp_busy->resp!=true){
6        RoboCompData<datasayin> req_say;
7        req_say->text = "Hello";
8        req_say->override = true;
9        RoboCompData<dataresponse> resp_say;
10       speechcomm->query<say>(req_say, resp_say);
11        ...
12     }
13     ...
14 }
```

Listing 3: Extract of code using the *query* communication pattern

The implementation of the communication patterns is also generated from the CDSLs automatically depending on the middleware selected. For each proxy or publisher, a high-level communication pattern class is generated. Internally, these classes wrap up the proxies and publishers of the underlying middleware. An instance method of the specific communication pattern will be used to invoke an operation, where its arguments (request/response data structure variables) are used to identify the exact operation inside the communication pattern object (for instance, lines 4 and 10 in listing 3).

A. A case of study: integrating Nerve in RoboComp

1) Nerve overview: As a case of study, we have extended RoboComp to support *Nerve*, a lightweight C++ middleware for networked robotics [16]. *Nerve* has been designed to guarantee the scalability and quality-of-service(QoS) of real-time and critical components. It uses several frameworks provided by ACE to implement communication policies between local components which execute as threads, using message queues and zero-copy buffering. *Nerve* also makes use of the OMG's Data Distribution Service for real-time systems(DDS) standard, which is a completely decoupled publish/subscribe communication model with a wide set of QoS for distributed components. Although *Nerve* relies on the OpenSpliceDDS open source implementation of DDS [17], it is worth noting that different DDS products could be integrated in a seamless way.

Data in *Nerve* are defined as topics, which is a concept inherited from DDS. Topics are data structures (types) identified by a name and an associated quality of service. These QoS policies may define reliability, missed deadline or resource limits among others, and must be applied carefully to obtain the best performance in a specific deployment. As in other OMG specifications, the definition of topic types is done using an IDL (in this case, a subset of the CORBA IDL), which can be compiled to primitives and data structures for specific programming languages.

Nerve has been designed to choose the fastest communication mechanism between components executing parallel tasks, which depends on the way these tasks are deployed on the final robotics system. For instance, inter-thread communication modes are faster than inter-process communication ones (we obtain better data transfer rates) whether components are executed within the same physical node. In short, the main features of *Nerve* include:

- The encapsulation of critical tasks as platform-independent services that can be executed as threads or processes at deployment time.
- The adoption of an asynchronous execution model, in which services react to events available from their own event queue.
- The internal selection of the fastest mechanism for communications: zero-copy buffering for threads, shared memory for processes running within a node and reliable multicast for networked processes. Users will be unaware of the selected mechanism.
- The adoption of a standard protocol in the distributed case by adopting the OMG's Data Distributed Service recommendation.

Although *Nerve* is mainly based on the publish/subscribe communication model for distributed communications, recently, we have also included a mechanism that emulate Remote Method Invocations, it being now possible to fulfill the communication requirements in Robocomp usig its communication patterns.

2) Extending the automatic code generation process for Nerve in RoboComp: In order to use *Nerve* in RoboComp, we have extended the automatic code generation process for CDSL, IDSL and PDSL: a) *Nerve* IDL files can be generated with the appropriate topics types that are obtained from the parameters of the operations defined in the IDSLs, b) from CDSLs is also possible to generate the component code with all the files that wrap the communication details of Nerve, hiding them from developers, and c) PDSL language can be also used to generate the appropriate code for parameters and their initialization values.

3) Generating Nerve-based components: Figure 6 shows the generation process of a RoboComp component using the RoboComp DSL Editor tool. Shaded boxes represent files that are middleware independent. To clarify the code generation process, both middleware (Ice and *Nerve*) integrated in

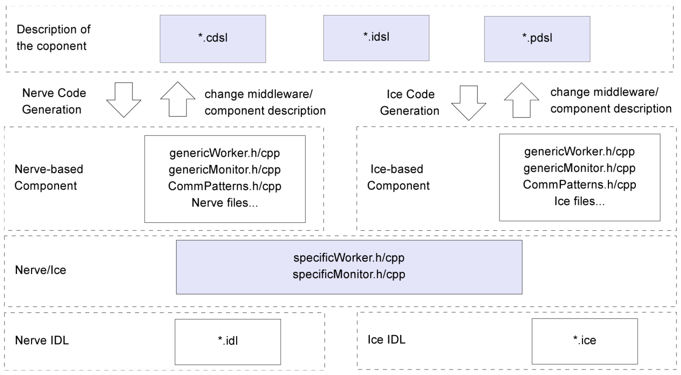


Fig. 6: Screenshot of the RoboComp Generation Process. Shaded boxes indicate those parts that are modified by the user and are middleware independent

RoboComp are depicted. Middleware must be selected before the component is created from CDSL. Then, the component skeleton (with specific, generic parts and IDL files) are generated automatically. It is worth noting that the specific part of a component is middleware independent because it is only created the first time the component is generated, whilst the other files (generic part, IDLs, etc) are always overwritten after changes and modifications. When an existing Nerve-based component is going to be deployed in an Ice-based components system, developers must regenerate the Nerve-based component first by repeating the above process, although selecting Ice as its target middleware. From a developer's point of view, no further changes in the source code of the existing component are needed.

VI. RoboComp InnerModel Simulator

As stated in the Introduction, one of the main efforts taken recently in RoboComp has been the design and construction of a robotics simulator. This effort has been partially mitigated by the reuse of the InnerModelDSL elements and of the OpenSceneGraph visualization technology that was already employed for monitoring and debugging purposes. Combining these components along with a careful design has taken us to the RoboComp Innermodel Simulator (RCIS), a 3D simulator. The most important feature of this simulator is that it is also a native RoboComp component. Being so, it can implement all the interfaces of the existing components in the hardware abstraction layer, i.e. cameras, lasers, kinect, motors, bumpers, ultrasound, tactile and any others that may come in the future. The rest of the components in a certain deployment graph can communicate to these interfaces as if they were the original components, facilitating enormously the development cycle of complex algorithms. Figure 7 shows a screenshot of RCIS.

Having complete control over the simulator kernel allows us to adapt it to our needs. For example, it is very useful to be able to activate or deactivate the physics engine or to modify the level of noise in the simulated sensors and actuators, Also, we plan to introduce semi-autonomous humans in the system to simulate and develop HR interaction behaviors. We should be able to, for example, connect the OpenNI tracking software to a synthetic RGBD sensor located in the robot

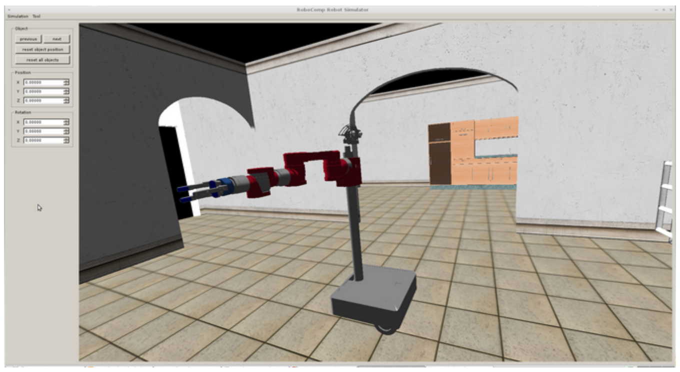


Fig. 7: Screenshot of the RoboComp InnerModel Simulator with a model of the robot Locky in RoboLab's Living Lab.

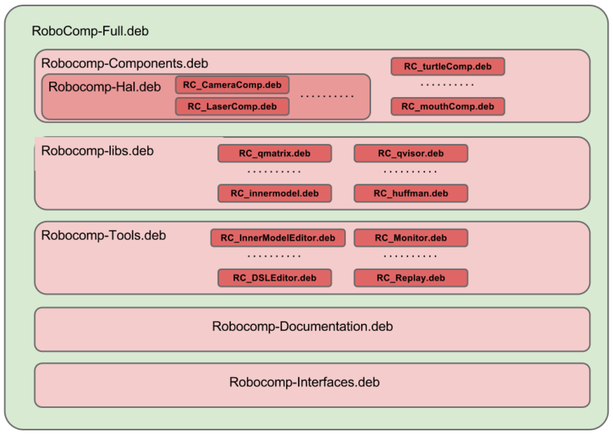


Fig. 8: Distribution of RoboComp in Debian packages

inside the simulator and track the evolution of the synthetic human figure.As a side effect of these developments we have built InnerModelViewer, a graphic editor of InnerModelDSL files. This tool has proven of great utility in the modelling of new robots and scenes using meshes and partial models created in other modelling programs such as Blender. Finally, a last feature of RCIS that is hard to find in other simulators is that it provides a flexible interface to control objects on the fly. This interface is implemented as a RoboComp interface -an IDSL- so it can be accessed from any other component. These components can create and transform elements in the scene graph. As a consequence, RCIS may be used as an internal modelling and simulation system. This use of a full-fledged simulator as a cognitive module has been proposed before in theories of consciousness [18] and we plan to include it as the central element of a new cognitive architecture being developed on top of RoboComp, called RoboCog. The simulation capabilities of RCIS can be used internally to predict the outcome of the robot actions on its represented environment. The robot itself would occupy the central place in the simulator and the objects and agents around it would be modelled and updated by the robot's perceptive system. Sensor models in the original RCIS now can be used to generate the sensorial data that the model of the robot would perceive when interacting with its modelled environment. Furthermore, the robot's self-model could be temporarily cloned to execute and evaluate plans computed by an opportunistic task-planner. The unfolding of alternative courses of action in the internal simulator and the interleaved execution of the partially validated plan creating a real course of action, is part of our current ongoing research on RoboCog.

VII. Packaging RoboComp

RoboComp is a framework designed for robotics software developers but might also be used by final software users. Currently, the standard workflow involves downloading the entire source code from the svn repository, manually installing required dependencies, compiling the components/tools needed and running them, usually along with some new code or components written by the user. Most of roboticists work implies the use of several common unmodified components directly downloaded from the repository and compiled without any modifications. Because of this, with the exception of some work on building one component from scratch or adding some small changes to an existing one, the rest of the components source code usually remains unchanged. Final users might also be interested in taking a look into RoboComp as a tool to make a robot move without any development involved or just to test some devices. In this cases, dealing with the entire framework source code becomes nothing but a handicap. In order to ease these situtations, a new way of distributing RoboComp has been introduced. The whole framework has been broken down into different parts, compiled and packaged for easy distribution and usage. To this end, the widely extended Debian packages format has been used, since most of current Linux distributions accept them as a binaries installation source.

In order to allow a smooth coexistence of binaries and sources, RoboComp will use two main locations: the default installation directory placed on `/opt/robocomp` , and a working space placed under the user home directory `∼/RC_worckspace` . Both of them will maintain a similar structure with the only difference that the installation directory will not contain sources. Instead, any needed sources should be manually placed by the developer under the workspace directory for any modifications. When the work on user components is finished they can be installed by running `#make install` as superuser, and thus placed under the installation directory so they can be used as if they had been installed from packages. When a component is installed its binaries and other relevant non-source files, such as default configuration, are placed under the installation directory and the PATH variable is updated in order to make them executables as regular commands.

Figure 8 shows the packages that came out of the RoboComp breakdown and packaging process. First, there is one package that contains the whole documentation of the framework. Then, for the tools, a package has been created for each of them, so you can choose the ones you need and, at the same time, you can have them all by just installing the virtual package robocomp-tools. Components packages work in a similar way. They came also in individual packages and Robocomp-Components is a virtual package containing all stable components. Some other group of components are also available through virtual packages like robocomp-hal or robocomp-vision. The package robocomp-classes contains classes needed for RoboComp development. With this packaging approach RoboComp becomes more easily distributable and better usable for roboticists not interested in the underpinnings of the supporting framework. It makes installation easy and fast since users do not have to take care of dependencies, compilation and environment variables.

VIII. Conclusions and future work

A. Conclusions

In this paper we have described recent progress with the RoboComp framework. We have introduced the new model driven approach based on DSLs recently implemented that includes: a) the CDSL used to describe general component properties; b) the IDSL used to describe component interfaces; c) the DDSL used to describe the deployment plan of local and networked components; d) the PDSL used to describe templates of component parameters and e) the InnerModelDSL used to describe robot kinematics, sensors and scenes. These new tools have been integrated in an Eclipse-based syntax-aware editor that allows users to create and maintain their components reducing development times. Users can focus on the implementation of the algorithms instead of wasting time on low-level details of the framework and the middleware. Moreover, RoboComp's components structure allows developers to create and modify existing components, interfaces and parameters without interfering with existing parts of the component, improving the lifecycle of the whole system.

Until now, the dependence of RoboComp in Ice has prevented developers from adapting components to new requirements such as real-time or a specific QoS. Therefore, we have also improved the DSLs to make RoboComp a middleware-independent robotics framework. This way, three communication patterns (*command*, *query* and *publish*) (adapted from [15]) are now used to specify the operation semantics and to abstract users about the middleware to use at run-time. Thus, we have proved this solution integrating RoboComp with Nerve, a novel middleware which provides our framework with real-time and QoS features.

The RoboComp InnerModel Simulator along with the InerModelDSL constitutes a strong alternative to those currently available. Moreover the run-time interface of the simulator allows the developer to use this tool for other purposes such as a cognitive simulation tool. All these new improvements constitute a great advance of RoboComp in integrating state of the art Software Engeneering methods that will guarantee increased levels of scalability and adaptability in the future.

B. Future Work

A future need for the DSLs is the ability to represent hierarchical representation of groups of components. In the future one component should automatically be created to act as a proxy for all incoming communications to the group. This rearrangement will make a group of components appear as a single one to the rest, at the cost of a certain increase in communication delay. We believe this additional level of abstraction is necessary to handle hundred of running components, a common situation in future complex robotic scenarios.

The middleware independence opens the possibility for RoboComp components to communicate with components in other frameworks. Since all communication code used in the user programmed part of the components is middleware independent, it should be possible to specify in the CDSL that a certain required proxy comes from a, for instance, ROS component. There is now work in progress towards model-based interconnection of different frameworks.

Finally, the new RobCog architecture that will use RCIS as its internal modelling and simulation cognitive module has also started initial developments and we plan to run soon the first tests in the new humanoid social robot Loki, built in close collaboration with the University of Castilla-La Mancha.

Acknowledgment

This work has been partially supported by the Spanish Ministerio de Ciencia e Innovación TIN2011-27512-C05-04 and AIB2010PT-00149, and by the Junta de Extremadura projects IB10062.

References

[1] J. He, X. Li and Z. Liu. "Component-based Software Engineering: the Need to Link Methods and their Theories." *Proc. of ICTAC 2005*, Lecture Notes in Computer Science 3722, pp. 70-95, 2005.
[2] L.J. Manso, P. Bachiller, P. Bustos, P. Nuñez, R. Cintas and L. Calderita. "RoboComp: a Tool-based Robotics Framework". *In Proc. of Int. Conf. on Simulation, Modeling and Programming for Autonomous Robots* (SIMPAR). Pages: 251-262. 2010.
[3] M. Henning and M. Spruiell. "Distributed Programming with Ice", *ZeroC*. 2009.
[4] ROS open source community. "ROS: The meta-operating system for robots." *Available at http://ros.org* 2011.
[5] C. Schlegel, A. Steck, D. Brugali and A. Knoll "Design Abstraction and Proccesses in Robotics: From Code-Driven to Model-Driven Engineering" *In 2nd International Conference on Simulation, Modeling and Programming for Autonomous Robots* (SIMPAR) 2010.
[6] R. W. Claus, N. Wang, D. C. Schmidt and C. ORyan, "Overview of the CORBA Component Model". *Component Based Software Engineering Putting the Pieces Together* Pages: 1-16. 2011.
[7] "Adaptive Communications Environment" *http://www.cs.wustl.edu/∼schmidt/ACE-overview.html*
[8] N. Ando, S. Kurihara, G. Biggs, T. Sakamoto and H. Nakamoto. "Software Deployment Infrastructure for Component Based RT-Systems." In *Journal of Robotics and Mechatronics*. Vol.23, no.3, pp. 350-359. 2011.
[9] A. W. Brown "Model Driven Arquitecture: Principles and practice" *Sofware and systems modeling* Pages: 314-327. 2004.
[10] Object Management Groups Robotic Technology component standard, version 1.0. Available at http://www.omg.org/spec/RTC/1.0/
[11] D. C. Burnett, J. Carter, J. Barnett, M. Bodell, R. J. Auburn and R. Alkolkar "State Chart XML (SCXML): State Machine Notation for Control Abstraction"
[12] Object Management Group: Data Distribution Service for Real-time Systems (DDS), version 1.2, 2007.
[13] Gerkey, B.P., Vaughan, R.T., Howard, A. "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems." *In Proceedings of the 11th International Conference on Advanced Robotics*. pp. 317-323, 2003.
[14] Martínez J, Romero-Garcés A, Manso L, Bustos P "Improving a robotics framework with real-time and highperformance features." *Simulation, Modeling, and Programming for Autonomous Robots*, Lecture Notes in Computer Science, vol. 6472, Springer, 2010; 263274.

[15] Schlegel, C.: "Communication Patterns as Key Towards Component Interoperability" *In Software Engineering for Experimental Robotics. STAR Series, vol. 30, pp.183210* (Springer, Heidelberg) 2007.

[16] Cruz, J. M., Romero-Garcés, A., Rubio, J. P. B., Robles, R. M. and Rubio, A. B., "A DDS-based middleware for quality-of-service and high-performance networked robotics." *Concurrency Computat.*: Pract. Exper.. doi: 10.1002/cpe.2816 2012.

[17] PrismTech. OpenSplice DDS. Available at http://www.opensplice.com 2011.

[18] H, Owen, and R. Goodman. "Robots with internal models Journal of Consciousness Studies" 10 (4): 1-45. 2003.