2014 IEEE International Conference on Robotics & Automation (ICRA)
Hong Kong Convention and Exhibition Center
May 31 - June 7, 2014. Hong Kong, China

Modeling and Reusing Robotic Software Architectures: the HyperFlex Toolchain

Luca Gherardi and Davide Brugali

Abstract— **During the last years robotic researchers have been able of developing extremely complex applications. The complexity of these applications is reflected by the variety of functionalities involved, which are provided by a significant number of components. Although the reuse of software components is becoming a best-practice, the reuse of reference architectures, which model sub-systems providing functionalities common to a great number of applications, is still uncommon.**

This paper provides two contributions to this topic: (a) a development process that defines how reference architectures can be exploited for building robotic applications, (b) the HyperFlex software toolchain, which has been developed for supporting the design and the reuse of reference architectures. The idea presented in this paper is simple yet powerful: instead of building complex applications by reusing single components, even more complex applications can be developed by reusing reference architectures of mature sub-systems.

I. Introduction

Robot software systems are typically concurrent, distributed, embedded, real-time and data intensive. Computation performance has always been a major requirement, especially for autonomous robots that process sensory information and have to react in a timely fashion to events occurring in the environment. Nowadays however, due the increasing complexity of robotic applications, modularity, reusability and composability are also considered important factors. In order to satisfy these conflicting requirements, robotic control systems have evolved from monolithic applications, which ran on a single processor, to distributed component-based architectures [1].

These software systems are characterized by complex interactions between components. Managing concurrent access to shared resources by multiple activities while guaranteeing real-time performance is one of the main issues. For this reason research teams developed robotics-specific component-based frameworks, which offer mechanisms for real-time execution, synchronous and asynchronous communication, data flow and control flow management, and system configuration (see [2] for a survey).

Thanks to these software frameworks, robotic researchers have developed hundreds of open source components and designed extremely complex applications, such as robots that cooperate to make pancakes [3] or that can fetch a drink from the fridge and deliver it to a human [4]. The complexity of these applications is reflected by the variety of functionalities involved, which are provided by a significant number of components. Designing these components in such a way that they can be reused in different applications is a difficult task, and even more challenging is designing the system architecture. The reason is that these tasks require advanced software engineering techniques, which are not always mastered by robotic software developers.

L. Gherardi is with the Institute for Dynamic Systems and Control, ETH Zürich, Switzerland. lucagh@ethz.ch. D. Brugali is with the Dept. of Engineering, University of Bergamo, Italy brugali@unibg.it

In the last years few research teams have tried to address this problem by developing robotics-specific software toolchains. These tools are designed according to the Model Driven Engineering (MDE) paradigm and facilitate the design of robotic component-based applications. An example is BRIDE, which has been developed in the context of the European Project BRICS [5] and allows the users to graphically design models of new (or legacy) software components. These models describe the components' interfaces and are decoupled from their implementations. Software engineers design these models and robotic developers can compose them for designing their applications and generating a configuration file for the system deployment [6].

These toolchains provide the users with the possibility of reusing the model of the same component for designing different applications. However the same operation is not possible with models of larger portions of a system. This strong limitation prevents software engineers to define reference architectures for mature functional sub-systems (e.g. robot navigation and robot manipulation) and distribute them as configurable units that can be composed for building complex applications.

A reference architecture for a specific domain is an architecture template for all the software systems in that domain. It defines the fundamental components of the domain and the relations between them. The architecture for a particular sub-system is the result of the configuration of the reference architecture [7]. For example, according to this definition, a 3D perception expert can define a reference architecture for a sub-system providing perception functionalities (i.e. image acquisition, filtering, feature extraction, registration, segmentation, etc.). This reference architecture can be subsequently configured by robotic developers and integrated with other components and/or sub-systems for developing complex mobile manipulation applications.

The definition of reference architectures is already a best practice in other domains (for example the automotive community has defined the AUTOSAR standard [8]). Their adoption provides two major benefits. First, they can be designed by developers with a software engineering background and reused by other developers that do not master software engineering techniques. Second, researchers can focus on the development of algorithms related to their specific domain and exploit reference architectures for other functionalities, which are required by the applications but are not part of the developers' competencies. Based on these arguments we claim that the definition of reference architectures would boost the development, the testing and the benchmark of new components, sub-systems and entire applications.

This paper provides two main contributions: (a) a development process that defines how reference architectures can be configured and integrated in order to build complex robotic applications; and (b) a meta-model and a graphical editor that support the users during the development process and are integrated with the HyperFlex toolchain [9][10]. The two contributions are the natural extension of what has been developed in BRICS: a set of principles and tools that support the software engineers and the robotic developers in the task of designing, reusing and composing robotic systems. The idea presented in the paper is simple yet powerful: instead of building complex applications by reusing single components, we want to provide the capability of building even more complex applications by reusing and composing reference architectures of entire sub-systems.

The rest of the paper is organized as follows. Sec. II introduces the development process. Sec. III describes the meta-model and the graphical editor. Sec. IV exemplifies by means of a case study the concepts previously defined. Finally Sec. VI and Sec. V present related works and draw relevant conclusions.

II. The Development Process

This section introduces the guidelines that describes how reference architectures can be used for boosting the development of new applications. Two possible processes, which can be complementary (as presented in Sec. IV), are described below.

A. *Integrating customer-specific components*

The first process that exploits reference architectures is depicted in Fig. 1. The reference architecture defines a functional sub-system template in terms of: (a) interfaces of customer-specific and customer-independent components, (b) connections between them and (c) components' configuration properties. Customer-independent components are general components that can be downloaded and reused in several applications (e.g. path planning). Customer-specific components are instead components the implementation of which has to be developed from scratch because is strongly coupled with the developer's requirements (e.g. hardware drivers).

The development process is made of two activities. During the first one (*Customer specific components development*) the reference architecture is used as interface specification for the implementation of the customer-specific components. During the second activity (*Integration*) the information provided by the reference architecture is used for the definition of the connections between customer-specific and customer-independent components and for the configuration of the components' parameters. These two activities customize the template provided by the reference architecture and produce the concrete architectural model of the functional sub-system (*Reference Architecture Configured Model*). Note that these models must conform to a software framework specific meta-model, such as the Orocos component meta-model presented in Sec. III or a ROS component meta-model.

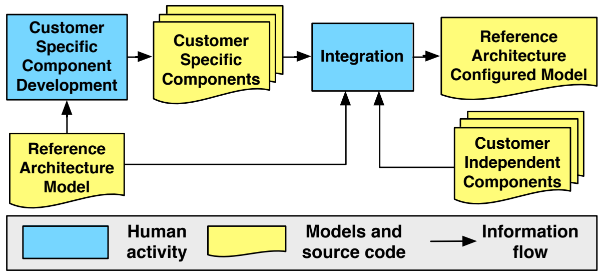

Customer Specific Component Development
Customer Specific Components
Integration
Reference Architecture Configured Model
Reference Architecture Model
Customer Independent Components
Human activity
Models and source code
Information flow

Fig. 1: Customer-specific components customize the reference architecture of a functional sub-system.

This process can be exemplified as follows. A reference architecture for controlling a robotic arm defines the interfaces of the components providing all the required functionalities (e.g. inverse kinematics and trajectory following) and the interfaces of the component needed for interfacing the robot. This component, being customer-specific, has to be developed and integrated in the reference architecture. The result is a concrete sub-system for controlling a manipulator, which can be deployed or alternatively reused as reference architecture for building more complex systems.

B. *Reusing reference architectures in complex applications*

The second process that exploits reference architectures is depicted in Fig 2. In this case a reference architectures is a reusable building block, which is composed with other sub-systems to develop complex applications. We assume that all the components of the reference architecture are available. They parameters can be configured but they need not to be implemented (the reference architecture can be the result of the previous process). For example an application developer can design a sub-systems providing high-level functionalities (e.g. trajectory generation) and integrate it with the reference architecture designed for controlling the arm.

The development process is made of two activities. During the first one (*New Functionalities Development*) the application developer implements new sub-systems providing functionalities that are application-specific and are not yet available. These sub-systems leverage on and coordinate the functionalities provided by the reference architectures. During the second activity (*Integration*) the architecture of the final application is designed by integrating the application-specific sub-systems with the configured reference architectures. Note that when several reference architectures are composed, they typically provide different capabilities that do not overlap (e.g. manipulation and perception). This ensures that two sub-systems designed by different parties do not rely on the same components.

The main difference with respect to the development process presented in Sec. II-A is that in this case the reference

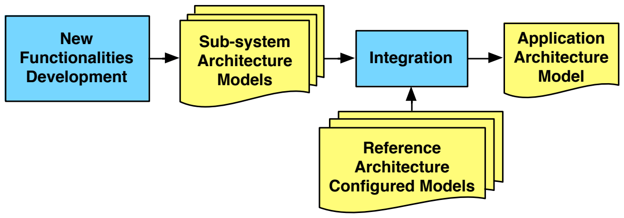

New Functionalities Development
Sub-system Architecture Models
Integration
Application Architecture Model
Reference Architecture Configured Models

Fig. 2: Configured reference architectures can be composed with custom sub-systems for building complex applications.

architecture is just a reusable black-box. It does not provide specifications about the architecture of the final application. This approach allows the developers to (a) focus on the implementation of the application-specific components that provide the functionalities for which they are expert; and (b) leverage on existing solutions for other functionalities required by the application.

III. Meta-Model and Tools

A component-meta model describes the architectural elements that can be used for modeling software architectures and how they can be composed. This section briefly introduces the main concepts of Orocos and presents the meta-model and the editor developed for supporting the design and the reuse of reference architectures. Due to limited space we focus only on Orocos, however the meta-models and the tools are available also for ROS. Note that, in order to be deployable, an architectural model must conform to a specific software framework. The transformation of a framework-independent model into an framework-specific model is an action that cannot be fully automated, due to the architectural mismatches that exist between the different robotic software frameworks (communication mechanisms, behavioral models, etc.). This is the main reason for which architectural models cannot be completely modeled by means of a framework-independent meta-model. The use of a framework-specific meta-models is instead required.

A. Orocos

Orocos is a software framework particularly suited for the design of hard real-time robotic systems. An Orocos system is a flat graph made of several task contexts (i.e. components). Task contexts interact according to two paradigms: (a) by exchanging data and events asynchronously through input/output data ports (data flow communication paradigm), and/or (b) by synchronously or asynchronously calling operations provided by other task contexts (client/server communication paradigm). Finally properties allows the configuration of the Task Context's execution behavior.

Orocos applications are deployed by means of a tool called *Deployer*, which is in charge of (a) loading the binary files containing the implementation of the various components, (b) instantiating them, (c) setting the desired values for the components' properties, (d) connecting the components' interfaces, and (e) starting the application. The information required by the deployer for executing these operations is encoded in a file (script or XML), which is called *deployment file* and describes the application architecture.

B. Orocos Component Meta-model

The Orocos meta-model allows developers to model the architecture of Orocos systems. Although a meta-model of Orocos is provided with BRIDE, we have redefined it in order to overcome the following limitations: (a) the absence of composition mechanisms, which makes hard the definition and the reuse of reference architectures, and (b) the impossibility of modeling service-based communications. In this section, in order to avoid ambiguities, we call our meta-model the *HyperFlex Orocos component meta-model*.

Figure 3 depicts the HyperFlex Orocos component meta-model, which is organized in three parts for sake of readability. White classes were defined for BRIDE, gray classes have been designed for the HyperFlex meta-model. An Orocos `System` contains a `Composite`, which hierarchically aggregates `TaskContexts` according to the Composite design pattern. In the rest of the paper we use the word *component* (i.e. `AbstractComponent`) when we introduce concepts that apply to both task contexts and composites.

Note that a *composite* is only a modeling entity. It does not map to any implementation entity defined in Orocos. Orocos allows the *service* composition, but not the composition of *components*. The *composite* is a powerful mechanism that allows us to introduce the hierarchical composition at model level and to design and distribute reference architectures.

Figure 3a depicts the entities used to model the mechanism for the data-flow and event-flow communication. Task contexts may have several input and output *data ports* (`TCInputDataPorts` and `TCOutputDataPorts` respectively), which are typed by a `DataType` and are connected by means of `ConnectionPolicies`. Input port also defines the boolean attribute `eventPort`. If true the component computation is triggered by new events received by this port, if false the component computation is triggered periodically according to the component's period.

A Connection Policy defines the communication channel between two ports and is characterized by a name, a type (data, buffer, or circular buffer), a lock policy (unsync, locked or lock free), a buffer size (which is always 1 when the type is data), and a transport (Orocos, MQueue, ROS), which defines the middleware used for the communication. A Connection Policy holds a reference to the two ports when it is used for connecting two Orocos task contexts, while it specifies only the source (output port) or the target (input port) when it is used for connecting an Orocos task context to a component than runs under a different software framework, e.g. ROS. In this case the field `note` is used for specifying the name of the topic on which the data has to be published (or read).

A composite may expose a subset of the ports of its components in order to provide/require data to/from components defined in other composites. For this purpose, a composite define a set of `CompInputDataPorts` and/or `CompOutputDataPorts`, which hold a reference to the exposed ports. OCL constraints ensure that connections are

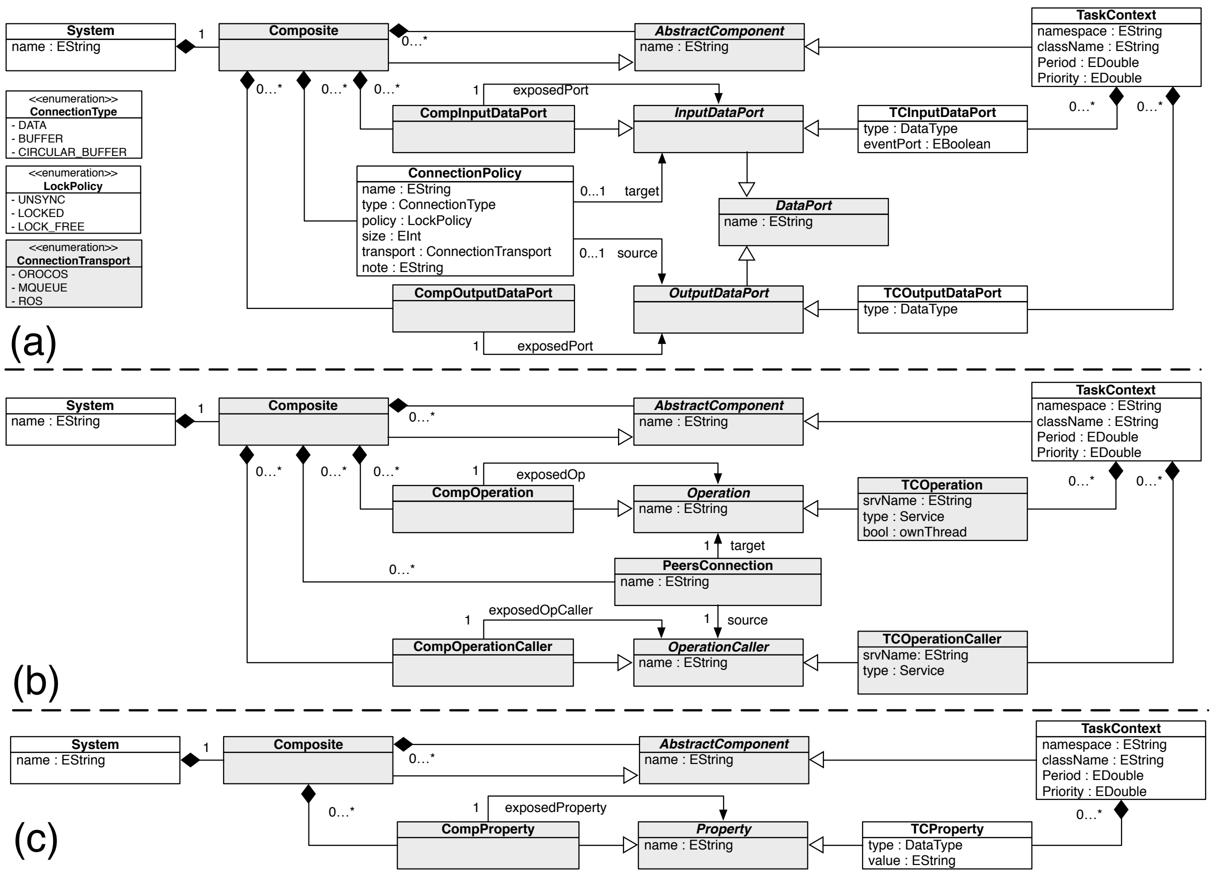


Fig. 3: The Orocos component meta-model (data-flow interfaces (a), service interfaces (b), component properties (c))

defined between ports that are typed by the same data type and belong to components defined in the same composite.

Figure 3b depicts the entities used to model the mechanisms for service-based communication. Task contexts can define provided and required service interfaces (`TCOperation` and `TCOperationCaller` respectively). They are typed by a `Service` (operation signature) and can be promoted at the level of the composite interfaces (`CompOperation` and `CompOperationCaller` respectively). Operations and operation callers can be connected by means of `PeersConnection` entities. OCL constraints ensure that peers connections are created between operations and operation callers that conform to the same service and belong to components defined in the same composite.

Figure 3c depicts the mechanisms for the definition of properties. A `Property` provides an interface for setting the value of a parameter defined in the implementation of a task context (e.g. the PID parameters). Properties, as well as the other interfaces, can be promoted at the composite level.

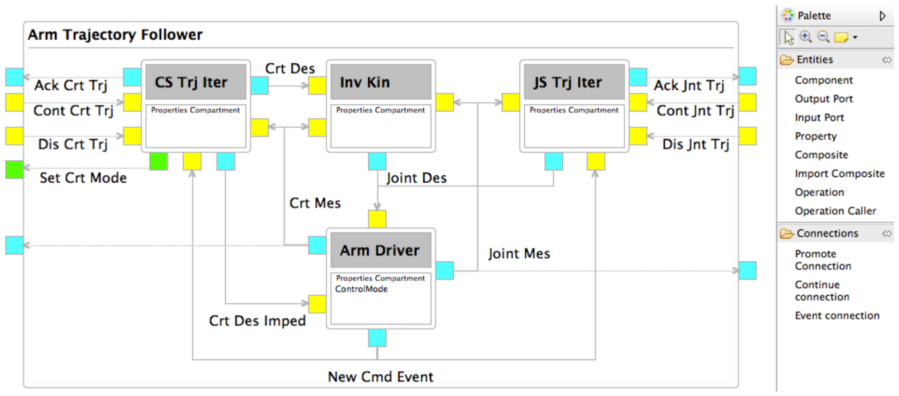
C. Tool support

On the top of the component meta-model presented above we have developed an Eclipse plugin that allows the design and the reuse of reference architectures (some screenshots are depicted in Fig. 4). This tool provides the means for modeling software components, connecting their ports/services and wrapping them into a composite, which promotes some of the components' interfaces. Note that, being the composite a modeling entity, a call to a promoted interface is the equivalent of a call to the associated component interface. When this is not the desired behavior, it is useful to define an additional component inside the composite (i.e. coordinator), which exposes its interfaces to the composite level and coordinates the execution of the other components defined in the composite (the interfaces of these components have to be connected to the coordinator and not anymore exposed). The editor provides a mechanism for importing an existing composite in a higher-level composite, allowing in this way the hierarchical composition.

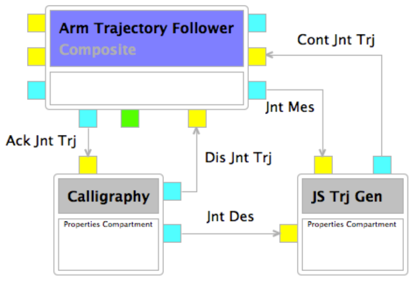
Once the application architecture has been designed, a model-to-text transformation provided by the editor generates the deployment file, which can be used by the Orocos deployer for launching the application.

IV. CASE STUDY

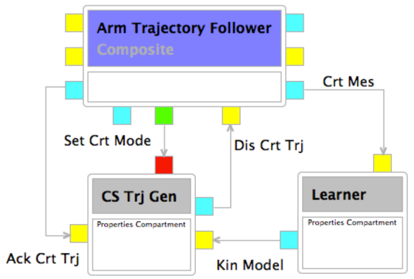
In this section we illustrate by means of a case study how reference architectures can boost the development of different systems. We have analyzed the architectures of two existing real applications and we have refactored them in order to decouple the commonalities from the variabilities. The two applications were developed by researchers of the IDSC group at ETH Zurich. In the first one a KUKA Lightweight Arm (LWR) is used for drawing Japanese characters [11], while in second the same arm is used for opening



(a) The reference architecture of the Arm Trajectory Follower



(b) The Calligraphy architecture



(c) The Learning architecture

Fig. 4: The case study architectures: white rectangles depicts components (grey header) and composites (blue header), yellow rectangles input ports, cyan squares output ports, green squares operations, and red squares operation callers.

a cabinet door and learning its kinematic model [12]. These applications have been chosen because they are based on a set of common functionalities but execute two significantly different tasks. This characteristic allows us to emphasis the advantages of defining and reusing reference architectures. The result of their refactoring consists of a new design for the two applications. They are now built on the top of a reference architecture, which encodes the commonalities and provides functionalities for controlling a robotic arm.

Sections IV-A and IV-B describe how the activities of the development processes presented respectively in Sec. II-A and II-B have been applied to the case study.

A. Integrating customer-specific components

Figure 4a depicts a reference architecture providing low-level functionalities for controlling a robotic arm and following cartesian space and joint space trajectories (the figure is a screenshot of HyperFlex). The components defined in this architecture can be interfaced with a general arm, provided that there is a driver for controlling it and that this driver offers the same interfaces described in the architecture (being the driver an hardware abstraction, the cartesian position/impedance control and the joint control can be implemented in a software component, or in a composite, when not provided by the hardware). The reference architecture is a composite made of the following components:

- *Arm Driver*: implements the joint and cartesian control of the arm. It defines a property for setting the control mode (joint or cartesian space) and provides ports for reading set points and publishing measured data (both in joint and cartesian space). Finally it has an output port for sending event notifications when a new command is required (e.g. previous command completed).
- *Joint Space Trajectory Iterator* (`JS Trj Iter`): receives as input a continuous (polynomial function) or discrete joint trajectory (vector of joint configurations) and interacts with the Arm Driver for following it. It reads the current joint configuration and sends back the next desired configuration upon request (triggered by the event `New Cmd Event`).
- *Cartesian Space Trajectory Iterator* (`CS Trj Iter`): receives as input a continuous (polynomial function or spline) or discrete cartesian trajectory (vector of poses) and interacts with the Arm Driver for following it. It provides a service for setting the control mode, which can be either *position* or *impedance*. This component reads the current end effector pose and produces as output, upon request (event `New Cmd Event`), the next pose. In compliant mode the position is sent directly to the Arm Driver while in position mode it is sent to the Inverse Kinematics component.
- *Inverse Kinematics* (`Inv Kin`): implements the velocity and position inverse kinematics and is configured according to an XML file that describes the kinematic model of the arm. It receives as input a desired cartesian pose and using the data produced by the Arm Driver computes the corresponding joint configuration.

The *Arm Trajectory Follower* composite exposes a subset of the data ports provided by its components, in particular the output regarding the measured data, the events for notifying the completion of a trajectory (`Ack Crt Trj` and `Ack Jnt Trj`), and the continuous and discrete trajectories inputs. Moreover it promotes the operation for setting the cartesian control mode.

We assume that this reference architecture is designed by a software expert, who makes it available to robotic developers. The first step that leads to the implementation of the applications described above consists of configuring the reference architecture according to the process presented in Sec. II-A. During the first activity the *Arm Driver* interfaces defined in the reference architecture are used as specification for implementing a component that provides a driver for the KUKA LWR (customer-specific component). During the second activity this component is integrated with the other components of the reference architecture, providing in this way a reusable composite.

The process described in Sec. II-A allows the development of a trajectory follower for a general arm, in this specific case the KUKA LWR. The only task required to the developer is the implementation of the arm driver. The reference architecture relieves the developers from the tasks of (a) defining the driver interface, (b) implementing the other components and (c) defining the sub-system architecture.

B. Reusing reference architectures in complex applications

The process described in Sec. II-B allows the development of different applications, which are built on the top of the Arm Trajectory Follower. This subsection presents the architecture of the two applications described above.

Figure 4b depicts the architecture of the first application. Two new components, the *Calligraphy* and the *Joint Space Trajectory Generator* (`JS Trj Gen`), use the functionalities provided by the Arm Trajectory Follower composite. The Calligraphy component reads a set of XML files, which contains the instructions for drawing Japanese characters, and generates for each of them a set of discrete trajectories, which defines how the strokes that compose a character have to be drawn. The Calligraphy component sends to the JS Trj Generator the joint configuration corresponding to the pose in which the brush has to be moved for drawing the first stroke. The generator computes a continuous joint trajectory and sends it to the Arm Trajectory Follower, which executes it and notifies the Calligraphy component when the brush is in the desired pose. When the Calligraphy component receives this event, it sends the stroke discrete trajectory to the Arm Trajectory Follower and waits until the stroke is completed.

Figure 4c depicts the architecture of the second application. The functionalities provided by the reference architecture are used by two components, the *Cartesian Space Trajectory Generator* (`CS Trj Gen`) and the *Learner*. In the initial configuration the arm is attached to the handle of a cabinet door (whose kinematic model is unknown) and is controlled in compliant mode. The CS Trj Generator sends to the Arm Trajectory Follower a discrete cartesian trajectory consisting of a straight line. Due to the single degree of freedom of the door, which is on the Z axis, the arm will not follow the desired trajectory. The Learner component reads the intermediate poses of the arm and use them for estimating the kinematic model of the door, which is sent to the CS Trj Generator. Using this model the CS Trj Generator is able to compute a more appropriate trajectory, which takes into account the degree of freedom of the door.

The two applications have been designed by reusing the Arm Trajectory Follower as a black box. Thanks to this approach developers can focus on the design of application-specific functionalities and are relieved from the task of developing low-level components. Note that the architecture of the two applications are not meant to represent reference architectures. In fact they share only the Arm Trajectory Follower and for the rest are significantly different.

V. Related Works

During the last decade advanced software engineering techniques have been progressively exploited for the development of reusable robotic systems [13], including component-based architectures [14] and Model Driven Engineering [15].

The European Project BRICS adopted these techniques and produced some interesting results. Two of them are the BRICS Component Model (BCM) [16] and BRIDE.

The BCM is a software framework-independent component model, which provides a set of guidelines for reasoning about the design of component-based robotic systems. However, being the BCM software framework independent, it cannot be used for modeling real systems. For this reason it is necessary to define software framework specific component meta-models, similar to the one presented in Sec III. Note that the BCM provides an elementary composition mechanism, which simply specifies that component can encapsulate sub-components. This mechanism is not enough for modeling reference architectures. We extended it by applying the Composite Pattern and the concept of interface promotion.

BRIDE is an Eclipse plugin that supports the robotic developers during the process of designing component based systems. It provides the users with the possibility of: (a) loading a set of previously modeled components and/or designing new components; (b) connecting the components interfaces in order to define their interactions; (c) generating the configuration file that describes the architecture and can be used for the deployment. The HyperFlex toolchain presented in this paper is the natural extension of BRIDE and improves it in two directions. First, it provides a powerful composition mechanism that allows the reuse of models with a larger level of granularity. Second, it supports all the communications paradigms provided by Orocos (the client/server paradigm is not supported in BRIDE).

The Proteus project [17] is another model-driven robotics initiative that provides a component meta-model and a software toolchain. Similar to BCM, the Proteus component-model contains a very basic composition mechanism, which is not enough for modeling reference architectures. Indeed the relationship between the interfaces of a composite and the interfaces of its components is not clearly presented in the paper.

Another model-driven approach from the robotics domain is OpenRTM, a robotic software framework that implements the RT-Middleware [18] specification. Differently from ROS and Orocos, which do not provide an explicit meta-model, the OpenRTM meta-model is explicitly defined and integrated with a software toolchain that provides features similar to the ones provided by HyperFlex. As well as the component model presented in this paper, OpenRTM provides the primitives for hierarchical composition. However in OpenRTM the composite is used for coordinating the execution of its components. In our case instead, the composite is a modeling and packaging entity, which allows the design and the distribution of reference architectures.

A development process for designing robotic applications is presented in a SmartSoft paper [19]. However according to this process the *SmartSoft System Integrator* is in charge of connecting and configuring existing components according to the requirements of a specific application, and for this reason his work "is rarely reusable by others" [19]. In contrast, in our approach we promote the definition of sub-system reference architectures that are configurable and easily reusable in several applications.

Finally *MoveIt!* [20] is a Motion Planning Framework recently developed in the context of the ROS community. It defines a plugin-based architecture, which allows the users to easily change and configure the implementation of the different functionalities (e.g. kinematics, collision detection, etc.). In this direction MoveIt! provides a reference architecture for motion planning systems. However, differently from the work proposed in this paper, MoveIt! does not follow a model-based approach. This makes harder the integration with other model-based techniques, such as the ones related to software variability management [9].

VI. Conclusions and Future Works

The aim of this paper was promoting the definition of reference architectures that model mature robotic sub-systems and provide functionalities common to a great number of applications. We presented a development process that defines the guidelines for exploiting reference architectures, and the open source HyperFlex toolchain, which provides the tools that support the development process.

The case study demonstrated how reference architectures can be used and composed for building complex applications. In our opinion, the design of component-based reference architectures for robotics functional sub-systems and their distribution by means of remote repositories similar to the Ubuntu Advanced Packaging Tool (APT) can significantly boost the development of new applications and bring great benefits to the robotics community.

In the future we plan to integrate Orocos and ROS meta-models and editors with the tools designed for modeling and resolving robotic software variability [9][21][22], which were previously integrated with BRIDE.

Acknowledgments

The authors would like to thank Nico Huebel and Gajamohan Mohanarajah for the valuable comments. The research accomplished at the University of Bergamo received founding from the European Project FP7 BRICS.

References

[1] D. Brugali and P. Scandurra. Component-based robotic engineering (part I). *Robotics & Automation Magazine, IEEE*, 2009.

[2] D. Brugali and A. Shakhimardanov. Component-based robotic engineering (part II). *Robotics & Automation Magazine, IEEE*, 2010.

[3] M. Beetz, U. Klank, I. Kresse, A. Maldonado, L. Mosenlechner, D. Pangercic, T. Ruhr, and M. Tenorth. Robotic Roommates Making Pancakes. In *International Conference on Humanoid Robots (Humanoids)*, 2011.

[4] J. Bohren, R. B. Rusu, E. G. Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mosenlechner, W. Meeussen, and S. Holzer. Towards Autonomous Robotic Butlers: Lessons Learned with the PR2. In *International Conference on Robotics and Automation (ICRA)*, 2011.

[5] BRICS. `http://www.best-of-robotics.org`.

[6] N. Hochgeschwender, L. Gherardi, A. Shakhirmardanov, G. Kraetzschmar, D. Brugali, and H. Bruyninckx. A Model-based Approach to Software Deployment in Robotics. In *International Conference on Intelligent Robots and Systems (IROS)*, Tokyo, Japan, 2013.

[7] A. E. Hassan and R. C. Holt. A Reference Architecture for Web Servers. In *Working Conference on Reverse Engineering*, 2000.

[8] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange. Autosar–a worldwide standard is on the road. In *International Congress on Electronic Systems for Vehicles*, 2009.

[9] L. Gherardi. *Variability Modeling and Resolution in Component-based Robotics Systems*. PhD thesis, Università degli Studi di Bergamo, 2013.

[10] L. Gherardi and D. Brugali. The HyperFlex Toolchain. `http://robotics.unibg.it/hyperflex`.

[11] S. Mueller, N. Huebel, M. Waibel, and R. D'Andrea. Robotic Calligraphy - Learning How to Write Single Strokes of Chinese and Japanese Characters. In *International Conference on Intelligent Robots and Systems*, 2013.

[12] M. Waibel, M. Beetz, J. Civera, R. D'Andrea, J. Elfring, D. Galvez-Lopez, K. Haussermann, R. Janssen, J.M.M. Montiel, A. Perzylo, B. Schiessle, M. Tenorth, O. Zweigle, and R. van de Molengraft. *Robotics Automation Magazine, IEEE*, 2011.

[13] D. Brugali and E. Prassler. Software Engineering for Robotics. *Robotics & Automation Magazine, IEEE*, 2009.

[14] D. Kortenkamp, R. Simmons, and D. Brugali. Robotic Systems Architectures and Programming. *Springer Handbook of Robotics - Second Edition*, 2013, in press.

[15] C. Schlegel, A. Steck, D. Brugali, and A. Knoll. Design Abstraction and Processes in Robotics: from Code-driven to Model-driven Engineering. In *International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)*. 2010.

[16] H. Bruyninckx, N. Hochgeschwender, L. Gherardi, M. Klotzbücher, G. Kraetzschmar, and D. Brugali. The BRICS Component Model: a Model-based Development Paradigm for Complex Robotics Software Systems. In *Annual ACM Symposium on Applied Computing (SAC)*.

[17] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane. RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications. In *International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)*. 2012.

[18] N. Ando, S. Kurihara, G. Biggs, T. Sakamoto, H. Nakamoto, and T. Kotoku. Software Deployment Infrastructure for Component Based RT-Systems. *Journal of Robotics and Mechatronics*, 2011.

[19] C. Schlegel, A. Steck, and A. Lotz. Robotic Software Systems: From Code-Driven to Model-Driven Software Development. *Robotic Systems - Applications, Control and Programming*, 2012.

[20] MoveIt! `http://moveit.ros.org`.

[21] L. Gherardi and D. Brugali. An Eclipse-based Feature Models Toolchain. In *Italian Workshop on Eclipse Technologies (EclipseIT)*, 2011.

[22] D. Brugali, L. Gherardi, A. Biziak, A. Luzzana, and A. Zakharov. A Reuse-Oriented Development Process for Component-based Robotic Systems. In *International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)*, 2012.