ArchGenTool: A System-Independent Collaborative Tool for Robotic Architecture Design

E. Ruffaldi[1], I. Kostavelis[2], D. Giakoumis[2], D. Tzovaras[2]

***Abstract*— Complex robotic architectures require a collaborative effort in design and adherence to the design in the implementation phse. ArchGentTool is a collaborative architecture generation tool which supports the design of the robotic architecture in a multi-level fashion. It comprises high-level conceptual analysis of the system to be designed, as well as low-level implementation breakdown of its functional components, acting complementary to the ROS framework. The tool facilitates reusability and expandability of the architecture to any robotic system, as it can be adapted to different specifications. A case study with the RAMCIP service robot is presented.**

I. INTRODUCTION

Notwithstanding the plethora of laborious work that has already been conducted in the area of robotics applications, the determination of a common architecture design framework remains an active research topic. This statement is proved by considering that the abundance of robotic software that has been developed during the last decades, is tightly dependent on the existing hardware specifications and their limitations [5]. Therefore, the existing architecture of the systems can not be expanded to follow the hardware advances and, consequently the re-design of a system is mandatory in order to embrace more functionalities.

Moreover, robot development is a combined field that requires joined efforts of software and hardware engineers for the design of a system architecture. Consequently, the adoption of methodologies that tackle the problem partially i.e independent software or hardware architecture, does not comprise an ample solution during the implementation of a robotic system. In accordance with the statement in [8], "*A good architecture model facilitates decision making and acts as a mediator between requirements and final implementation.*" However, it should be also stressed that a useful architecture model is the one that determines a blueprint for the developers to reproduce the envisioned system efficiently and in a structural manner.

The architecture of a robotic system refers to how a system is divided into subsystems, and how those subsystems interact [10]. Following this definition, the proposed work introduces the Architecture Generation Tool (ArchGenTool), which is a stand-alone toolchain based on a Domain Specific Language (DSL). It encompasses a consistent *hybrid* architecture design tool that facilitates joint collaboration during the system design phase. The ArchGenTool consists of a high-level conceptual description and diagrammatical representation, as well as low-level implementation of software and hardware entities, capitalizing on features illustrated in Fig. 1 which supports:

- ◇ the high-level architecture that conceptually analyzes the envisioned system;
- ◇ the low-level implementation that describes in detail the functional components;
- ◇ the mapping of the functional components to ROS-wise modularities, e.g. node, server, actionlib etc.;
- ◇ the logical analysis and diagnostics during the architecture design;
- ◇ the collaborative design among developers;
- ◇ the straightforward expandability to capture additional technical specifications;
- ◇ the detailed documentation generation in structured report forms;

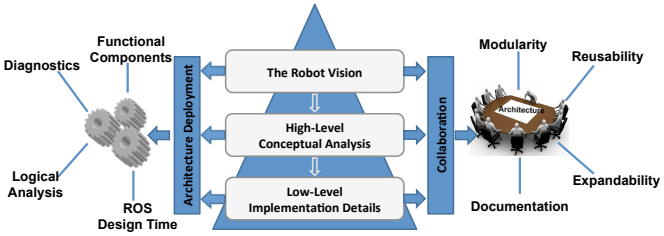

Fig. 1: Outline of the ArchGenTool basic features

II. BACKGROUND

The technical literature from a robotic design aspect classifies the architectural styles into three categories: *hierarchical*, *behavioral*, and *hybrid* [10]. The *hierarchical* model emphasizes on the high-level structure and restricts low-level horizontal communications, retaining thus poor flexibility in contemporary complex robotic systems. The *behavioral* model is build directly on groups of software modules that operate concurrently and interact with each other, a strategy which is strictly objectives-specific and eliminates the extensibility of the high-level architecture on non-trivial objectives. The *hybrid* architecture, which is the most common method in contemporary robotic systems, combines both reactive and deliberative structure and facilitates the design of efficient low-level details with a connection to high-level reasoning.

More specifically, this connection has been established through the use of DSLs which provide solutions at the level of abstraction of the problem domain, an extensive

*This work was supported by the European Commission

[1]E. Ruffaldi is with Scuola Superiore Sant'Anna, Pisa, Italy e.ruffaldi@sssup.it

[2]I. Kostavelis, D. Giakoumis and D. Tzovaras are with CERTH, ITI, Greece {gkostave, dgiakoum, tzovaras}@iti.gr

discussion of which can be found in [3]. DSLs are typically connected to a toolchain that determine the low-level (implementation level) in more detailed manner. Such a paradigm is pursued by the ROBOTML [4], which is based on a DSL to define a robotic ontology. The latter, is connected with the Papyrus modeling plugin [1] of the Eclipse platform capable of producing diagrammatic and code output of the studied system based on the Unified Modeling Language (UML). A more robotics-specific approach is BRIDE, which has been developed in the context of the European Project BRICS [2] and also comprises an Eclipse plugin. BRIDE allows the developers to graphically design models of new software components. BRIDE offered reusability capabilities of the same component for designing different applications. However, this was feasible only for systems with deteriorated scale and, therefore, its reusability capacity is not possible with models of larger portions of a system. Although hybrid systems proved adequate to thoroughly determine the architecture of a robotic system, the connection among high and low level depends on existing software platforms (Eclipse) that lacks the immediacy with the robotics hardware.

A more sophisticated solution that tackles the connection of high and low level architecture is the HyperFlex toolchain [5]. This set of tools acts as a graphical design tool for software architecture of robotic systems. It automatically generates the configuration files for various software frameworks such as ROS [7], Orocos [1], and SCA. However, this is not a stand-alone tool, yet it is also based on Eclipse software and comprises a natural extension of BRIDE, by allowing the reuse of models with a larger level of granularity and by supporting all the communications paradigms provided by Orocos. An extension to this work is the Robotics Run-time Adaptation (RRA) framework [6], capable of partially resolving the variation points at deployment-time and postponing the resolution of the remaining variation points at run-time. RRA also reasons about the run-time variability to choose the configuration that suits to the dynamic environment changes. This work is defined in an adaptation model which is also integrated in an Eclipse plugin. In addition, the same authors in [8] introduced the Architecture Modeling and Analysis Language (AMAL), which enables architecture development based on custom requirements or by integrating existing heterogeneous system paradigms. This method is relied on the Open Semantics Framework and comprises part of the SafeRobots ecosystem. The IDE of this language is also based on the Eclipse framework.

ArchGentTool is a *hybrid* architecture design tool which links the high-level abstraction of the system and the low-level implementation details utilizing the ROS framework. To the best of our knowledge, for the first time *a stand-alone architecture* generation toolchain (Eclipse-independent) is introduced herein, which due to its modular nature allows reusability and scalability of the existing models into larger systems.

[1]http://www.eclipse.org/papyrus/

[2]http://www.best-of-robotics.org

III. CONCEPT

The ArchGenTool concept is based on the handling of the DSL in a human-readable textual form. It involves the decomposition of the system's specification in pieces, much like a software program and allows to perform the collaboration by means of the code versioning using the GitHub web platform. Each developer can experiment with the architecture design, integrating the comments from others using branching, keeping track of the changes along time, and even executing verification tests on its own proposed changes. Quality measures of the architecture can be obtained and the designers can assess the progress of the process, such as detailing the implementation or specifying timing requirements. While the textual representation is useful for collaboration, the visual component is important for understanding the relationships of the specified components, for reporting and for sketching.

The underlying DSL is based on a component-based structure of the robotic architecture in which component instances exchange data via typed ports. The component semantics with ports, parameters and sub-components is similar to what is present in SysML although here is simplified taking into account the common patterns found in Robotics.

The ArchGenTool follows a toolchain approach in processing the architecture specification, much like a compiler with several backends and intermediate analysis steps. All the inputs are specified in YAML that is a text-based light-markup language widely used in ROS and OpenCV applications [3]. This description format is lighter than full markup languages such as XML but at the same time allows to specify variety of structures such as lists and dictionaries. The input YAML files are loaded and used for building the internal DSL model.

Then intermediate stages are executed to analyze the model, to compute the dependencies and to apply the required verifications. The verification stage is aimed at identifying disconnected or partially connected components, type mismatches, and it gives the possibility to higher level diagnostics associated to semantics of the component types (e.g. filter vs actuation components). One example of the analysis stage is constituted by the timing evaluation that identifies or infers the timing of the components and can support bandwidth-latency estimates inside the architecture (see V-B).

Finally the architecture is transformed by the backend(s) of the tool into visual representation, in structured report form and in a self-standing packaged YAML representation. These operations can be applied to the whole architecture or can be limited to a given layer.

Two layers of the architecture have been defined: the high-level functional architecture module and the low-level implementation architecture module. The first deals with functional components (see Sec. IV), presenting their role, the interdependencies and the data being exchanged. The low-level module is instead aimed at describing the tangible architecture implemented over a robotic middleware, ROS

[3]http://yaml.org/

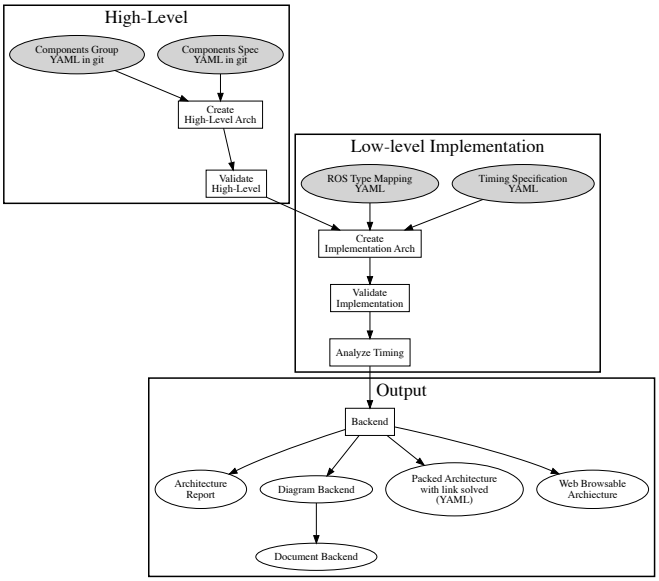

Fig. 2: Workflow. On top the input YAML files are transformed into a high-level architecture. The architecture is then extended with the implementation details. Then, analysis phases can be applied before the execution of the backends.

in this case, while preserving the connection with the high-level module. The connection between the high-level and the low-level modules is considered important because it allows to keep track of the conceptual flow, and whenever applicable, the fulfillment of functional specification requirements, defined in the high-level module when considering the actual system implementation. The overall structure of the ArchGenTool is shown in Fig. 2 presenting the steps of the toolchain.

IV. High-Level Module: Functional Specifications

The high-level module of the ArchGenTool is responsible to decompose the robotic system into its basic elements by identifying all the building blocks required for the materialization of the system and their dependencies in terms of inputs and outputs. The functionalities of the system, which have been established during the requirements identification phase, are organised in the ArchGenTool as the "Group of Components". Each "Group of Components" gathers all the core elements related to a specific task and, therefore, can be decomposed into the "Functional Components". The latter correspond to the core software elements, which are the building blocks to be developed for the construction of the robotic system. Each "Functional Component" is associated with descriptive information of the *Functionality*, the *Operational Requirements* and the *Performance* indicators. The *Performance* indicators are each described with the target criteria for their fulfillment, i.e the "Accuracy" and the "Execution Time". The "Functional Components" are then connected by means of "Inputs" and "Outputs" that induce a dependency network between them. These dependencies can be considered in an aggregated fashion and inherited among the "Groups of Components"."Inputs" and "Outputs" are intended in the digital and physical sense allowing to describe the different types of exchanges between the " Functional Components". In the description of the architecture each component lists a series of input/output among the components described in a conceptual manner. The inputs to a component can refer to a specific output of given component, to all the outputs of a given component or all the outputs of a component group. Figure 3 exhibits an example diagram as output of the high-level module, where the diagram associated to an "Environment Modeling" group of components for an assistant robot operating in human populated environments. This group refers to the environment understanding based on the sensing information stemming from visual sensors.

V. Low-Level Module: Implementation details

The implementation details module ArchGenTool provides explicit details for each one of the "Functional Components". Complementary to the high-level module of the architecture, this module studies the interactions among software elements, their frequency of execution and the specifications of their communication. The implementation level describes the components that realize the architecture as expressed in the chosen robotic software middleware. ArchGenTool supports the connection between the high-level functional architecture module and the low-level implementation module for keeping track of requirements, description and adherence to the high-level plan. Therefore, it is apparent that the low-level module of the architecture design should be expressed in a framework similar to the implementation framework under which the development of the core functional elements will be realized. In order to achieve this, a mapping procedure that facilitates the dual correspondence of "Functional Components" and data types exchanged between each other is conducted. In the current implementation of the ArchGenTool the chosen middleware is ROS because it presents a large ecosystem of packages and features, and ease of testing how the designed architecture is mapped into a real computational graph.

A. Mapping Architecture to ROS

The ROS mapping routine creates a component graph consistent with the graph of the high-level module. This graph retains the connection between the components of the two levels. In the ROS framework, the computation unit is the ROS node. Each high-level functional component is mapped to one or more ROS nodes that, when seen as a whole, expose the same interface of the high level module of the architecture.

The ArchGenTool uses a ros-type-mapper, which is a third party YAML file designed to describe the mapping between the high-level types to the ROS standard types, each provided with the name of the message type (e.g. sensor_msgs/Image) and a comment about characteristics of the message (e.g. the camera resolution). This is essential due to the fact that it allows the developers to identify, from the architecture

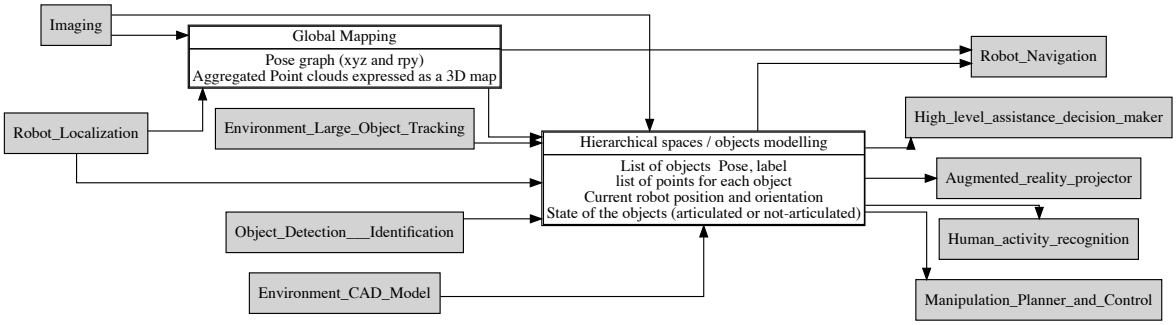

Fig. 3: Example of component diagram derived from the high-level module concerning a specific Component Group that describes robot mapping and environmental modeling. The components of the Group are shown in white color, while the external components that may belong to other groups are illustrated in gray color.

design phase of the system, the complementary to the basic software components, resulting to an orthogonal system construction. This way intermediate parsers and plugins are avoided retaining thus the modularity and reusability of the developed software.

ROS type communication messages for some of the components identified in the high-level module will be already available as standard ROS message types, e.g. sensor_msgs/Image for a high-level type "Time stamped synchronized RGB and Depth images", while others will be created with details filled-in at later stages of development. An additional attribute of the ArchGenTool is that the comments specified in the ros-type-mapper allow to track the progress and the mapping procedure during the architecture design process.

The ArchGenTool creates the implementation component graph from the high level components and then performs a validation of the provided architecture. Specifically, it verifies that all components have been mapped. Each high-level functional component is visualized in a separate graph where all the implemented ROS nodes are shown with their description. Additionally, the ROS nodes that provide or consume data are also exhibited with their name. Edges describe the connection with the data exchanged provided as edge label, with an asterisk when all the topics produced by the input node are exchanged with the current processing node. Figure 4 illustrates an example of an implementation graph. Currently all the communication between components is assumed to be performed via standard ROS semantics, but additional semantics could be added via attributes of the edges connecting the components.

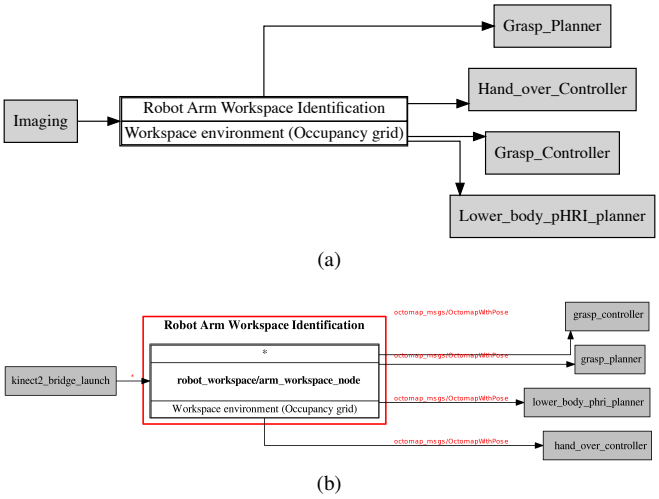

Fig. 4: High-level graph (a) and implementation graph (b) containing a single component. The box in the center is the ROS node implementing the component with the name in the middle. Incoming and outgoing ROS components are shown on left and right of (b). Edges show the exchanged data.

B. Timing Specification

One important aspect of the architecture design is the definition and analysis of timing interaction between components. In the ROS framework, for example, some nodes have a periodic behavior in publishing topics, such as the case of sensor nodes or planners of arm joint values. Other nodes are instead responding to changes in the subscribed topics and produce output correspondingly. In general nodes are not specifying timings and they can be obtained in the running system from the topics statistics.

The ArchGenTool supports the specification and analysis of timing behavior associated to the implementation level. Each functional component can be classified in three behaviors: periodic, event-driven, on-demand. Periodic means that the node produces outputs or in general computes at a given rate, i.e. a node in ROS; event-driven means that output is produced when some of the inputs are changed, i.e. an action lib in ROS; and finally on-demand means that answers to requests, i.e. a service in ROS. The tool performs a graph annotation phase propagating the timing behavior and the update rate of components from source components to sink

components when the period is omitted. The annotated graph can be visually presented by coloring functional components depending on the resulting period. In this way the overall timing structure can be understood by the developer in a compact way. The support for computation times for a high-level plan of latency is demanded to future updates.

VI. Tool Implementation

ArchGenTool is implemented as a Python package with single command line script that allows to process the YAML specification, analyze the model and generate outputs depending on the chosen backend. In document-output mode the tool produces a Word document, in YAML-output mode the tool produces a single validated YAML document that expresses the whole architecture and finally in Web-output mode the tool generates a statically browsable representation. The backends share a graph generation phase based on Graphviz that allows to produce diagrams of the various levels of the architecture providing single detailed diagrams per functional component, per functional group or even of the overall architecture. The visual layout of the generated graphs is also controlled for readability and clarity.

The typical multiuser scenario is based on the storage of the YAML documents in a Git repository starting from one single YAML document and then extending and decomposing it while the number of components and functional groups grows. Syntax verification and architecture validation can be provided also as a Continuous Integration service connected to the git repository.

VII. RAMCIP CASE STUDY

The capacity of ArchGenTool for the architecture design of a robotic system is assessed on the "Robotic Assistant for MCI patients at home" (RAMCIP) project [4]. As a collaborative EU-funded research project RAMCIP involves expertises from several partners that contribute to the definition of the system architecture. RAMCIP is a complex robotic system aiming to establish a service robot, capable of proactively assisting older persons with Mild Cognitive Impairment or early dementia stages, in a wide range of daily activities, being at the same time an active promoter of the user's physical and metal health. To achieve these objectives the robot will be equipped with a mobile platform upon which an arm will be mounted with a dexterous hand attached to its end point. An elevation mechanism is foreseen to allow the robot to reach both higher and lower locations with the same robotic arm. Additionally, the robot will also bear display and projective augmented reality mechanisms facilitating human-robot communication. Through this short description it is concluded that the RAMCIP robot is a complex yet modular system the architecture design of which is a laborious task.

The overall architecture is described by 9 main YAML files, one per-group, plus supporting YAML files that provide details on the ROS type mapping and timing. Different decompositions are possible depending on the size of the teams and responsibilities.

[4]http://www.ramcip-project.eu

A. High-level

The high-level functional architecture has been designed starting from the definition of project Use-Cases and functional requirements. These elements brought to the identification of conceptual modules further decomposed in functional components. After the initial phase the architecture has been specified using ArchGenTool backed by GitHub for data storage and collaboration.

The resulting high-level architecture comprises 10 groups that have been decomposed in 61 functional components (6.1 in average per group with a maximum of 21). For completeness the groups are: Robot State, Environment State, Environment Model, Human State, Human Model, Cognitive Reasoning, Robot Task Scheduler, Robot Action Planner and Communication Planner. The diagram of the groups and their internal components is shown in Fig. 5. In this stage the ArchGenTool has been used for identifying all the connections printing out them and provide a big picture of the overall architecture: it is also useful for working with stub components that can be later filled-in. The components are interconnected with a total of 138 connections (avg. 2.2), with 2.3 inputs and 1.8 outputs in average: sources correspond to sensors or user inputs, and sinks to actuators and visual feedback modules.

B. Implementation

After the definition of the high-level specification each functional component has been analyzed for mapping to a ROS implementation understanding which components can be taken from the ROS library or needs to be further developed. This analysis can be performed by each group-leader behind the components, detailing along time the sub-components used for the realization. Mapping types are instead agreed and entered cooperatively in the shared mapping file. The content produced by the nodes is mapped from the high-level to the ROS message types, producing, in this example a mapping of 138 entities. Further refinements have been performed during the later stage of integration detailing and amending elements of the architecture.

VIII. DISCUSSION

The toolchain is based on a specific computational graph model that balances description complexity for easing design and at the same time supporting the translation to a middleware such as ROS. At this stage it is worth discussing the differences between the computational graph model of ArchGenTool and other systems such as ROS and the System Modeling Language (SysML) that is widely used for describing software and hardware systems [2]. The computational graph is a directed graph in which typed data flows between nodes along edges. Edges are not connecting nodes directly but by means of ports: every node exposes a set of input and output ports, and edges connect the ports of different nodes. The difference between the three approaches is in how these ports are specified and connected.

In SysML nodes (parts in the Internal Blocks Diagrams) expose ports, called flow ports, each described with a name

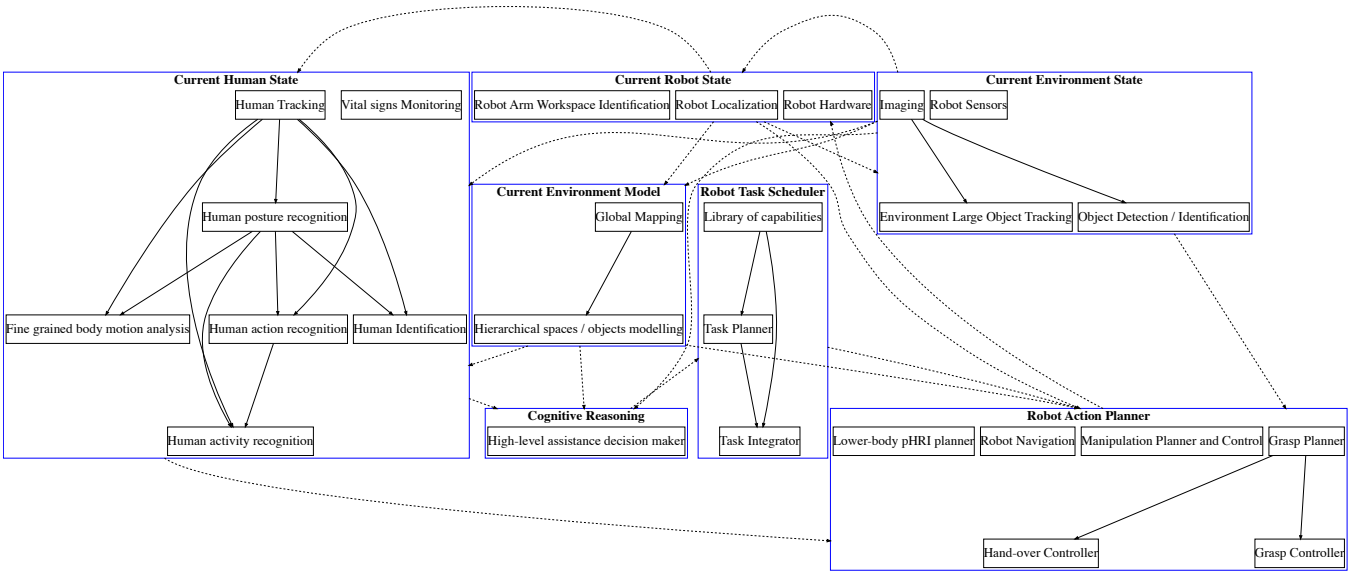

Fig. 5: Diagram of the RAMCIP component groups and their internal components, where the Human Model and Communication components have been removed for space motivations.

and type, and then connections specify the connecting nodes and ports, constrained to have the same type. In ROS every node has ports, that is published/subscribed topics, with name and type, then the connections link nodes by matching the name of the topic. Being it a dynamic and flexible publisher-subscriber system, topic names are matched without involving nodes. In ArchGenTool every node has ports with a single property, the type, and then the connections are specified expressing the input node and port. The high-to-low level mapping performed by ArchGenTool maps every high node into a set of ROS implementation nodes with fully specified connection to keep the high-low correspondence.
The implementation level describes only the software interconnections but, based on the same description methodology and tooling, it is possible to extend it to take into account the (1) the transformation graph of the architecture and (2) the physical partitioning of the architecture, that is the mapping to the different parts of the robot system such as the computer units involved.

IX. CONCLUSIONS

The approach proposed by ArchGenTool allows to experiment with different solutions while receiving feedbacks about architectural characteristics. The textual-based toolchain is aimed to reduce design time and tooling while the diagram generation provides the top-view understanding of the architecture. The tool is provided as an Open Source ROS package available at `https://github.com/eruffaldi/archgentool`.

There are clearly several possible improvements. Some are related to the expressiveness of the language, mainly for dealing with physical specifications and system partitioning, but also for resource sharing and implementation variants. Others are related to the import, much like libraries, of existing ROS packages and node. Finally others are related to the analysis performed over the architecture, such as improved timing, data exchange rates and adherence to requirements. This could allow the use of ArchGenTool with other component based frameworks with tighter requirements such as CoCo [9].

ACKNOWLEDGMENTS

This work has been supported by the EU Horizon 2020 funded project “Robotic Assistant for MCI Patients at home (RAMCIP)” under the grant agreement with no. 643433. The authors also acknowledge the effort and support of Ugo Cupcic and Toni Oliver from Shadow Robot that contributed to the development of ArchGenTool and the design of RAMCIP architecture.

REFERENCES

[1] H. Bruyninckx, “Open robot control software: the orocos project,” in *ICRA*, vol. 3. IEEE, 2001, pp. 2523–2528.
[2] S. Chhaniyara *et al.*, “Sysml based system engineering: A case study for space robotic systems,” in *62nd Int. Astronautical Congress*, 2011.
[3] P. C. Clements, “A survey of architecture description languages,” in *8th int. workshop on software specification and design*. IEEE, 1996.
[4] S. Dhouib *et al.*, “Robotml, a domain-specific language to design, simulate and deploy robotic applications,” in *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2012.
[5] L. Gherardi and D. Brugali, “Modeling and reusing robotic software architectures: the hyperflex toolchain,” in *ICRA*. IEEE, 2014.
[6] L. Gherardi and N. Hochgeschwender, “Model-based run-time variability resolution for robotic applications,” in *Software Engineering (ICSE)*, vol. 2. IEEE, 2015, pp. 829–830.
[7] M. Quigley *et al.*, “Ros: an open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.
[8] A. Ramaswamy *et al.*, “Architecture modeling and analysis language for designing robotic architectures,” in *Control Automation Robotics & Vision (ICARCV)*. IEEE, 2014, pp. 1911–1916.
[9] E. Ruffaldi and F. Brizzi, “Coco - a framework for multicore visuo-haptics in mixed reality,” in *SALENTO AVR, 3rd Int. Conference on AR, VR, Graphics*. Springer, 2016, pp. 339–357.
[10] R. Simmons *et al.*, “Architecture, the backbone of robotic systems,” in *ICRA*, vol. 1. IEEE, 2000, pp. 67–72.