A Software Product Line Approach for Configuring Cloud Robotics Applications

Luca Gherardi, Dominique Hunziker, and Gajamohan Mohanarajah
Institute for Dynamic Systems and Control
Swiss Federal Institute of Technology
Zurich, Switzerland
Email: lucagh at ethz.ch

***Abstract*—The computational requirements of the increasingly sophisticated algorithms used in today's robotics software applications have outpaced the onboard processors of the average robot. Furthermore, the development and configuration of these applications are difficult tasks that require expertise in diverse domains, including software engineering, control engineering, and computer vision. As a solution to these problems, this paper extends and integrates our previous works, which are based on two promising techniques: Cloud Robotics and Software Product Lines. Cloud Robotics provides a powerful and scalable environment to offload the computationally expensive algorithms resulting in low-cost processors and light-weight robots. Software Product Lines allow the end user to deploy and configure complex robotics applications without dealing with low-level problems such as configuring algorithms and designing architectures. This paper discusses the proposed method in depth, and demonstrates its advantages with a case study.**

***Index Terms*—Robotics, Cloud Computing, Software Product Lines**

I. Introduction

In recent years, robotics applications have evolved from monolithic programs running on a single workstation to component-based and distributed software systems running on a network of heterogeneous processors. Robotics researchers are designing robots with increasingly complex behavior, such as robots that cooperate to make pancakes [1] or that can fetch a drink from the fridge and deliver it to a human [2].

The software systems required to enable and control these behaviors require the integration of several software components that provide a wide spectrum of functionalities, such as kinematics and dynamics, control and estimation, path planning and obstacle avoidance, and environment perception and computer vision. These software are thus computationally expensive and require powerful processors that can add to the overall weight, volume, and cost of the robot.

The emerging trend in next-generation robotics toward using small, inexpensive processors (similar to those found on mobile phones) will further exacerbate the problem. Indeed, even though these processors provide satisfactory computational power for a small cost, the onboard execution of computationally expensive functionalities is prohibitive.

In this paper we address this problem of "doing more with less" by adopting and extending our previous work on Rapyuta [3], an open source robotics Platform-as-a-Service (PaaS) on top of which robotics developers can design robotics Software-as-a-Service (SaaS) applications. Rapyuta allows the outsourcing of some or all of a robot's onboard computational processes by providing secured, customizable computing environments in the cloud. Rapyuta's WebSocket-based communication server provides bidirectional, full duplex communications between the physical robots and their computing environments. The computing environments allow robots to easily access the RoboEarth [4] knowledge repository, enabling them to share knowledge about object models, environment maps, and action recipes, and to learn from each other's experiences.

Rapyuta can store a library of ready-to-use software components that cover the entire spectrum of robotics functionalities; it can also provide a powerful and scalable computational environment where these components can be composed and executed. Yet despite these advantages, designing and configuring robotics software architectures remains a complex and time-consuming task. Decisions such as what components to use, how to compose them, and where to deploy them (on the robot vs. in the cloud) require that the designer be competent in several domains at once (e.g., software engineering, control theory, computer vision, etc.); yet the typical end user has not mastered all these areas. Consequently, hiding low-level configuration details and allowing end users to simply select their desired requirements would make the deployment of SaaS applications easier.

To achieve this goal we have extended previous work on variability management in robotics Software Product Lines[1] (SPLs) [6], [7]. In particular we adapted our model-driven development process in order to provide support for robotics cloud-computing peculiarities. Following this approach we define a reference architecture, the variation points of which can be specialized for developing several SaaS robotics applications. We describe the SPL variability by means of three models. The first model defines the architectural variability of the reference architecture in terms of stable and variable elements (computing environments, components, connections, and properties). Stable elements are reused in all the applications while a selection of variable elements characterizes a specific application. The second model defines the functional variability by means of Feature Models [8]. Finally, the third

[1] A set of software-intensive systems that share a common set of features and are developed from a common set of core assets in a prescribed way [5].

978-1-4799-5063-8/14 $31.00 © 2014 IEEE
DOI 10.1109/CLOUD.2014.104

model describes how it is possible to deploy a specific application by resolving the variability in the reference architecture according to a selection of features or requirements.

One of the main differences between SaaS and classical SPL is the multitenancy factor. In SaaS the objective is to deploy a single instance of the application and dynamically customize it for all the users. However, in robotics it is necessary to shift the focus from the entire application to the set of components that provide a specific functionality. Sharing a unique instance of a set of components between all the clients is sometimes desirable. This is, for example, the case for a component that provides the functionality to read bar-codes placed on boxes that must be manipulated. In other cases, it may be preferable to share component instances only between a restricted group of users. For example, the component processing the data produced by a robot that is flying and taking pictures should be private (or shared only between a specific group of users). In order to support the sharing of components, we allow the developers to specify the sharing mode for each of them. Furthermore, by adopting the Extended Features Models [9] and enriching the features with an attribute that defines the sharing mode, we allow the end user to decide how the functionalities should be shared.

This paper focus on the development of SaaS robotics SPLs, which exploit the cloud for outsourcing computation and can be easily configured by the end users. The first contribution of the paper is an approach for integrating cloud computing and SPLs. The approach takes into account the peculiarities of robotics and provides guidelines for handling variability in robotics cloud applications, which are not based on web-services, but on a set of components that interact through different communication paradigms. The second contribution is a set of meta-models and tools, which are integrated with the HyperFlex toolchain [10], are openly available on GitHub [11], and support the design and configuration of cloud robotics SPLs based on Rapyuta.

The rest of the paper is organized as follows: Section II presents a set of related works. Section III provides an overview of the approach, which is then elaborated in Sec. IV and V. Section VI exemplifies the approach by means of a case study. Finally, Section VII draws relevant conclusions.

II. Related Work

This section discusses related work on the topics addressed in this paper. It also introduces ROS, the robotics software framework used for the development of Rapyuta.

A. Cloud Robotics

The idea of robots with remote brains can be traced back to the 90s ([12], [13]). During the past few years this idea has gained traction (mainly due to the availability of computational/cloud infrastructures) and several efforts to build a cloud computing framework for robotics have emerged. The DAvinCi Project [14] used ROS as the messaging framework to get data into a Hadoop [15] cluster, and showed the advantages of cloud computing by parallelizing a 2D localization and mapping algorithm. It used a single computing environment without process separation or security; all inter-process communications were managed by a single ROS master, which is less general than Rapyuta; does not have a robot specific data-base and is unfortunately not available for public. The ubiquitous network robot platform (UNR-PF) [16] focuses on using the cloud as a medium for establishing a network between robots, sensors, and mobile devices. The project also made a significant contribution to the standardization of data-structures and interfaces. Finally, rosbridge [17], an open source project, focused on the external communication between a robot and a single ROS environment in the cloud. However none of these projects addressed computation scalability and DAvinCi does not provide ubiquity in terms of connectivity. Section IV-A explains how Rapyuta addresses some of these remaining challenges.

B. Robotics Software Product Lines

Robotics literature includes few papers dealing with SPL (e.g. [18], [19], [20]). Some of these focus on the design of SPLs for mature and stable domains of industrial robotics (e.g. manipulators used in car manufacturing). Others present specific case studies in which the SPL approach has been applied to refactor existing mobile robot applications. Differently from our approach these works do not adopt three orthogonal models for representing the variability information, but mix everything in one or two models. More recently [21] proposed a domain specific language for pick and place applications. In contrast to our approach, this work and the one documented in [22] focus on source code generation and not on the configuration of the system architecture.

C. Variability Management in the Cloud

Many papers present approaches for customizing cloud computing environments. Some of them are based on SPLs, while others leverage different techniques (e.g. [23], [24]). As described in [25], SPLs can be applied at three different levels of cloud computing: Infrastructure-as-a-Service (IaaS), PaaS, and SaaS. SPLs for the first two levels are presented in [26]. This paper focuses only on the third level.

In [27] the authors present a general vision of how applications can be configured using an SPL approach. Another vision of how SPL can be applied to cloud applications is reported in [28]. Differently from our work, in both cases the implementation is left to future works. In [29] three orthogonal models are used to describe the variability; however there is a one-to-one mapping between features and architectural elements. Conversely, in our approach a feature can be mapped to several elements and/or transformations that have to be applied on the architecture template.

Most of the mentioned papers focus only on the variability related to the control-flow, without considering the data-flow and the message-flow. These problems are addressed in [30], which adopts three orthogonal models to represent variability. However, differently from our approach, the configuration engine only allows the preservation and removal of services,

and other types of transformations are not possible (e.g. migrating components). A similar approach is presented in [31], where services are organized in behavioral units. Finally, [32] introduces a slightly different approach, which models the functional variability through the Orthogonal Variability Model (OVM). In contrast to our work, OVM does not allow the selection of more variants of the same variation point. Moreover the variability is addressed at the code level and not on the architecture level.

D. Discussion on Related Work in Cloud Computing

Beyond these considerations, the most important difference with respect to our work is that most of the papers discussed in this section deal with Service Oriented Architectures (SOA). The mismatches between SOA and robotics applications make it hard to use existing solutions for robotics. Such issues include programming languages, the number of threads, and the communication paradigms. In fact, robotics applications are multithreaded and require stateful protocols to exchange information asynchronously and synchronously with the robot. For example, a general PaaS platform such as Google App Engine is not well suited for robotics since it exposes only a limited subset of program APIs required for web applications, allows only a single process for a limited amount of time, and does not expose sockets, which are required for robotic solution. To the best of our knowledge this is the first paper that combines robotics, cloud computing, and SPLs.

E. ROS: The Robot Operating System

ROS [33] is the most widely spread component-based robotics-specific software framework. A ROS system is a computation graph made of a set of components, usually called *nodes*. Nodes interact by exchanging asynchronous messages (publish/subscribe paradigm) and/or by invoking synchronous services (client/server paradigm). ROS messages are organized by topics, which correspond to information subjects that allow subscribers to recognize the information they are interested in. When a node receives a message, an associated handler performs some computation on the message payload and possibly generates a new message to publish the computation results. ROS services are instead identified by a unique name. Clients do not have to know the service provider since the binding is performed at runtime according to the service name.

III. Approach Overview

Figure 1 depicts the overview of the approach proposed for designing robotics SaaS SPLs and deploying their applications. It conforms to the common approaches to variability modeling. The design-time part is accomplished by the robotics developers and the deployment-time part by the end users.

At design-time the development process starts with the *Reference Architecture Design*, which produces the *Template System Model* (TSM), and the *Functional Variability Modeling*, which produces the *Feature Model* (FM).

TSM specifies the set of components needed for building all the possible SaaS applications and the set of stable connections

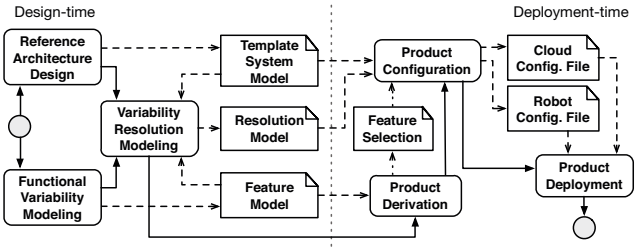

Fig. 1: Overview of the proposed approach: rounded rectangles depict activities, rectangles with a folded corner configuration files and models, continuous arrows activity flow, and dashed arrows the files and models flow (input/output).

between them. This model conforms to Rapyuta's component meta-model (Sec. IV) and organizes components in containers (robot or cloud execution environments). A TSM encodes four types of variation points: a) components can be optional, b) components can be migrated to a different container, c) new connections can be created, and d) component implementations and parameters can be configured. The family of applications that can be deployed by customizing the TSM share a set of common functionalities while differing in others. This functional variability is symbolically represented by means of the FM.

The TSM and the FM are orthogonal, meaning the models have no dependencies on each other. The *Variability Resolution Modeling* activity defines the *Resolution Model* (RM), which specifies how a selection of variants of the FM variation points influences the resolution of the TSM variation points. In other words the RM describes how the reference architecture must be configured according to the requirements specified by the end user. The RM conforms to the resolution meta-model introduced in Sec. V.

Once the three models have been defined the SaaS SPL can be configured. The configuration is realized at deployment time by means of the *Resolution Engine* (RE), which executes a set of model-to-model (M2M) and model-to-text (M2T) transformations. The RE receives a selection of features, which reflect the requirements of the desired application and is created by the end user during the *Product Derivation*. During the next activity, *Product Configuration*, the RE resolves the TSM architectural variability by applying a subset of the M2M transformations encoded in the RM, resulting in the architectural model of the desired application. Finally, this intermediate result is transformed into a set of configuration files by means of M2T transformations.

IV. Rapyuta: A Cloud Robotics Platform

This section introduces Rapyuta and presents the meta-model we have defined for modeling the TSM of applications distributed on the robot and in the cloud. We introduce the minimal abstraction of Rapyuta with regard to the relevance for this paper (more details can be found in [3]).

A. Overview

Rapyuta's primary purpose is to facilitate the outsourcing of computationally heavy application components from the robot to the cloud. Rapyuta uses a combination of ROS and Web Socket communication protocols to provide a standardized and secure messaging system and a component-based infrastructure. To separate the individual components shared by different groups and provide scalability these ROS nodes are executed in secured cloud computing environments, also known as *Containers*. Each container runs its own ROS environment, preventing direct access to the container internals from the outside. Similarly, the robot contains its own ROS environment that runs in the robot's computing environment. In order to facilitate the exchange of messages between different computing environments, Rapyuta allows the definition and connection of special interfaces. Inside each container an *Endpoint*, a special ROS node, part of and managed by Rapyuta, provides these aforementioned interfaces and acts as a proxy for all the connected interfaces in other computing environments.

Using different computing environments allows the introduction of a sharing mode for individual components and provides a simple mechanism for sharing select information with third parties. The sharing mode can be one of the following three values: a `private` component is only accessible by its owner; a `protected` component is shared between all robots of the same group; finally, a `public` component is available for every robot in Rapyuta. Note that the sharing mode is not yet available in Rapyuta, but is under development.

B. Component Meta-Model

Rapyuta's component meta-model defines the architectural elements needed for modeling the TSM and how they can be composed. On top of this meta-model we have implemented the *Architecture Editor* (see Fig. 4), which allows the developer to design distributed architectures. To keep the modeling task as simple as possible, the meta-model abstracts Rapyuta as much as possible. Note that, strictly speaking, the components running on the robot are not part of Rapyuta. However, in order to model the entire application and not just the cloud side, this meta-model also includes a *special* container that models the ROS environment running on the robot side.

Rapyuta's component meta-model is depicted in Fig. 2. A system contains multiple *Containers*, each one containing an endpoint. Containers encapsulate ROS nodes, the behavior of which can be modified by means of parameters. Nodes have a property that defines the sharing mode, the value of which is by default `private` for nodes running on the robot. To communicate between themselves, the nodes have associated interfaces that are both asynchronous (publisher/subscriber) and/or synchronous (service server/client). Finally, the endpoints have specialized implementations of these interfaces that allow nodes of different computing environments to communicate while using the endpoints as proxies.

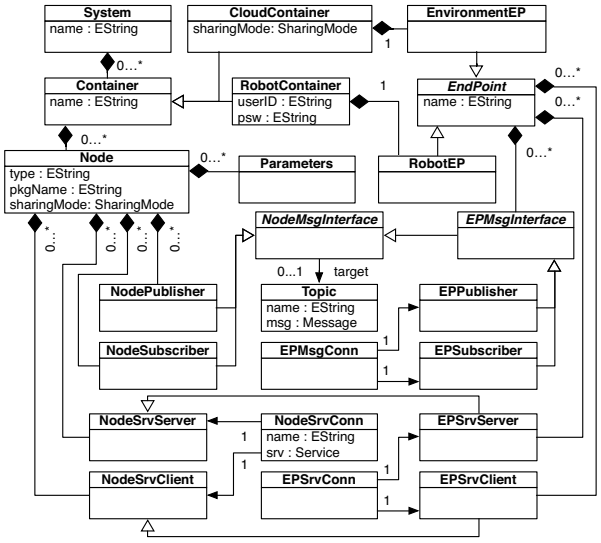

Fig. 2: Rapyuta's component meta-model

C. Deployment

In order to simplify the application deployment robotics software frameworks provide a tool that instantiates and configures the application components according to a configuration file defining the application architecture. For example ROS provides `roslaunch`, which is a tool able to deploy the application defined in a XML-based configuration file. Rapyuta provides a similar tool that uses a JSON-based configuration file for the application deployment.

To deploy an application distributed on the robot and Rapyuta, multiple configuration files are necessary. Each robot requires a Rapyuta configuration file for the declaration of the endpoint's interfaces and connections as well as all the other necessary configuration required on the cloud side of the application. Additionally, since Rapyuta cannot launch the nodes on the robot, a separate `roslaunch` file is needed to deploy the nodes in the robot's ROS computing environment.

The *Architecture Editor* implements a set of M2T transformations that transform the architecture model into the configuration files required for deploying the application. These transformations are the same as the ones used during the variability resolution phase (see Sec. V-C). In this way the tool is not coupled to the SPL approach, but can be also used for modeling architectures of independent applications.

V. Variability Modeling and Resolution

This section introduces the models and tools used for modeling and resolving the SaaS SPL variability.

A. Feature Meta-Model

The functional variability is modeled according to the Extended Feature Models formalism (see [8] and [9] for a detailed description). A feature model is a hierarchical composition of features and is organized as a tree. Features represent the application's functionalities and/or requirements and can be mandatory or optional. Parent features are connected to

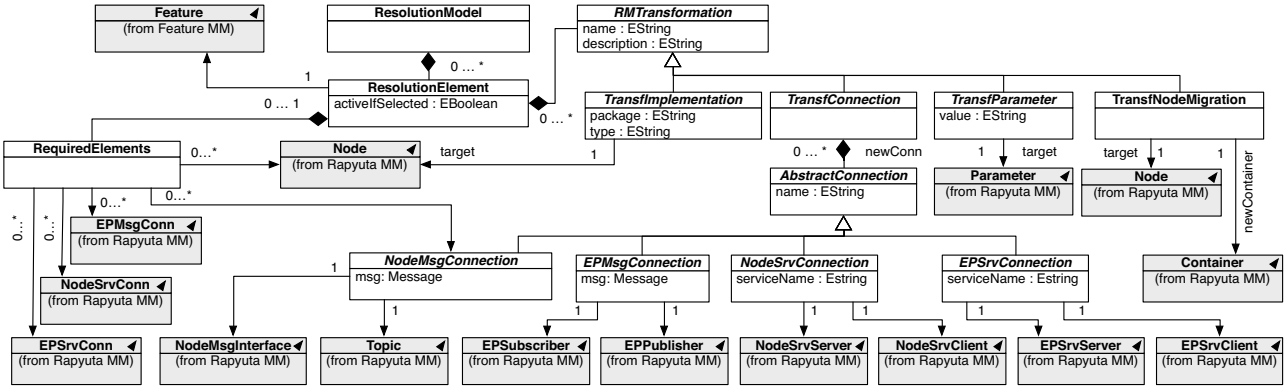

Fig. 3: The resolution meta-model: white classes belong to Rapyuta's resolution meta-model while gray classes with the arrow on the top-right corner belong to the feature meta-model and Rapyuta's component meta-model.

child features by means of edges that represent the containment relationships. Two containment types are available: specialization and aggregation. In the first case the parent feature is a variation point, while the children are the possible variants. In the second case the parent feature is made of the child features. The feature models also allow the definition of constraints between features, which express dependencies or incompatibilities between functionalities. Feature Models can be designed by means of the *Feature Editor* depicted in Fig. 5

The feature model describing the functional variability of the SaaS SPL conforms to the meta-model presented in [6]. We extended the meta-model by defining the class `SharingConfig`, which inherits from the entity `Attribute` and specifies the property `mode`. This attribute provides a mechanism for postponing at deployment-time (end user responsibility) the decision of the sharing mode of the functionality associated to the enriched feature. Similar to the component meta-model, the property `mode` can assume three values. A `public` functionality is shared with all the users of the SaaS SPL independently of the specific application. A `protected` functionality is shared between the users of the same group. A `private` functionality, as well as the information on which the functionality works, is not shared.

B. Resolution Meta-Model

The *Resolution Model* conforms to the meta-model depicted in Fig. 3. It maps functional variability to architectural variability in terms of M2M transformations, which are executed at deployment-time to resolve the variability of the TSM and to generate the deployment files. The design of the RM is supported by a software tool called *Resolution Editor*.

A Resolution Model is a collection of *Resolution Elements*, which associate a *Feature* with a set of *Required Elements* and a set of *Transformations*. *Resolution Elements* define the boolean attribute `activeIfSelected` that specifies when the resolution element is active: if `true`, the resolution is active when the associated feature is selected; if `false`, the element is active when the associated feature is not selected.

Required Elements are architectural elements of the TSM (e.g. components) that must be present in the architecture of the desired application when the resolution element is active. *Transformations* are actions to be performed on the architectural elements defined in the TSM and are grouped into four types. The *Implementation* transformation specifies the implementation that must be used for a target component. A specific implementation is identified by a package name and a type. The *Connection* transformation specifies a set of connections that must be created for configuring the architecture. Four different connection types are available that encapsulate the information required for creating the connections. The *Parameter* transformation defines the value for a target component parameter. Finally, the *Node Migration* transformation defines the container in which a target component must be migrated.

C. Variability Resolution

At deployment time the end users of the robotics SaaS SPL adopt the *Feature Selector* tool for creating a selection of features reflecting the requirements of their application. The *Resolution Engine* (RE) receives this selection and performs the following actions on the TSM.

As a first step, a copy of the TSM is created. This copy is the model on which the RE operates. The RE iterates the active resolution elements defined in the RM (active resolution elements are detected according to the selected features) and executes the associated transformations for each of them. Additionally, a list with the required elements of the active resolution elements and the architectural elements on which the transformations operate is maintained. In particular, when the feature associated with the resolution element defines the attribute `SharingConfig`, the attribute value is used for setting the property `sharingMode` of all the nodes contained in the required elements of the resolution element. Once the iteration of the resolution elements is finished, the nodes and connections that are not contained in the list are removed from the TSM copy. The resulting model describes the architecture of a SaaS application, which belongs to the SPL and satisfies the requirements corresponding to the selected features. Finally, this model is transformed into the configuration files

required for the deployment of the application (ROS XML and Rapyuta JSON configuration files).

Although the transformations are self-explanatory and their execution changes the system architecture as described above, it is worth analyzing the *Node Migration* transformation further. This transformation is in charge of migrating a node from one container to another. While the migration itself is simple, the problem of maintaining the same topology of connections between the interested node and other nodes or topics is not trivial. In order to satisfy these constraints the interfaces of the original container endpoint and the interfaces of the new container endpoint must be modified to reflect the interfaces of the node. For instance, suppose that a node has a publisher P connected to a topic T and must be moved from the container C_i to the container C_j. In this case the transformation adds to the endpoint of C_i a publisher connected to the topic T, and to the endpoint of C_j a subscriber connected to a new topic T (created in C_j), which is then connected to the publisher P of the migrated node. Finally, the publisher of the endpoint contained in C_i and the subscriber of the endpoint contained in C_j are connected.

VI. Case Study

This section provides a case study related to the design of a robotics SPL for cloud-based 3D mapping applications and shows how the end user can deploy one of its applications by specifying his or her requirements. 3D mapping is an important robotics task, which consists mainly of building a point cloud representation of the environment where each point consists of a 3D location and a color. Sensors on the mobile robot produce color (RGB) images and/or depth images that encode the distance of each pixel. The goal of 3D mapping is to merge the sensor data into a point cloud and estimate the pose of the moving camera.

In this scenario the main source of variability is the hardware, i.e., the end user can execute one of the SaaS applications belonging to the SPL with different robot configurations. We consider two such configurations. The first robot configuration consists of a differential drive mobile base, an RGB-D camera, and an onboard processor. The mobile base, an iRobot Create, allows the robot to move only along arcs. The RGB-D camera is a special camera that, in addition to a color image (RGB), also produces a depth image. Finally, the onboard processor allows the execution of computationally light tasks. The second robot configuration consists of an omnidirectional mobile base, a stereo camera, an onboard processor, and an additional laptop. The mobile base, a KUKA youBot, allows the robot to move in any possible direction without constraints. The stereo camera is a composition of two RGB cameras. Given the relative position between cameras, it is possible to reconstruct the 3D point cloud of the scene. The onboard processor allows the execution of computationally light tasks and the additional laptop allows the onboard execution of some computationally expensive algorithms.

The exploration strategy depends on the last variation point: the environment in which the robot operates. For example, an in-place rotation works well for a small room, whereas a straight line translation is preferred for a narrow corridor.

A. The Architectural Model

Figure 4 depicts a screenshot of the HyperFlex plugin [11] we developed for modeling Rapyuta architectures. The figure represents the TSM of the cloud-based 3D mapping SPL. In order to illustrate how the components interact all the connections have been drawn. However, connections between optional component interfaces are not stable and are not defined in the TSM of the mapping SPL.

The *Depth Camera* (optional) node reads data from the depth camera and outputs an RGB image and a depth image (a frame is the composition of the two). Similarly, the nodes *RGB Camera L* and *R* (optional) read data from the RGB cameras and output RGB images. The *Stereo Vision* node (optional) merges the two images and produces a single RGB image and a depth image, i.e., a frame. Given a frame produced by the *Depth Camera* or the *Stereo Vision*, the *Visual Odometry* estimates the pose of the camera. It outputs keyframes every time the camera's estimated pose passes a certain threshold relative to the previous keyframe. A keyframe mainly consists of a frame, the camera pose, and a frame ID.

The *Map Optimizer* receives the keyframes as input and optimizes the keyframe poses by solving a pose graph optimization problem. In a pose graph the nodes represent keyframes and edges represent the relative poses between keyframes. After each optimization step, the *Map Optimizer* updates the keyframe poses in the database and publishes a notification that the pose was updated. *Map Optimizer* has a parameter called `CameraInfo`, which stores the path of the file describing the camera properties. This information is required for rectifying the raw image data. The *Database* stores the optimized keyframes and provides services for adding keyframes and updating their poses, and for retrieving a map. Here a map is defined as a set of keyframes. The *Map Merger* node retrieves two maps from the *Database* and tries to find correspondences, i.e., tries to add a new edge for connecting the two pose graphs. If a match is found, all poses are moved to a common coordinate frame.

The notifications produced by the *Map Optimizer* triggers the *Robot Controller*. Based on the next target pose, the *Robot Controller* generates a corresponding trajectory, which is a set of time indexed poses that the robot must follow. The trajectory computation is influenced by the robot's mobile base type (Roomba vs youBot). The *Robot Controller* retrieves the next position from the *Strategy Planner*, which retrieves the map associated with the robot and, based on the map, decides the next best target position for the robot in order to improve the map. The *Trajectory Follower* receives the trajectory and sends twist (linear and angular velocity) commands to the *Robot Driver* to execute this trajectory.

Except for *Database*, *Map Merger* and *Strategy Planner*, which are shared between robots that collaborate for the mapping (`protected` mode), a new instance of the other components is launched for each robot (`private` mode).

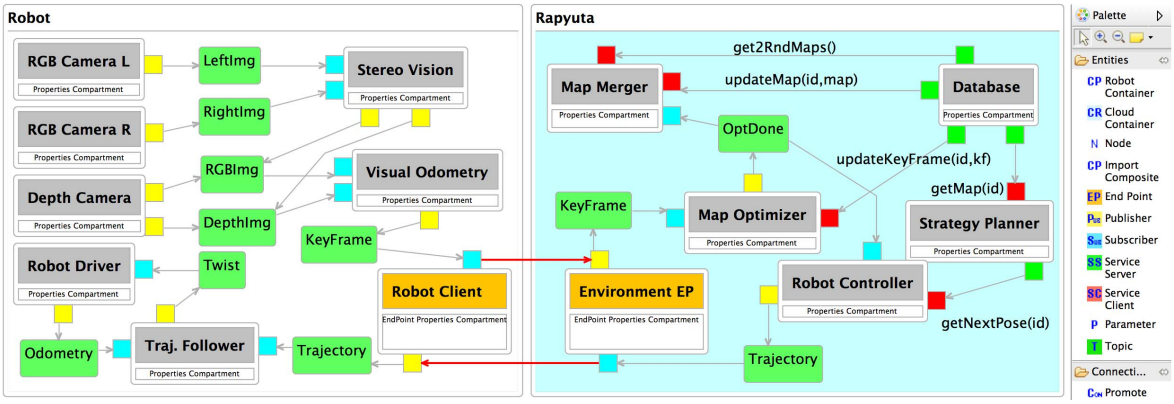

Fig. 4: TSM of the case study: grey rectangles are used to depict nodes, orange rectangles for Rapyuta's endpoints, green rectangles for topics, small yellow squares for topic publishers, small cyan squares for topic subscribers, small green squares for service servers, and small red squares for service clients. The direction of the arrows defines the flow of the data from publishers to topics and from topics to subscribers. Additionally arrows with labels connect service servers to service clients.

B. The Case Study Variability

The FM depicted in Fig. 5 describes the variability in the requirements of the mapping SPL. It has been organized according to the requirements variability discussed above. Three variation points are related to the robot hardware: one for the *Mobile Base* (*youBot* XOR *Roomba*), one for the *Perception Sensor* (*Stereo* XOR *Depth* cameras), and one for the presence of an *Additional PC*, which allows the deployment of the *Map Optimizer* component on the robot. The environment in which the robot operates can be a *Room* or a *Corridor*. This last variation point influences the navigation strategy, that is the implementation of the *Strategy Planner* component.

Note that the feature names do not directly represent functionalities. Indeed we want to hide the low level details to end users and allow them to deploy their 3D mapping applications by simply specifying the configuration of their robot and environment. On the other hand, there is a clear mapping between requirements and required functionalities.

C. The Resolution Model

The RM describes how the variability of SPL architecture can be resolved according to a selection of features. Some of the resolution elements defined in the Resolution Model of the 3D mapping SPL are described below. Due to limited space

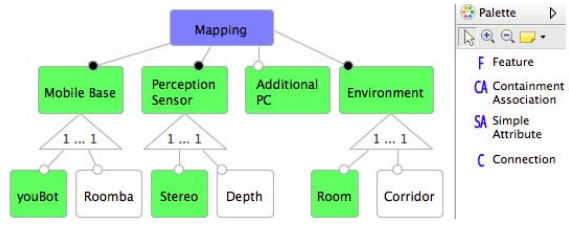

Fig. 5: FM of the case study: selected features are in green. White circles mark optional features, black circles mandatory ones. Triangles depict the containment selection cardinality.

we report a single resolution element for each variation point (the element name reflects the associated feature).

The resolution element *youBot* defines an *Implementation* transformation, which sets the implementation of *Robot Controller*, *Trajectory Follower*, and *Robot Driver* in order to take into account the kinematic model of the youBot. The resolution element *Stereo* defines by means of a *Connection* transformation how the *Stereo Vision* component must be connected to the topics *RGBImg* and *DepthImg*. A *Parameter* transformation defines the value that must be used for the parameter `CameraInfo` in the *Map Optimizer* component. Finally, the *RGB Camera L* and *R* are declared as required components and the connections between their interfaces, the topics *LeftImg* and *RightImg*, and the interfaces of *Stereo Vision* are declared as required connections. The resolution element *Additional PC* defines a *Node Migration* transformation, which moves the *Map Optimizer* component to the robot container. The resolution element *Room* defines an *Implementation* transformation, which sets the implementation of the *Strategy Planner* component for exploring a room. Finally, a resolution element associated to the root feature (*Mapping*), which is always selected, defines as required elements the components *Database*, *Map Merger*, and *Visual Odometry* as well as all the stable connections. Note that the other stable components need not be inserted in the required element list, because the transformations operate on them. This ensures that they will be preserved during the variability resolution phase.

D. Variability Resolution

The FM depicted in Fig. 5 highlights a selection of features that activate the resolution elements described above. The Resolution Engine receives this selection as input and produces as output the configuration files for the application deployment. The architecture configured according to this selection does not contain the *Depth Camera* component, which is required only when the *Depth* feature is selected. Since an additional PC is present, the *Map Optimizer* is moved to the robot

container and the interfaces of the endpoints are modified. For example the robot endpoint provides a new publisher, a new subscriber, and a new service client (these interfaces reflect the *Map Optimizer* interfaces).

VII. Conclusions

This paper proposed a new approach for designing Robotics SaaS SPLs based on Rapyuta, the robotics open source PaaS framework. It allows robotics developers to relieve end users from the low-level decisions required for configuring the architecture of complex systems distributed on the robot and the cloud. The approach encodes the SPL variability by means of three models. The first two represent orthogonal aspects: the reference architecture (TSM) and the functional variability (FM). The third (RM) is the glue between them and specifies how the variability can be resolved. Our approach is supported by open source meta-models and tools [11], which help the developers during the design of the aforementioned models. The tools also support the end user during the requirements specification phase, and automatically generates the configuration files for the desired application. Finally, we described a case study where this new approach was applied to build a robotics 3D mapping SPL.

In the future we plan to investigate how to handle constraints originating from scenarios in which several robots share the same instances of cloud components. In such a case, when the application of the first robot is deployed, the shared components are configured according to the selection of features. This configuration can impose constraints on the features selectable by additional robots. We plan to model and enforce these constraints, in order to avoid the generation of incompatible configurations. Furthermore, we plan to investigate how this approach can be integrated with the RoboEarth system [4], in order to automatically generate the selection of features according to the robot capabilities.

References

[1] 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*, 2011.

[2] 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.

[3] G. Mohanarajah, D. Hunziker, M. Waibel, and R. D'Andrea, "Rapyuta: A cloud robotics platform," *IEEE Transactions on Automation Science and Engineering*, February 2014, accepted.

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

[5] "Software Engineering Institute. Software product lines overview," http://www.sei.cmu.edu/productlines, 2013.

[6] L. Gherardi, "Variability Modeling and Resolution in Component-based Robotics Systems," Ph.D. dissertation, Università degli Studi di Bergamo, 2013.

[7] D. Brugali, L. Gherardi, 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)*, Tsukuba, Japan, November 2012.

[8] K. Kang, "Feature-oriented Domain Analysis (FODA) Feasibility Study," DTIC Document, Tech. Rep., 1990.

[9] K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker, "Generative Programming for Embedded Software: An Industrial Experience Report," in *Generative Programming and Component Engineering*, 2002.

[10] L. Gherardi and D. Brugali, "Modeling and Reusing Robotic Software Architectures: the HyperFlex toolchain," in *International Conference on Robotics and Automation (ICRA)*. IEEE, May 2014.

[11] "The HyperFlex Toolchain," https://github.com/Robotics-UniBG/HyperFlex, 2013.

[12] K. Goldberg and R. Siegwart, Eds., *Beyond Webcams: an Introduction to Online Robots*. Cambridge, MA, USA: MIT Press, 2002.

[13] M. Inaba, S. Kagami, F. Kanehiro, Y. Hoshino, and H. Inoue, "A Platform for Robotics Research Based on the Remote-Brained Robot Approach," *I. J. Robotic Res.*, 2000.

[14] R. Arumugam, V. R. Enti, K. Baskaran, and A. S. Kumar, "DAvinCi: A Cloud Computing Framework for Service Robots," in *International Conference on Robotics and Automation*. IEEE, May 2010.

[15] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010.

[16] K. Kamei, S. Nishio, N. Hagita, and M. Sato, "Cloud Networked Robotics," *Network, IEEE*, May-June 2012.

[17] G. T. Jay, "Rosbridge." [Online]. Available: http://www.rosbridge.org/

[18] J. M. Thompson and M. P. Heimdahl, "Structuring Product Family Requirements for n-dimensional and Hierarchical Product Lines," *Requirements Engineering*, 2003.

[19] R. Weiss, J. Doppelhamer, and H. Koziolek, "Bottom-Up Software Product Line Design: A Case Study Emphasizing the Need for Stakeholder Commitment," in *SEI Architecture Technology User Network Conference (SATURN'09)*, 2009.

[20] K. C. Kang, M. Kim, J. Lee, and B. Kim, "Feature-oriented Re-engineering of Legacy Systems into Product Line Assets: a Case Study," in *International Software Product Lines Conference (SPLC)*, 2005.

[21] T. Buchmann, J. Baumgartl, D. Henrich, and B. Westfechtel, "Towards A Domain-specific Language For Pick-And-Place Applications," in *International Workshop on Domain-Specific Languages and models for Robotic systems (DSLRob)*, 2013.

[22] E. Jung, C. Kapoor, and D. Batory, "Automatic Code Generation for Actuator Interfacing from a Declarative Specification," in *International Conference on Intelligent Robots and Systems (IROS)*, 2005.

[23] X. Etchevers, T. Coupaye, F. Boyer, and N. de Palma, "Self-Configuration of Distributed Applications in the Cloud," in *International Conference on Cloud Computing*, 2011.

[24] Y. Huang, Z. Feng, K. He, and Y. Huang, "Ontology-Based Configuration for Service-Based Business Process Model," in *2013 IEEE International Conference on Services Computing*, 2013.

[25] K. Schmid and A. Rummler, "Cloud-based software product lines," in *International Software Product Line Conference (SPLC)*, 2012.

[26] E. Cavalcante, A. Almeida, T. Batista, N. Cacho, F. Lopes, F. C. Delicato, T. Sena, and P. F. Pires, "Exploiting Software Product Lines to Develop Cloud Computing Applications," in *International Software Product Line Conference (SPLC)*, 2012.

[27] S. T. Ruehl and U. Andelfinger, "Applying Software Product Lines to Create Customizable Software-as-a-Service Applications," in *International Software Product Line Conference (SPLC)*, 2011.

[28] J. Schroeter, P. Mucha, M. Muth, K. Jugel, and M. Lochau, "Dynamic Configuration Management of Cloud-based Applications," in *International Software Product Line Conference (SPLC)*, 2012.

[29] G. H. Alferez and V. Pelechano, "Systematic Reuse of Web Services through Software Product Line Engineering," *European Conference on Web Services*, Sep. 2011.

[30] T. Nguyen, A. Colman, and J. Han, "Modeling and Managing Variability in Process-based Service Compositions," *Service-Oriented Computing*, 2011.

[31] I. Kumara, J. Han, A. Colman, T. Nguyen, and M. Kapuruge, "Sharing with a Difference: Realizing Service-Based SaaS Applications with Runtime Sharing and Variation in Dynamic Software Product Lines," *International Conference on Services Computing*, 2013.

[32] R. Mietzner, "A Method and Implementation to Define and Provision Variable Composite Applications, and its Usage in Cloud Computing," 2010.

[33] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an Open-source Robot Operating System," in *ICRA workshop on open source software*, 2009.