A reference architecture for deploying component-based robot software and comparison with existing tools

Nico Hochgeschwender, Geoffrey Biggs and Holger Voos

***Abstract*—This article discusses the problem of deploying component-based software for a robotic system, including both the initial deployment and re-deployment at run-time to account for changing requirements and conditions. We begin by evaluating a set of tools used for all or part of the deployment activity. The evaluated tools are the OMG DEPL specification, Chef, Ansible, Salt, Puppet, roslaunch and Orocos Deployer/ROCK. These tools were chosen to cover a range of capabilities and styles. The evaluation identifies a set of core roles found in the deployment activity, and based on this we propose a reference architecture for a set of tools that satisfy the deployment activity. This reference architecture provides a foundation for future work in developing and evaluating tools that can be used in deployment.**

I. Introduction

Deploying software on real, heterogeneous robot systems is a challenging and nebulously-defined exercise. It encompasses activities beginning as early as releasing software and ending as late as altering and changing software during run-time to meet changing and emerging requirements [1] [2]. What "deployment" means can even vary according to the software system being deployed, the tools being used, and the development process being followed.

For the purpose of this paper, software deployment is defined as an activity with the goal to make an application ready to use. This encompasses "*tak[ing] the requirements of the software to be deployed, along with the resources of the target environment on which the software will be executed, and decid[ing] how and where the software will be running in that environment*". We base this definition on the deployment and configuration specification introduced by the Object Management Group (OMG) [3].

To employ this definition in the context of robotics, it is worth emphasizing that robots are expected to perform many different tasks over a long period of time. To do so, they need to cope both with varying requirements for the software and with changing resources. Both requirements and resources are induced by changing tasks, goals, and environmental features. Thus, deployment in robotics is a frequent activity performed throughout the complete life-cycle of robot software systems, namely from design-time to run-time. To this end, reliable deployment tools covering the full range of the deployment task and ensuring the availability of robotic applications are required. Those tools should be available not only for robot software developer, but also for the robots themselves as we aim to reduce human intervention in order to deploy increasingly unsupervised robots in increasing long-running applications.

In this work we select and evaluate some existing tools used for all or part of the deployment task. These tools are taken from the robot domain and from other domains where deployment is a mature task. The starting point of the evaluation is the observation that the tools available for software deployment are not sufficient to deal with many use cases appearing in robot software deployment. In particular, the tools tend to be inflexible and do not cover use cases involving responding at run-time to changes in requirements and resources. The evaluation identifies the core roles found in each assessed tool. Based on the evaluation we generalize, propose and describe a reference architecture for tools used in the task of deploying component-based robot software, according to the definition of deployment given above. Further, we give some insight into the suitability of some tools to solve the deployment task as defined by the reference architecture.

II. Analysis of existing deployment tools

To identify the frequently-occurring roles in the deployment process, we analyse a range of well-known tools that are used for deployment. We include both tools used in robotics and, for a broader perspective, tools from outside the robotics domain. Most of these are designed for static deployment and are more commonly known as configuration management tools. Popular examples include Chef and Ansible which are mature tools used by system administrators on a daily basis. By contrast, tools that can handle automatic dynamic deployment are mostly found in the robotics domain, and even so there is only one example, the ROCK tool.

The tools covered in this evaluation have been selected based on popularity in their fields and to give a good coverage of the style of tools available. Space restrictions and the number of tools in existence prevent this from being an exhaustive list of every tool with relevance to the extensive deployment task.

The purpose of the analysis is to identify commonly-occurring roles in the deployment process. These roles may be performed by a human or by some part of the tool, whether

Geoffrey Biggs is with the Robot Innovation Research Centre, AIST, Japan geoffrey.biggs@aist.go.jp Nico Hochgeschwender and Holger Voos are with the Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg nico.hochgeschwender@uni.lu.

an independent component (such as a daemon) or as part of a monolithic entity. We have therefore tried to re-use role names across tools to identify commonalities.

A. OMG "Deployment and Configuration of Component-based Distributed Applications" specification

The Object Management Group (OMG) provides a specification for the deployment of component-based software applications. This specification is titled "Deployment and Configuration of Component-based Distributed Applications" (DEPL) [3]. It is primarily intended to be used with component-based software designed and implemented using the OMG's other technologies, such as CORBA and IDL.

We begin with this specification as it in some ways can act as a base reference model for the static deployment of an application into a distributed computing environment, which is a use case that fully encompasses deploying into an environment consisting of single computing node. It identifies some common role names for use in the analysis of other tools. The architecture of DEPL is shown in Figure 1.

The DEPL's architecture is rooted in two overarching requirements.

- DEPL is used for static application deployment. Although there are hints in the specification of monitoring node resource usage, there is nothing specified in the way of responding to changing conditions and re-deployment at run-time. However, this is also not explicitly forbidden, leaving an implementor of the specification open to providing such a feature if they choose to do so.
- DEPL is, like the CORBA applications it is intended for, designed to be used for distributed system deployment. This means it has an extensive data model and facilities in its architecture for working with multiple computing nodes.

These two requirements mean that DEPL has no explicit role for monitoring the state of the application or system after deployment in order to respond to changes by altering the deployed application. The "NodeManager" entity may fulfill this requirement, but there is nothing in the specification on how this may be achieved nor on how the entities responsible for deployment should behave. What context monitoring (in the form of resource monitoring) there is in DEPL is intended to be used when the application is first deployed, as part of determining which nodes should host which components of the software.

The architecture's design also stems from a third requirement, that of allowing multiple vendors to provide different parts of the deployment infrastructure. The result is the somewhat awkward division of deployment execution responsibilities amongst the ExecutionManager, NodeManager, DomainApplicationManager and NodeApplicationManager entities. The stated goal is to allow the tool responsible for managing deployment to be separate from the tool responsible for performing deployment actions on each node.

In DEPL we can identify the following roles as being involved in some aspect of deployment.

- **System selection.** The selection of what to deploy is performed by a planning tool. This is not explicitly defined in the DEPL specification, but its presence is made clear.
- **System deployment.** The ExecutionManager and the DomainApplicationManager are responsible for managing the deployment of the application.
- **Deployment infrastructure.** The NodeManager and the NodeApplicationManager, an instance of which runs on each node in the domain (the distributed system into which the application is being deployed), are responsible for performing the actual deployment, starting the application's components.
- **State store.** The state of the domain, in terms of available resources, is stored and used during by the system selection role to decide on which node to place each component of the application.

B. Chef

Chef [4] is a style of tool known as a "configuration management" tool. Its intended purpose is to manage the software installed and running on a server. It is most commonly used in server management, particularly for services providing World Wide Web and Internet- or Intranet-based application content.

The architecture of Chef is shown in Figure 2. Like all configuration management tools, the most notable aspect is the reliance on a human administrator to perform the system selection role. Chef provides a tool, Chef Manage, to aid in this task but it is the human administrator who decides what to deploy and where to deploy it.

The system deployment role is spread across two entities in Chef. The central Chef Server, of which there is only one instance, instructs clients to deploy systems when it is itself instructed to do so by the human administrator. The majority of the system deployment role is performed by the Chef Client directly.

The Chef Client is also responsible for the deployment infrastructure role, performing the actual instantiation and initialisation of the part of the system being deployed on its node.

Chef includes an entity known as "Bookshelf" that fulfills a role we call the "catalogue". This provides access to the cookbooks (descriptions of deployable systems) as well as supporting data needed for deployment by Chef Client instances.

Chef also includes another role not found in the DEPL architecture. An entity known as "Ohai" performs a state monitoring role. One instance of this runs on each computing node. It gathers information about the state of the node and the (sub-)system deployed to that node. This information is used by the Chef Client entity while performing its system deployment and deployment infrastructure roles. However, none of this information is used by the system selection role.

Although much of the detailed state of each node and the deployed system is stored individually on each Chef Client instance and used solely by that instance, some state about

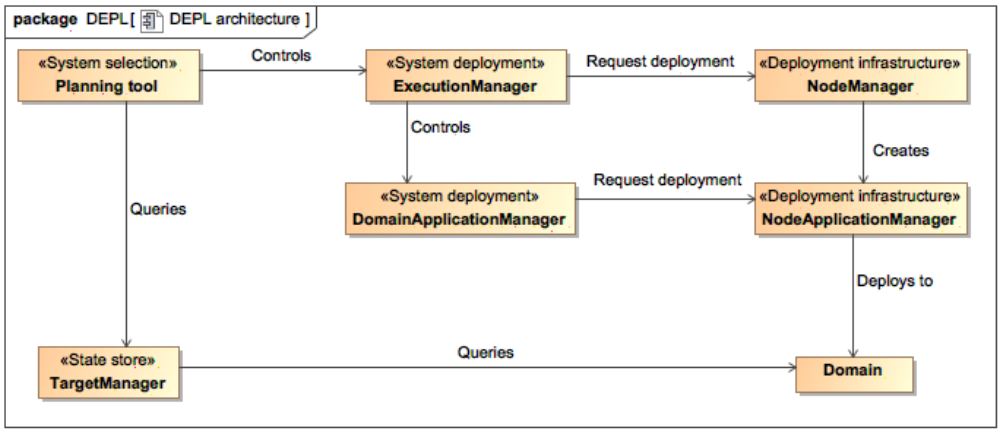

Fig. 1: The architecture of the OMG's Deployment and Configuration of Component-based Distributed Applications specification, specified in terms of the roles involved and the parts of the architecture that fulfills them

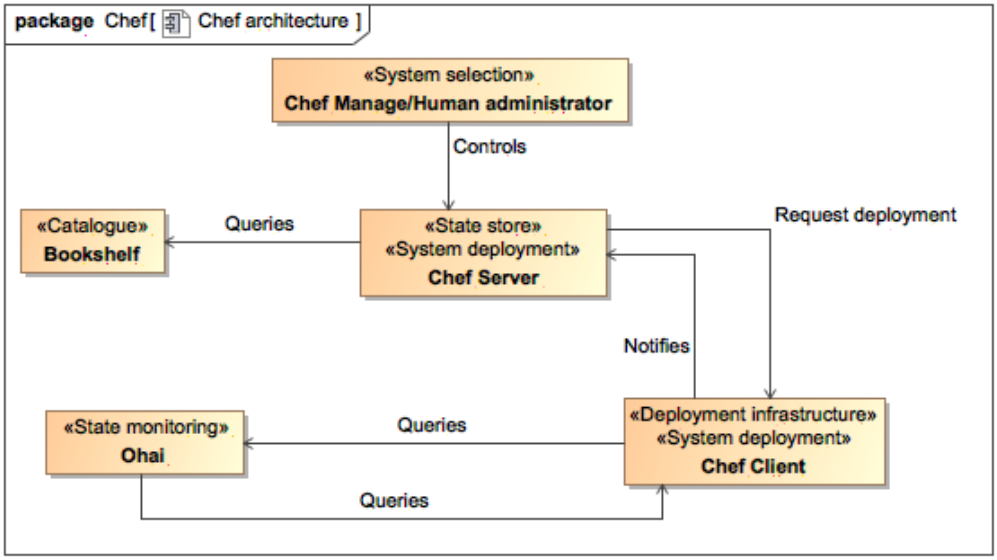

Fig. 2: The architecture of Chef, modelled using a UML-like component diagram

the deployment on each node is returned to the Chef Server. Thus the Chef Server partly fulfills the state store role.

Like DEPL, Chef is a tool for static deployment. Changes in deployment are primarily intended to be initiated by the human administrator fulfilling the system selection role. However, Chef does have some capability to dynamically deploy software to nodes. This is done via a semi-independent deployment infrastructure, known as the "push jobs client", and separate interface to the system selection tool (the Chef maintenance interface).

C. Ansible

Ansible [5] is a configuration management tool in the same vein as Chef. It is used to automate tasks on a collection of servers. These tasks can be as small as executing a single command (for example, to configure the host name of a server) to as broad as starting a suite of software in the correct order with pre-programmed responses to different conditions on the server or errors.

The architecture of Ansible is shown in Figure 3. The core architecture is very simple, consisting of a single entity, the Ansible Automation Engine. This architecture is very similar to the roslaunch architecture (described in Section II-F).

Like Chef, the system selection role selection role is played by a human administrator. Ansible itself in general is not capable of choosing when to deploy a system (known as a "playbook" in Ansible, and not necessarily a long-running application but also including one-shot tasks). It does, however, have the capability to chain system deployments together. This chaining can include conditionals, allowing a choice of the next playbook to deploy once the current playbook completes.

The administrator may perform their system selection role via the Ansible Tower web-based software interface. Ansi-

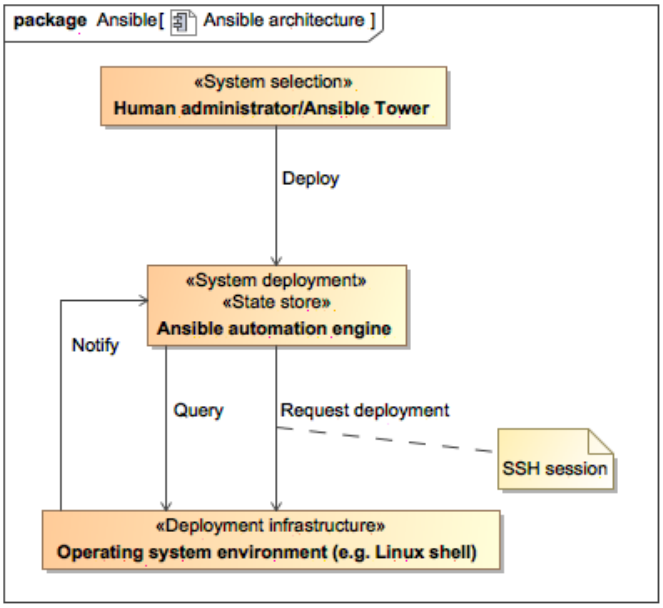

Fig. 3: The architecture of the Ansible configuration management tool

ble Tower provides various forms of support to the human administrator, such as graphically displaying the status of deployments collected from the deployment infrastructure role via real-time queries. It also provides a graphical interface for controlling the chaining of playbooks. However it ultimately is just an interface for the human administrator.

Notably, Ansible does not include a separate entity fulfilling the catalogue role. Although it can be told to use a directory of playbooks stored on disk, it is flexible about where playbooks come from. For the aspects of the state store role that relate to maintaining current system state, the automation engine queries servers as needed to get this information. A caching system is available, although it is not clear whether this can cache system information or just variables used in the playbooks for use in follow-up playbooks when chained.

The deployment infrastructure role is, similar to roslaunch, provided by the operating system environment on each computing node. Ansible uses an SSH session to access the deployment environment and executes commands much as a human administrator would. For cases where software to be executed is not currently present on the server, Ansible relies on its ability to execute generic commands to copy (using SCP) or install (using the deployment environment's package manager) software before execution[1].

D. Salt

Salt [6] describes itself as a tool for "event-driven orchestration", and fits into the same configuration management category as Chef and Ansible. It features a highly-modular architecture, with flexibility being a lead design goal. Salt

[1]There are modules for Ansible that abstract this capability to provide a more user-friendly interface.

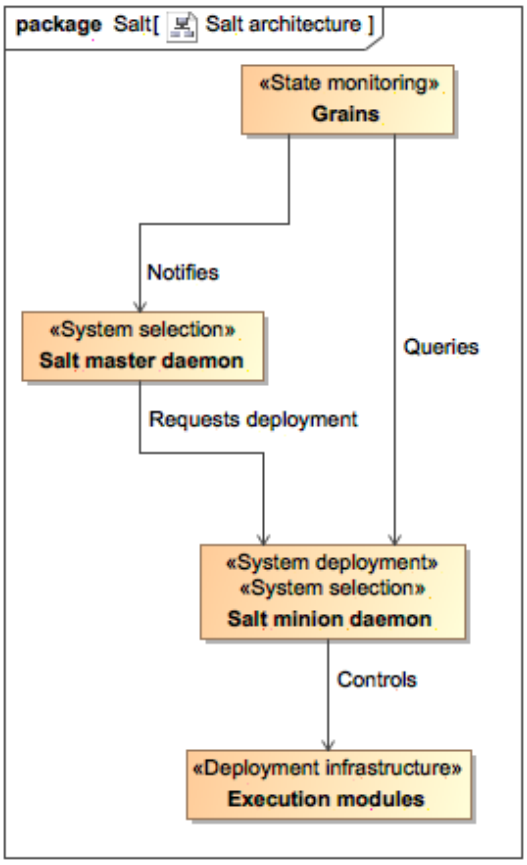

Fig. 4: The architecture of Salt

uses an agent-based (client/server) architecture, albeit a highly-distributed one. The architecture of Salt is shown in Figure 4.

In Salt, there is no entity fulfilling the state store role. Instead, Salt relies on gathering information at the time it is needed using instances of an entity called "Grains". These fulfill the state monitoring role. Salt claims that this ensures that information is always up-to-date when used, but the cost is increased latency and, for distributed systems, increased network traffic. However, these disadvantages are reduced by Salt's approach to sub-system selection.

Salt relies on distributed decision making for controlling system selection and deployment. The central controller does not completely fulfill the system selection role; it does not make decisions about what each computing node (called a "minion" in Salt's terminology) in the network should do. Instead, Salt sends the same configuration states to each node. Each state, known as a formula, provides actions to be executed to bring the computing node to the desired state with the necessary system deployments. They include decision points that are used by each node to determine which (sub-)system it should deploy. This reduces the load on the Salt master, but at the cost of making the configuration of every node known to every other node (a potential security risk in some environments).

Many of Salt's processes are event-driven. The computing

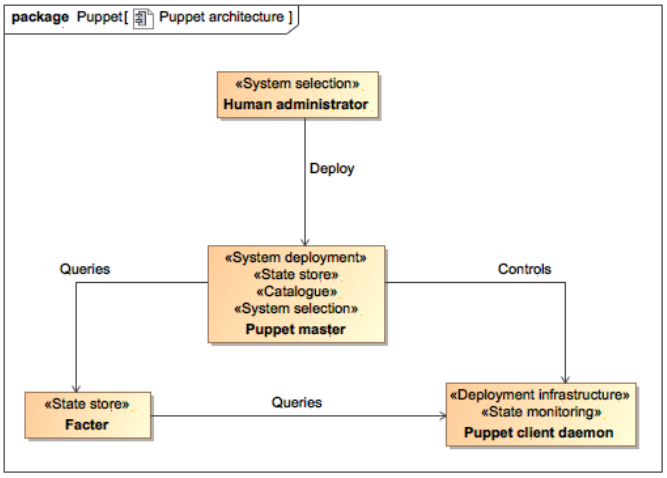

Fig. 5: The architecture of Puppet

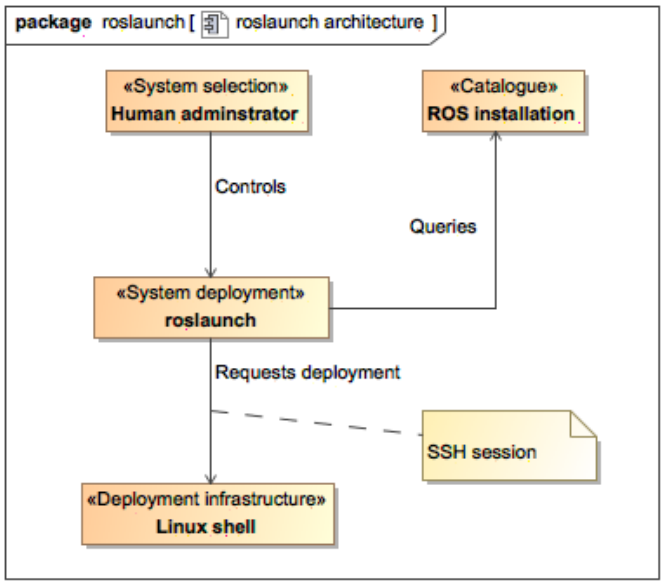

Fig. 6: The architecture of the roslaunch tool

nodes have a significant level of autonomy within the confines of the configurations provided by the Salt master. They will alter their current deployments automatically based on events received.

E. Puppet

Puppet [7] is a configuration management tool used to manage a large set of servers, like Chef and Ansible. It is used to automate the deployment of necessary software to a set of servers. Like Chef but unlike Ansible, Puppet uses a agent-based (server-client) architecture. This architecture is shown in Figure 5.

In Puppet, a deployment action is controlled by the Puppet master server and performed by a Puppet agent daemon instance on each node to which software is being deployed. The master server determines what should run on each node and provides that node with instructions specific to it. The Puppet master server is fulfilling the role of system deployment, while the Puppet agent fulfills the role of deployment infrastructure.

Puppet specifies deployment goals (the desired configuration of a computing node) in "catalogs". Each catalog specifies how a specific node should be set up, including which software should be deployed on it. These catalogs are generated on-demand by the Puppet master server from one or more manifests. One catalog is downloaded to a Puppet client when it must be deployed.

The manifests describe the necessary deployment of software and configuration across a set of computing nodes, which typically cooperate and therefore represent a single distributed system. The manifests themselves are stored by the Puppet master server. This means that the Puppet master server is fulfilling the catalogue role.

The system selection role is performed by a combination of a human administrator andthe Puppet master server. Conditionals specified in the manifests currently active are used during compilation of the catalog for each node. This allows Puppet to automate the configuration of many types of nodes. It does not directly provide dynamism to the deployment, although the use of periodic automatic deployment-change checks and redeployment can provide some measure of dynamism.

A small level of automated re-deployment is provided using a simple event system in the Puppet clients. By specifying a particular type of dependency between two deployed entities, it is possible to ensure that one is re-deployed automatically when the other re-deploys (usually due to a manual trigger). This gives the Puppet client daemon the state monitoring role.

F. roslaunch

`roslaunch` [8] is the application launching tool provided by ROS.

The architecture of `roslaunch`, which is relatively simple, is shown in Figure 6. It uses an "agent-less architecture". This term means that computing nodes where ROS components (which are known as "nodes" in ROS, but with a different meaning to the computing node in a network architecture) are deployed to do not need to run a persistent daemon specific to the deployment infrastructure. Instead, ROS relies on SSH to provide remote access to nodes, and therefore uses the SSH daemon for access. It does not rely on any daemon for node-local launching and management of components, using instead the standard Linux shell infrastructure to fulfill the deployment infrastructure role.

`roslaunch` is a relatively simple tool, designed only to start a deployment, maintain the deployment in a running state if part of it should terminate early, and shut down the deployed system when instructed to or when an error occurs. It does not feature any dynamic deployment capabilities, and its state monitoring features are limited to monitoring the system it launches for premature termination.

The role of system selection is performed entirely by a human administrator. `roslaunch` provides no facilities

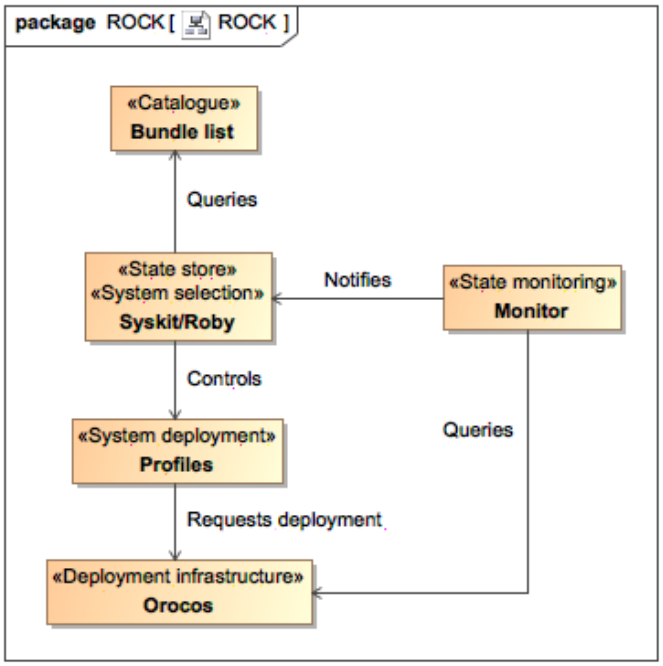

Fig. 7: The architecture of the Robot Construction Kit (ROCK) tool

for automating this. The tool itself is responsible for the system deployment role, controlling each computing node's deployment infrastructure to bring about the deployment of the chosen system.

G. Orocos Deployer/ROCK

The Orocos `deployer` [9] is a software component responsible for both loading and configuring components developed in the Orocos ecosystem. The `deployer` takes an XML file providing a description of the system to deploy. This contains instructions for importing, loading and configuring components (i.e. setting names and properties) and their connections to each other. From an architectural point of view the Orocos `deployer` is a relatively simple, centralized tool, yet the underlying deployment API exposed by the scripting methods of the `deployer` component is rich in terms of the possible configuration options. These include communication policies and component scheduling options. The architecture is nearly identical to that of `roslaunch`. As in `roslaunch`, the Orocos deployer fulfills the "system deployment" role. The "system selection" role is fulfilled by a human administrator and the XML description fulfills the "catalogue" role.

Another deployment approach in the Orocos ecosystem is the `Syskit` tool which can be found in the Robot Construction Kit (ROCK). ROCK is based on the Orocos RTT framework and its underlying component model. The `Syskit` tool aims to simplify the management and hence deployment of large component-based robot software systems. To this end, `Syskit` introduces the concept of "bundles" to group a set of components together, declaring dependencies to other bundles as necessary. These bundles fulfill the "catalogue" role. The bundles and corresponding components can be browsed by the `Syskit` tool to infer information such as component type and exposed data and service ports. To make the bundles executable they are refined by "profiles". Profiles fulfill the "system deployment" role. Profiles are executed by the `Roby` tool. `Roby` fulfills the "system selection" role. The information provided by profiles and bundles is also used to automatically compose different bundles together by matching port types and making connections. This automatic composition, which also resolves redundancies, is performed by the `Roby` plan management framework. This also makes it possible to declare policies on how to cope with errors and failures and, if necessary, which bundles need to be re-deployed.

III. Reference Architecture for a Deployment System

Having evaluated a variety of tools used for some or all of the deployment activity, we now propose a reference architecture for deploying component-based robot software. According to Taylor *et al.* [1] a reference architecture is defined as "*the set of principal design decisions that are simultaneously applicable to multiple related systems, typically within an application domain, with explicitly defined points of variation*". In this work we consider component-based robot software to be related systems within the domain of robotics. Reference architectures are considered to be *best practices* in other domains. Examples can be found in well-established domains, such as automotive [10] [11] and avionics [12].

Introducing a reference architecture for robot software systems is necessary. Deployment is a recurring yet often underestimated activity that is often prone to errors. Thus, a reference architecture provides not only a way to organize deployment, but also indicates how we can devise a template solution for recurring applications. It is worth noting that such a template solution is applicable and tailorable for many diverse applications.

The proposed reference architecture is depicted in Fig. 8 as a high-level role-based diagram. Here, each role forms an important part of the deployment process. These roles have been identified in Sec. II. The diagram assembles the roles, each of them providing and requiring activities, for example, notifying or querying other roles. Before each role is explained in detail, we explain two principal design decisions of the reference architecture.

- Firstly, the knowledge relevant for deployment, for example about software components and execution platforms, shall be explicitly stored and provided by one role, making deployment knowledge accessible for other roles. For example, the "State repository" role stores the current state of deployed components and is queried by several other roles, such as the "Sub-system selection" role.
- Secondly, deployment is performed in two steps. The first step deals with selecting a (sub-)system ("Sub-system selection" role), which is a set of components realizing certain functional features suitable for the task at hand.

The second step deals with deciding where, for example on which computing node, the selected components will be executed. Having such a step-wise approach enables developers to implement application-specific and domain-specific tools fulfilling the sub-system selection role without dealing with deployment concerns. These may, for example, be tailored for perception, manipulation or control.

A. Sub-system catalogue

The "Sub-system catalogue" role deals with the knowledge about what can be deployed and the requirements to do so. These requirements can include information such as the existence of specific hardware, and the properties of that available hardware (e.g. the amount of available memory). Further, the catalogue contains information about where software components can be deployed to and what the current state of those locations or computing resources is. [13] gives insights on the type of information required for deployment, which this role will typically include.

B. State repository

The "state repository" is a dynamic storage containing the current state of the deployed software and the deployment environment. This state information is constantly updated by the "Context monitoring" role. For example, in case of sudden errors or crashes of deployed components.

C. Context monitoring

One or more tools fulfilling the "Context monitoring" role are composed in the reference architecture in order to provide the contextual information needed to select (see III-D) and deploy sub-systems (see III-E). To this end, context monitors collect and interpret all the measurements required to infer the current state of the robots' environment, platform (including mechanical structure, sensors, actuators and computational elements) and intelligence features such as tasks, behaviors and skills. Context monitors make this information accessible to other roles by inserting it in the "State repository". Very often robotic systems already acquire those measurements and take important decisions based on their values. However, the acquisition of these measurements is usually hard- coded in the implementation of the software or hardware components that reason about them. Thus, introducing dedicated context monitoring will make components more reusable. In general, context monitors are application-specific and employ various context representations (e.g. logic-based vs. probabilistic-based approaches) [14] suitable for the task at hand.

D. Sub-system selection

The "sub-system selection" role is in charge of selecting one or more (sub-)systems suitable for the task at hand. Activation is either triggered – in a reactive manner – by changes in context information inserted in the "State repository", or by higher-level roles. In order to select the sub-systems suitable for the task to be performed, different methods and algorithms such as rule-based approaches or constraint solvers can be used. Depending on the employed selection algorithm, different types of queries may be applied to retrieve the information required to carry out selection.

E. Sub-system deployment

The "sub-system deployment" role is responsible for deploying one or more sub-systems and their corresponding components into the target deployment environment. To this end, the sub-system deployment role takes the requirements of the to-be-deployed software from the sub-system catalogue role performer and, combined with the state of the platforms, tries to find a suitable match.

F. Deployment infrastructure

The "Deployment infrastructure" role is responsible for bringing up and taking down each software component of the selected (sub-)system and making or breaking corresponding connections between those software components. The infrastructure is informed by the "Sub-system deployment" role which software components on which computing nodes should be started or stopped, respectively. In order to execute software components on certain platforms the infrastructure may require additional, execution-relevant knowledge. For example, how software components are mapped to executable primitives, such as processes and threads, or where the binaries of components are located, may need to be known in order to instantiate the components.

IV. Conclusion

In this article we have evaluated a variety of tools used for part or all of the task of deploying robot software. From the evaluation we have proposed a reference architecture that identifies the core roles that must be filled to satisfy the complete deployment activity. These roles are sub-system catalogue, state repository, sub-system selection, sub-system deployment, deployment infrastructure, and context monitoring.

While some of the identified roles may be filled by a human, and in several of the tools evaluated this is the case, the ideal is for the entire deployment activity to be automated. To achieve this, all roles must be filled by software tools that interact with the other roles to provide an integrated tool chain.

This reference architecture provides a foundation on which a set of interacting software tools can be built. Each tool must fulfill one or more of the identified roles. If such a set of tools is assembled, the task of deploying and re-deploying software on a robotic system both before and during run time will be achievable.

In future work we will implement and evaluate the reference architecture for robotic scenarios where re- deployment of component-based robotic software at run-time is desirable. To implement the reference architecture we favor a run-time environment that can be used by different robot systems. Having such a run-time environment would foster the development of reusable robot- and application-agnostic elements of the reference architecture, such as the sub-system selection.

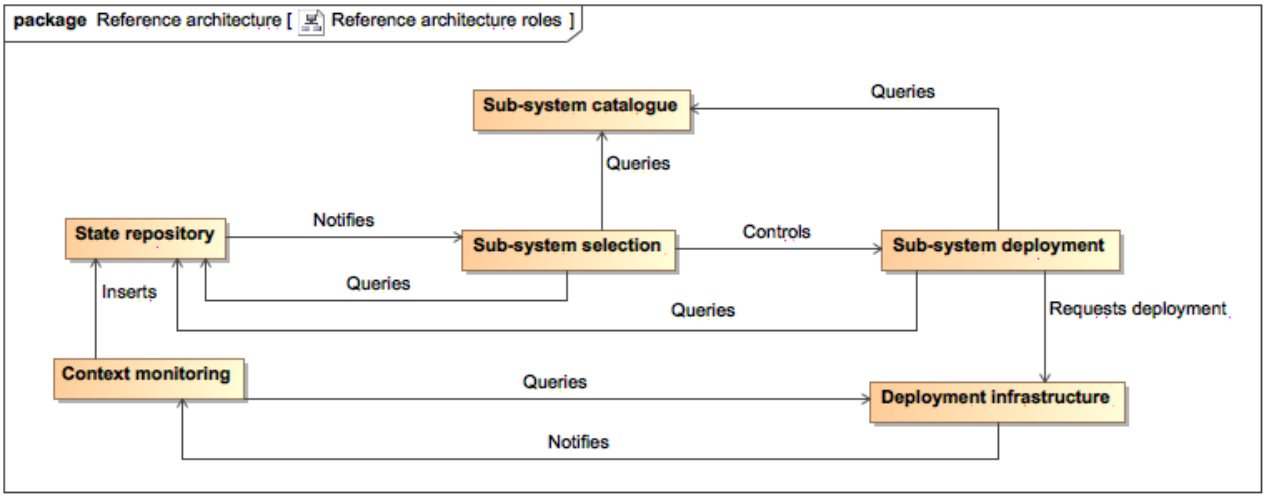

Fig. 8: The proposed reference architecture

REFERENCES

[1] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[2] A. Dearle, "Software deployment, past, present and future," in *2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 269–284.

[3] Object Management Group, "Deployment and configuration of component-based distributed applications specification," http://www.omg.org/spec/DEPL/4.0/, 2004, [Online; accessed 03-January-2016].

[4] "Chef configuration management," https://www.chef.io/, 2017, [Online; accessed 15-October-2017].

[5] "Ansible configuration management," https://www.ansible.com/, 2017, [Online; accessed 15-October-2017].

[6] "Salt intelligent orchestration," https://saltstack.com/, 2017, [Online; accessed 15-October-2017].

[7] "Puppet configuration management," https://puppet.com/, 2017, [Online; accessed 15-October-2017].

[8] "Roslaunch tool for launching ros nodes," http://wiki.ros.org/roslaunch, 2017, [Online; accessed 15-October-2017].

[9] "Orocos deployer," http://www.orocos.org/stable/documentation/ocl/v2.x/doc-xml/orocos-deployment.html, 2017, [Online; accessed 15-October-2017].

[10] "Autosar (automotive open system architecture)," https://www.autosar.org/, 2017, [Online; accessed 15-October-2017].

[11] U. Eklund, O. Askerdal, J. Granholm, A. Alminger, and J. Axelsson, "Experience of introducing reference architectures in the development of automotive electronic systems," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–6, May 2005.

[12] L. Wang, D. Ma, Y. Zhao, X. Zhao, and Y. Wang, *An Approach to Develop Architecture of ARINC653-Based Avionic Software*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 257–262.

[13] N. Hochgeschwender, L. Gherardi, A. Shakhimardanov, G. K. Kraetzschmar, D. Brugali, and H. Bruyninckx, "A model-based approach to software deployment in robotics," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, November 3-7, 2013*, 2013, pp. 3907–3914.

[14] D. Calisi, A. Farinelli, G. Grisetti, L. Iocchi, D. Nardi, S. Pellegrini, D. Tipaldi, and V. A. Ziparo, "Uses of contextual knowledge in mobile robots," in *Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence on AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, ser. AI*IA '07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 543–554.