2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
Congress Center Hamburg
Sept 28 - Oct 2, 2015. Hamburg, Germany

RRA: Models and Tools for Robotics Run-time Adaptation

Luca Gherardi and Nico Hochgeschwender

Abstract— **Robotics applications are characterized by a huge amount of variability. Their design requires the developers to choose between several variants, which relate to both functionalities and hardware. Some of these choices can be taken at deployment-time, however others should be taken at run-time, when more information about the context is known. To make this possible, a software system needs to be able to reason about its current state and to adapt its architecture to provide the configuration that best suites the context.**

This paper presents a model-based approach for run-time adaptation of robotic systems. It defines a set of orthogonal models that represent the system architecture, its variability, and the state of the context. Additionally it introduces a set of algorithms that reason about the knowledge represented in our models to resolve the run-time variability and to adapt the system architecture. The paper discusses and evaluates the approach by means of two case studies.

I. Introduction

In recent years robotics developers have adopted component-based robotics-specific software frameworks (e.g. ROS) to design applications with increasingly complex behavior (e.g. robots that cooperate to cook [1]). These systems integrate several components, which provide a wide spectrum of functionalities, such as control, motion planning, and perception. Most of these components are openly available and, typically, several components provide slightly different implementations of the same functionality. An example is the variety of path planning components designed during the years. In order to enhance exchangeability, different functionality implementations can be harmonized [2]. This allows the developer to exchange the functionality implementation based on the particular requirements of their application, without having to reimplement the other components.

In addition to functional variability, other variability dimensions consist of robotics hardware and environment. The same measurements can be provided by different sensors, which have to be chosen according to the task and the environment. For example a GPS is suitable for outdoor localization but is not reliable when used indoor.

These variability dimensions make the development of robotics applications challenging and time consuming. If on the one hand the large availability of software components should support the reusability, on the other hand decisions such as what components to use (i.e. which algorithm for a certain functionality) and how to configure and compose them, require the developers to master competencies in several domains. It is therefore necessary to introduce robotics-specific approaches to address software variability issues.

Luca Gherardi is with the Institute for Dynamic Systems and Control, ETH Zürich, Switzerland. `lucagh at ethz.ch`.

Nico Hochgeschwender is with the Department of Computer Science, Bonn-Rhine-Sieg University, Sankt Augustin, Germany. `nico.hochgeschwender at h-brs.de`.

A. Variability Management in Robotics Systems

Previous works demonstrated how robotics variability can be addressed by integrating Software Product Lines[1] (SPLs) techniques with robotics-specific meta-models[2] and tools that support the developers [3][4]. These approaches allow the design of reference architectures that can be configured to derive different applications. Reference architectures define the set of software components and how they interact. Additionally, they explicitly define the architectural variation points and variants (e.g. optional or alternative components), and map them to functional requirements. At deployment time, the developer chooses the requirements of his application and lets the tools automatically select the required architectural variants for each variation point. The variants selection and the process of configuring the reference architecture according to the selected variants is called *variability resolution*. This process leads to the deployments of the desired application and is the key concept of this paper.

Model-based SPLs hide the low-level configuration details from the system integrator and improve the reusability, which is not anymore limited to the component level but is lifted to the architecture level. However, not all the variability can be optimally addressed at deployment-time and not all the run-time conditions in which the robot will operate can be foreseen at design-time. One problem consists of selecting the best variant for each variation point. Although some variation points can be addressed at deployment-time to deploy one application instead of another, other variation points, which strictly relate to the application behavior, can be better resolved at run-time, when more information on the context (current state of robot embodiment, situatedness and intelligence) is available. For example, the choice of the obstacle avoidance algorithm can be better performed at run-time, when the density and the dynamics of the obstacles are known. A second problem relates to the elicitation of the possible variants, which are not always known at design-time or are too many to be enumerated. An example is the upper bound used to compute a joint torque. Based on the material of the object that has to be moved, this value can be chosen in the continuous range $[q_{min}, q_{max}]$ (infinite variants). Some objects are known a priori, others can be recognized only at run-time by accessing a knowledge repository.

[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.

[2]A model defining a language to express domain-specific information.

B. Addressing Run-time Variability

These issues lead to the necessity of (a) postponing the resolution of some variation points at run-time and (b) continuously reasoning about the run-time variability to choose the run-time configuration (i.e. variants selection) that best suits the current context and provides the best Quality of Service (QoS). According to this view the resolution of the deployment-time variability allows the configuration of the SPL reference architecture and the deployment of a specific application. Later, the architecture configured at deployment-time becomes the reference architecture for the run-time variability resolution, which dynamically adapts its reference architecture to address the changes in the context.

We present Robotics Run-time Adaptation (RRA), a model-driven approach that addresses runtime adaptation by modeling and resolving run-time variability. Its elements of novelty are: (a) a set of meta-models that explicitly define the run-time variability and the rules for its run-time resolution; (b) a set of algorithms and tools that reason about the aforementioned information to adapt the software architecture; (c) the integration of the proposed approach with an approach that addresses the deployment-time variability [3]. The meta-models, the algorithms and the tools are openly available.

The paper is organized as follows. Sec. II introduces the deployment-time variability resolution. Sec. III presents the models and algorithms for run-time variability management. Sec. IV evaluates the approach. Finally, Sec. V discusses related work and Sec. VI draws relevant conclusions.

II. Deployment-time Variability Resolution

This section introduces the concepts of the deployment-time variability resolution that are relevant in this paper. The deployment-time models and the tools are presented in [3]. Figure 1 depicts how the run-time variability resolution (right part) builds on the top of the deployment-time approach (left part). The two approaches are independent, however, modeling the run-time variability on the top of the deployment-time variability, allows us to model the run-time adaptation for a family of applications instead of a specific one.

Three problems have to be addressed to resolve the deployment-time variability based on user's requirements. First, the formalization of the SPL reference architecture and its architectural variability. Second, the identification of the functionalities provided by the SPL applications, their relationships, and their constraints (i.e. functional variability). Third, the definition of a mapping between the architectural and the functional variability, which allows the resolution of the first based on the variants selected for the second.

The first issue is addressed by means of the *Template System Model*, which specifies the components required to build all the possible applications and the interactions between them (aka connections). This model encodes several variation points: (a) components can be optional, (b) components' implementation can be changed, (c) new connections can be created, and (d) components' parameters can be configured.

The functional variability is symbolically represented by means of a *Feature Model* [5], which is a hierarchical composition of features and is organized as a tree. Features (depicted as rectangles) represent application's functionalities and/or requirements and can be mandatory or optional (denoted with a white or black circle on the top, respectively). We distinguish between deployment-time and run-time features, which differ for the binding time of the variation points encoded with the features. The former have to be bound at deployment-time while the latter at run-time (deployment-time features relate to variability that do not depend on the context, hence the bounding of deployment-time feature cannot be reconsidered at run-time). Parent features are connected to child features by means of edges representing containment relationships. Two kinds of containments are available. Aggregation, where the parent feature is made of the child features, and specialization, where the parent feature is a variation point, while its children are the possible variants. Variants can be alternative or complementary (depicted with a white and a black arc, respectively). Finally this model allows the definition of dependencies or incompatibilities between functionalities. The *Feature Editor* [6] allows the design of *Feature Models*.

The mapping between architectural and functional variability is defined by the *Deployment-time Resolution Model*. It encodes a set of model-to-model (M2M) transformations that specify, for each variation point of the *Template System Model*, which variant has to be selected based on the variant selected for the corresponding variation point in the *Feature Model*. In other words the *Resolution Model* specifies how the SPL reference architecture has to be configured according to the requirements selected in the *Feature Model*.

Based on a selection of deployment-time features that reflect the requirements of the application, the *Deployment-time Resolution Engine* uses the transformations defined in the *Deployment-time Resolution Model* to resolve the *Template System Model* deployment-time architectural variability and to produce the *Deployment-time Configured System Model*. This model defines the architecture of the desired application and contains only run-time variation points, which will be resolved during the run-time variability resolution.

III. Run-time Variability Resolution

This section presents the run-time variability resolution by means of an academic case study. After introducing the case study, we describe the problems that had to be addressed to design a run-time adaptation framework.

A. Object Detection and Storing Case Study

A mobile manipulator equipped with a laser scanner and an RGB-D camera operates in a workspace made of two rooms (A and B) connected through a corridor. The rooms are equipped with furnitures while other robots and people move in the corridor (for reasons of safety the maximum velocity in the corridor is limited to 0.5 m/s). In each room, there are two racks containing different objects made of different materials. Objects are identified by means of barcodes. An annotated grid-map of the environment is provided, where the locations of the racks are tagged. The racks

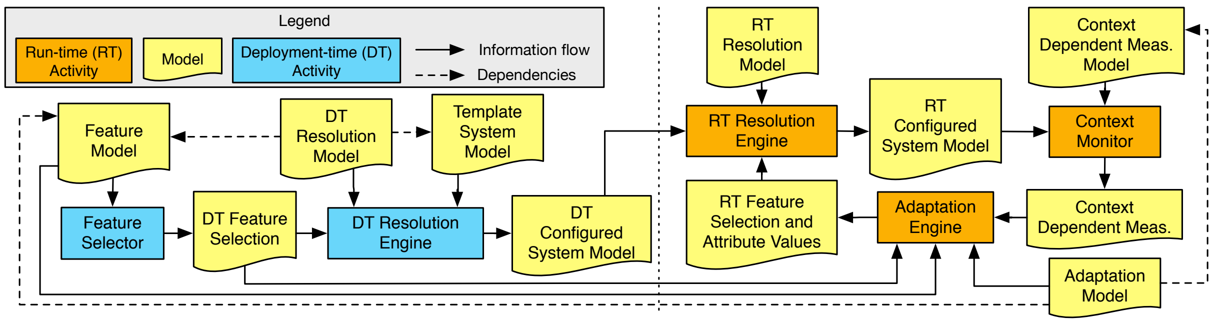

Fig. 1. An overview of the model-based variability management approach.

in area A are small (objects are placed closer to each other) whereas the racks in area B are large (objects are not placed densely). The robot interacts with a remote server using one of the two available communication channels, which have different latencies and power consumptions and have to be chosen in order to improve the QoS (the low latency channel is preferable when the battery level is high). The robot can be asked to execute two different tasks. The goal of the first task (detection) is to identify the position of the objects placed on different racks. The remote server specifies the object to be identified (in terms of rack ID and barcode) and expects back the object position. This task requires software components providing navigation and perception functionalities. The goal of the second task (manipulation) is to move all the objects to a different rack. The remote server specifies the next object to be moved, its material and the target rack. In addition to the components required for the first task, components providing the manipulation functionality are also needed.

In order to understand the goal of RRA, one can think about the following scenario. The user selects one of the two tasks and the design-time variability resolution configures the SPL reference architecture to deploy the application that executes that task. When the application is deployed, the run-time variability resolution monitors the context and according to this information adapts the architecture generated at deployment-time. For example, when the battery level goes below a certain threshold, the run-time adaptation switches to the lower consumption communication channel.

Our goal is to generate the application architecture and trigger its adaptation automatically. Additionally we want to decouple the information required to adapt the run-time architecture from the implementation of the robotic functionalities, in such a way to make the latter more reusable.

B. Run-time Resolution Initialization

The first problem addressed by RRA relates to the system initialization, which requires a reference architecture for the run-time adaptation and a model of the functional variability.

The reference architecture is specified by means of the *Deployment-time Configured System Model*, which specifies the components that can be used during the application life cycle. The run-time variability-resolution continuously adapts this model to provide the configuration that best suits the current context. Here for configuration we refer to the process of (a) activating and deactivating components, (b) changing their implementations, (c) changing the connections between them, and (d) configuring their parameters.

The configuration of the *Deployment-time Configured System Model* is stored in the *Run-time Configured System Model*, which reflects the software architecture running the robot control system. This means that the adaptation and the run-time variability resolution are performed on the model and then applied to the real system. After the initialization the *Run-time Configured System Model* is an exact copy of the *Deployment-time Configured System Model*. Both these models conform to a component meta-model, which is specific to a certain software framework and defines the architectural elements needed to model the system architecture. These meta-models are deeply discussed in [3][4].

Fig. 2 depicts the *Deployment-time Configured System Model* for the execution of the manipulation task. In order to keep the example simple, most of the components are an abstraction of a set of components, which together provide the same functionalities. The *Task Manager* coordinates the task execution. It receives instructions on the objects that have to be manipulated from a remote server. The communication with the remote server takes place through one of the radio channels, which are alternative (i.e. only one of the two components can be present), and are managed by the *Radio Switch*. Based on the query, the *Task Manager* interacts with the navigation components to move the robot in proximity of the desired rack. Once the robot reaches the rack, the *Task Manager* interacts with the *Object Detector* to retrieve the position of the desired object. Finally the *Task Manager* coordinates the activities that grasp the object, store it on the robot, and finally place it into another rack.

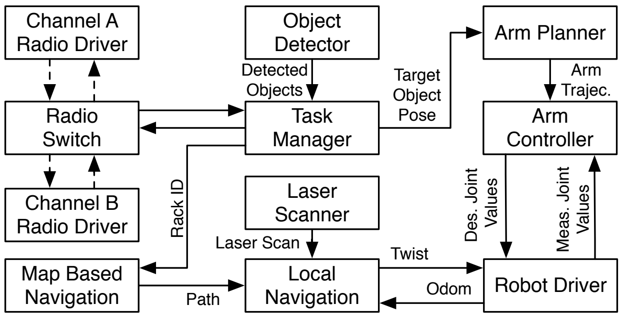

Fig. 2. Case study *Deployment-Time Configured System Model*. Rectangles depict components, continuous arrows stable connections from publishers to subscribers (topics are implicitly represented), dashed arrows optional connections. The interfaces used by the context monitor are not depicted.

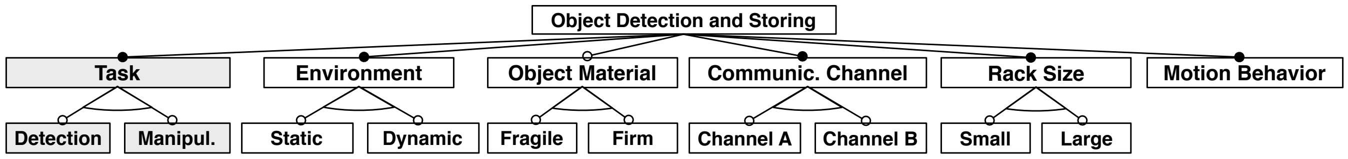

Fig. 3. The case study *Feature Model*. Deployment-time features are depicted in gray, run-time features in white.

We decided to use a *Feature Model* to represent the run-time functional variability. This allows for an easy integration with the deployment-time variability resolution. In fact, the *Feature Model* can be the same used in that phase. Fig. 3 depicts the case study *Feature Model*. Since this paper focuses on run-time variability, we simplified the deployment-time variability to a single variation point, which is the task. The alternative variants are *Detection* and *Manipulation*. The deployment-time resolution includes the manipulation's components into the *Deployment-time Configured System Model* only when the *Manipulation* feature is selected.

The run-time variability consists of the following variation points. (a) The obstacle avoidance algorithm has to be decided based on the dynamics of the obstacles. The *Static* feature is used inside the rooms while the *Dynamic* feature is adopted when the robot is in the corridor. (b) The object material is an optional variation point, which must be bound only when the *Manipulation* feature is selected (example of feature dependency). Based on the material the grasping parameters of the *Arm Controller* component have to be properly configured. (c) The communication channel defines which radio component has to be used and its connections with the *Radio Switch*. (d) The *Rack Size* influences the configuration of the algorithm that computes the object poses. The *point cloud subsampling* and the *region of interest* must be configured accordingly. (e) The *Motion Behavior* specifies a maximum velocity parameter, which influences the computation and execution of the rover trajectories. This parameter makes the robot movements smoother or more reactive. Later in this section we explain how a variation point is encoded in this mandatory feature.

As depicted in Fig. 1, the selection of run-time features for the aforementioned variation points is computed according to the set of adaptation rules and run-time information that are described in the rest of this section. The feature selection is the input that triggers the variability resolution and consequently the architecture adaptation.

C. Monitoring the Context

The second problem addressed by RRA relates to the context monitoring, which means collecting the information that will drive the run-time adaptation. We call this information *Context Dependent Measurements*. Robotics systems already acquire these measurements and take important decisions based on their values. However, the acquisition of these measurements is usually hard-coded in the implementation of the components that reason about them. We claim that making this information explicit allows for a higher reusability of the components. Indeed, if the relevant measurements are collected and computed only by specific components (from now on *Context Monitors*), the other components of the system can be relieved from this task. As a result their implementation is more reusable because it is decoupled from *Context Dependent Measurements*, which are typically application specific. We designed RRA to provide the formalism needed to model *Context Dependent Measurements*.

Context Dependent Measurements are typically computed starting from the outputs of different system's components (e.g. the component that drives the robot may publish the battery level). To make this explicit, we have defined the *Context Dependent Measurements Model* (Fig. 4), which describes the relationship between a *Context Dependent Measurement*, the interface producing the information required for its computation, and the data type produced by the interface. Additionally, the model describes functions that operate on these data types. This is required because in most cases one is not directly interested in the output of a component, but in the information that can be retrieved from the last output or from a series of outputs (e.g. by running an estimator). For example, given the output of a *Laser Scanner* component, one might need to apply a function to retrieve the distance to the closest obstacle. Interface data types and functions are modeled by means of the *Data Type* meta-model (Fig. 4). Data types can be hierarchically composed and define functions. Note that we are interested in modeling functions that operate on a certain data type and produce as output a *Context Dependent Measurement*.

Notably, a Model-2-Text transformation (M2T) operating on the *Context Dependent Measurements Model* and the *Data Types Model* allow the automatic generation of the *Context Monitor*. For each *Context Dependent Measurement* the generated component defines: (a) an input interface associated to the output interface publishing the required information, (b) an optional function to process the information and create the *Context Dependent Measurement*, and (c) an output interface to publish it.

Table I presents an excerpt of the *Context Dependent*

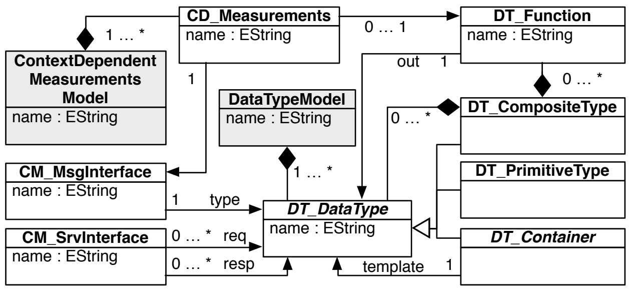

Fig. 4. The Context Dependent Measurements and Data Types meta-models. The prefixes specify the meta-model in which the classes are defined. (CD: CDM meta-model. DT: Data Type meta-model. CM: Component meta-model.)

Measurements Model defined for the case study. For each measurement the table reports the component producing the data, its interface, and the optional function that transforms the data into the measurement.

D. Selecting the Run-time Features

The third problem addressed by RRA relates to the selection of the functionalities (run-time features) required by the robot to operate at its best based on the state of the context. This problem can be decomposed in two subproblems: (a) modeling the required information, (b) reasoning about it.

The information acquired by the *Context Monitor* is not sufficient to automatically generate the feature selection. We realized that two other pieces of information are required. First, a set of measurements that cannot be acquired with sensors but the values of which is always the same. We call them *Context Independent Measurements*. Examples from the case study are the communication channels' properties, which describe the latencies and the power consumptions. Second, a set of constraints, typically called *Non-Functional Requirements*, which do not focus on *what* the application should do, but rather on *how*. A possible example from the case study is the safety velocity limit that must be respected when the robot is in the corridor. The difference between *Context Independent Measurements* and *Non-Functional Requirements* is that the former define information that is useful to make run-time decisions while the latter express constraints that must be respected to achieve the goal.

We propose to model this information in the *Feature Model* by enriching features for which the developer wants to specify a *Context Independent Measurement* or a *Non-Functional Requirement* with an attribute. To achieve this goal we have extended the Feature Meta-model presented in [3], where the entity *Attribute* specifies four properties: a name, a data type, a value and a unit of measurement. In addition to the standard properties, *Non-Functional Requirement* attributes also specify one of the following *tags*: min, max, avg, and $count$ (e.g. max indicates that the attribute value is the maximum allowed for the non-functional requirement represented by the attribute).

The idea of modeling *Non-Functional Requirements* in the *Feature Model* and not with a specific model (e.g. the SysML requirements diagram) is mainly motivated by the fact that *Non-Functional Requirements* are strictly related to the information provided by the features. In fact, *Non-Functional Requirements* typically provide constraints for the requirements modeled by the features. In a similar way, [7] models *Non-Functional Requirements* as feature's attributes. However, they use them only at deployment-time.

CDM	Component	Interface	Function
BattLev	Robot Driver	Battery Level	—
SemLoc	Map Based Nav.	Semantic Location	—
ObstDist	Laser Scanner	Laser Scan	getClosestObst
Material	Radio Switch	Object Material	—

TABLE I
CASE STUDY CONTEXT DEPENDENT MEASUREMENTS.

Feature	Attribute	Type	Value	Unit
Channel A	Latency	CIM	High	—
Channel A	PowerConsump	CIM	0.5	$\%/min$
Dynamic	VelocityLimit	NFR (max)	0.5	m/s
Motion Behavior	MaxVel	CV	[0.4, 0.8]	m/s

TABLE II
CASE STUDY ATTRIBUTES.

Another information to be modeled, as described in the introduction, is the *Continuous Variability*, i.e. variation points for which the number of variants can be infinite. We propose to model the *Continuous Variability* by means of attributes. For example, in the case study we have a feature *Motion Behavior*, which encodes the maximum velocity. A possibility for modeling this variation point would have been defining two sub features with fixed velocity limits. However one might want to change the limit more precisely, for example according to the battery level and the distance to the closest obstacle. We addressed this problem by enriching the feature *Motion Behavior* with a *Continuous Variability* attribute, which adds to the standard attribute's properties an upper and a lower bounds. In the case study an attribute called *MaxVel* specifies that the maximum velocity can be between 0.4 and 0.8 m/s. Differently from the attributes introduced before, *Continuous Variability* attributes do not have a fixed value but store the current one. Note that the *MaxVel Continuous Variability* is not in contrast with the *VelocityLimit Non-Functional Requirement*. In fact *VelocityLimit* is a constraint on the allowed values for the attribute *MaxVel*, which is valid only when the robot is in the corridor.

Table II reports some of the attributes defined for the case study. The column *Type* uses the following abbreviations: *Non-Functional Requirements* (NFR), *Context Independent Measurements* (CIM) and *Continuous Variability* (CV).

Once the aforementioned information is modeled, the second sub-problem consists of using it to generate the selection of features that will trigger the adaptation. Ideally the logic that produces the selection should be decoupled from the components that provide the functionalities. We decided to model it by means of a model called *Adaptation Model*. It specifies how the *Context Dependent* and *Independent Measurements* trigger a set of actions that generate a selection of features that respect the *Non-functional Requirements*.

The *Adaptation Model* defines a set of atomic rules, which consist of a condition under which the rule is fired and an action. Rule conditions compose *Non-functional Requirements*, *Context Dependent*, and *Context Independent Measurements* by means of logical, relational and math operators. Three types of actions are defined: feature selection, feature deselection and modification of *Continuous Variability* attribute values. Rule-sets compose atomic rules that manipulate the same features or the same attribute. Atomic rules of the same set must have different priorities.

The adaptation logic is implemented by the *Adaptation Engine*, a component that is in charge of reasoning about the state of the context and producing the selection of run-time features and values for *Continuous Variability* attributes

```
import caseStudy.contextDepMeasModel as cdm

rule Location:
 if( cdm.SemLoc == "RoomA" OR "RoomB" )
  activate_feature(Static)
 else activate_feature(Dynamic)

rule RackSize:
 if( cdm.SemLoc == "RoomA")
  activate_feature(Small)
 else if( cdm.SemLoc == "RoomB")
  activate_feature(Large)
 else deactivate_feature((Large AND Small)

rule Radio:
 if( cdm.batteryLevel > "50" )
  select_feature_from_variants_of(Communic. Channel)
  where_attribute MIN(Latency)
 else select_feature_from_variants_of(Communic. Channel)
  where_attribute MIN(PowerConsump)

ruleSet MaxVelocity:
 rule maxVelBattObst priority 2:
  set_attribute(MaxVel) with_value(MaxVel.min + (MaxVel.
      max-MaxVel.min)*(BattLev/100)*[1-e^(-ObstDist)])
 rule maxVelCorridor priority 1:
  if( cdm.SemLoc == "Corridor" )
   set_attribute(MaxVel) with_value(VelocityLimit)

rule ObjMaterial:
 if( cdm.material == "Glass" )
  activate_feature(Fragile)
 else activate_feature(Firm)
```

Fig. 5. Examples of the adaptation rules defined for the case study.

that best suits the context and at the same time satisfies the *Non-Functional Requirements*. The *Adaptation Engine* algorithm periodically iterates all the rules and executes the active ones. Atomic rules of a rule-sets are evaluated and executed starting from the one with smaller priority (higher number). Note that, when the run-time resolution is built on the top of the deployment-time resolution, not all the run-time features are selectable (e.g. if a deployment-time feature is not selected, its run-time sub-features cannot be selected at run-time). Therefore the *Adaptation Engine* needs to know the selection of deployment-time features. In this context, a problem that will need to be addressed is verifying that two or more rules with opposite effects are not fired at the same time (e.g. two rules that select and deselect the same feature).

Fig. 5 exemplifies some rules defined for the case study. They are expressed by means of a Domain Specific Language (DSL) implemented with Xtext. For example the rule *Radio* allows the selection of the radio channel that provides the best compromise between latency and power consumption, while the rule-set *MaxVelocity* exemplifies a situation in which *Non-Functional Requirements* are used to set and limit the value of a *Continuous Variability* attribute (*MaxVel*).

The DSL defined for the *Adaptation Model* is similar to other Event-Condition-Action languages found in event-driven frameworks. However, we decided to implement a new DSL in order to make seamless the integration with all the models described in this paper.

The *Adaptation Engine* is configured by means of the *Adaptation Model*. Once configured this component has an input interface for each *Context Dependent Measurement* produced by the *Context Monitor*, and implements the rules described by the *Adaptation Model*. Notably, the developers do not have to implement the *Adaptation Engine* but only describe its adaptation rules by writing an *Adaptation Model* with the editor provided with our toolchain. This makes the implementation of the *Adaptation Engine* highly reusable. In contrast with the *Context Monitor*, the *Adaptation Engine* does not have output interfaces, but transmits to the *Run-time Resolution Engine* a selection of run-time features and the values of the *Continuous Variability* attributes.

E. Adapting the Run-time Architecture

The last problem addressed by RRA relates to the reconfiguration of the system's architecture based on the selection of run-time features. Here the approach defined for the deployment-time variability resolution can be reused. In particular we decided to encode in a model called *Run-time Resolution Model*, the rules describing how the architectural variability of the *Run-time Configured System Model* has to be resolved based on the output of the *Adaptation Engine*. The *Run-time Resolution Model* associates to each run-time feature a set of transformations, which have to be performed when the feature is selected. This model can be used to instruct the component in charge of executing the architecture reconfiguration: the *Run-time Resolution Engine*. Differently from the *Deployment-time Resolution Engine*, the run-time version applies the transformations not only to the architectural model, but also to the architecture of the running system. This is done by invoking an adaptation API, which executes the transformations on the running system. For this reason, because of the different inputs, and to allow the independent execution of deployment-time and run-time variability management, we separated deployment-time and run-tim resolution in two engines, which are based on the same software library.

Developers can define the following transformations: (a) component can be launched and stopped, (b) their implementation and parameters can be changed, and (c) connections can be created and removed. The meta-model defining the transformations is an extension of the meta-model used to model the *Deployment-time Resolution Model* [3]. To support the run-time resolution we have defined a new transformation, which uses the value of a *Continuous Variability* attribute to set the value of a component's parameter.

Note that the *Run-time Configured System Model* is always a configuration and a subset of the *Deployment-time Configured System Model* (e.g. if the feature *Detection* is selected, the manipulation components are not present in the template and cannot be used at run-time).

One may argue that the architecture adaptation could be triggered directly by the adaptation rules without the need for generating the feature selection (i.e. run-time requirements). However, we believe that the proposed approach allows for a better separation of the concerns. Indeed, the developers in charge of designing the adaptation rules can focus on the definition of the functionalities required for the possible states of the context and need not to care about architectural issues. These are indeed modeled in the resolution model, which can be designed by a software architecture expert. Notably, the *Adaptation Model* is architecture independent.

Some examples of the transformations defined in the case study *Run-time Resolution Model* are the following. (a) Based on the feature selected for the variation point *Communic. Channel*, one of the *Radio Driver* components is launched and the other is stopped. The connections with the *Radio Switch* are changed accordingly. (b) A parameter of the *Local Navigation* component is configured according to the value of the *Continuous Variability* attributes *MaxVel*. (c) The feature selected for the *Environment* variation point influences the implementation used for the obstacle avoidance algorithm in the *Local Navigation* component.

IV. Evaluation

In order to evaluate the approach we implemented a simpler case study. The goal is to simulate a quadrocopter moving along a trajectory and use RRA to recover from a motor-failure. Note that fault-recovery is not the main target of RRA, but just one of its possible applications. Therefore here the focus is not on how the recovery algorithm works, but rather on how the architecture is dynamically adapted.

The architecture of the case study is a ROS system made of the following components: a trajectory generator, a controller, an estimator and a vehicle simulator. An additional *Proploss* controller, which allows the vehicles to recover in case of motor failure [8], is available but not launched at deployment-time. The *Context Monitor* listens the outputs of the vehicle simulator (drawn current, desired and measured motor speed) and analyzes the discrepancy between expected and measured current in order to infer the number of motors that are working as expected. The *Adaptation Engine* uses this measurement to resolve a variation point made of two variants: a feature `Proploss` is selected only when less than four motors are functioning. Based on this feature the *Run-time Resolution Engine* adapts the architecture as follows. When the feature `Proploss` is not selected, the architecture is not modified. Otherwise the standard controller node is stopped and replace with the *Proploss* controller.

The case study has been executed on a standard laptop. Fig. 6 depicts the current drawn by the motor, the desired collective thrust produced by the new controller (multiplied by 4) and the altitude of the vehicle. The motor breaks after 366.3 seconds and the new controller starts sending commands after 0.5 seconds. During this small time the context information is processed, the standard controller is stopped and the *Proploss* controller is launched. A video of the case study is available at `lucagherardi.it/RRA`.

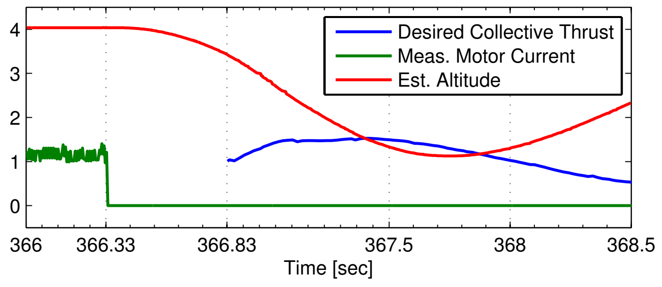

Fig. 6. The plot shows how the controller is switched in 0.5 seconds.

V. Related Work and Discussion

In robotics, run-time adaptation concepts such as intelligent methods to cope with run-time failures (e.g. cognizant failure [9]) have been studied in the context of the layered robot control architectures. Interesting concepts are presented in [10] and [11]. These architectures employ specific languages and frameworks (e.g. ESL [9] and PLEXIL [12]), which encode execution knowledge that is related to failure procedures and context monitoring and that is crucial for run-time adaptation. RRA is inspired by these approaches in the sense that we aim to systematize their concepts in a well-formalized model-based software engineering approach.

In software engineering, run-time adaptation approaches have been studied mainly from an architectural perspective (e.g. [13] and [14]), where architectural models are used to express topological and behavioral adaptation constraints.

Other run-time adaptation approaches are instead based on the MAPE (Monitor-Analyze-Plan-Execute) principle [15]. Interestingly, MAPE can be considered as a variant of a three-layered robot control architecture, for example the one introduced in [10]. A survey on run-time adaptation approaches based on MAPE is presented in [15]. These approaches vary in how the context is monitored and how the adaptation is planned and executed in order to satisfy functional and non-functional requirements. In our approach the adaptation is triggered by a set of event-based rules. Differently [16] adopts an approach that aims to optimize non-functional requirements whereas [17] utilizes planning methods to synthesize component architectures in order to meet new requirements. We argue that as long as the interface of the adaptation engine doesn't change, our framework can be easily extended with new adaptation algorithms.

Robotics Run-Time Adaptation follows MAPE and introduces the following novel design decisions. First, being RRA based on explicit models, it does not modify the run-time architecture directly (e.g. in [18]) but it modifies the feature selection, which then triggers architecture adaptation. As a consequence the adaptation design is abstracted from the architectural level to the functional level, which allows us to decouple the adaptation rules from the architecture.

Second, being RRA based on models that describe three orthogonal layers (i.e. functional and architectural variability, and adaptation rules), developers can easily change the models describing one of these layers without affecting the others. For example, in contrast to [13] and [16], RRA allows developers to easily change the underlying component model (e.g. switch from ROS to Orocos). Developers must change the *Template System Model* (or *Run-Time Configured System Model* if they adopt only the run-time part) and the *Context Dependent Measurement Model*. However they can reuse the original *Feature Model* and the original *Adaptation Model*. Similarly developers can model functional run-time variability for a different set of requirements with a new *Feature Model*, and reuse the original architectural model.

Third, the orthogonal nature of RRA promotes a structured development process, where developers with different skills (e.g. software architect, robotics system integrator, etc.) focus on their specific domains and encode their knowledge in dedicated models. As claimed in [19], this kind of approaches enables modeling by and for reuse and makes easier the development of robotic applications of increasing complexity.

VI. Conclusions

This paper presented RRA, a model-based approach for run-time adaptation of component-based robotics systems. The approach relies on a set of orthogonal models that represent the system architecture, its run-time variability, the state of the context, and the information required for resolving the variability. We also designed a set of algorithms and tools, which continuously reason about the knowledge represented in the aforementioned model, resolve the run-time variability, and consequently adapt the system architecture. Finally, we presented how this approach can be integrated with other approaches for the resolution of deployment-time variability in robotics SPLs. Thanks to our approach, the information regarding the variability is not anymore hard coded in the software components, but can be explicitly modeled, thus increasing the level of flexibility and reusability. Despite being simple, the evaluation case study demonstrated how the approach is feasible and how the run-time adaptation can be achieved quickly enough to recover a flying machine.

RRA has been implemented for ROS but can be easily extended to accommodate other frameworks. The *Context Monitor* is generated by means of a M2T transformation and is a roscpp node. The *Adaptation Engine* is instead a rosjava node and uses the *Adaptation Model* and the java reflection to dynamically create the subscribers that listen for the outputs of the *Context Monitor*. The *Run-time Resolution Engine* is an eclipse plugin, which uses ROS tools to launch the initial nodes and execute the run-time reconfiguration of the running system. Models and tools are openly available [20].

We intend to improve our approach from several points of view. First, we want to support *Context Dependent Measurements* computed from the output of different components. A problem to be addressed is the synchronization of the different inputs. Second, we want to investigate different adaptation strategies for the run-time replacement of the components (e.g. should a new component start before or after the termination of the one that has to be replaced? Should we preserve the component's state after its termination?). Third, we want to extend our approach to multi-machine systems by integrating applications deployment models [21].

Acknowledgments

The first author would like to thank Mark Müller for the *Proploss* controller and the Flying Machine Arena team for the control, estimation, and simulation infrastructure [22]. Additionally he would like to thank Raffaello D'Andrea for founding the research. The second author is recipient of a PhD scholarship from the Graduate Institute of the Bonn-Rhein-Sieg University, which he gratefully acknowledges.

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] D. Brugali, W. Nowak, L. Gherardi, A. Zakharov, and E. Prassler. Component-based refactoring of motion planning libraries. In *International Conference on Intelligent Robots and Systems (IROS)*, 2010.

[3] L. Gherardi. *Variability Modeling and Resolution in Component-based Robotics Systems*. PhD thesis, University of Bergamo, 2013.

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

[5] K.C. Kang. Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document, 1990.

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

[7] S. Soltani, M. Asadi, D. Gašević, M. Hatala, and E. Bagheri. Automated planning for feature model configuration based on functional and non-functional requirements. In *International Software Product Line Conference*, 2012.

[8] M. W. Mueller and R. D'Andrea. Stability and control of a quadrocopter despite the complete loss of one, two or three propellers. In *International Conference on Robotics and Automation (ICRA)*, 2014.

[9] E. Gat. ESL: a language for supporting robust plan execution in embedded autonomous agents. In *Aerospace Conference, 1997. Proceedings., IEEE*, volume 1, pages 319–324 vol.1, Feb 1997.

[10] R. Peter Bonasso, David Kortenkamp, David P. Miller, and Marc Slack. Experiences with an architecture for intelligent, reactive agents. In Michael Wooldridge, Jörg P. Müller, and Milind Tambe, editors, *Intelligent Agents II Agent Theories, Architectures, and Languages*, volume 1037 of *Lecture Notes in Computer Science*, pages 187–202. Springer Berlin Heidelberg, 1996.

[11] Rachid Alami, Raja Chatila, Sara Fleury, Malik Ghallab, and Félix Ingrand. An architecture for autonomy. *International Journal of Robotics Research (IJRR)*, 17(4):315–337, 1998.

[12] Vandi Verma, Tara Estlin, Ari Jónsson, Corina Pasareanu, Reid Simmons, and Kam Tso. Plan execution interchange language (PLEXIL) for executable plans and command sequences. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS)*, 2005.

[13] D. Garlan, S-W Cheng, A-C Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based Self-adaptation With Reusable Infrastructure. *Computer*, 2004.

[14] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From goals to components: A combined approach to self-management. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 1–8. ACM, 2008.

[15] M. C. Huebscher and J. A. Mc Cann. A Survey of Autonomic Computing – Degrees, Models, and Applications. *ACM Compututing Surveys*, 2008.

[16] A. Lotz, J. F. Inglés-Romero, C. Vicente-Chicote, and C. Schlegel. Managing run-time variability in robotics software by modeling functional and non-functional behavior. In *Enterprise, Business-Process and Information Systems Modeling*. Springer Berlin Heidelberg, 2013.

[17] Hossein Tajalli, Joshua Garcia, George Edwards, and Nenad Medvidovic. Plasma: A plan-based layered architecture for software model-driven adaptation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 467–476, New York, NY, USA, 2010. ACM.

[18] G. Edwards, J. Garcia, H. Tajalli, D. Popescu, N. Medvidovic, G. Sukhatme, and B. Petrus. Architecture-driven self-adaptation and self-management in robotics systems. In *SEAMS*, 2009.

[19] D. Alonso, C. V. Chicote, F. Ortiz, J. Pastor, and B. Alvarez. V3CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development. *Journal of Software Engineering for Robotics*, 2010.

[20] RRA Toolchain. `https://github.com/lucaghera/RRA`.

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

[22] S. Lupashin, M. Hehn, M. W. Mueller, A. P. Schoellig, M. Sherback, and R. D'Andrea. A platform for aerial robotics research and demonstration: The Flying Machine Arena. *Mechatronics*, 2014.