Declarative Specification of Robot Perception Architectures

Nico Hochgeschwender[1,2], Sven Schneider[1],
Holger Voos[2], and Gerhard K. Kraetzschmar[1]

[1] Bonn-Rhein-Sieg University, Computer Science, Sankt Augustin, Germany
nico.hochgeschwender@h-brs.de
[2] University of Luxembourg, SnT Automation Research Group, Luxembourg

Abstract. Service robots become increasingly capable and deliver a broader spectrum of services which all require a wide range of perceptual capabilities. These capabilities must cope with dynamically changing requirements which make the design and implementation of a robot perception architecture a complex and tedious exercise which is prone to error. We suggest to specify the integral parts of robot perception architectures using explicit models, which allows to easily configure, modify, and validate them. The paper presents the domain-specific language RPSL, some examples of its application, the current state of implementation and some validation experiments.

1 Introduction

Service robots operating in industrial or domestic environments are expected to perform a wide variety of tasks in different places with often widely differing environmental conditions. This poses many challenges for the perception-related parts of the control software (here referred to as **robot perception architecture** (RPA)), which includes recognizing and tracking manipulable and non-manipulable objects, furniture, people, faces, and recognizing gestures, emotions, sounds, and speech.

Designing a single set of perception components that performs all these perceptual tasks simultaneously, robustly, and efficiently would require enormous effort and would result in unmanageable complexity. To meet the challenges of service robotics we need concepts, methods, and tools for designing and developing RPAs in a very flexible manner. Ultimately the robot should be able to adjust to the wide range of situations autonomously (e.g. by dynamically selecting a set of perceptual components into an RPA configuration for a particular task). Fig. 1 illustrates the concept, where the components shown in red compose the RPA configuration active when the pose of a person is required. To do so, explicit knowledge representation about its available perception capabilities/functionalities (as depicted in Fig. 1) are required.

However, many RPA design decisions remain nowadays implicit. These decision concern the robot *platform*, robot's *tasks*, and the *environment* in which the

D. Brugali et al. (Eds.): SIMPAR 2014, LNAI 8810, pp. 291–302, 2014.

robot operates. Some examples include the selection and configuration (e.g. resolution and data frequency) of sensors, the selection and parameterization of filters and feature detectors (see also [1]), and the selection, configuration and/or training of classifiers. Also, the domain experts connect all these perceptual components into a coherent RPA. We argue that implicit design decisions are a major cause for the inflexibility of today's RPAs as if any of the implicit assumptions is changing, the task to adapt the RPA remains challenging and is prone to errors.

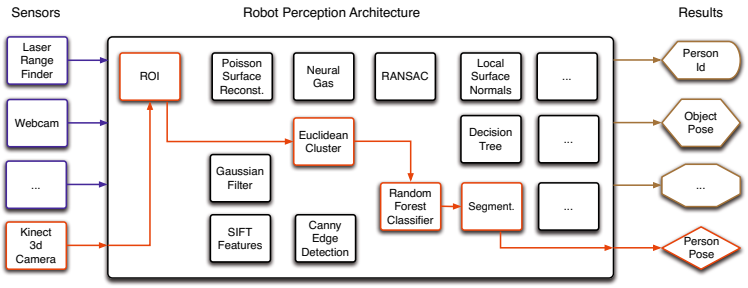


Fig. 1. The design space of Robot Perception Architectures (RPAs) includes the following constituents: *i*) heterogenous sets of sensors (blue boxes), *ii*) processing components (black boxes), *iii*) task-relevant information and knowledge (brown boxes), and *iv*) perception graphs (red visualized path)

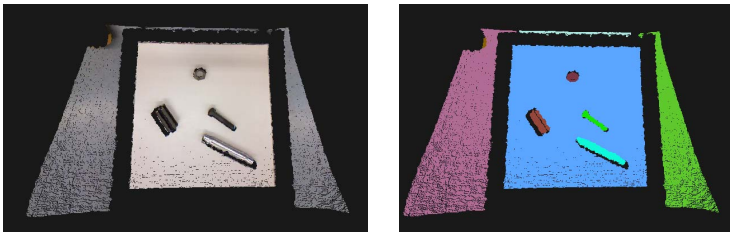
Providing RPAs with the aforementioned cabilities requires to model the design decisions in an explicit and computable manner [2]. In the work presented here, the Model-Driven Engineering [2] approach is adopted for the design and development of RPAs as it enables modeling *for* and *by* reuse. More precisely, we introduce a set of meta-models and a corresponding domain-specific language (DSL) which enables the declarative and explicit specification of the integral parts of RPAs (see Sec. 2). We also show how concrete domain models are reused in an architecture facilitating the demand-driven selection and execution of perception graphs stored in a (model) repository (see Sec. 3).

2 RPSL: Robot Perception Specification Language

In the following we introduce the meta-models (abstract syntax) of the RPSL which is a textual domain-specific language (DSL) [2]. The RPSL allows to specify the integral parts of RPAs in an explicit manner. To identify the domain abstractions required to specify RPAs a domain analysis was performed on existing RPAs [3] which have been integrated on various robot platforms. Ranging from people detection, recognition and tracking to object recognition, pose estimation and categorization the assessed functionalities cover a wide range of perceptual capabilities required for today's service robots. Several core domain concepts were identified and described, namely components, algorithms, and

perception graphs. These domain concepts correspond roughly to the structural constituents of RPAs as shown in Fig. 1. We also identified conceptual spaces as a cross-cutting domain. We apply a MDE approach using the Eclipse Modeling Framework (EMF)[1]. Each domain is specified in the form of an Ecore model. Based on the Ecore models, we developed a DSL using the Xtext framework[2].

Example: Color-Based Region Growing Segmentation. To exemplify the domain abstractions introduced with RPSL we use a standard segmentation method commonly applied in robotics, namely color-based region growing segmentation [4]. An example of the output of the system is shown in Fig. 2. Here, a scene with industrial objects such as screws and nuts is segmented. We assume the following setup: *i)* A RGB-D camera provides a $3D$ point cloud with RGB information at a resolution of 640×480 pixels. *ii)* Another component implements the color-based region growing segmentation. The component takes the whole point cloud as an input and provides a list of segmented regions, based on a certain configuration such as color range and minimum/maximum number of points per segmented region.



(a) Before Segmentation (b) After Segmentation

Fig. 2. Example of the color-based region growing segmentation

2.1 Modeling Components

We propose component-based development for RPAs similar to [5]. The core idea is to use components as building blocks and to design RPAs by composing components with clearly defined interfaces. Component-based development fosters structured design and development and is nowadays the predominant software development approach in robotics [6].

The Component Meta-Model (CMM) (see Fig. 3) borrows the core structural elements, such as components and ports, from the BRICS Component Model (BCM) [6] and has been enriched through RPA-specific aspects. The main objective of the CMM is to model `Components`, the basic building blocks of RPAs. Each `Component` contains `Ports` of type `InputPort` and `OutputPort`, which serve as endpoints for communication between `Components`. Comparable

[1] `http://www.eclipse.org/modeling/emf/`
[2] `http://www.eclipse.org/Xtext/`

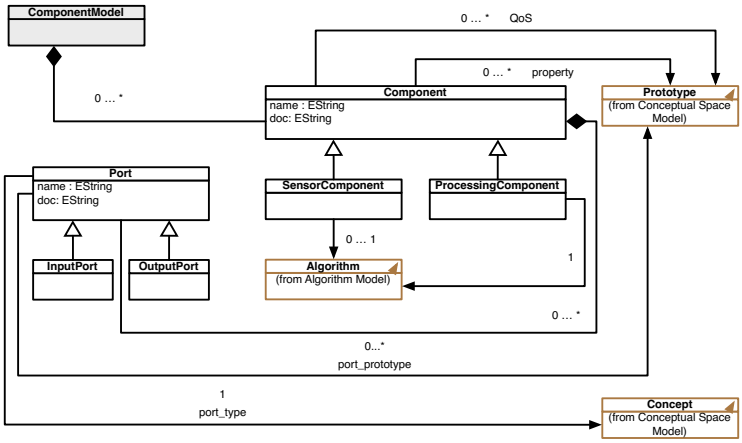


Fig. 3. The Component Meta Model (CMM)

to robot software frameworks such as Orocos RTT and others[3] `Ports` are typed. In RPSL, the type is modeled using the conceptual space meta-model (CSMM) presented in Sec. 2.3. Based on our domain analysis [3] we identified and distinguish between two types of `Components`, namely `SensorComponents` and `ProcessingComponents`.

`SensorComponents` are used to model sensors (e.g. cameras and laser range finders) and `ProcessingComponents` are used to model purely computational components which implement functionalities such as filters and feature detectors. `ProcessingComponents` link to an `Algorithm` modeled using the algorithm meta-model (see Sec. 2.2). Nowadays, some sensors preprocess data internally (e.g. noise filtering) before the sensor data is delivered. Thus, `SensorComponents` might link to an `Algorithm` as well. Each `Component` has `properties` such as configuration parameters, which are specific to the actual functionality (e.g. the minimum number of points per region for the region-growing segmentation), and `Quality of Service (QoS)` characteristics, such as worst case execution time. Similarly to `Ports`, the `QoS` characteristics and `properties` are modeled using the CSMM. `Components` are the basic first-class primitives in our meta-model. Currently, we do not support composition of components on a CMM level; hierarchies will be introduced later with the Perception Graph Meta-Model (PGMM) (see Sec. 2.4). `Ports` of `Components` are solely intended for data exchange among `Components` and not for configuration concerns (e.g. configuration ports found in other component models [7]). This design decision implies that the domain expert needs to specify all components together with particular configuration values. This approach fosters the modeling of feasible configuration values for components, as the domain expert is led to provide them. This also provides the possibility (if needed at a later stage) to reduce the design space (e.g. through grouping components).

[3] See [7] for a discussion on Component Models found in robotics.

2.2 Modeling Algorithms

The main objective of the Algorithm Meta-Model (AMM) (see Fig. 4) is to model *meta-information* about the algorithms which are integrated in components[4].

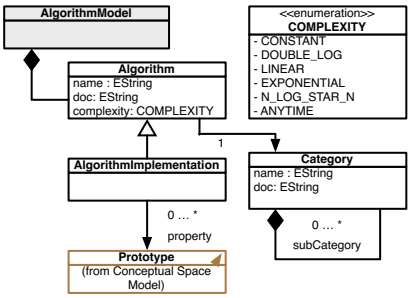


Fig. 4. The Algorithm Meta Model (AMM)

We distinguish between `Algorithm` and `AlgorithmImplementation`, where `AlgorithmImplementation` models a particular implementation of an `Algorithm`. Each `Algorithm` belongs to a `Category` (e.g. filter, feature descriptor) and provides information about its `Complexity`. Grouping algorithms in certain categories is feasible and *best practice*. In fact, every major computer vision and perception library, such as PCL [8] or OpenCV [9], is organized in categories sharing some properties. The distinction in `Algorithm` and `AlgorithmImplementation` is useful because it enables the domain expert to model different implementations of a particular algorithm. The color-based region growing algorithm, for instance, could either be implemented naïvely on the CPU, while another implementation is optimized for GPUs. Both implementations are specializations of the same `Algorithm` (e.g. `RegionGrowing`), but have different `properties` such as precision.

2.3 Modeling Conceptual Spaces

RPAs are producing heterogenous output spanning multiple levels of abstractions and ranging from raw sensor data and subsymbolic representations to symbolic information. The type of output depends on the task of the robot and on other functional components demanding the information. For example, in a pick-and-place scenario a decision making component might be interested in the types/names (symbolic information) of present objects, whereas a grasping component demands information about the pose of an object. Hence, a knowledge representation approach which enables us to model data produced by RPAs

[4] Please note, we do not model the algorithms themselves, e.g. in terms of steps and procedures.

on various levels of abstractions is required. In [10], Gärdenfors introduced Conceptual Spaces (CS) as a knowledge representation mechanism, which is used here and extended for RPSL. A CS contains the following constituent parts:

- A **Conceptual Space** is a metric space where **Concepts** are defined as convex regions in a set of domains (e.g. the concept *Color*).
- A **Domain** includes a set of **Domain Dimensions**[5] that form a unit and are measurable (e.g. the domain dimension *Red* of the RGB color model).
- An **Instance** is a specific vector in a space (e.g. the RGB color *red* with the values 255 (red), 0 (green), and 0 (blue)).
- A **Prototype** is an **Instance** which encodes typical values for a **Concept**.

The vector-based representation of the CS framework allows to apply similarity measures such as Euclidian distance to decide to which concept an instance belongs. In [11], Chella *et al.* showed that the CS framework enables the systematic integration of different knowledge representations as required in robotics. Further, the CS representation facilitates computationally efficient implementations based on the vector-based approach.

The Conceptual Space Meta-model (CSMM))(see Fig. 5) is a formalization of the CS framework as an Ecore model. In RPSL its purpose is to model the input and output of computational components of RPAs. Additionally, we use it to model `QoS` and general `properties` of `Components` and `Algorithms`. The CSMM contains several `Concepts`, where `Concepts` may contain `subConcepts` thereby supporting hierarchical concept structures. For each `Domain` a `DomainDimension` is defined. According to Gärdenfors, a `DomainDimension` is measurable. As RPAs deal with different types of data we introduce four `DomainDimensions` based on the work of Stevens [12], namely `NominalDimension`, `OrdinalDimension`, `IntervalDimension`, and `RatioDimension`. Each dimension permits to apply a set of logical and mathematical operators suitable to model different data.

Example: The color-based region growing segmentation component produces a list of regions. For each region the points belonging to it, the number of points, and the average RGB value of the points in the region are stored. We model a region as a `Concept` named `Region` referring to three `Domains`, namely `PointCloud`, `NumberOfPoints`, and `AvgColor`. We now exemplify the `AvgColor` domain which is decomposed into four `DomainDimensions` each of them of type `IntervalDimension`. Three of them are used to model the RGB color model and one for the standard deviation σ of the color distribution of the region. Each `IntervalDimension` for the RGB color model is equipped with an `Interval` ranging from 0 to 255 and of type `Integer` whereas the range for σ is from 0 to 1 and of type `Double`. Furthermore, a prototype is defined, which declares typical values for each `Domain`. For instance, depending on the configuration of

[5] In [10], a domain dimension is named quality dimension. For the sake of avoiding confusion with the term Quality of Service, we renamed it domain dimension.

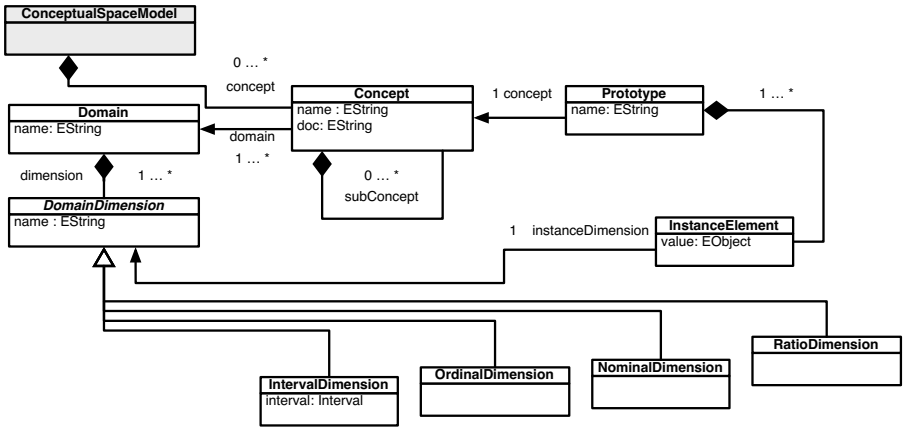


Fig. 5. Excerpt of the Conceptual Space Meta Model (CSMM). For the sake of readability some elements are not shown: the set of mathematical operators applicable on each domain dimension, the `Interval` class for the `IntervalDimension`, and the full set of attributes for each dimension class.

the algorithm the `NumberOfPoints` would be an `IntervalDimension` with an interval from 50 (minimum number of pixels per region) to 10,000.

2.4 Modeling Perception Graphs

The Perception Graph Meta-Model (PGMM) (see Fig. 6) enables the composition of components (see Sec. 2.1) in a directed acyclic graph (DAG) of components. A `PerceptionGraph` (PG) consists of two types of `Elements` (where `Element` refers to exactly one `Component`), namely `Nodes` which have at least one successor and `Leafs` without successor. To explicitly model successors, `Nodes` are connected through `Connections`, which have one `InputPort` and one `OutputPort` and refer to exactly one `Element`. Hence, we ensure that we do not connect two `InputPorts` or `OutputPorts` with each other. The PGMM enables the domain expert to model PGs which can easily be reused. Ranging from simple filtering pipelines to more elaborated PGs with multiple input, output and processing branches.

2.5 Modeling Constraints

Once domain concepts are represented as meta-models, we can also define constraints on concrete domain models conforming to these meta-models. Similarly to [13], we use the Object Constraint Language (OCL)[6] to model two types of constraints, namely atomic and composition constraints. Here, atomic constraints are valid for single meta-models whereas composition constraints appear when we compose meta-models (e.g. CSMM and PGMM in Sec. 2.4).

[6] `http://www.omg.org/spec/OCL/2.3.1/`

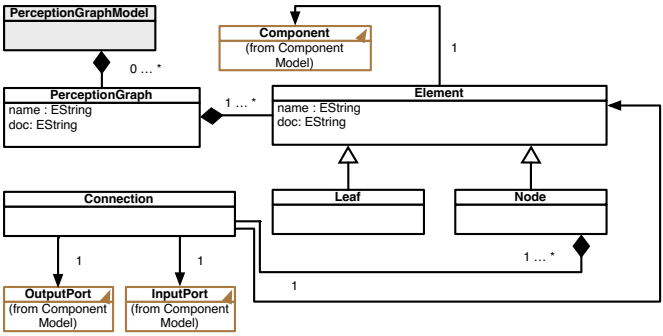


Fig. 6. The Perception Graph Meta Model (PGMM)

Beside atomic constraints, such as ensuring non-empty names and IDs, we check the following composition constraints: *i)* Each `Element` of type `Leaf` in a `PerceptionGraph` refers to a `Component` with at least one `OutputPort`. This ensures that a `PerceptionGraph` always provides an output. *ii)* Each `PerceptionGraph` does not have any directed cycles. This ensures the DAG property.

2.6 Modeling Demand

So far, we have modeled the integral parts of RPAs. To allow for demand-driven selection of PGs, we need abstractions to express demands. For that, we introduce the concept of a `Request`, which encodes an expected piece of information which is to be provided as the output of a `PerceptionGraph`. As inputs and outputs are modeled with the CSMM, a `Request` needs to know which `Concepts` are available in our architecture (see also Fig. 8). We introduce the concept of a `PrototypeRequest`. A `PrototypeRequest` consists of *i*) the prototype, which is a concrete instance of a concept (i.e. all properties have a specific value), *ii*) a distance, which determines how close values must be to the prototype, and *iii*) a distance measure. An example of a `PrototypeRequest` is shown in Fig. 7. The general idea of a `ProtoypeRequest` is that a *client* (the component expressing the demand) defines an expected value for each `DomainDimension` of a `Concept` which is later used to compute the most suitable PG. The suitability is defined by the `Request` in terms of `Similarity` which contains a `Metric` (e.g. a Euclidian distance or Jaccard distance) and a corresponding distance value which is interpreted as the maximally-allowed deviation. Assuming the `Request` can be fulfilled, the *client* expects some sort of data which is specified in the `Data` entry. Here, we support either one sample or a list of samples (in our example a list of Regions).

```
from myconcepts import Region as R

PrototypeRequest segmentedRegions {
 Prototype regionPrototype {
  R.AvgColor.Red = 220
  R.AvgColor.Green = 20
  R.AvgColor.Blue = 60
  // ...
 }

 Similarity similarity {
  Metric m = SIMILARITY_METRIC.EUCLIDIAN_DIST
  Distance  d = 0
 }

 Data data {
  List_of Region
 }
}
```

Fig. 7. An excerpt of a `PrototypeRequest` modeled with the `Request` DSL for the segmentation example

3 RPSL Run Time Environment

To validate the abstractions introduced by the RPSL and to realize the demand-driven selection and execution of previously modeled PGs we implemented the RPSL run time architecture shown in Fig. 8. During design time, a domain expert uses an editor to model concrete domain models of `PerceptionGraphs` using previously modeled `Components` and `Concepts`. The `PerceptionGraphs` and `Concepts` are stored in dedicated repositories which are accessible at run time by the run time architecture, which is bound to receive several requests each of which encodes a demand for a concrete piece of information that can be

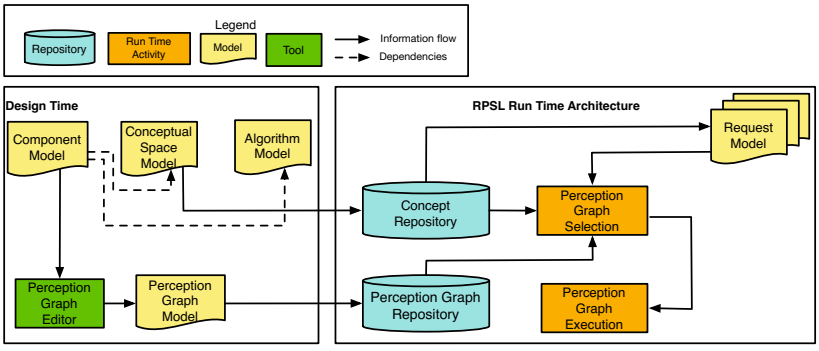


Fig. 8. The RPSL Run Time Architecture

provided by a stored `PerceptionGraph`. This demand is modeled with the abstractions introduced in Sec. 2.6. Based on the request and the models stored in the repositories we need to *select* and *execute* a `PerceptionGraph` (see activities in Fig. 8). Demand-driven selection and execution of perception graphs is beneficial for two reasons: First, it is not necessary anymore to deploy PGs before they are actually required at run time as for many tasks the sequence which PG needs to be active is not known a priori. In particular, for resource-constrained robots (e.g. micro air vehicles) this is advantagous as the use of resources like memory can be optimized. Second, requests are made explicit, which faciliates step-wise development and systematic testing of RPAs.

Algorithm 1. Selection of a perception graph based on a request

Input: Request R, set of perception graphs $PG = \{pg_1, pg_2, ..., pg_i\}$
Output: Set of candidates $C = \{\{c_1, c_2, ..., c_i\}$ where $C \subset PG$

```
for all Output o_i in PG do
  if Concept C_R of R matches the concept C_i of o_i then
    if R is of type PrototypeRequest and o_i includes Prototype p_i then
      // Compute similarity dist. with given measure between C_R and p_i
      d_i ← R.similarity.m(C_R, p_i)
      if d_i <= R.distance then
        C ← C ∪ pg_i
```

Perception Graph Selection and Execution. To select a PG matching a particular `Request`, we apply Algorithm 1 which iterates over each PG stored in the repository and assesses the outputs it provides. As a result, a `Request` can yield any of the following situations: *i*) no PG matches, *ii*) exactly one PG matches, *iii*) several PGs match. In the case of a `PrototypeRequest`, the PG with the shortest distance is executed.

Implementation and Experiments. The architecture shown in Fig. 8 is work in progress, implemented in Java and Ruby, and will be released soon as open source.[7] For each meta-model presented in Sec. 2, we already provide an Eclipse-based textual editor which enables the specification and storage of domain models. The run time system of the RPSL architecture contains two main modules: PG `selection` and PG `execution`. The `selection` module realizes the platform-independent algorithm described above, whereas the `execution` modules is platform-dependent. The modeled PGs are independent of a particular framework in which the PGs are implemented. For execution of the PGs, the PG primitives are mapped to framework-specific primitives. So far, we have a direct mapping to ROS nodes implementing the PGs, but we plan to realize this step with a Model-2-Model transformation from the PGMM to a framework-specific meta-model. To assess the overall approach, we modeled several PGs ranging from smaller examples such as the different variants of the region growing PG

[7] `https://github.com/nicoh/RPSL`

to more complex PGs, such as combined region of interest, segmentation, and shape-extraction PGs. For each PG we also tested corresponding `Requests`. It is possible to select a PG which matches the `Request` even in a repository which is unstructured (different PGs stored for different purpose).

4 Discussion and Related Work

To the best of our knowledge the presented approach is the first which applies the MDE approach to the design and development of RPAs. Although MDE approaches are becoming popular in robotics, they mainly focus on subdomains such as coordinate representations [14].

The CSMM turned out to be a general-purpose meta-model which is applicable not only to describe the input and output of components but also their properties and QoS characteristics. We intend to consider this information also in the selection process to select the fastest or most reliable PG. To which extend this information will be reflected in the `Request` remains open and will be investigated in ongoing work. To model these *non-functional* properties we will also investigate the meta-models proposed in [15].

Even though with `PrototypeRequests` we can already select PGs we foresee the implementation of different types of `Requests` such as `ConstraintRequests`. Here, a `ConstraintRequest` would define constraints (e.g. inequality constraints such as >, <= etc.) on `DomainDimension` which is beneficial when several PGs provide the same concept at their output, but their characteristics differ (e.g. different `Intervals` for the same `DomainDimension`).

To define acceptable names and terms for the `Concepts` is crucial for the usage of the proposed approach. Hence, we will investigate ontologies proposed in other projects such as [16]. In contrast to the work by Moisan *et al.* [1], our approach does not depend on a feature-model representation as it is mainly driven by the integral architectural parts of RPAs.

We also plan to enrich the PG `execution` through deployment models [13] such that PGs can be deployed on platforms which increase the performance.

5 Conclusion

This paper presented the Robot Perception Specification Language (RPSL) which enables the explicit specification of RPAs. We showed how to model and store ready-to-use perception graphs and to efficiently select the most appropriate perception graph at run time.

Acknowledgement. Nico Hochgeschwender is recipient of a PhD scholarship from the Graduate Institute of the Bonn-Rhein-Sieg University, which he gratefully acknowledges.

References

1. Moisan, S., Rigault, J.-P., Acher, M., Collet, P., Lahire, P.: Run time adaptation of video-surveillance systems: A software modeling approach. In: Crowley, J.L., Draper, B., Thonnat, M. (eds.) ICVS 2011. LNCS, vol. 6962, pp. 203–212. Springer, Heidelberg (2011)
2. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer 39(2), 25–31 (2006)
3. Hochgeschwender, N., Schneider, S., Voos, H., Kraetzschmar, G.K.: Towards a robot perception specification language. In: Proceedings of the 4th International Workshop on Domain-Specific Languages and Models for ROBotic Systems (DSLRob) (2013)
4. Zhan, Q., Liang, Y., Xiao, Y.: Color-based segmentation of point clouds. International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences 38, 248–252 (2009)
5. Biggs, G., Ando, N., Kotoku, T.: Rapid data processing pipeline development using openrtm-aist. In: 2011 IEEE/SICE International Symposium on System Integration (SII), pp. 312–317 (2011)
6. Bruyninckx, H., Klotzbücher, M., Hochgeschwender, N., Kraetzschmar, G., Gherardi, L., Brugali, D.: The brics component model: A model-based development paradigm for complex robotics software systems. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC 2013, pp. 1758–1764. ACM, New York (2013)
7. Shakhimardanov, A., Hochgeschwender, N., Kraetzschmar, G.K.: Component models in robotics software. In: Proceedings of the Workshop on Performance Metrics for Intelligent Systems, Baltimore, USA (2010)
8. Rusu, R.B., Cousins, S.: 3D is here: Point cloud library (pcl). In: Proceedings of the International Conference on Robotics and Automation (ICRA) (2011)
9. Bradski, G.: The OpenCV Library. Dr. Dobb's Journal of Software Tools (2000)
10. Gärdenfors, P.: Conceptual spaces - the geometry of thought. MIT Press (2000)
11. Chella, A., Frixione, M., Gaglio, S.: A cognitive architecture for artificial vision. Artif. Intell. 89(1-2), 73–111 (1997)
12. Stevens, S.S.: On the Theory of Scales of Measurement. Science 103, 677–680 (1946)
13. Hochgeschwender, N., Gherardi, L., Shakhirmardanov, A., Kraetzschmar, G., Brugali, D., Bruyninckx, H.: A model-based approach to software deployment in robotics. In: 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 3907–3914 (November 2013)
14. Nordmann, A., Hochgeschwender, N., Wrede, S.: A survey on domain-specific languages in robotics. In: Brugali, D., Broenink, J., Kroeger, T., MacDonald, B. (eds.) SIMPAR 2014. LNCS (LNAI), vol. 8810, pp. 193–204. Springer, Heidelberg (2014)
15. Ramaswamy, A.K., Monsuez, B., Tapus, A., et al.: Solution space modeling for robotic systems. Journal for Software Engineering Robotics (JOSER) 5(1), 89–96 (2014)
16. Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M.: RobotML, a domain-specific language to design, simulate and deploy robotic applications. In: Noda, I., Ando, N., Brugali, D., Kuffner, J.J. (eds.) SIMPAR 2012. LNCS, vol. 7628, pp. 149–160. Springer, Heidelberg (2012)