RPSL meets Lightning: A Model-based Approach to Design Space Exploration of Robot Perception Systems

Loïc Gammaitoni and Nico Hochgeschwender

***Abstract*— The design space of a robotic application defines at a meta level what are all of its possible implementations. Those possibilities are called design alternatives and differ on many different aspects, one being preferred to the other depending on how, where, when or what the application should do. Design Space Exploration (DSE) is the process of reviewing those design alternatives, prior to their implementation, with intention to verify that the set of all design alternatives to be implemented covers all the possible scenarios in which the application is to be executed. In this paper we address two challenges related to DSE, namely, (1) the formal definitions of design spaces, a non-trivial task due to the many dimensions to be taken into consideration, and (2) the automatisation of DSE, that is, enabling a domain expert to review design alternatives corresponding to a given design space effortlessly. In this paper, we address those challenges in the context of robot perception software systems by combining two already existing technologies, namely RPSL for the specification of robot perception system's design spaces and Lightning, a language workbench that we use to formalise RPSL and obtain, from RPSL specifications, corresponding design alternatives.**

I. Introduction

The development of complex robotic applications meant to be deployed in dynamic environments is a challenging, time-consuming and error prone exercise. Indeed, designers of such systems not only need to cope with vastly heterogeneous distributed hardware, but also need to compose a large variety of sophisticated features[1] required for the task at hand. Although, many robotic features are nowadays available in the form of reusable software components [1], the task of choosing and composing features while meeting functional and non-functional application requirements remains challenging. This challenge is known as the *variability* problem [2] [3].

Take as an example a service robot performing an insertion task in an industrial environment (see Fig. 1). In order to perform this task, not only planning and control features, but also a broad set of perception features are required. Those features have to provide vital information for answering questions such as *where to grasp the object?*, *is the cavity large enough to insert the object?*, *is the cavity within reach?*, and so on. Not only the large number of required features to achieve such tasks, but also their inter-dependencies (e.g. an object tracking capability requires a detection capability) are contributing factors to the large *functional variability* of real-world systems [4] [5].

Developing a single set of perception features answering all those perception-related questions simultaneously and efficiently would result in unmanageable complexity. Hence, remarkable solutions (e.g. methods and algorithms) for solving some perceptual issues have already been developed [6]. In order to provide those solutions, domain experts perform a creative, experimental process which yields one or more *perception architectures*. Such an architecture implements not only a perception feature (e.g. by composing and configuring algorithms provided by libraries such as PCL [7] and OpenCV [8]), but also implicitly encodes a set of design decisions made by the expert at design time, e.g., choice of a robot platform and its sensing equipment, of the environment in which the robot will operate, of the tasks the robot should perform, and so on. All of these design decisions have an impact on structural, functional, non-functional and behavioral aspects of the perception architecture and are thus contributing factors to the large *architectural variability* of robot perceptions systems.

The combination of both architectural and functional variability form what we call the *design space* of a robot perception system. This design space needs to be *explored* by the domain expert in order to identify potential *design alternatives* satisfying possible new requirements that could arise from different applications. This exercise is known as *design space exploration* (DSE) [9]. Considering the aforementioned insertion task, design alternatives could differ in the implementation of, e.g., sensing feature by choosing to use either RGB-D or other modalities. Another example would be in the choice of how to implement cavity detection, e.g., using either fast or robust detection algorithms.

Model-based approaches are getting popular in robotics [10] to structure and manage specific aspects of a design space. Those approaches propose domain-specific languages allowing the intuitive representation of functional [3], architectural [2][11][12][13] and platform [2][11][12] variabilities. Although the development of robotic systems greatly benefits from those approaches (e.g. through model validation, model reuse and code generation), providing model-based solutions to design space exploration remains challenging. In particular, modeling design spaces as a whole –i.e., taking into account all variabilities – and using those models to perform a systematic and eventually (semi)-automatic exploration has not yet been achieved.

In this paper, we take a step toward providing such a method. For the specification of design spaces, we propose

Loïc Gammaitoni is with the University of Luxembourg, Luxembourg loic.gammaitoni@uni.lu Nico Hochgeschwender is with the Bonn-Rhein-Sieg University, Sankt Augustin, Germany nico.hochgeschwender@h-brs.de. Note that both authors contributed equally to this work.

[1]In the context of this work a feature is considered to encode a robotic functionality such as a sensing, planning and motion control functionality.

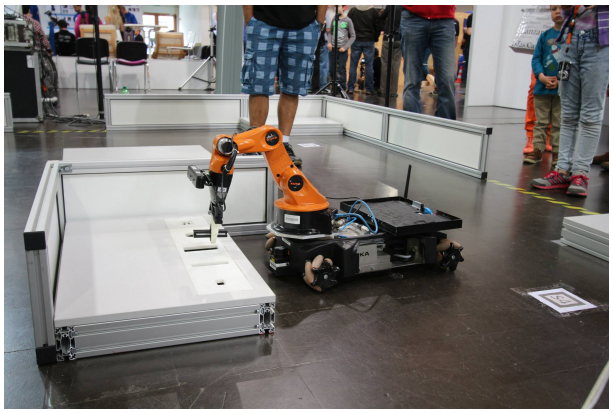

Fig. 1. A youBot robot performing an insertion task at a service area.

to use an extended version of an existing robotics language, the Robot Perception Specification Language (RPSL)[14], allowing the definition of both functional and architectural variability of robot perception systems. We also propose a new approach to DSE consisting in importing domain models expressed in RPSL into a framework based on Lightning [15], an Alloy [16] based tool allowing, given Alloy models, the generation of conforming instances and their depiction using a domain specific visualization [17].

The remainder of the paper is structured as follows. We first list in the next section some objectives our approach should fulfill. We then detail our approach in Sec. III before presenting the framework it relies in Sec. IV. We validate the approach with the help of a case study in Sec. V before closing the paper with discussions and presentation of related and future works.

II. DSE Objectives

In the following, we list several objectives our model-based approach to design space modeling and exploration should meet.

O1: Domain models should express both architectural and functional variability

We saw in previous section that a design space covers many variabilities. In robot perception systems the main variabilites are functional and architectural variability. The language we will use to model design spaces should thus allow the definition of both kinds of variabilities in a same domain model.

O2: The structural constraints of all variabilities should be formally defined

The language used to define domain models should be accompanied with some constraints specifying the well-formedness of design alternatives with respect to the variabilities considered – *e.g.* a design alternative cannot implement two mutually exclusive features.

This formally defines, given a domain model, what are valid design alternatives.

O3: Mechanisms to generate all possible design alternatives from a given domain model should be provided

As DSE is about reviewing valid design alternatives for a given design space then our approach should provide mechanisms to automatically obtain from a domain model the set of all possible (and valid with respect to structural constraints) design alternatives.

O4: Mechanisms to let domain experts guide the exploration should be provided

A domain expert might want to review design alternatives having certain properties solely, e.g., execution time below a given threshold, usage of components having a certain degree of precision, a.s.o.. . . Our approach should thus let the domain expert filter out valid design alternatives that do not meet his expectations regarding those properties.

O5: Design alternatives should be depicted in a syntax the domain expert is familiar with

Design alternatives obtained while performing the DSE should be represented intuitively in order to minimize the cognitive effort the domain expert needs to provide to inspect them. This has as effect to increase both the speed and the quality of the domain space exploration.

In the remainder of this paper, we present a solution to DSE fulfilling the aforementioned objectives.

III. An Approach to DSE

The approach to DSE we propose is depicted in Fig. 3. Firstly, the domain expert defines the design space in RPSL, a suitable domain-specific language for robot perception systems' design space specification. Design alternatives conforming to the given RPSL specifications and to the additional constraints optionally provided by the domain expert can then be obtained from the Lightning framework (see Sec. IV). In this section we introduce the building blocks of this approach namely: RPSL, the formal specification language Alloy (enabling the generation of conforming instances) and the Lightning language workbench (the tool our framework is based on).

A. RPSL: Robot Perception Specification Language

The starting point of our work was the selection of a language achieving O1. The RPSL [14] was a good candidate as it provides suitable abstractions – namely, perception graphs – enabling domain experts to express the architectural variability of robot perception systems. In particular, with RPSL a domain expert can represent multi-stage perception systems by composing sensing and processing *components* in a *perception graph*. In RPSL a perception graph is a directed acyclic graph where sensor and processing components are nodes. Here, sensing components represent sensors such as cameras and processing components encapsulate perception-related functions consuming and producing data in a data-flow oriented manner. For RPSL to fulfill O1, we extend the language with feature models, a suitable abstraction to represent functional variability [4], with intent that each leaf feature represents a perception capability realized by one or several perception graphs. This feature to perception graph mapping is given in a so called *Resolution Model*. Note that in a resolution model, a given feature is mapped to one or

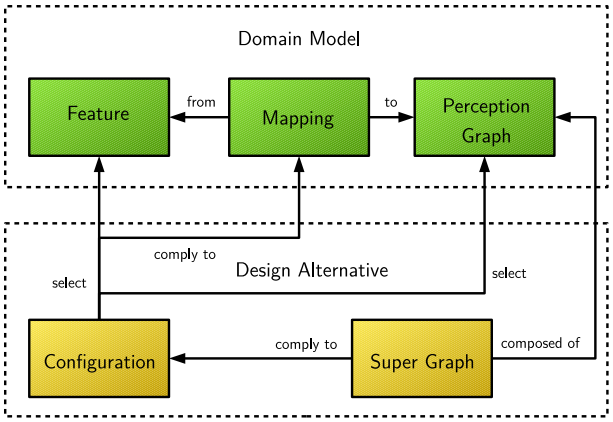

Fig. 2. Structural overview of an RPSL domain model and its conforming design alternative.

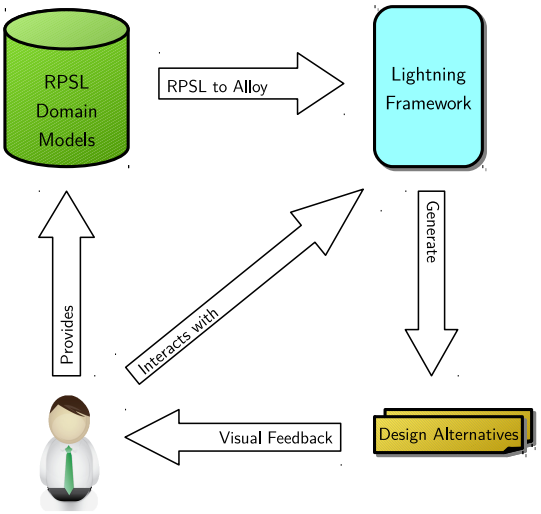

Fig. 3. Visualization of the DSE approach. Here, a domain expert shown on the bottom-left provides RPSL domain models, interacts with the framework and inspects visualized design alternatives.

more perception graph as a feature can be implemented by different perception graphs, with different characteristic – *e.g.*, time complexity, precision, *a.s.o.*.... To discriminate perceptions graph by their characteristics, we introduce the notion of weights assigned to each component and varying in function of how the component performs with respect to a selected characteristic. This is a step towards the fulfillment of O4 – *i.e.* less suited alternatives can be filtered out with respect to a selected characteristic).

Given a domain model, a conforming design atlernative consists of a selection of features and of a selection of exactly one perception graph per selected feature. We call these selections a *configuration*. The design alternative also contains what we call a *super graph*, the super graph being a well-formed composition of all the selected perception graphs. Relations between concepts of the domain model and of the design alternatives are depicted in Fig. 2. We note that those design alternatives should conform, as stated by O2, to a set of implicit structural constraints that will be listed in Sec. IV-B.

B. Alloy

Originally, the structural constraints of RPSL domain models was given pragmatically by the Ruby implementation RPSL comes up with. In this work, we provide Alloy specifications for domain models and design alternatives, so that their structural constraints is explicitly and non-ambiguously given – hence fulfilling O2.

Alloy [16] is a formal modeling language from MIT based on relational calculus and transitive closure, allowing the definition (in an *Alloy model*) of concepts – namely *signatures* –, relations between concepts – namely *fields*– and constraints on those relations – namely *facts*.

Alloy comes with its dedicated tool, the *Alloy Analyzer*, which relies on SAT solvers to find in a finite domain, given an Alloy model, the set of all instances conforming to its constraints. This mechanism, called *Alloy analysis*, will be used to achieve objective O3.

The graphical representation of instances generated by Alloy follow the structure of their model of origin, that is elements typed by given signatures are represented by boxes and relations between those elements are depicted by an arrow. In order to enhance the readability of such instances using the domain knowledge, and hence fulfilling objective O5, we make use of Lightning, an Alloy-based language workbench.

C. The Lightning Language Workbench

Lightning is an Eclipse plug-in allowing the definition and instantiation, using Alloy, of *domain specific modeling languages* (DSML) [15]. A DSML has a well defined *concrete syntax* specifying how instances of the language are to be depicted and a well defined *semantics* giving meaning to the language [18]. In this work, we have defined in Lightning domain models and design alternatives as full-fledged languages with well defined concrete syntax. This enables us to provide intuitive [17] (domain specific) visual feedback for the domain space exploration we propose to carry out.

In the next section, we introduce and explain the Alloy models composing the Lightning-based framework used in our approach. Those models specify the set of all valid RPSL domain models and design alternatives as well as their structural constraints.

IV. The Lightning-Based DSE Framework

In this section, we introduce the Lightning framework which our DSE approach is based on. This framework consists of Alloy models formalizing (1) RPSL – i.e., defining the set of valid RPSL domain models –, (2) valid design alternatives given a domain model –i.e., structural constraints–, and (3) a domain specific visualization of design alternatives. The relation between those Alloy models is given in Fig. 4.

Note that models being all expressed in Alloy, they were validated using the same lightweight approach we build our domain space exploration on (consisting in refining the models until generated instances match the expectations).

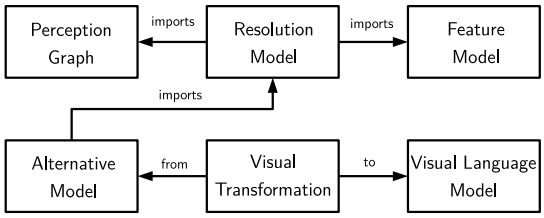

Fig. 4. Structure of the Lightning-based RPSL framework

A. RPSL Formalization

The RPSL formalization is done by defining in Alloy the three core RPSL artifacts – *i.e.*, Perception Graph, Feature Model and Resolution Model.

1) Feature Model: A Feature Model in RPSL is a tree of features where each child feature is a realization of its parent. Features can be mutually exclusive or require one another. The relationship between parent and children in an RPSL feature tree are of two kinds, *specification* or *containment*. Their semantics differ when it comes to feature selection. When a parent feature is selected to be part of a design alternative, any children feature can be selected in the case of a containment relationship, while exactly one should be selected in the case of a specification relationship.

The Alloy model defining this Feature Model is given in Listing 1. Constraints were left aside for readability and conciseness sake but can be found in the full version of the case study[2].

```
module AbstractFeatureTree

abstract sig FeatureTree{
  root: Feature
}
abstract sig Feature{
  spec: set Feature,
  contain: set Feature,
  excluded: set Feature,
  required: set Feature
}
```

Listing 1. Alloy model of RPSL Feature models (well-formedness constraints omitted)

2) Perception Graph: A Perception Graph in RPSL is a directed acylcic graph whose nodes represent sensor and processing components. Each component has output ports, while processing components also have input ports, both of arbitrary type[3]. Input and output ports can be connected by an edge only if they share the same type. It is possible to define graphs containing solely one or several processing components in which case, some input ports might not be connected. However, the composition of two (or more) such graphs is said to be well-formed if and only if all input ports are connected to some output ports. Note that each component can be assigned integer-valued weights as a discriminating factor with respect to a given property (see Sec. III-A). The weight of a graph is the sum of weights of its components.

[2]https://github.com/nicoh/rpsl-alloy

[3]We only assume the existence of different types without entering into details as this is not the focus of this paper. For more details about types we refer the reader to [14].

The Alloy specification of those Perception Graphs is given in Listing 2.

```
module AbstractPerceptionGraph

abstract sig PerceptionGraph{
  components: set Component,
  connections : set Output -> Input,
}
abstract sig Port {
  type:Concept
}
abstract sig Concept{}
abstract sig Input,Output extends Port {}
abstract sig Component {
   input: disj set Input,
   output: disj set Output,
   weight: Int
}
abstract sig SensorComponent,ProcessingComponent extends
     Component {}
```

Listing 2. Alloy model of RPSL Perception Graph (well-formedness constraints omitted)

3) Resolution Model: The resolution model maps one or several imported RPSL perception graphs to each leaf feature declared in the imported RPSL feature model. We do not provide an Alloy abstraction for Resolution models as the mapping they emobody can be directly derived from the RPSL specifications (see Sec. V-B). The resolution model is used by the alternative model to know given a selection of features which are the graphs that are candidate for a well-formed graph composition.

B. Design Alternatives

The structural constraints are given in the *Alternative* model defining for a given domain model the set of valid design alternatives.

Design alternatives are composed of a configuration defining the set of selected features and for each selected feature exactly one selected perception graph. It also defines the notion of *super graph* resulting from the composition of selected perception graphs. In Listing 3 we give the Alloy representation of this Alternative model.

```
module AlternativeModel
open ResolutionModel

one sig Configuration{
  selectedFeatures: set Feature,
  selectedGraph: set PerceptualGraph,
}{
  all f:selectedFeatures | one p:PerceptualGraph| f->p in
     feature2Graph.mapping and p in selectedGraph
  no disj x,y:selectedFeatures.~*(spec+contain) | x.
     excluded=y
  selectedFeatures.required in selectedFeatures.~*(spec+
     contain)
  selectedFeatures.(contain+spec)=none
}
one sig SuperGraph extends PerceptualGraph{
}{
  no c : components| c.input not in connections[Output]
    components=Configuration.selectedGraph.@components
  this.contains[Configuration.selectedGraph]
}
```

Listing 3. Alloy model of RPSL Alternative Model

The structural constraints expressed in Alloy in Listing 3 are, in the same order:

- with respect to configurations:

  - For each selected feature there should be exactly one perceptual graph mapped to this feature in the resolution model and selected in the configuration.
  - The set of selected feature should not be composed of features which are excluding each other or their parents.
  - The set of selected feature should contain at least one leaf feature implementing each required feature.
  - All selected features should be leaf features.
- with respect to the super graph resulting from the given configuration
  - All input ports should be connected to an output port
  - The components present in the super graph are those composing the selected graphs
  - The super graph contains all selected graphs.

Design alternatives being formally defined, it is possible to define in Lightning a domain specific visualization to represent them graphically.

C. Domain Specific Visualization

The domain specific visualization of RPSL design alternatives is given as a model transformation from the Alternative model to a visual language model.

This transformation is expressed in F-Alloy [19], a sub-language of Alloy allowing the specification of efficiently computable model transformations[4]. An excerpt of such transformation is given in Listing 4.

```
module VisualisationTransformation
open AlternativeModel
open VisualLanguageModel

one sig CREATE{
  mainFrame : Component -> INVISIBLE_CONTAINER,
  inputFrame: ProcessingComponent -> INVISIBLE_CONTAINER,
  component : Component -> RECTANGLE,
  inputPort: Input -> RECTANGLE,
    outputFrame: Component -> INVISIBLE_CONTAINER,
  outputPort: Output -> RECTANGLE,
  arc: Output -> Input -> CONNECTOR,
}
pred guard_component(c:Component) {
  c in SuperGraph.components
}
pred value_component(c:Component, r:RECTANGLE) {
  r.color=(c.weight=1 implies WHITE else (c.weight=2
     implies YELLOW else ORANGE))
}
pred guard_arc(o:Output, i:Input) {
  o->i in SuperGraph.connections
}
pred value_arc(o:Output, i:Input, c:CONNECTOR) {
  c.source=CREATE.outputPort[o]
  c.target=CREATE.inputPort[i]
  c.color=RED
}
```

Listing 4. Excerpt of an F-Alloy transformation from Alternative Model to Visual Language Model

This excerpt contains all the mappings used to define how the super graph of a design alternative is to be rendered and a selection of two pre and post conditions (called guard and value predicates) applying to two of those mappings, namely, `component` and `arc`. The `component` mapping defines, according to its associated guard predicate, that each component composing the `SuperGraph` resulting from the current configuration is to be rendered as a rectangle. The value predicate then assigns a color to that rectangle given the weight associated to the component it represents. The `arc` mapping and its associated guard and value predicate together define, that for each pair of output and input port connected in the resulting `SuperGraph`, a red connection has to be created between their visual representations (defined by the `inputPort` and `outputPort` mapping, respectively).

In the next section, we present how an RPSL designer can use this framework to explore design spaces expressed in RPSL using a real world application.

V. Case Study

In this section, we illustrate our approach, depicted in Fig. 3, with the help of a case study.

A. The Pick & Place Case Study

The Pick & Place case study we use to illustrate our approach has been developed in the context of a recent robot competition, namely RoboCup@Work [20]. In this case study, a youBot mobile manipulation robot (see Fig. 1) is deployed in a factory-like environment which is composed of service areas. Each service area represents a region of the factory having a specific purpose for a particular task. For example, areas to load objects, to insert objects into object-specific cavities and to place objects into containers. Depending on a goal specification given by some factory worker the task of the robot is to pick objects such as screws, nuts and profiles from containers and to place and eventually insert them at corresponding service areas.

The functional variability of this scenario is given in RPSL in Listing 5 and depicted graphically in the upper-part of Fig. 5.

```
rpsl.feature_model do
  name "Pick and Place"
  add_feature "Application", :is_root
  add_feature "ServiceArea", :is_mandatory ,
                             :child_of = "Application"
  add_feature "ObDetection", :child_of = "Application"
  add_feature "ObRecognition", :requires = "ObDetection",
                               :child_of = "Application"

  add_feature "ContRecognition", :child_of = "Application"
  add_feature "CavRecognition", :child_of = "Application"
end
```

Listing 5. Feature Model used in the Pick & Place case study specified in RPSL

The feature model contains five leaf features representing in the same order the following perceptual functionality required for the pick and place task:

- The service area detection feature allows to delimit the service area by detecting the dominant plane in the surrounding of the robot. This information is required by all other features as objects, container and cavities are all lying on this dominant plane.
- The object detection feature provides a bounding box for each object present in the service area.

[4]Lightning relies on this formalism to efficiently provide its domain specific visualization support.

- The object recognition feature provides, when possible, a pose and a label for each detected object. Object detection is thus required by this feature.
- The container recognition feature provides, when possible, a pose and a bounding box for each container present in the service area
- the cavitiy recognition feature provides, when possible, a pose for each cavity present in the service area

The RPSL specification of a perception graph associated to the Service Area feature is given in Listing 6.

```
rpsl.sensor_component do
   name "Kinect"
   add_port :out, "outCloud", "xyzRGB"
end

rpsl.processing_component do
   name "PlaneDetect"
   add_port :in, "inCloud", "xyzRGB"
   add_port :out, "outPlane", "Plane"
end

rpsl.perception_graph do
   name "Service Area 1"
   connect "Kinect", "outCloud", "PlaneDetect", "inCloud"
end
```

Listing 6. a Perception Graph implementing the Service Area feature expressed in RPSL

This perception graph specification proposes to perform the service area detection by using a kinect and a plane detection algorithm (e.g. RANSAC). Note that the kinect is declared as a sensor component whose output port is called outcloud (typed xyzRGB) and the plane detection algorithm is declared as a processing component whose input port is called inCloud (typed xyzRGB) and output port is called outPlane (typed Plane).

We note that this perception graph composes the super graph depicted in Fig. 5 as it is a possible implementation of the selected feature `ServiceArea`.

B. RPSL to Alloy

In section IV, we have provided a set of Alloy models providing a generic definition of RPSL artifacts. The first step in using our approach is the translation of RPSL specifications, like the ones given in Listing 5 and 6 , into Alloy models conforming to those generic definitions and constrained so that the only instance obtainable by Alloy analysis corresponds to the given RPSL specification. This translation is done automatically as it is very straightforward: each element of the specification is declared as a singleton extending the appropriate type and fields of those elements having their value bound by constraints. We illustrate this translation by giving in Listing 7 the Alloy model derived from the RPSL specification of the perception graph given in Listing 6.

C. Generation and Visualization of Design Alternatives

Design Alternatives are obtained by performing an Alloy analysis on the alternative model (e.g., Listing 3).

The Alloy analysis produces conforming instances by translating constraints of the analyzed model into a boolean formula which is then solved by an off-the-shelf SAT-solver (miniSAT, SAT4J, a.s.o. . .).

```
module Service1Graph
open AbstractPerceptualGraph

one sig Service1Graph extends PerceptualGraph{}{
        components = PlaneDetect + Kinect
        connections = outCloud -> inCloud
        compGraph = Kinect -> PlaneDetect
}
one sig  inCloud extends Input {}{
        type=xyzRGB
}
one sig  outPlane extends Output {}{
        type=Plane
}
one sig  outCloud extends Output {}{
        type=xyzRGB
}
one  sig Kinect  extends SensorComponent{}{
  input = none and output = outCloud
 }
one sig PlaneDetect extends ProcessingComponent {}{
  input = inCloud and output = outPlane
}
```

Listing 7. Alloy translation of the RPSL snippet given in Listing 6

In our framework, instances obtained by Alloy analysis are then given as input to the F-Alloy interpreter embedded in the Lightning tool along with the model transformation given in Listing 4. This interpreter builds from the transformation specification and its input the corresponding visual language instance that can then be rendered to the user. Such a visual feedback, obtained from the analysis of the Alternative Model (given in Listing 3) is depicted in Fig. 5. In this figure, we see the domain specific visualization of an Alternative Model instance. The tree in the upper part of the visualization represents the feature tree of this case study, in which selected features are highlighted in green. For readability's sake the alternative model to Visual language transformation was modified to mask requirement arrows. This change can be undone at anytime by the user. The lower part of the visualization depicts the super graph resulting from the composition of perception graphs mapped in the resolution model to the highlighted selected features. Note that this super graph was not specified in RPSL and is resulting from the Alloy analysis of those well constrained models.

The red and green squares surrounding each component are their output and input ports, respectively. Note that, the `PlaneDetect` component appears in yellow as it is assigned a weight of 2 in our RPSL specification. The black box in the top left corner lists additional properties of the selected configuration (here the total weight of the super graph implementing the features selected).

D. Guiding the Exploration

Domain experts can further guide the exploration by defining additional constraints in the alternative model or by changing the weights assigned to each component. Adding constraints has as effect to reduce the number of instances conforming to the Alternative model, thus narrowing the set of design alternatives to be considered. This mechanism becomes useful when the domain expert is interested in design alternatives showcasing specific properties. We list

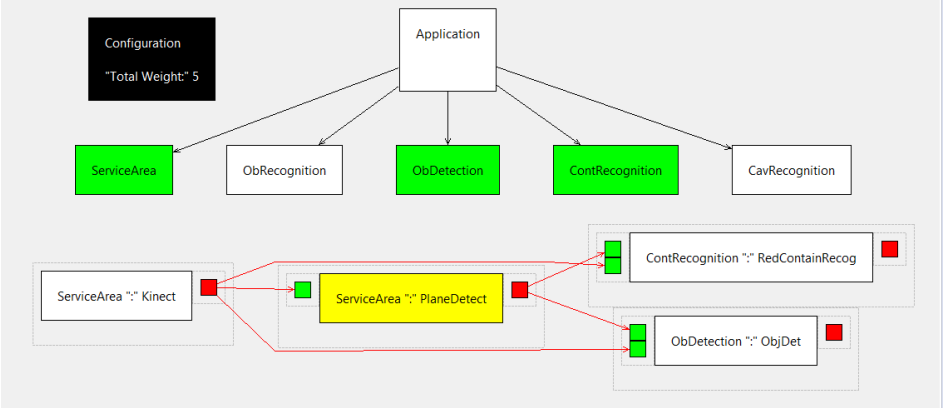

Fig. 5. Visualization of a given Pick & Place configuration obtained with the framework

in the following some examples of constraints used to guide the DSE of our case study:

- Specific Feature/Component Selection: We were interested in reviewing all the design alternatives implementing the `ObRecognition` feature and whose supergraph contains a sensor providing `xyzRGB` data in order to ensure that ObjRecognition can be carried for this kind of input data. The Alloy constraints used to express this is:

```
ObRecognition in Configuration.selectedFeatures
and some (SensorComponent & xyzRGB.~type.~output )&
SuperGraph.components
```

- Optimal Solution Selection (with respect to the attributed weights): We were interested in reviewing design alternatives with exactly three features implemented and having a minimal weight.

`#Configuration.selectedFeatures=3 and SuperGraph.getWeight[]< n` with n incrementally increasing until a design alternative is found.

VI. Related Work and Discussion

We have presented in this paper a solution to the design space exploration problem by formalizing a robotics DSL – namely RPSL – in a formal modeling language – namely Alloy – with the intent of using Alloy's analysis mechanism to obtain all possible design alternatives belonging to a certain design space. As the proposed approach builds up on methods and tools from several domains, namely DSE, Robotics and Software Engineering we will discuss related work individually bellow.

A. Design Space Exploration

In the embedded systems and design automation domain the DSE problem is very often framed as an optimization problem. Here, one or more, often non-functional, conflicting objectives such as timing, energy and costs need to be optimized for a particular task such as the co-synthesis of hardware/software units for embedded systems (see e.g. [21] [22]). In contrast, our approach is focused on exploring design alternatives which differ in their structural appearance. This appearance carries enough domain knowledge so that a domain expert can seamlessly perform DSE as shown in Sec. V, without prior knowledge of intermediary languages.

B. Robotics

Our approach to DSE in the presence of functional and architectural variability is based on the meta-models introduced by Gherardi and Brugali [3] [4]. Similar to their work we also employ resolution models to compose functional with architectural variability, but in a RPSL-specific manner –i.e., an architecture is represented as a perception graph. In addition, we make the following small, yet important changes: a feature resolution in our work yields one or more perception graphs in order to express different architectures, with different characteristics for the same feature. This allows us to compose different perception graphs in the exploration phase. Furthermore, we rigorously specified the structural constraints of the feature, perception graph and resolution models in order to prevent the creation of erroneous design alternatives. Interestingly, from a DSL developer perspective the redefinition of the RPSL meta-models in Alloy revealed some errors in the original RPSL implementation such as the possibility to select non-child features.

C. Software Engineering

Works in [23] have already shown that design space exploration can benefit from approaches based on Model-Driven Engineering advances. More precisely, authors have shown that a framework allowing the definition of a specification language and the exploration of design spaces defined in that language can be implemented.

Our work provides an alternative solution to the problem of providing a DSE framework by reusing already existing tools and technologies rather than implementing a framework from scratch.

The advantages of our approach is that, the RPSL to Alloy transformation being provided, an RPSL expert can directly perform design space exploration without learning any new intermediate language. Guiding the design space exploration

through the addition of Alloy constraints requires some basic Alloy knowledge which can be seen as a limitation to our approach. Nevertheless, a general trend is to define graphical representations for constraint languages [24], [25] to make them more user-friendly, suggesting the possibility that such a language could also be defined for Alloy.

Our work also differ from [23] with the domain specific visualization of design alternatives provided by the framework. The visualization definition can be used out of the box by neophytes but can also easily be modified by Alloy experts. It is still to be determined whether or not the visualisation we provide is suitable for other case studies.

VII. Conclusion and Future Work

In this paper, we proposed a model-based approach for design space exploration which enables a domain expert to systematically explore the design space of robot perception systems. Such a design space is composed by functional and architectural variability. In this work we have presented an extension of the RPSL language allowing the expression of both those variabilities. We have also presented an approach to the exploration of design spaces specified in RPSL relying on existing MDE technologies, namely the formal language Alloy and the Lightning language workbench and have validated this approach through the DSE of the Pick & Place case study. The efficiency of the provided DSE framework was showcased when design alternatives, such as the one depicted in Fig. 5, were generated directly from the RPSL specification of the case study, without intervention of the expert.

A limitation of the current approach is the restriction of the design space to two types of variabilities, namely architectural and functional variability. In reality, a domain expert would also be interested in taking into consideration other design space dimensions (e.g., deployment variability). In a future work we will investigate the definition of multi-dimensional design spaces. This could be achieved by using jointly several DSLs or by providing a general language to express them. It would then be interesting to see whether or not the approach proposed in this paper can still be applied to explore such multi-dimentional design spaces.

References

[1] D. Brugali and A. Shakhimardanov, "Component-based robotic engineering (part ii)," *IEEE Robotics Automation Magazine*, vol. 17, no. 1, pp. 100–112, March 2010.

[2] C. Schlegel, A. Steck, D. Brugali, and A. Knoll, "Design abstraction and processes in robotics: From code-driven to model-driven engineering," in *2nd International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Darmstadt, Germany, 2010.

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

[4] L. Gherardi, "Variability modeling and resolution in component-based robotics systems," Ph.D. dissertation, 2013.

[5] S. Moisan, J.-P. Rigault, M. Acher, P. Collet, and P. Lahire, "Run Time Adaptation of Video-Surveillance Systems: A software Modeling Approach," in *ICVS, 8th International Conference on Computer Vision Systems*, 2011.

[6] B. Siciliano and O. Khatib, *Springer Handbook of Robotics*, 2007.

[7] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

[8] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[9] E. Kang, E. Jackson, and W. Schulte, "An approach for effective design space exploration," in *Proceedings of the 16th Monterey Conference on Foundations of Computer Software: Modeling, Development, and Verification of Adaptive Systems*, 2011.

[10] A. Nordmann, N. Hochgeschwender, D. Wiegand, and S. Wrede, "A survey on domain-specific modeling and languages in robotics," *Journal for Software Engineering Robotics (JOSER)*, 2016.

[11] A. Ramaswamy, B. Monsuez, and A. Tapus, "Saferobots: A model-driven framework for developing robotic systems," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sept 2014, pp. 1517–1524.

[12] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "Robotml, a domain-specific language to design, simulate and deploy robotic applications," in *Proceedings of the Third International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, ser. SIMPAR'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 149–160.

[13] Ortiz, al.Francisco J., D. Alonso, F. Rosique, F. Sánchez-Ledesma, and J. A. Pastor, "A component-based meta-model and framework in the model driven toolchain c-forge," in *Proceedings of the 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, 2014, pp. 340–351.

[14] N. Hochgeschwender, S. Schneider, H. Voos, and G. K. Kraetzschmar, "Declarative specification of robot perception architectures," in *Simulation, Modeling, and Programming for Autonomous Robots: 4th International Conference*.

[15] L. Gammaitoni, P. Kelsen, and C. Glodt, "Designing languages using lightning," in *International Conference on Software Language Engineering*, 2015.

[16] D. Jackson, *Software abstractions*, 2012.

[17] L. Gammaitoni and P. Kelsen, "Domain-Specific Visualization of Alloy Instances," in *ABZ*, 2014.

[18] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*, 2008.

[19] L. Gammaitoni and P. Kelsen, "F-alloy: An alloy based model transformation language," in *International Conference on Theory and Practice of Model Transformations*, 2015.

[20] G. K. Kraetzschmar, N. Hochgeschwender, W. Nowak, F. Hegger, S. Schneider, R. Dwiputra, J. Berghofer, and R. Bischoff, "RoboCup@Work: Competing for the Factory of the Future," in *Proceedings of the 18th RoboCup International Symposium*, 2014.

[21] H. Oh and S. Ha, "Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, 2002.

[22] R. Hourani, R. Jenkal, W. R. Davis, and W. Alexander, "Automated design space exploration for dsp applications," *Journal of Signal Processing Systems*, 2009.

[23] T. Saxena and G. Karsai, "Mde-based approach for generalizing design space exploration," in *International Conference on Model Driven Engineering Languages and Systems*, 2010, pp. 46–60.

[24] C. Kiesner, G. Taentzer, and J. Winkelmann, "Visual ocl: A visual notation of the object constraint language," *http://tfs.cs.tu-berlin.de/vocl*, 2002.

[25] S. Kent, "Constraint diagrams: visualizing invariants in object-oriented models," in *ACM SIGPLAN Notices*, 1997.