Software Variability Composition and Abstraction in Robot Control Systems

Davide Brugali[1], Mauro Valota[1]

University of Bergamo, Dalmine, Italy, brugali@unibg.it, m.valota@studenti.unibg.it

Abstract. Control systems for autonomous robots are concurrent, distributed, embedded, real-time and data intensive software systems. A real-world robot control system is composed of tens of software components. For each component providing robotic functionality, tens of different implementations may be available.
The difficult challenge in robotic system engineering consists in selecting a coherent set of components, which provide the functionality required by the application requirements, taking into account their mutual dependencies. This challenge is exacerbated by the fact that robotics system integrators and application developers are usually not specifically trained in software engineering.
Current approaches to variability management in complex software systems consists in explicitly modeling variation points and variants in software architectures in terms of Feature Models.
The main contribution of this paper is the definition of a set of models and modeling tools that allow the hierarchical composition of Feature Models, which use specialized vocabularies for robotic experts with different skills and expertise.

1 Introduction

Control systems for autonomous robots are concurrent, distributed, embedded, real-time and data intensive software systems. The computational hardware of an autonomous robot is typically interfaced to a multitude of sensors and actuators, and has severe constraints on computational resources, storage, and power. Computational performance is a major requirement, especially for autonomous robots, which process large volumes of sensory information and have to react in a timely fashion to events occurring in the human environment.

In recent years, a variety of software frameworks have been specifically designed for developing robot control systems that are designed as (logically) distributed component-based systems (see [9] for a survey). Currently, the Robot Operating System [1] is the most widely used robotic framework in research and development. It offers mechanisms for real-time execution, synchronous and asynchronous communication, data flow and control flow management.

A real-world robot control system is composed of tens of components. For each component providing a robot functionality, tens of different implementations may be available. The initial release of ROS in year 2010 already contained

hundreds of open source packages (collections of nodes) stored in 15 repositories around the world [1].

Clearly, building complex control applications is a matter of system integration more than of capabilities implementation. The difficult challenge consists in selecting a coherent set of components that provide the required functionality taking into account their mutual dependencies.

In previous papers [14, 15] we have presented the HyperFlex Model-driven toolchain and approach for the design of software product lines for autonomous robotic systems.

The key characteristics of HyperFlex are the support to the design and composition of architectural models of component-based functional subsystems, the possibility to symbolically represent the variability of individual functional subsystems using the Feature Models formalism, and the automatic configuration of functional subsystems according to selected features. The HyperFlex approach builds on our experience in developing software architectures for robotic control systems in the context of the EU FP7 BRICS project [7].

The novel contribution of this paper is the description of the new functionality of the HyperFlex toolchain and the definition of a set of guidelines that enable the composition and abstraction of the variability models of individual functional subsystems and of the integrated control system.

Feature models usually don't scale up when the number of variation points and variants becomes substantial, because a single and huge feature model is too complex to maintain and to be understood and processed by humans. Our aim is to simplify the system configuration phase by supporting the definition of feature models at multiple levels of abstraction using specialized vocabularies for each expert involved in system configuration.

System configuration is a crucial phase, which requires to select, integrate, and fine tune the robot functionalities (developed by domain experts) according to the available resources (requiring maintenance by qualified engineers), the environment conditions (often beyond the control of the application engineer), and the task to be performed (often specified by unskilled users).

The paper is structured as follows. Section 1.1 presents the background information about the HyperFlex approach and toolchain by means of a simple example. Section 2 reports on the related works. Section 3 presents the novel contribution of this paper. It extends the previous example with two case studies of variability composition and illustrates the new models and meta-models of the HyperFlex toolchain. The relevant conclusions are presented in Section 4.

1.1 The HyperFlex Approach and Toolchain

HyperFlex is a Model-driven engineering (MDE) [21] environment (available open source on GitHub [2, 15]) that supports the development of flexible and configurable robotic control systems. It builds on state-of-the-art approaches in software variability modeling and resolution as described in Section 2 and consists in a set of Eclipse plugins for the definition and manipulation of three types of software models:

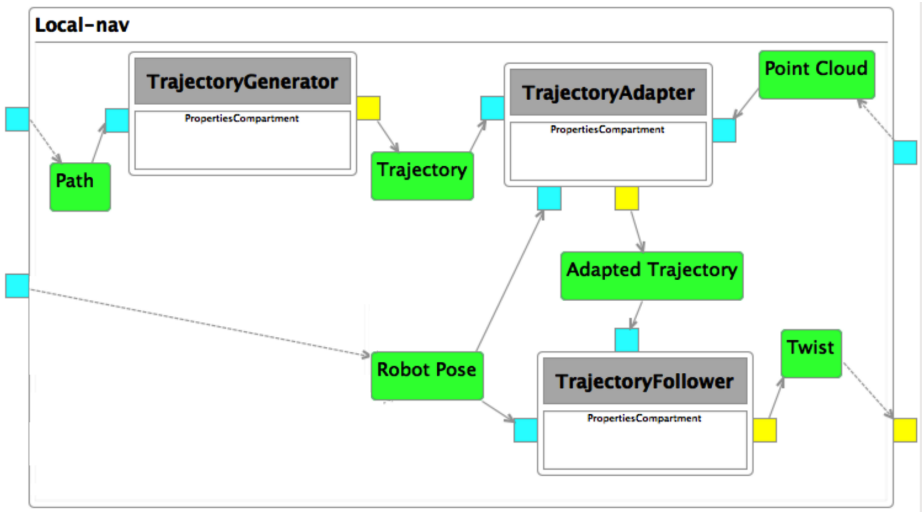

Fig. 1: The architectural model of the Local Navigation system

- *Architectural Models* represent the structure of control systems in terms of component interfaces, component implementations, and component connectors. The HyperFlex approach promotes the design and composition of domain-specific software architectures for common robotic functionality (e.g. robot navigation), which capture the variability in robotic technologies (e.g. various algorithms for trajectory generation).
- *Feature Models* symbolically represent the variant features [22] of a control system; symbols may indicate individual robot functionality (e.g. marker-based localization) or concepts that are relevant in the application domain, such as the type of items that the robot has to transport (e.g. liquid, fragile, etc.), which affect the configuration of the control system.
- *Resolution Models* define model-to-model transformations, which allow to automatically configuring the architecture and functionality of a control system based on required features. Eventually, the configured architectural model is used to deploy the control system on a specific robotic platform.

As an example, Figure 1 represents the architectural model of the Local Navigation system of an autonomous mobile robot. Local navigation is the set of functionality that allow the robot to autonomously move from its current position towards a goal position, while avoiding collisions with unexpected obstacles (i.e. moving people) in an indoor environment such as a hospital or a museum.

Architectural components define provided and required interfaces (depicted respectively as yellow and cyan squares), which can be connected by means of registers (green rectangles) according to the topic-based publish/subscriber paradigm supported by the ROS framework.

The *TrajectoryGenerator* receives as input a robot path and produces as output a trajectory, which specifies linear and angular velocity for each one of

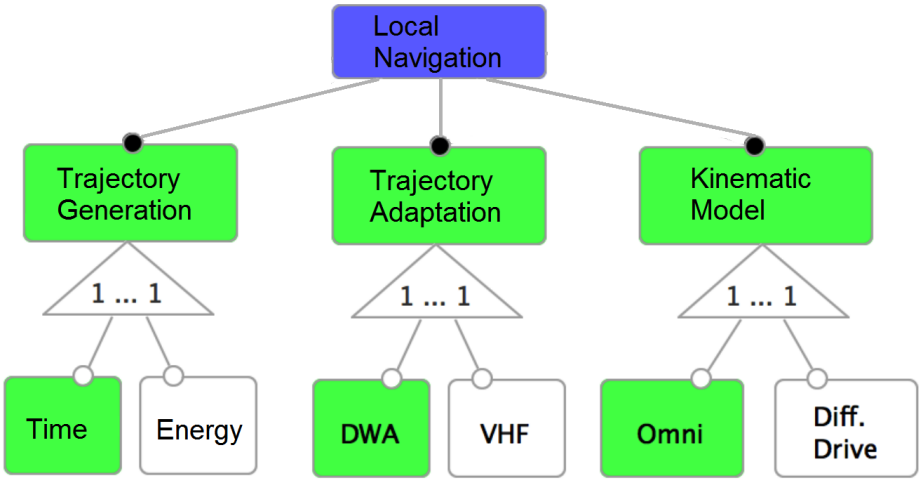

Fig. 2: The Feature Model of the Local Navigation system

its waypoints. The *TrajectoryAdapter* receives as input the generated trajectory and produces as output a modified trajectory that avoids unexpected obstacles detected by the robot sensors. The *TrajectoryFollower* receives as input a trajectory and implements a feedback control loop that periodically reads the current robot pose and generates a twist (i.e. linear and angular velocities along the three axis) to minimize the distance to the path.

Figure 2 represents the *Feature Model* of the Local Navigation System, where the selected features are marked in green. The algorithms for *Trajectory Generation* are usually named by the function that is optimized, namely: minimum jerk or minimum time. The algorithms for *Trajectory Adaptation* are classified as reactive, i.e. they use only sensor data to generate robot control commands, such as the Vector Field Histogram (VFH), or as deliberative, i.e. they evaluate alternative paths and choose the best in terms safe distance to obstacles and minimum distance to the original path, such as the Dynamic Windows Approach (DWA).

The robot kinematics (Differential Drive or Omnidirectional) affects the implementation of all the three functionality.

Figure 3 represents the *Resolution Model* for the Local Navigation System. As an example, the selection of feature *DWA* triggers the four transformations that are indicated by the red arrow. They create the connections between the ports of the *TrajectoryAdapter* component that implements the DWA algorithm and the rest of the system.

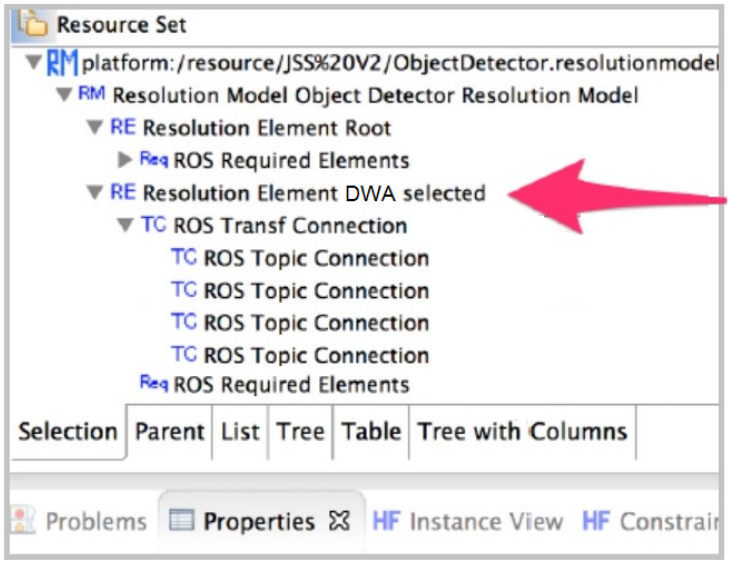

Fig. 3: A portion of the Resolution Model of the Local Navigation system

2 Related Works

The following subsections illustrates the related works in three areas: (i) approaches to variability modeling, (ii) approaches to Feature Models composition, and (iii) variability modeling approaches in robotics.

2.1 MDE for software variability management

Hyperflex follows a common approach to model the variability of a software system, which consists in defining four models: (a) the architectural model defines the software architecture of the system in terms of implementation modules (classes, aspects, agents, components) and their interconnections; (b) the variability model describes the functional variability of the system using a symbolic representation (e.g. feature models [11] or the OMG CVL language [17]); (c) the resolution model defines the mapping between the symbols of the variability model and the implementation modules of the architectural model; (d) the configuration model consists in a specific set of variants for the variation points defined in the variability model.

In our approach the first three models are completely orthogonal, i.e. they can vary independently, while the configuration model is an instance of the variability model.

The *Talents* approach [19] aims at modeling and composing reusable functional features for configuring the behavior of a software system. A graphical

environment simplifies feature composition. The Talents approach models functional features at the level of instances of a class in an object-oriented programming language. In contrast, HyperFlex models functional features at the level of software components and component-based systems and thus is more adequate to model variability in robotic control systems.

In GenArch [10] the variability model and the configuration model are represented using the same meta-model, while in OMG CVL [17] the variability model and the resolution model are not explicitly separated.

The approach described in [16] defines three modeling categories, i.e. *Commonality*, *Variability*, and *Configuration*. The *Commonality* describes the architecture of a system, in terms of components, sub-components, ports and connectors. These architectural elements can be enriched with variation points, which represent the *Variability* and define how the common parts can be configured. For example, a variant for a component variation point can specify that a new sub-component has to be included in the component. Finally, the *Configuration* describes the selection of variants for all the variation points. The architectural model and the configuration conform to the MontiArc meta-model. Differently from HyperFlex, this approach condenses all the information in a single model.

2.2 Feature Models composition

A survey of recent papers that propose techniques for Feature Model composition can be found in [5]. The surveyed approaches mostly focus on model composition techniques that are dedicated to support semantics preserving model composition. HyperFlex is a complementary approach, as it focuses on the automatic generation of Feature Model instances in a tree of variability models that are assumed to be semantically coherent and correct.

The approach described in [20] defines a set of composition constraints that specify how the features of the lower level feature models have to be selected according to the configuration of the higher level feature model. Differently from our approach, they don't adopt a component model for the architecture.

The *Compositional Variability* [4] approach supports the hierarchical composition of architectural models and feature models. The associations between a high-level feature model and a low level feature models are defined by means of the so called *Configuration Links*, which are similar to the feature dependencies defined in the HyperFlex *Refinement Model*. Differently from HyperFlex, this approach defines an abstract component model and does not provide the capabilities for modeling domain-specific component-based systems.

2.3 Variability Modeling Approaches in Robotics

In recent years, several model-driven approaches and tools for the development of robotic systems have been proposed, such as OpenRTM [6], Proteus [12], and Smartsoft [18].

In particular, the SmartSoft model-driven approach supports robotics variability management by modeling functional and non-functional properties of

robot control system. The approach addresses two orthogonal levels of variability by means of two domain specific languages: (a) the variability related to the operations required for completing a certain task and (b) the variability associated to the quality of service.

These two variability levels are more related to the execution of a specific application (in the paper the example is a robot delivering coffee), while the HyperFlex approach supports modeling the variability of functional systems and the variability of the family of applications resulting from the composition of these functional systems.

3 Variability Composition and Abstraction with HyperFlex

HyperFlex allows structuring a complex control system as a hierarchical composition of functional systems. As an example, Figure 4 shows the architecture of the *Robot Navigation* composite system, which integrates the *Local-nav* subsystem described in the previous section, with either the *Marker-nav* subsystem or the *Map-nav* subsystem. These components implement two strategies (map-based and marker-based) for generating the robot path between a start position and a goal position that is passed to the *Local-nav* subsystem.

If a geometric map of the environment is available, the robot is able to plan a geometric path in the free space. This strategy requires the robot to estimate its current position with respect to the map reference frame accurately. On the contrary, if artificial visual markers have been placed on the floor or on the walls, a camera mounted on the robot can detect them and the robot can navigate by following a path defined by a specific sequence of visual markers. In this case, the robot needs only to estimate its relative position with respect to the next visual marker. Figure 4 represents the situation where the *Local-nav* subsystem receives the robot path from the *Marker-nav* subsystem.

An interesting challenge that needs to be faced when using feature models to represent the variability of a software product line is the definition of an appropriate vocabulary for naming variation points and variants.

The clear separation of the symbolic representation of the system variability from its architectural model allows the definition of multiple Features Models for the same software system that are meaningful for system integrators with different needs and expertise.

In this context, HyperFlex allows the composition of Feature Models according to two different strategies, that we call *Bottom-up functionality composition* and *Top-down specification refinement*. These two composition strategies are meant for two types of stakeholders in software development for robotics:

- The community of researchers, who keep implementing new algorithms for common robot functionalities as open source libraries, need tools that simplify the configuration of robotic control systems during test trials in various operational conditions.

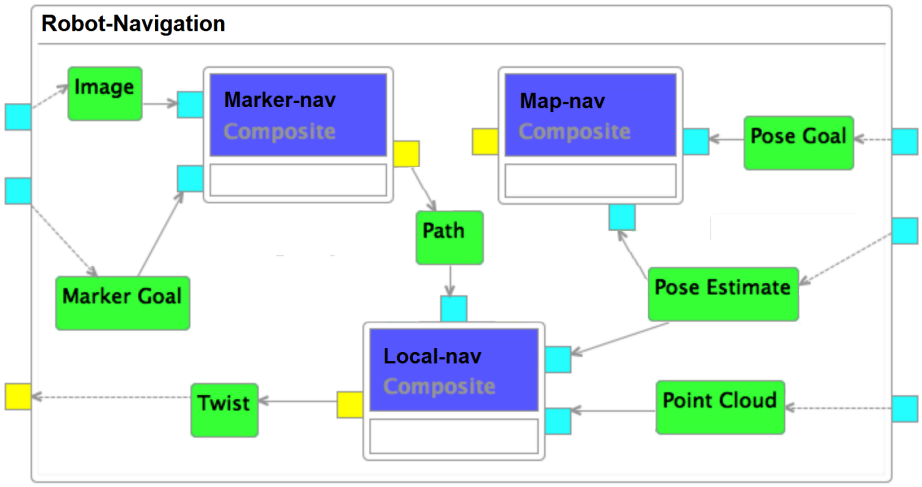

Fig. 4: The Architectural Model of the composite Navigation system

– System integrators, who are expert in specific application domains, need tools for the configuration of robot control systems according to specific application requirements.

3.1 Bottom-up Functionality Composition

Typically, the expert in robotic functionalities is interested in a representation of the control system variability that highlights the different algorithms implemented in the robot control system. For example, in [8] we have analyzed the variability in software library that implement motion planning algorithms. In this context, the relevant features are the type of bounding-box used by the collision-detection algorithm, the sampling strategy, and the type of kinematic model (e.g. single chain, multiple end-effectors).

Feature Models can be hierarchically composed to reflect the composition of functional systems. At each level the feature names abstract the relevant concepts of the corresponding functional system composition level.

For example, Figure 5 shows three Feature Models that represent the variability of the composite Navigation system depicted in Figure 4. In particular, the Feature Model of Figure 5.A has two leaf features, namely *Marker Navigation* and *Map Navigation*, that represent two alternative variants of the *Navigation Strategy* variation point.

When the system engineer selects the *Marker Navigation* feature, the HyperFlex tool creates a new instance of the corresponding Feature Model depicted in Figure 5.C. Subsequently, the system engineer can select features of lower-level Feature Models for specific configuration properties. For example, in Figure 5.C the manual selection of Feature *Aruco* triggers a model-to-model transformation that configures the architectural connectors of the component implementing the

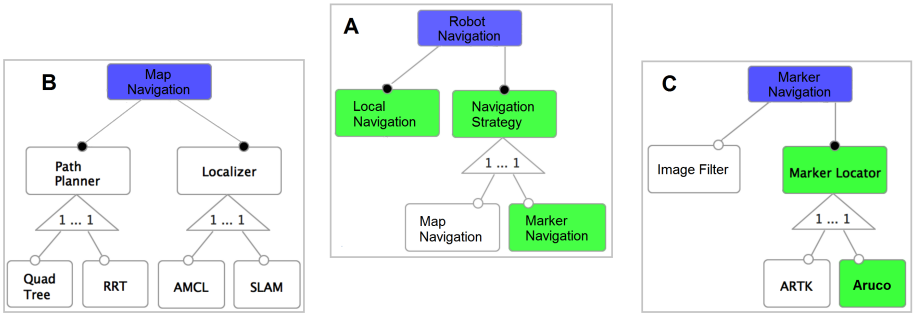

Fig. 5: The Feature Models of the (A) composite Navigation system, (B) the Marker-based Navigation subsystem, and (C) the Map-based Navigation System

Aruco algorithm [13] for marker localization. The approach is not limited to two levels but can be hierarchically extended. Systems made of subsystems can be further composed in order to design more complex systems.

3.2 Top-down Specification Refinement

The application domain expert is interested in a representation of the system variability that specifies the application requirements supported by the robot control system more than its specific functionality. Figure 6 depicts the feature model of a robot control system for logistics applications. It is structured around three main dimensions of variability in application requirements, namely the type of environment, the type of load that the robot should handle, and the available equipment.

For this purpose, HyperFlex supports the composition of Feature Diagrams representing variability ad different level of abstractions. At each level the feature names abstract the relevant concepts of the specific domain: low-level names represent functional and technical terms while high level names are closer to the application requirements. This approach ensures that the terminology is well known by the system integrators that operates on a specific level.

During the variability resolution process, the application domain expert operates only on the highest-level Feature Model and the selected features trigger the automatic selection of features in the lower levels Feature Models.

For example, the robot operational environment could be a space with narrow passages and only static obstacles (see feature *Warehouse* in Figure 6) or populated by moving obstacles in crowded areas (see feature *Airport* in Figure 6). According to the operational environment, the robot should be configured with different algorithms: a slow and complete motion planner is adequate for moving among static obstacles in narrow passages; instead, a fast and approximate motion planner is needed for dynamic environments.

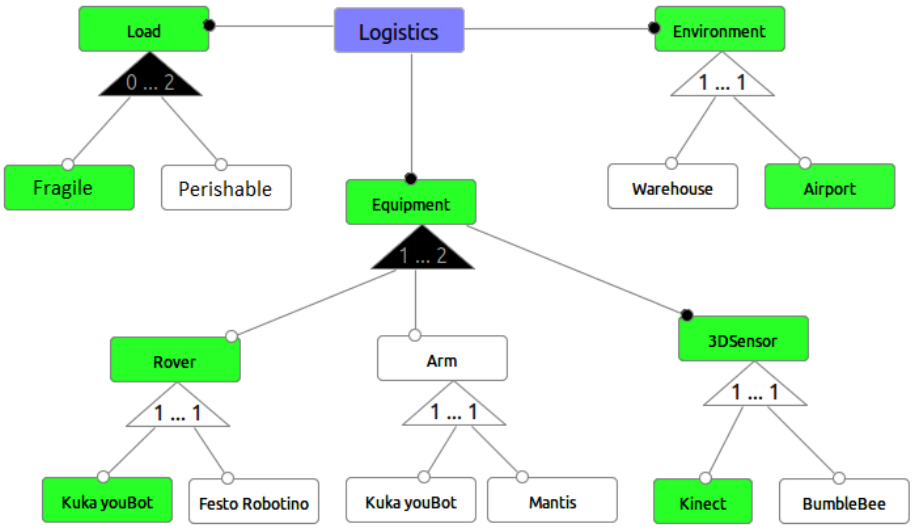

Fig. 6: The Feature Models of the Logistics application

HyperFlex provides a tool that allows to specify that the feature *Warehouse* in the *Logistics* FM is linked to the feature *QuadTree* in the *Map Navigation* FM of Figure 5.B, while the feature *Airport* is linked to feature *RRT*.

If the logistic task consists in transporting objects, the system integrator should select one of the available rovers. Here, the selection of feature *Kuka youbot* in the *Local Navigation* FM will trigger the selection of feature *Omni* in *Logistics* FM of Figure 2.A, which corresponds to the algorithms for omnidirectional rovers. Similarly, if the feature *Fragile* is selected in the FM of Figure 6) then the feature *Jerk* is automatically selected in the FM of Figure 2.A.

Clearly, the system integrators can focus on the specification of the application requirements and should not be concerned with the functionality that implement them.

3.3 Refinement Model

In this section we illustrate the models, meta-models, and tools that allow the composition of Feature Models and the automatic generation of their instances according to the composition strategies described in the previous section.

The proposed appraoch consists in defining a new transformation model (called *Refinement Model*) that specifies links between the features of a parent Feature Model and the features of its child Feature Models. Figure 7 shows an example, where *FM_A* is a parent Feature Model and *FM_B* and *FM_C* are child Feature Models.

This approach does not require to modify the meta-model defined for Feature Models and thus promotes the reuse of existing Feature Models.

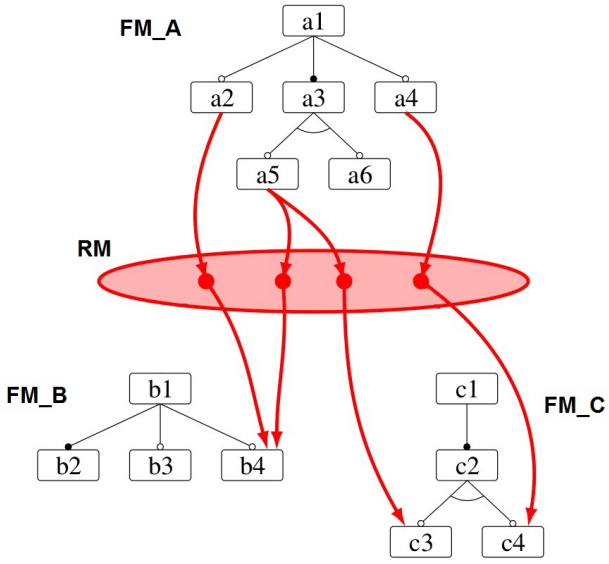

Fig. 7: The links between the features of different Feature Models

Figure 8 depicts the metamodel that we have defined for creating Refinement Models. The top level class is the *FeatureRefinementModel*, which has a link to the parent Feature Model and encapsulates a list of instances of class *FeatureRefinementPolicy*, one for each child Feature Model.

A Feature refinement policy is a collection of *FeatureRefinementElement*, which store the links between a feature of the associated parent Feature Model and a set of features of the associated child Feature Models. The proposed metamodel imposes the following rules to the definition and use of Feature Refinement Models.

When a new istance of the parent Feature Model is created, the istances of the child Feature Models should be empty, i.e. none of the features is selected. This condition allows to create instances of the child feature models incrementally.

When a feature of the parent FM is selected, all the linked features should be included in the istance of the child FM. This means that it is not possible to define *FeatureRefinementElement*s that remove a previously inserted feature.

If a feature of the parent FM (e.g. feature a5 in Fig. 7) needs to be linked to several features of different child FMs (e.g. features b4 and c3), one *FeatureRefinementElement* should be created for each child FM and added to the corresponding *FeatureRefinementPolicy*.

Several features of the parent FM (e.g. features a2 and a5 in Fig. 7) can be linked to the same feature of a child FM (e.g. feature b4) by creating a *FeatureRefinementElement* for each feature of the parent FM and adding all of them to the same *FeatureRefinementPolicy* associated to the child FM.

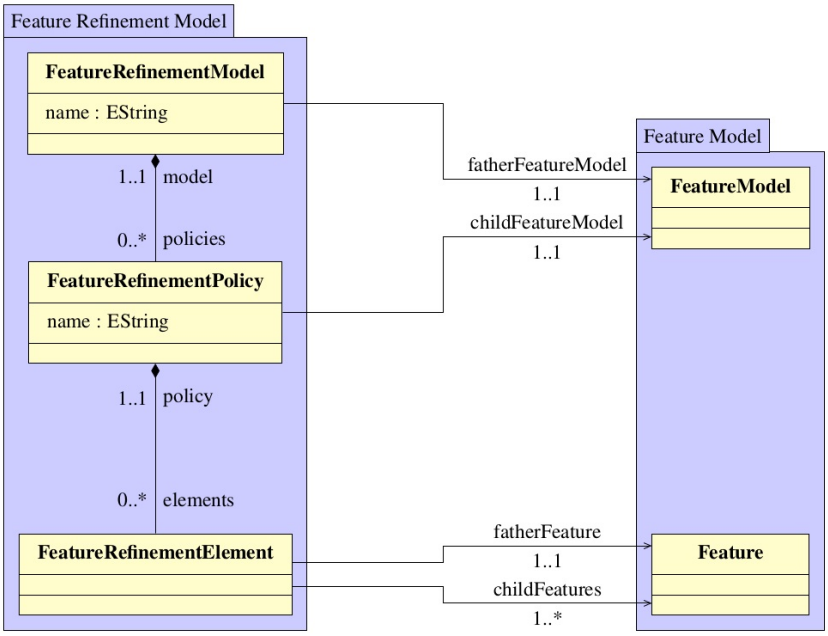

Fig. 8: The Feature Refinement Meta-Model

This set of rules allows to minimize the amount of memory used by the feature refinement tool, which needs to load only two Feature Models at a time (the parent FM and one child FM) during the feature refinement process. The tool takes as input an instance of the parent FM (created with the HyperFlex editor) and a *FeatureRefinementModel* associated to it to generate an instance of each child FM automatically.

It should be noted that some features of the parent FM (e.g. feature a6 in Fig. 7) might not be linked to any feature of the child FMs and vice versa (e.g. feature b3).

The former case corresponds to the situation where the parent FM is used to configure directly some variation points of a functional subsystem as in the example of Section 3.2. In this case, feature b3 would be associated to a model to model transformation of the subsystem architecture as described in Section 1.1. The latter case requires manual selection of some features of the child FM as in the example of Section 3.1.

Feature Models can include constraints that limit the set of possible combinations of selected features. For examples, features *c3* and *c4* in Figure 7 are mutually exclusive. It is not necessary to replicate the constraint in the parent Feature Model (i.e. *FM_A*), becuase the HyperFlex tool is able to report constraint violations in child FM to the user with the indication of the selected features in the parent FM that caused them.

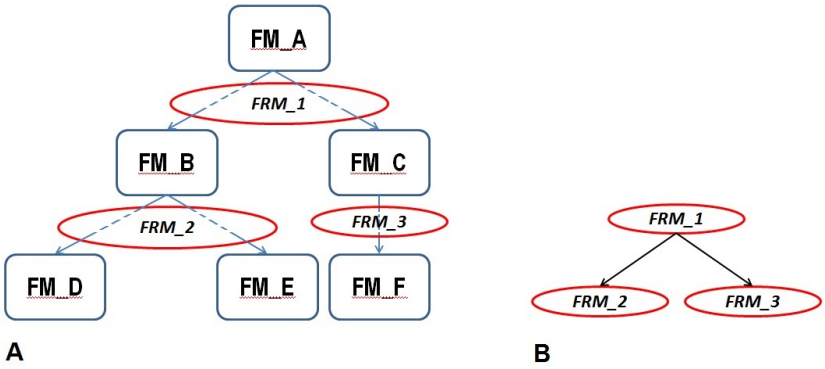

Fig. 9: The composition of Feature Refinement Models (A) and the Feature Refinement Tree (B)

The HyperFlex toolchain includes an Eclipse Wizard that supports the model designer in defining the *FeatureRefinementModels* by means of a set of intuitive Eclipse Forms.

3.4 Refinement Language

The Feature Refinement Model described in the previous section defines a tree structure between a parent Feature Model and a set of child Feature Models. Starting from a manual selection of features in the parent FM, the HyperFlex tool generates instances of the child Feature Models automatically.

This structure can be extended to trees with an arbitrary number of levels by connecting Feature Refinement Models hierarchically. Here, the hierarchy imposes an order according to which the Feature Refinement Models are processed in order to create an instance of each intermediate and leaf Feature Model.

Figure 9.A illustrates a simple example with six Feature Models (*FM_A*, ..., *FM_F*) and three Feature Refinement Models (*FRM_1*, ..., *FRM_3*). Figure 9.B shows the hierarchical dependencies between Feature Refinement Models.

The refinement process starts when an instance of Feature Model *FM_A* is created manually. The HyperFlex tool processes the *FeatureRefinementPolicy* and the *FeatureRefinementElement* defined in *FRM_1* and generates an instance of *FM_B* and *FM_C*. These instances are then used as input for processing the Feature Refinement Models *FRM_2* and *FRM_3* and generating an instance of the Feature models *FM_D*, *FM_E*, and *FM_F*.

Figure 10 shows the Xtext [3] grammar of the language used to define the tree structure of Feature Refinement Models. The keyword *Node* has an identifier (*ID*), a content, and a list of children nodes. The keyword *Content* indicates that each node of the tree can embed a *Feature Refinement Model*, a sub-tree, or the path (URI) to a file that stores a sub-tree. The keyword *Tree* indicates that there is the possibility to specify the algorithm for traversing its children nodes. Here,

```
grammar org.hyperflex.featurerefinementmodels.xtext.editor.FeatureRefinementLanguage
with org.eclipse.xtext.common.Terminals
generate featureRefinementLanguage
"http://www.hyperflex.org/featurerefinementmodels/xtext/editor/FeatureRefinementLanguage"

FRL_Root :
      aliases += (Alias)* 'ROOT' rootTree = Tree trees += Tree*;

Alias : Model | FRL_File;

Model : 'FEATURE_REFINEMENT_MODEL' name = ID importURI = STRING;

FRL_File :
      'FEATURE_REFINEMENT_LANGUAGE' name = ID importURI = STRING;
Tree :
      'TREE' name = ID ':' mode = ('BFS' | 'DFS' | 'SUB') '=' (rootNode = Node |
      multiNode ?= '{''ROOT' rootNode = Node nodes += (Node)* '}');

Child : '=>' node = [Node] ;

Node :
      'NODE' name = ID '(' ((content = Content ')') | (empty ?= ')')) childNodes += (Child)* ;
Content :
      (modelContent ?= 'MODEL' ':' model = [Model] | treeContent ?= 'TREE' ':' tree = [Tree]
       | frlContent ?= 'FRL' ':' file = [FRL_File] );
```

Fig. 10: Xtext Grammar for the Feature Refinement Language

BFS stands for Breadth-first search and *DFS* stands for Depth-first search. The third modality (i.e. *SUB*) is used for sub-trees and indicates that it should be used the search algorithm of the parent tree. Figure 11 exemplifies the use of the *Feature Refinement Language* to build a tree of five nodes.

4 Conclusions and future works

In this paper, we presented the functionality, models, and metamodels of the HyperFlex model-driven toolchain for composing Feature models according to different composition strategies. HyperFlex has been conceived for simplifying configuration and deployment of complex control systems of autonomous robots. Nevertheless, the proposed approach to variability modeling and composition can be applied to any application domain.

Our current work aims at exploiting the approach presented in this paper to develop dynamically adaptive robotic systems. Robotic engineers can define several variation points (resources, algorithms, control strategies, coordination policies, cognitive mechanisms and heuristics, etc.). Depending on the context, the system dynamically chooses suitable variants to realize those variation points. These variants may provide better quality of service (QoS), offer new services that did not make sense in the previous context, or discard some services that are no longer useful.

References

1. ROS: Robot Operating System. `http://www.ros.org`, 2007.
2. The HyperFlex Toolchain. `http://robotics.unibg.it/hyperflex/`, 2014.

Fig. 11: An example of Feature Refinement Tree

3. Eclipse Xtext. `https://eclipse.org/Xtext/`, 2016.
4. A. Abele, H. Lönn, M.-O. Reiser, M. Weber, and H. Glathe. Epm: a prototype tool for variability management in component hierarchies. In *Proc. of the 16th Int. Software Product Line Conference-Volume 2*, pages 246–249. ACM, 2012.
5. M. Acher, P. Collet, P. Lahire, and R. France. Comparing approaches to implement feature model composition. In *Modelling Foundations and Applications*, pages 3–19. Springer, 2010.
6. N. Ando, S. Kurihara, G. Biggs, T. Sakamoto, H. Nakamoto, and T. Kotoku. Software deployment infrastructure for component based rt-systems. *Journal of Robotics and Mechatronics*, 23(3):350–359, 2011.
7. R. Bischoff, T. Guhl, E. Prassler, W. Nowak, G. Kraetzschmar, H. Bruyninckx, P. Soetens, M. Haegele, A. Pott, P. Breedveld, et al. Brics-best practice in robotics. In *Robotics (ISR), 2010 41st International Symposium on and 2010 6th German Conference on Robotics (ROBOTIK)*, pages 1–8. VDE, 2010.
8. D. Brugali, W. Nowak, L. Gherardi, A. Zakharov, and E. Prassler. Component-based refactoring of motion planning libraries. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ Int. Conference on*, pages 4042–4049. IEEE, 2010.
9. D. Brugali and P. Scandurra. Component-based robotic engineering (part i)[tutorial]. *Robotics & Automation Magazine, IEEE*, 16(4):84–96, 2009.
10. E. Cirilo, U. Kulesza, and C. Lucena. A product derivation tool based on model-driven techniques and annotations. *Journal of Universal Computer Science*, 14(8):1344–1367, 2008.
11. E. Cirilo, I. Nunes, U. Kulesza, and C. Lucena. Automating the product derivation process of multi-agent systems product lines. *Journal of Systems and Software*, 85(2):258–276, 2012.
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 *Simulation, Modeling, and Programming for Autonomous Robots*, pages 149–160. Springer, 2012.

13. S. Garrido-Jurado, R. Muoz-Salinas, F. Madrid-Cuevas, and M. Marn-Jimnez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280 – 2292, 2014.
14. L. Gherardi and D. Brugali. An eclipse-based feature models toolchain. In *6th Italian Workshop on Eclipse Technologies (EclipseIT 2011)*, 2011.
15. L. Gherardi and D. Brugali. Modeling and Reusing Robotic Software Architectures: the HyperFlex toolchain. In *IEEE International Conference on Robotics and Automation (ICRA 2014)*, Hong Kong, China, May 31 - June 5 2014. IEEE.
16. A. Haber, H. Rendel, B. Rumpe, I. Schaefer, and F. Van Der Linden. Hierarchical variability modeling for software architectures. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 150–159. IEEE, 2011.
17. O. Haugen, A. Wasowski, and K. Czarnecki. Cvl: Common variability language. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 277–277, New York, NY, USA, 2013. ACM.
18. A. Lotz, J. F. Inglés-Romero, C. Vicente-Chicote, and C. Schlegel. Managing runtime variability in robotics software by modeling functional and non-functional behavior. In *Enterprise, Business-Process and Information Systems Modeling*, pages 441–455. Springer, 2013.
19. J. Ressia, T. Grba, O. Nierstrasz, F. Perin, and L. Renggli. Talents: an environment for dynamically composing units of reuse. *Software: Practice and Experience*, 44(4):413–432, 2014.
20. M. Rosenmüller and N. Siegmund. Automating the configuration of multi software product lines. In *VaMoS*, pages 123–130, 2010.
21. D. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.
22. M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Softw., Pract. Exper.*, 35(8):705–754, 2005.