Software Architecture and Task Plan Co-Adaptation for Mobile Service Robots

Javier Cámara
University of York
javier.camaramoreno@york.ac.uk

Bradley Schmerl
Carnegie Mellon University
schmerl@cs.cmu.edu

David Garlan
Carnegie Mellon University
garlan@cs.cmu.edu

ABSTRACT

Self-adaptive systems increasingly need to reason about and adapt both structural and behavioral system aspects, such as in mobile service robots, which must reason about missions that they need to achieve and the architecture of the software executing them. Deciding how to best adapt these systems to run time changes is challenging because it entails considering mutual dependencies between the software architecture that the system is running and the outcome of plans for completing tasks, while also considering multiple trade-offs and uncertainties. Considering all these aspects in planning for adaptation often yields large solution spaces which cannot be adequately explored at run time. We address this challenge by proposing a planning approach able to consider the impact of mutual dependencies between software architecture and task planning on the satisfaction of mission goals. The approach is able to reason quantitatively about the outcome of adaptation decisions handling both the reconfiguration of the system's architecture and adaptation of task plans under uncertainty and in a rich trade-off space. Our results show: (i) feasibility of run-time decision-making for self-adaptation in an otherwise intractable solution space by dividing-and-conquering adaptation into architecture reconfiguration and task planning sub-problems, and (ii) improved quality of adaptation decisions with respect to decision making that does not consider dependencies between architecture and task planning.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**; *Software safety*; *Software design tradeoffs*; *Software design techniques*; • **Computer systems organization** → *Robotics*;

KEYWORDS

self-adaptation, assurances, architecture, mobile robotics

ACM Reference Format:
Javier Cámara, Bradley Schmerl, and David Garlan. 2020. Software Architecture and Task Plan Co-Adaptation for Mobile Service Robots. In *IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '20), October 7–8, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3387939.3391591

1 INTRODUCTION

Increasingly, self-adaptive systems need to consider adapting both structural and behavioral aspects of the system to achieve their goals. An important class of such systems is mobile service robots, which typically perform tasks such as fetching mail and escorting a visitor to an office [50]. To do this, a robot must carry out actions such as navigating between locations and interacting with humans. These systems operate in environments where obstacles might dynamically appear, light conditions may change, batteries might require recharging, and sensors may fail. Hence, these systems must adapt both their structure and behavior at run time to respond to changes in their operating environment, as well as to system changes that might include software and hardware faults. For example, a robot tasked with navigating to a specific location within a building might experience failure in the camera it is using to navigate. At that point, the system should (i) change the architecture (structure) to a new legal configuration that does not require the camera to navigate, and possibly (ii) replan the way in which the task (behavior) is going to be completed, since there might be more than one way of reaching the target location (e.g., through alternative paths), and the current one might be sub-optimal for the new architecture configuration in which the camera is not available.

However, obtaining the best combination of software architecture configuration and task plan specification is not straightforward due to the mutual dependencies between them. In general, the outcome of executing the same task specification (e.g., reaching the target location through a specific path) in different software configurations will differ depending on factors like energy efficiency of the configuration or safety aspects related to the accuracy of the sensors and navigation algorithms used. Conversely, adapting the task specification (e.g., the robot needs to find a new path because an obstacle is blocking the way) might require reconfiguration because the current architecture configuration may not be suitable for the new path, e.g., which might include dark corridors in which camera-based navigation is not an option anymore.

Accounting for such dependencies in decision making for adaptation poses a challenge because the size of the combined solution spaces for the architecture reconfiguration and task re-planning can easily become too large to be adequately explored at run time.

To deal with that problem, the MORPH reference architecture [6] suggests separating as much as possible structural reconfiguration and behavior synthesis problems. This is achieved by including a *goal model manager* component in its *goal management layer* that is in charge of decomposing requirements into achievable reconfiguration and behavior problems, which are assigned to different solvers. Then, the *strategy management layer* includes a *negotiation process* that is assumed to guarantee consistency among reconfiguration and behavior strategies when they are selected for

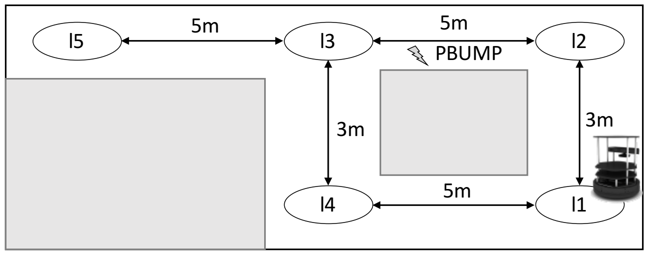


Figure 1: Simple mobile robotics scenario.

execution. However, our research has shown that such negotiation is not straightforward nor always feasible when there is a need to account not only for functionality, but also to optimize multiple extra-functional concerns (e.g., safety, timeliness, energy efficiency) and their trade-offs. This is because the quantitative guarantees associated with reconfiguration and behavior strategies depend heavily on different pieces of information from multiple heterogeneous models (e.g., energy consumption, software architecture, physical space, safety). These pieces of information are inter-related and change at run time, potentially invalidating any quantitative guarantees associated with pre-computed strategies. This calls for a solution able to *co-adapt* structure and behavior in a scalable way, as opposed to reconciling a posteriori pre-computed reconfiguration and behavioral strategies that might not provide quantitative guarantees.

In this paper we contribute an approach based on quantitative synthesis and verification that *separates planning into architecture reconfiguration and task planning problems, dividing and conquering the solution space while still considering their mutual dependencies*. This effectively co-adapts task plan specifications and architecture to optimize adaptation decisions. The approach is instantiated using Alloy [25] and the probabilistic model checker PRISM [29], which are used in tandem to reason quantitatively about the outcome of adaptation decisions. This allows run-time adaptation of the architecture and task plans, under uncertainty, and in a rich trade-off space. With respect to our previous work using quantitative verification for self-adaptation [10, 11, 36], that only dealt with structural adaptation, this approach dynamically constructs strategies that also control the functional behaviour of the system layer components, elaborating explicitly the distinction and coordination between configuration and behavior control.

Our results show that our approach: (i) enables automated run-time decision-making for self-adaptation in what was a priori an intractable solution space by dividing-and-conquering adaptation into sub-problems and (ii) improves the quality of adaptation decisions with respect to decision-making that reasons about structural adaptation only.

2 MOTIVATING SCENARIO

Mobile indoor service robots operate in environments where obstacles might dynamically appear, light conditions may change, and batteries may require recharging. They are also limited in what they can sense, creating uncertainty in their location, chances of colliding against obstacles, and the resources that they may have left to complete a plan. In spite of this uncertainty, they must attempt to ensure safe operation, effective use of resources like battery, and timeliness of completing a task.

We illustrate our approach using a simple scenario (Figure 1), in which the mission of the robot is navigating to a target location (l5) from an initial location (l1) in the shortest possible time, with a limited battery, and without bumping into obstacles or walls. To achieve this goal, the robot can perform physical actions (e.g., move between locations) and change its configuration (e.g., change a sensor, its localization algorithm, or its speed setting).

While accomplishing the mission, the main concerns are: (i) *timeliness* — the robot should get to the target in the shortest possible time, (ii) *safety* — the robot should arrive at the target location without bumping into obstacles, and (iii) *efficiency* — the robot should minimize the energy used to get to the target location.

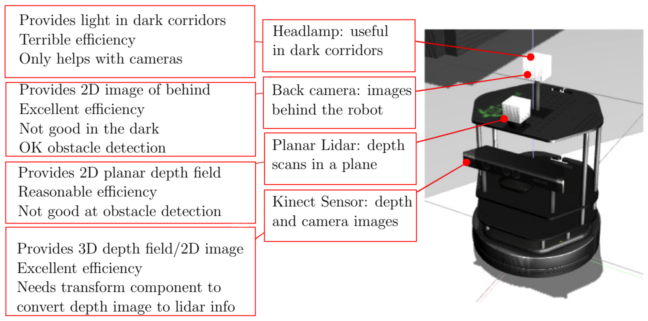
The problem that we want to solve is synthesizing specifications for the architecture and behavior of the robot to successfully complete the mission, attending to the criteria described above, despite situations that include component or sensor failure, obstacles blocking corridors, and unexpectedly low battery level.

To inform decision-making, we have four models that capture different aspects of the domain.

Architecture. Captures the modes in which the robot can operate, determined by the software components that can be employed at run-time (e.g., associated with sensor input processing, low-level navigation algorithms), their connections, and configuration parameters (e.g., speed, localization accuracy). This model includes:

• The current software configuration, including components, connections, and parameter settings. Configurations in this architecture contain three component categories (Figure 2):

1) *Sensing*. Manages physical sensors that capture information from the environment. **(a)** *Kinect*: Provides an image where each pixel also includes depth information that can be used to estimate distances to walls and obstacles. **(b)** *Lidar*: This sensor provides range information in a 2D plane covering 180°. **(c)** *Back Camera*: The robot has a rear facing camera that can be used by software to position the robot using visual information.


Provides light in dark corridors
Terrible efficiency
Only helps with cameras
Provides 2D image of behind
Excellent efficiency
Not good in the dark
OK obstacle detection
Provides 2D planar depth field
Reasonable efficiency
Not good at obstacle detection
Provides 3D depth field/2D image
Excellent efficiency
Needs transform component to convert depth image to lidar info
Headlamp: useful in dark corridors
Back camera: images behind the robot
Planar Lidar: depth scans in a plane
Kinect Sensor: depth and camera images

Category	Name	Energy cost	Accuracy	Requires
Sensing	lidar	Medium	Bad	-
	kinect	Excellent	Good	laserscanNodelet
	camera	Excellent	Medium	markerRecognizer headlamp (when dark)
Localization	amcl	Excellent	Excellent	-
	mrpt	Medium	Good	-
	aruco	Bad	Good	-
Auxiliary	laserscanNodelet	N/A	N/A	-
	markerRecognizer	N/A	N/A	-
	headlamp	Really bad	N/A	-

Figure 2: Mobile robotics architecture configuration space.

2) *Localization*. Produces information about the position of the robot based on information captured by a sensor, or processed by another component. **(a)** *AMCL*: Implements a general-purpose localization scheme that fuses depth sensor information and odometry to estimate the pose of a robot on a known map [15]. **(b)** *MRPT*: Implements an alternative localization algorithm [4]. **(c)** *ArUco*: Takes an image from a camera and identifies the relative position of the camera to visual markers that are placed in the environment [41], using that knowledge to determine the position of the robot. This algorithm relies on a well lit environment to see the markers.
3) *Auxiliary*. Perform auxiliary functions and impact qualities like energy efficiency, localization accuracy and obstacle detection (which affects safety) in different ways. **(a)** *Headlamp*. The robot has an LED lamp to make visual information available in dark locations. **(b)** *Laserscan Nodelet*. Transforms data from Kinect to Lidar format, so that it can be consumed by standard localization components (i.e., AMCL and MRPT).
• The repertoire of behavior primitives that the robot can execute to change its configuration. We assume a pair of primitives to enable/disable every component in the architecture (e.g., enable_kinect, disable_lidar). For a configuration parameter (like speed setting), there is a primitive that sets the new value of the parameter (e.g., set_speed(value)).
• A formal model of the architectural style that prescribes which configurations are valid, encoding for instance, that some components require additional auxiliary components to interoperate with the rest of the system (c.f. table in Figure 2).

The architecture is represented in Acme [17], and was derived from an architecture discovery for ROS similar to [44].

Physical Environment. Describes the space that the robot is navigating. This model is captured as a graph, where nodes correspond to physical locations, and arcs correspond to trajectories between them. Arcs are tagged with attributes *distance* (5m and 3m for horizontal and vertical arcs in Figure 1, respectively) and *risk* (a probability value in [0,1] that captures the likelihood that the robot will bump into an obstacle when moving between the endpoints of the arc). In this simplified scenario, we assume that all arcs have a risk probability of 0, except for arc (l2, l3), which has a bump risk probability PBUMP which varies, depending on the robot's configuration (e.g., a configuration with higher speed or less accurate obstacle detection will have a higher probability). This is modeled as an annotated graph.

Power. This model is queried to determine: (i) available resources (i.e., remaining battery energy), and (ii) cost of operations in a given configuration (e.g., how much energy it takes to move to a location with a given speed and set of sensors/components enabled). This is a linear regression model that was was derived from experimentation on a Turtlebot [43], combined with documented power specifications of sensors, e.g. for the Lidar [2].

Task. Describes the task to accomplish, and its progress. In this scenario, we assume that the progress of the mission captures the location of the robot, its remaining battery level, the history of actions executed, and the remainder of actions to be executed. It also includes the repertoire of behavior primitives that the robot can execute to carry out an action in the physical environment. For simplicity, we assume that in this scenario the robot can carry out two types of action: (i) move between two locations in the map, and (ii) change its configuration, i.e., execute a sequence of reconfiguration primitives to enable/disable software components and/or set new values of its speed setting. This is modeled as a graph of robot actions represented using Instruction Graphs [35].

Analyzing system properties requires combining pieces of information from different models, which do not provide insight if considered individually. For example, the topology of the physical environment, operations, or the power model are not useful on their own, but they can be combined e.g., to estimate metrics like time to complete the robot's mission, or the overall probability of bumping into an obstacle along the way. Moreover, even in this simplified scenario, the size of the combined reconfiguration and task plan solution spaces is in the order of magnitude of 10^6 states and will grow exponentially with additional components and map waypoints. This makes moderate problem instances intractable for online monolithic planning approaches.

Hence, despite the apparent simplicity of this scenario, its characteristics make evident the need for approaches to: (i) integrate information from heterogeneous models and different levels of abstraction to generate and provide assurances about system behaviors, (ii) reason in a multi-dimensional space of concerns, (iii) capture and factor in uncertainties that affect the system, and (iv) scale when dealing with potentially intractable solution spaces.

3 APPROACH

Our approach for reasoning about self-adaptation is intended mainly for run-time use in the *planning* activity of MAPE-K, which is concerned with determining *how* to best adapt the system at run time.

Considering the characteristics of the scenario and the challenges described in Section 2, a suitable approach to our problem should be: (i) reusable in a new context with low reengineering overhead, (ii) able to reason about multiple system aspects, trade-offs, and under uncertainty, (iii) able to scale, and (iv) able to provide formal guarantees about the solutions generated.

To satisfy those requirements, we propose an approach that hinges on a carefully orchestrated use of disparate formal methods like structural model synthesis and probabilistic model checking. Concretely, the rationale for our approach entails: (i) separating reconfiguration from mission planning, and (ii) carrying out reconfiguration planning before task planning to avoid exploring a larger state space in which candidate task plans are carried out in configurations that are later found to be infeasible. Figure 3 illustrates our approach, which divides the planning into two major stages: *Architecture reconfiguration planning*, which determines the legal software architectures of the robot given the current available and failed components (Step 1) and generates an adaptation strategy to adapt the robot to this new architecture (Step 2), and *Task planning* which replans the mission (Step 3) and determines the best architecture/mission combination through quantitative planning in PRISM. The result of this is a combined plan that adapts the software architecture of the robot and the robot mission to satisfy the quality trade-offs that are required (Step 4). The circled numbers and references to tables and listings in Figure 3 are elaborated on later.

(A) *Model-View Projection* (Section 3.1): (i) each problem domain model is projected into a view that abstracts its important features in an intermediate language (this projection is model type-specific

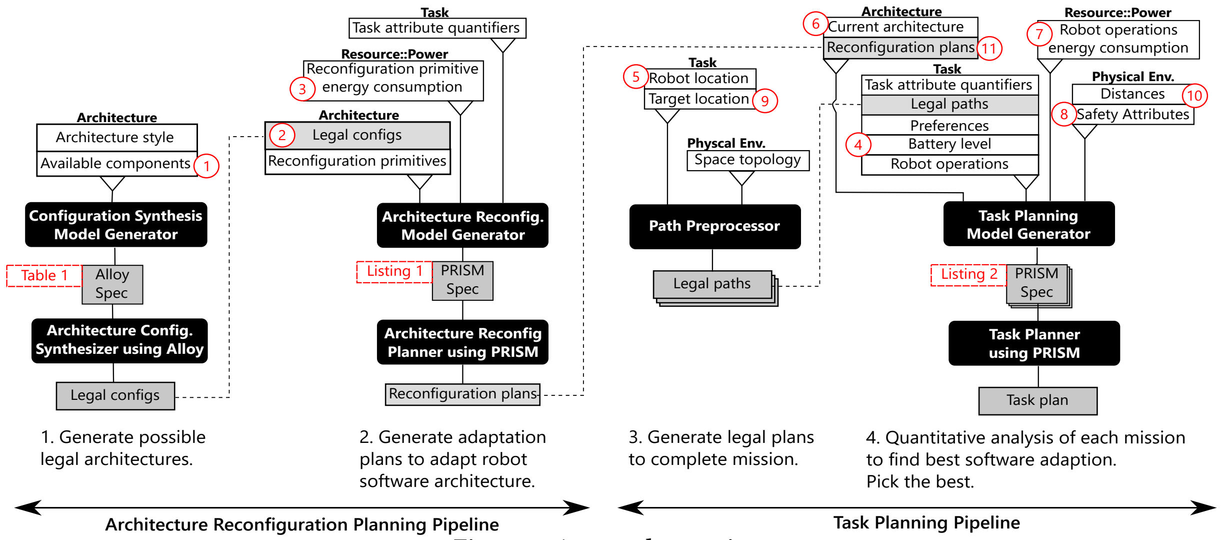

Architecture
Architecture style
Available components
1
Configuration Synthesis Model Generator
Table 1
Alloy Spec
Architecture Config. Synthesizer using Alloy
Legal configs
1. Generate possible legal architectures.
Task
Task attribute quantifiers
Resource::Power
Reconfiguration primitive energy consumption
3
Architecture
2
Legal configs
Reconfiguration primitives
Architecture Reconfig. Model Generator
Listing 1
PRISM Spec
Architecture Reconfig Planner using PRISM
Reconfiguration plans
2. Generate adaptation plans to adapt robot software architecture.
Task
5
Robot location
Target location
9
Physcal Env.
Space topology
Path Preprocessor
Legal paths
3. Generate legal plans to complete mission.
Architecture
6
Current architecture
Reconfiguration plans
11
Task
Task attribute quantifiers
Legal paths
Preferences
4
Battery level
Robot operations
Resource::Power
7
Robot operations energy consumption
Physical Env.
Distances
10
8
Safety Attributes
Task Planning Model Generator
Listing 2
PRISM Spec
Task Planner using PRISM
Task plan
4. Quantitative analysis of each mission to find best software adaption. Pick the best.
Architecture Reconfiguration Planning Pipeline
Task Planning Pipeline

Figure 3: Approach overview.

– for each type of model a model-view projector component must be written), and (ii) the relevant information is composed into Alloy and PRISM models used by the architecture reconfiguration planning and task planning pipelines (see Figure 3), by filling out templates as described in Section 3.2.

(B) *Adaptation planning* (Section 3.2), which entails planning model generation and plan synthesis both for architecture reconfigurations and tasks, and is divided into two planning pipelines:

(a) *Architecture reconfiguration planning pipeline* (Section 3.2.1). Architecture reconfiguration plan synthesis involves: (i) the **configuration synthesis model generator** produces an Alloy specification that encodes the constraints of the architectural style, along with the currently available set of components (both extracted from the architecture model through its corresponding model-view projector), (ii) using such an Alloy specification, the **architecture configuration synthesizer** generates a set of legal configurations, (iii) the **architecture reconfiguration model generator** encodes the set of legal configurations produced in the previous step as predicates into a PRISM specification that incorporates information from the other views, (iii) the architecture reconfiguration planner generates a set of reconfiguration plans via MDP policy synthesis from PRISM specifications produced in the previous step, in which the choices of what reconfiguration primitives must be executed to reach the target configuration is underspecified as nondeterminism. Plan synthesis is carried out by PRISM, guided by *task attribute quantifiers*, which are probabilistic temporal logic PCTL specifications that encode optimization of metrics like time or energy consumed, which are factored in as costs of the reconfiguration plans.

(b) *Task planning pipeline* (Section 3.2.2). Obtaining a task plan entails: (i) using the **task planning model generator** to produce a PRISM specification for task planning that integrates the actions that the robot can carry out in the physical world, along with multiple pieces of information from the different views, as well as the set of legal reconfiguration plans obtained from **(a)** – each reconfiguration plan is symbolically encoded as an alternative reconfiguration action in the task PRISM specification– and, (ii) the **task planner** generates a task plan via MDP policy synthesis that resolves the nondeterministic choices in which the selection of robot and reconfiguration actions is underspecified. In such a way, task planning is carried out in a context-sensitive manner with respect to the set of possible configurations in which the task can be executed, which might yield different outcomes in terms of timeliness, energy efficiency, and safety.

3.1 Model-View Projection

System models are complex engineering artifacts written in a formal or semi-formal language, each capturing a different facet of the problem domain (e.g., energy or time). Directly integrating models into composite analyzable behaviors is a difficult task. To ease this process, we first provide abstractions of the models, which, following prior research, we term *views* [23]. A view contains a description of sets, relations, functions, and predicates that a model provides for behavioral reasoning. For instance, a configuration view for a mobile robot can provide a set of available system configuration options (such as speed and sensors to use), domains of their values, and constraints on which options can be used together (e.g., a robot can use visual navigation only if its visual sensors are on).

We construct views and manage the relations between them. For example, an architecture configuration view exposes a set **confs**, and a power view exposes a function **pow**(**bp**, **c**) that returns the power needed to carry out behavioral primitive **bp** (e.g., move between locations **l1** and **l2**) in configuration **c** ∈ **confs**. Thus, the power view can be related to the configuration view.

We engineer views to expose model information relevant to behavioral reasoning and synthesis in the task domain.

3.2 Adaptation Planning

Our approach to adaptation planning is based on probabilistic model checking [28] which enables quantitative analysis and policy synthesis in systems that exhibit probabilistic behavior. These policies can be synthesized from MDP models that capture system behavior using standard temporal logics and model checkers like PCTL/PRISM and are guaranteed to achieve optimal expected probabilities and quantitative rewards [30].

In the remainder of this section, we first show how we combine MDP policy synthesis via model checking with formal description and reasoning about architectural styles for reconfiguration synthesis (Section 3.2.1). We then illustrate task planning (Section 3.2.2).

3.2.1 Architecture Reconfiguration Planning. Architecture reconfiguration planning is based on combining architectural synthesis techniques built on a formalization of architectural styles with plan synthesis that uses MDP policy synthesis via model checking. Architectural styles [46] characterize the design space of families of software systems as patterns of structural organization, defining a *vocabulary* of component/connector types and *constraints* that help designers constrain exploration to a set of legal system structures.

Definition 3.1 (Architectural Style). Formally, we characterize an architectural style as a tuple (Σ, C_S), where:

- $\Sigma = (CompT, ConnT, \Pi, \Lambda)$ is an architectural signature, such that:
  - $CompT$ and $ConnT$ are sets of component/connector types.
  - $\Pi : (CompT \cup ConnT) \rightarrow 2^{\mathcal{D}}$ is a function that assigns sets of symbols typed by datatypes in a fixed set $\mathcal{D}$ to architectural types $\kappa \in CompT \cup ConnT$. $\Pi(\kappa)$ represents the properties associated with type κ. To refer to a property $p \in \Pi(\kappa)$, we simply write $\kappa.p$. To denote its datatype, we write $dtype(\kappa.p)$.
  - $\Lambda : CompT \cup ConnT \rightarrow 2^{\mathcal{P}} \cup 2^{\mathcal{R}}$ is a function that assigns a set of symbols typed by a fixed set $\mathcal{P}$ to components and a set of symbols in a fixed set $\mathcal{R}$ to connectors. $\Lambda(\kappa)$ represents the ports of a component (conversely, the roles if κ is a connector), which define logical points of interaction with κ's environment.
- C_S is a set of structural constraints expressed in a constraint language based on first-order predicate logic similar to Acme [17] or OCL [51] constraints (cf. Table 1).

For the remainder of this section, we assume a fixed universe $\mathcal{A}_\Sigma$ of architectural elements, i.e., a finite set of components and connectors for Σ typed by $ConnT \cup CompT$. For a given architectural element $c \in \mathcal{A}_\Sigma$, we denote its type as $type(c)$. Moreover, we denote the set of components in $\mathcal{A}_\Sigma$ as $\mathcal{A}_{\Sigma Comp}$ ($\mathcal{A}_{\Sigma Conn}$ for connectors).

Formal characterization of architectural styles in Alloy [25] have proved to be a valuable tool for exploring rich solution spaces by synthesizing system configurations that satisfy the constraints imposed by an architectural style [3, 9, 13, 27, 32]. A *configuration* can be characterized as a a graph that captures the topology of a feasible structure of the system in the style.

Definition 3.2 (Configuration). A configuration in a style $\mathcal{A} = (\Sigma, C_S)$ is a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ satisfying the constraints imposed by C_S, where: $\mathcal{N}$ is a set of nodes, s.t. $\mathcal{N} \subseteq \mathcal{A}_\Sigma$, and $\mathcal{E}$ is a set of pairs typed by $\mathcal{P} \times \mathcal{R}$ representing *attachments* between ports and roles. We denote the set of legal configurations in $\mathcal{A}$ by $\mathcal{AG}$.

Example 3.3. We can characterize the space of configurations in our scenario by the signature CompT = {lidarT, kinectT,cameraT, ...}, ConnT = {laserScanTopicT, ...}, Π = { (lidarT, {Status, Consumption,...}),...}. Connectors correspond to topics to which the components publish and subscribe.[1] Component properties in Π include Status ∈ {Enabled, Disabled, Offline}, and Consumption, which captures a component's contribution to energy consumption. With these elements, we can specify a set of structural constraints that the style imposes on valid configurations (c.f. Table 1[2]).

[1]This architecture is based on the pub-sub middleware ROS [40].

To generate models for architecture reconfiguration planning, we need to be able to encode architecture configurations as predicates in our model. To do so, we define an encoding function $pred_{\mathcal{G}}$ that takes a configuration and returns a predicate:

$$pred_{\mathcal{G}}(\mathcal{G}) = \bigwedge_{\substack{n \in \mathcal{N}, \\ n.\text{Status=Enabled}}} pred_{\mathcal{N}}(\mathcal{N}) \wedge \bigwedge_{\substack{n \in \mathcal{N}, \\ n.\text{Status}\neq\text{Enabled}}} \neg pred_{\mathcal{N}}(\mathcal{N}) \tag{1}$$

Function $pred_{\mathcal{N}}$ takes a node in a configuration graph, and returns a predicate that encodes the presence of that component/connector in the graph. We leave implicit bindings between components and connectors, which are not required because we assume no rewiring primitives as possible reconfiguration actions (just enabling/disabling components). This encoding can be naturally extended to include in the conjunction additional terms encoding bindings.

We define a component status vector as the set of all possible component status values in the configuration space, i.e., $Comp_{\mathcal{S}} \equiv \{\text{Enabled, Disabled, Offline}\}^{|\mathcal{A}_{\Sigma Comp}|}$.

Definition 3.4 (Reconfiguration Planning Model). A parametric MDP model for reconfiguration is a tuple $(S, s_I, A, \Delta, r, \mathcal{GP})$, s.t.:

- $S \subseteq Comp_{\mathcal{S}}$ is the space of possible component status.
- $s_I : Comp_{\mathcal{S}}$ is a vector of initialization parameters for component status, which determines the starting reconfiguration point.
- $A \equiv \{\text{Enable, Disable}\} \times \mathcal{A}_{\Sigma Comp}$ is a set of actions to enable/disable components. We note the action (Enable, c) as Enable_c.
- $\Delta : S \times A \rightarrow S$ is a transition relation that differs from the standard MDP description, leaving out probabilistic extensions for unreliable reconfiguration primitives out of scope.
- $r : A \rightarrow \mathbb{R}^+$ maps reconfiguration actions to rewards (costs).
- $\mathcal{GP} \equiv \bigcup_{g \in \mathcal{AG}} pred_{\mathcal{G}}(g)$ encode legal configurations in $\mathcal{AG}$.

In practical terms, we build the MDP model as a set of modules in PRISM with the structure illustrated in Listing 1. In the listing, global variable turn (line 2) imposes a strict total order on which components can be reconfigured, eliminating unneeded interleavings and reducing remarkably the state space.

The model includes a set of $|\mathcal{A}_{\Sigma Comp}|$ modules (referred to in the listing as **N**). Every component's behavior is encoded as a module with the corresponding **enable** and **disable** actions. There is no explicit action for the **Offline** status, since this indicates that the component is unavailable (e.g, due to failure), and reconfiguration actions over it are out of the system's control. The parts highlighted in blue in the listing are those incorporated from the domain model views: ①[3] sets the current status of a component, obtained from the monitoring infrastructure and exposed through the architecture view, and a set of formulas ② encode the alternative legal configurations ($\mathcal{GP}$, obtained from function $pred_{\mathcal{G}}$ – Expression 1).

Finally, reconfiguration cost is encoded as a reward structure over reconfiguration actions (lines 16-20). For each action, we assign a cost ③ that can encode different attributes like time employed in carrying out the reconfiguration action, or energy spent.

[2]For clarity, representation of port-role attachments is left implicit in our formalization.
[3]Numbered elements in listings 1 and 2 correspond to those identified in Figure 3.

Table 1: Architectural element definitions and constraints of the pub-sub robotics software architecture (excerpt).

Type	Constraint	Description
General definitions	**abstract sig** componentT {pub: **set** topicT, sub: **set** topicT}	Component definition.
	abstract sig topicT {pub: **set** componentT, sub: **set** componentT}	Connector definition.
General constraints	**pred** publishesTo[c:componentT, t:topicT] t **in** c.pub	A component publishes to a topic.
	pred onlyPublishesTo[c:componentT, t:topicT] { publishesTo[c,t] **and all** t’:topicT-t | **not** publishesTo[c,t’] }	A component only publishes to a specific topic.
ROS definitions	**abstract sig** sensingT, localizationT **extends** component {}	Component type definitions.
	lone sig lidarT, kinectT, cameraT **extends** sensingT {}	Sensing component type definitions.
	lone sig laserScanTopicT, sensorMsgsImageTopicT, markerPoseTopicT **extends** topicT{}	Topic type definitions.
ROS constraints	**pred** {**all** c:kinectT | onlyPublishesTo[c,sensorMsgsImageTopicT]	Kinect only publishes to image sensor message topic.
	pred {**some** kinectT <=> **some** laserscanNodeletT}	Kinect requires laser scan translator component.

Synthesis of Reconfiguration Plans. Using a MDP model like the one described above, in which initialization of component status constants is done according to the monitored status of the components at run-time, we can carry out policy synthesis to determine the best legal reconfiguration plans. To illustrate the process, consider the PCTL property (2) that captures the reachability of a legal configuration (e.g., conf_M encoded in Listing 1, line 14).

$$\underbrace{\mathrm{R\{reconf\_cost\}_{min=?}}}_{\text{Reward quantifier}}\ \underbrace{\mathrm{[F\ conf\_M]}}_{\text{Path formula}} \qquad (2)$$

This property contains two parts: (i) a *reward/cost quantifier*, which indicates that the policy synthesis should resolve the nondeterminism in the model in such a way that it minimizes the accrued reward reconf_cost (Listing 1, line 16), and (ii) a *path formula*, which indicates that the paths in the model over which the reward has to be optimized are those that lead to states where the reachability predicate (i.e., arriving at legal architecture configuration conf_M) is satisfied. Model checking a property like the one described for every legal configuration with a probabilistic model checker can yield a set of policies. Translating from policies to sequential reconfiguration plans is straightforward, since the policies are deterministic. We define function $generate_{\mathcal{R}}$, which yields the set of reconfiguration plans for a given parametric MDP reconfiguration model $\mathcal{MR}$ and an initial configuration state i:

$$\bigcup_{g\in\mathcal{AG}} seq_\sigma(\sigma(\mathcal{MR}, i, \mathrm{R\{reconf\_cost\}_{min=?}[F}\ pred_{\mathcal{G}}(g)]).p) \qquad (3)$$

Where $\sigma(\mathcal{MR}, i, \phi)$ designates the model checking function that synthesizes a policy for a model $\mathcal{MR}$ from an initial state i, and a PCTL property ϕ (we denote by $\sigma(\mathcal{MR}, i, \phi).p$ the policy generated, whereas $\sigma(\mathcal{MR}, i, \phi).q$ is the quantitative result, i.e., the quantified cost for the policy generated in the Expression 2). Function seq_σ is a simple function that turns a deterministic policy into a sequential plan. We also assume a simple function *label_plan* that assigns a symbolic label to a sequential plan. We note the set of symbolic labels for a set of reconfiguration plans X as $labels(X)$.

```
1   mdp
2   global turn:[0..N] init 0;
3   const DISABLED=0; const ENABLED=1; const OFFLINE=2;
4   ...
5   const  compN_INIT; ① // Available components, obtained from architecture view
6   module compN // One per component
7       compN_done : bool init false;
8       compN_status:[0..2] init compN_INIT;
9       [compN_enable] (turn=N−1) & (compN_done=false) & (compN_status=DISABLED) −>
                (compN_status'=ENABLED) & (compN_done'=true);
10      [compN_disable] (turn=N−1) & (compN_done=false) & (compN_status=ENABLED) −>
                (compN_status'=DISABLED) & (compN_done'=true);
11      [] (turn=N−1) −> (turn'=N);
12  endmodule
13  ...
14   formula conf_M = (comp0_status=DISABLED | comp0_status=OFFLINE) & ... &
            (compN_status=ENABLED); ②  // One per legal configuration, generated by
15                                      // the architecture configuration synthesizer
16  rewards "reconf_cost"
17      ...
18      [compN_enable] true :  cost_compN_enable; ③  // Reconf. primitive energy cost,
19      [compN_disable] true :  cost_compN_disable ; // obtained from resource (power) view
20  endrewards
```

Listing 1: Architecture reconfiguration model structure.

3.2.2 *Task Planning.* In addition to the architectural model of the system, we assume a set of additional views that expose relevant information for task planning. These pieces of information can be combined into building blocks for rich planning model construction.

We begin by introducing the view that describes the physical environment in which the task takes place.

Definition 3.5 (Physical Environment View). A physical environment view $\mathcal{V}_{PE} \subseteq LOC \times TRA$ can be characterized as a simple directed graph, where LOC is a set of nodes representing locations, and $TRA \subseteq LOC \times LOC$ is a set of edges that represent valid trajectories between locations. We also equip the view with $\Pi_{LOC} : LOC \rightarrow 2^{\mathcal{D}}$ and $\Pi_{TRA} : TRA \rightarrow 2^{\mathcal{D}}$, which are functions that assign sets of symbols typed by datatypes in a fixed set $\mathcal{D}$ to nodes and trajectories in the map, capturing relevant attributes, e.g. length of a trajectory, plane coordinates or id of a location.

In addition to the set of reconfiguration primitives A, there is a set of actions O that can be carried out in the physical world by the robot. Executing both types of action consumes resources which can be encoded in a resource view. We illustrate this section with a single resource view that corresponds to energy.

Definition 3.6 (Resource View). A resource view $\mathcal{V}_R : \mathbb{R}^+ \times \mathcal{AG} \times (O \cup A) \rightarrow \mathbb{R}^+$ is characterized as a function that quantifies the amount of resources consumed by an action carried out during an arbitrary amount of time and in a given architectural configuration.

Execution of actions to accomplish a task also requires keeping track of the current progress of the task, as well as what has been done and remains to be done to satisfy system goals.

Definition 3.7 (Task View). In a task view $\mathcal{V}_T = (S, s_I, O, H, P, g)$: S is the state space of the task, $s_i \in S$ is the current state of the task, O is the set of robot operations, $H \in O^*$ is the history of executed actions, $P \in O^*$ are the pending actions, and g is an abstract specification of the task’s goal. Although we give a general definition, in our scenario we assume that a state is a pair in $\mathcal{V}_{PE}.LOC \times \mathbb{R}^+$ of localization in the map and battery energy.

Now that we have defined the views, we introduce a definition of the task planning model.

Definition 3.8 (Task Planning Model). A model for task planning is an MDP (S, s_I, A, Δ, r) where: $S \subseteq \mathcal{AG} \times \mathcal{V}_T.S$ is the state space,

```
1   module robot
2   b:[0..MAX_BATTERY] init  INITIAL_BATTERY;④ // Task view
3   l:[0..MAX_LOCATIONS] init  INITIAL_LOCATION;⑤ // Task view
4   c:[1..M] init init  INITIAL_CONFIGURATION;⑥ // Architecture view
5   rd: bool init false; collided: bool init false;
6         ... // One command per legal target configuration
7     [t_set_conf_M] (c!=conf_M) & (b>MIN_BATTERY+deplete_battery_reconfM⑪
               ) & (!rd) & (!stop) −> (c'=conf_M) & (rd'=true) & (b'=b−deplete_battery_reconfM);
8         ... // One command per combination of legal config/arc among adjacent map locations
9     [lx_to_ly] (l=lx) & (!stop) & (c=conf_M) −>p_col_conf_M_lx_to_ly⑧
               : (l'=ly) & (b'=b_upd_lx_ly⑦) & (collided'=true) + 1−(p_col_conf_M_lx_to_ly):
               (l'=ly) & (b'=b_upd_lx_ly) & (collided'=false);
10  endmodule
11  formula b_upd_lx_ly= c=conf_1? max(0,b-e_lx_ly_conf_1) : ... (c=conf_M?
               max(0,b-e_lx_ly_conf_M) : 0);⑦ // One per arc between adjacent map locations
12  ...
13  const INITIAL_LOCATION;
14  const TARGET_LOCATION;⑨ // Task view
15  formula goal = l=TARGET_LOCATION;
16  formula stop = goal | b<MIN_BATTERY;
17  rewards "time"
18        [lx_to_ly] true :c=conf_1 ? t_lx_ly_conf_1⑩ : ... c=conf_M ? t_lx_ly_conf_M :
                 MAX_BATTERY; // One per arc between adjacent map locations;
19        ...
20        [t_set_conf_1] true :c=conf_2 ? t_set_conf_2_conf_1⑪
                 : ... c=conf_M ? t_set_conf_M_conf_1 : 0; ... // One per legal target configuration
21  endrewards
22  rewards "energy"
23        stop : b;
24  endrewards
```

Listing 2: Task planning model structure.

$s_I \in S$ is a pair of architectural configuration c and the initial task state, $A \equiv O \cup labels(generate_{\mathcal{R}}(\mathcal{MR}, s_I.c))$ is the set of actions that include operations of the robot in the physical world, as well as symbolic actions that correspond to architecture reconfiguration plans from the initial configuration $s_I.c$, $\Delta : S \times A \longrightarrow \mathcal{D}(S)$ is a (partial) probabilistic transition function, and $r : S \times S \longrightarrow \mathbb{R}^+$ is a reward structure that maps transitions to rewards.

We build the MDP in a PRISM encoding of the robot behavior using the pattern illustrated in Listing 2, in which the robot module includes variables: **b** for capturing the battery energy level (initialized with the value exposed from the task view ④), **l** for robot location (with an initial value ⑤ from the task view), **c** for the robot's architecture configuration (initialized with information ⑥ from the architecture view), **rd** to keep track of whether reconfiguration has been carried out, and **collided** which captures if the robot has collided against some obstacle (lines 2-5). Based on these variables, we also define formulas that capture: *(a) goal satisfaction* (⑨, line 15) encoding that the robot location is the target location, and *(b) stop condition* (line 16), which holds when the robot satisfies the goal or does not have energy to operate. The module's behavior is defined by two groups of guarded commands:

• Reconfiguration actions (line 7). There is one reconfiguration action command for every legal reconfiguration. Action labels in the reconfiguration commands (e.g., [t_set_conf_M]) are obtained via $labels(generate_{\mathcal{R}}(\mathcal{MR}, s_I.c))$ (cf. Definition 3.8).

– Each reconfiguration command is guarded to make sure that:(a) the target configuration is not already active, (b) there is sufficient energy in the battery to reconfigure (**deplete_battery_reconf_X** ⑪, lines 7-8, are the energy costs of reconfiguration plans, computed using Expression 2), (c) reconfiguration has not been carried out, and (d) the stop condition does not hold.

– Each command updates: (a) the configuration to the target one, (b) the reconfiguration tracking variable, and (c) the battery level to reflect the energy consumed during reconfiguration.

• Task actions (line 9). These commands encode the actions of the robot in the physical world. In this case, these actions correspond to movements between locations in the physical environment. For every trajectory between locations **lx** and **ly** in the physical environment view $\mathcal{V}_{PE}.TRA$ (c.f. Definition 3.5), there is a set of commands (one per configuration) that encodes the movement of the robot between those locations. Every command:

– Checks in its guard that: (a) the current location is **lx**, (b) the stop condition has not been reached, and (3) the current configuration is the one to which the command corresponds.

– Includes two updates with probabilities **p_col_conf_A_lx_to_ly** ⑧ and its complementary outcome, which encode the likelihood of colliding while going through the trajectory in configuration **A**. This probability is one of the attributes with which we tag our physical environment view (c.f. Π_{TRA}, Definition 3.5). Each of these update: (a) the location variable, (b) the depletion of the battery, and (c) the collision status. For battery updating, the model defines a set of formulas (⑦, line 11) where different battery depletions are encoded per-configuration. These values (**e_lx_ly_conf_X**) are obtained by querying the resource (i.e. energy) view $\mathcal{V}_R$ with the specific action and configuration that the command corresponds to (c.f., Definition 3.6).

Our task planning model also incorporates a reward structure that enables quantifying metrics associated with planning concerns like timeliness and energy efficiency. Lines 17-24 illustrate reward structures used to capture timeliness and energy efficiency. For timeliness, line 18 shows how the structure encodes time reward for an action that moves the robot from location **lx** to **ly**, accruing different amounts of time **t_lx_ly_conf_X** ⑩, depending on the configuration that the robot is in. These times are computed by combining the distance attribute of the trajectories in the physical environment view ($\mathcal{V}_{PE}.TRA$) with the speed setting of the robot's configuration. Line 20 shows how time is accrued for reconfigurations. **t_set_conf_A_conf_B** ⑪ is the time it takes to reconfigure, obtained from the evaluation of the reconfiguration cost analogous to the one encoded in Expression 2, in which cost is expressed in terms of time. For energy efficiency, Line 23 encodes a simple reward that corresponds to the level of battery **b** when the robot reaches the **stop** condition.

Generating Task Plans. Generating a task plan entails:

(A) *Synthesis of candidate solution plans.* For generating task candidate plans, we instantiate a set of alternative probabilistic models $\mathcal{M}_{f_O}$ (cf. Definition 3.8), in which the set of robot operations is fixed to a specific path between its current and the target location. These paths are computed using Dijkstra's algorithm and ranked by ascending distance. We set a threshold n_{f_O} that fixes the number of paths to extract from the top of the ranking. We perform this step for efficiency reasons, because synthesizing paths as part of MDP policy synthesis is rather costly. Hence, the only decision left underspecified for model checking is selecting the best reconfiguration for a specific candidate solution model. This results in n_{f_O} runs of the model checker instead of one, but the computational cost is reduced orders of magnitude, compared to considering the full solution space in a single model checking run.

(B) *Quantifying the value of metrics* for the different concerns of every candidate solution entails model checking the candidate solution models in $\mathcal{M}_{f_O}$ against the PCTL properties in Table 2:

Table 2: PCTL properties for mission planning.

Name	PCTL Formula	Description
ϕ_t	$\mathsf{R}\{\mathsf{time}\}_{\mathsf{min=?}}[\mathsf{F\ goal}]$	Time to target location.
ϕ_e	$\mathsf{R}\{\mathsf{energy}\}_{\mathsf{max=?}}[\mathsf{F\ goal}]$	Remaining energy at target location.
ϕ_s	$\mathsf{P}_{\mathsf{max=?}}[\mathsf{F}\ (\mathsf{goal} \land \neg\mathsf{collided})]$	Non-collision probability.

(a) The process starts by generating the optimal policy for the priority concern for every candidate solution $\sigma_*^i \triangleq \sigma(m_{fo}^i, cs, \Phi).p$, where $\Phi \in \{\phi_t, \phi_e, \phi_s\}$ is the PCTL formula for the prioritized concern, $m_{fo}^i \in \mathcal{M}_{fo},\ i \in \{1 \ldots n_{fo}\}$, and $cs \in \mathcal{AG} \times \mathcal{V}_T.S$ (c.f., Definition 3.8) is the current state combining configuration and task state from which the solution is being computed.

(b) Once the optimal policy for the priority σ_*^i has been generated for each candidate, we quantify how good this candidate solution is by computing an attribute value vector:

$$\langle \sigma_{\sigma_*^i}(m_{fo}^i, cs, \phi_t).q, \sigma_{\sigma_*^i}(m_{fo}^i, cs, \phi_e).q, \sigma_{\sigma_*^i}(m_{fo}^i, cs, \phi_s).q \rangle$$

Where $\sigma_{\sigma_*^i}(\ldots)$ denotes that model checking is constrained to a version of the model in which the policy is fixed to σ_*^i. We designate the set of attribute vectors for $\mathcal{M}_{fo}$ as V_{fo}.

(C) *Candidate solution generation*, for which we maximize a utility function over attribute vectors (in the expression below, function $\sigma_\uparrow$ returns the optimal policy σ_*^i from which an attribute value vector was generated):

$$p_\uparrow \triangleq \sigma_\uparrow(\arg \max_{v \in V_{fo}} u(v))$$

4 EVALUATION

In this section, we evaluate our approach along two dimensions: (i) *Run-time tractability* – can our planning approach work at run time to adapt a robot using off-the-shelf robotic software and adapt to errors and changes in the environment to complete missions? and (ii) *Quality improvement* – does the approach generate plans with higher utility than the standard self-adaptive practice of considering only the structural aspects of the system?

We use a scenario similar to the one in Section 2, but with a more realistic robot environment with 58 intersections (used as waypoints for planning). Some corridors contain small obstacles that are out of the range of one of the robot sensors (Lidar), meaning that in some configurations the robot has a high chance of colliding with them. The obstacles are static, but unknown to the planner. Similarly, some corridors are known to be dark, which means that they will be non-navigable if the robot is using light-sensitive sensors. To enable visual localization, ArUco markers are spaced every 2m on the walls. Figure 4 shows the map used, with the known dark corridors and preplaced obstacles, as well as an example of ArUco markers in the lower right.

The robot has three speed settings: (i) safe (0.24m/s) meaning that collisions with walls or obstacles do not cause damage to them or the robot, but the probability of mission success is diminished because they may block the robot, (ii) normal (0.35m/s) the starting speed of the robot for all experiments, and (iii) high (0.68m/s). This setup allowed us to explore an interesting range of adaptations where multiple adaptations of architecture and missions could be found to complete the mission. To instigate adaptation, perturbations of the robot or environment include turning off lights in the environment, causing sensor failure, or killing the localization node (simulated software crash) while the robot is conducting a mission.

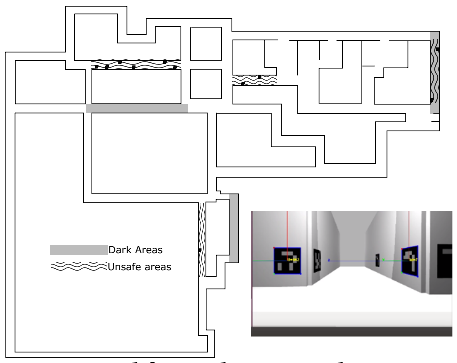


Figure 4: Map used for evaluation and ArUco markers.

The goal of the planner is to find plans that will optimally complete a mission by finding a combination of software architecture, path through the environment, and speed, to reach a goal. The path is a series of waypoints – the underlying robot navigation software controls how to get from one waypoint to another. In our scenario, the planner can be used to optimize three different utility functions: (a) *favor timeliness* will favor plans that reach the goal quickly; (b) *favor efficiency* will choose plans that minimize battery consumption; and (c) *favor safety* will search for plans that minimize the chance of damaging collisions with obstacles. This enables us to test the multi-dimensional quality aspect of our approach.

For efficiency, we defined the power consumption for each element in the architecture in milliwatt hours (mwh), based on physical experiments with the robot and documented power specifications of sensors. For timeliness and safety, we experimentally ran the robot at least eleven times over each segment in the map for every configuration and speed and calculated the average time and probability of collision from the collected data.

4.1 Experiment Setup and Results

4.1.1 Run-time tractability. To show that our approach is tractable and works on real robot software we integrated it as a planner in Rainbow [16] and ran Rainbow in a loop on top of (mostly) existing third-party robotic software written for the Turtlebot described earlier. These missions took as input the waypoints to visit and instructed the robot accordingly. We used the standard ROS node MoveBase[42] for navigation between waypoints.

We simulated the robot using the Gazebo simulator [18] v7.9, with a world generated from our map, including walls with spaced ArUco markers and physics (e.g., lighting, gravity, friction, mass). We added plugins to simulate power consumption, in addition to using standard simulations for the sensors and actuators. We customized Rainbow, developing probes, gauges, and models to interact with the robot. We also developed analyses for mission and robot state, architecture, and power. Finally, we wrote an adaptation manager implementing the approach described in this paper and integrated it with Alloy and PRISM. The adaptation manager synthesizes plans encoded as instruction graphs given to the robot which manages configuration and mission changes.

The starting condition of each test was a tuple {start_waypoint, target_waypoint, start_configuration, utility_preference}. To enable the comparison, we ran each test under three conditions: (A) no perturbations, and no adaptation; (B) same as A, but with perturbations, and (C) same as B but with adaptation enabled. When the test

	Start configuration									
Final configuration	amcl-kinect	amcl-lidar	aruco-camera	mrpt-kinect	mrpt-lidar	amcl-kinect	amcl-lidar	aruco-camera	mrpt-kinect	mrpt-lidar
amcl-kinect	—	1.00	23.95	3.22	1.11	—	0.10	0.13	0.10	0.07
amcl-lidar	2.89	—	**0.33**	9.11	**0.33**	0.06	—	0.11	0.14	0.13
aruco-camera	**11.33**	5.56	—	**13.67**	5.78	0.07	0.07	—	0.05	0.05
mrpt-kinect	1.22	**6.44**	**2.9**	—	**2.11**	0.09	0.07	0.06	—	0.06
mrpt-lidar	16.44	**4**	0.00	4.67	—	0.19	0.14	0.00	0.14	—
aruco-headlamp	**8.22**	3.89	0.71	**2.33**	1.22	0.07	0.29	0.92	0.11	0.93
none	**1**	0.00	**21.43**	**1**	**5.67**	n/a	n/a	n/a	n/a	n/a
	Reconfigurations (%)					Utility				

Figure 5: Differences in reconfigurations and predicted utility during comprehensive testing.

driver determined that the test was finished (mission completed, no solution found, or timed out), it used ground truth readings of the current state of the robot and location in the world to evaluate the mission. A test case was graded Pass if the C condition got closer to the target than B and did so with better utility (i.e., how much time it took, how much battery was left over, whether the robot hit anything in the environment). It was graded inconclusive if both the B and C conditions performed the mission similarly, and Fail if the C condition completed the missions worse than B.

In total, we ran 838 test triples (a total of 2514 runs), in which 272 of the cases failed A condition. These triples included a random distribution of perturbations including killing both node and sensor. Our results were that adaptation restored mission success and utility score. Of the 838 test cases, 646 passed, 21 failed, and 171 were inconclusive. Interestingly, of the 272 cases where A failed, adaptation allowed mission success in 170 (62.5%) of those cases.

4.1.2 Co-adaptation quality improvement. One of the important aspects of our approach is that allowing a planner to co-adapt task and architecture models will result in better plans. To evaluate this, we ran experiments comparing *Adapting only architecture vs. adapting architecture and mission*, exploring more exhaustively the adaptation state space. Exhaustive exploration is challenging because of the number of tests that need to be considered. For the example above, we want to test all missions under each starting configuration favoring each of the utilities, allowing each perturbation and lighting condition which yields 297,420 starting conditions for the planner. In addition to this we want to consider planners that allow changes to only the software, or both software and task resulting in 594,840 planner runs. Running each of these as real robot tasks is intractable.

We were able to reduce the number by carefully considering how to eliminate uninteresting and duplicated coniditions – not every mission will lead to different choices, and some missions subsume others. Thus we considered only missions: **(i)** where different starting configurations and utility preferences result in different paths through the map, and **(ii)** that subsumed the missions filtered in the first case, i.e., if we have two missions that follow the path $l1 \rightarrow l2 \rightarrow l3$ and $l0 \rightarrow l1 \rightarrow l2 \rightarrow l3 \rightarrow l4$, we eliminate the shorter path because choices about the mission/configuration are considered in the longer path.

Following these heuristics, the number of missions examined was reduced to 100. With all starting conditions, perturbations, utilities, and planner configurations, this resulted in 74400 runs, which was still intractable to run on real missions. Instead, We used the same planner code base as in Section 4.1.1, except that we short-circuited the code implementing the MAPE loop and invoked only the planner. Additionally, we did not run the robot simulation and therefore did not run missions to completion like in Section 4.1.1. We ran these tests on an Intel NUC i7 3.1GHz dual core Ubuntu system with 16GiB of memory.

Figure 5 shows the differences in reconfigurations and utilities when we disallow changes to the mission. Each column represents the robot starting configuration before perturbation, while each row represents the configuration the planner chose in response to a perturbation. Additionally, each row is divided among two different aspects of the plan: (a) the percentage of times the plan had a different final configuration, and (b) the difference in utility of the plan. Each cell is shaded by the degree of difference across all planner runs; green and normal font means that allowing mission changes caused a change more times than requiring the mission to be fixed, blue and bold means the other way around. In this case, we see differences that clearly point to the advantage of being able to adapt both the architecture and the mission. Regarding reconfigurations, the planner is more often unable to find a valid solution (21.43% more times when starting with aruco-camera and 5.67% more often when mrpt-lidar is the start configuration). Utilities are also markedly better when considering both models in tandem.

Table 3: Timing (in s) for different planning characteristics. Δ indicates the change when considering architecture only.

	Base (mean)	Base (stddev)	Δ (mean)	Δ (stddev)
Reconfig. planning	5.57	1.48	0.001	0.23
Path finding	3.59	0.25	-2.86	0.2
Task planning	5.94	0.45	-4.31	0.53
Total time	15.1	1.67	-7.17	0.63

Table 3 shows the time data for the planner under different conditions. The time to construct the Alloy and PRISM files from templates was negligible and so was not considered. First, in the base planner case, the planner takes on average 15.1s, spending 5.57s generating reconfigurations (which involves finding valid new configurations using Alloy and then PRISM to construct reconfiguration plans), 3.59s using Djikstra's algorithm to generate paths, and 5.94s model checking reconfigurations against the selected paths to generate the most optimal plan. When only considering reconfigurations, there is a significant difference in the amount of time that the planner takes (saving ≃7s, mostly because it does not need to generate alternative paths, only model checking on the existing path to determine the best reconfiguration). With an average task execution time of 224s, this represents a 3.1% increase in time using our approach. Furthermore, if we analyze the uncategorized results, we determine that for this additional time, adaptation finds 4636 (12.46%) more solutions with good utility (i.e., adaptation with > 0.5 utility). Also, the overall increase in utility is 0.09 (9%) across solutions found. This indicates that the extra time spent to find these better adaptations is worth it in this domain.

5 RELATED WORK

Our work draws from several research areas, such as self-adaptive systems [12, 24], formal specification/synthesis of software architectures [3, 27, 32], and probabilistic model checking [21, 28, 45].

Self-adaptive systems. Including our own prior work, discussed in Section 1, support for generation of optimal structure that also considers the richer context of task execution as a primary concern in self-adaptive systems is still limited. Tajalli et al. [49] introduce an approach that uses planning via model checking and architecture-based mechanisms for adaptation, and Sykes et al. [47, 48] introduce assembly of configurations, using extra-functional information to guide the process. These approaches do not account for any uncertainties or dependencies among domain facets that might impact task completion or qualities.

Self-adaptation in cyber physical systems (CPS) is a cross-layer concern, where using multiple feedback loops and models poses challenges that remain mostly unaddressed, mapping concerns to layers and adaptation mechanisms, coordination of adaptation mechanisms within and across layers, achieving system-wide consistency of adaptation [37]. Recent work started addressing some of these challenges: MORPH [6] is a reference architecture to improve consistency among configuration and behavior management that advocates separating reconfiguration and behavior strategy synthesis, and reconciling them a posteriori when they are selected for execution. Compared to MORPH, our approach attempts to preserve combined quantitative guarantees of the reconfiguration and behavior strategies by synthesizing them semi-independently (separate solvers, but exchanging information among them). Borda et al. [5] tackle compositional verification by proposing a language to model self-adaptive CPS, although without incorporating any solution synthesis mechanisms. Gerostathopoulos et al. [19] introduce an approach that uses homeostasis as a principle to maintain operational state by adjusting a fixed set of adaptation strategies.

Formal specification and synthesis of software architectures. Some architecture synthesis approaches [3, 32] focus on structural properties. In addition to that, more recent approaches [7, 8] also consider behavioral, quantitative, and probabilistic aspects of system descriptions to optimize quantitative guarantees of generated architectures under uncertainty. However, these have limited support to reason about the wider context of a system in terms of the tasks that have to be performed and the mechanisms available to achieve self-adaptation. Within the large body of work on software architecture for robotics [1], only a small subset of approaches consider the wider context of tasks or self-adaptive capabilities: Park et al. [38] introduce a task-based approach that generates optimal software architecture to make robots provide service reliably with limited resources. Edwards et al. [14] introduce an approach that relies on architectural middleware to ensure consistency between a system's architecture and its implementation. These approaches are not equipped to reason under uncertainty or produce specifications of task plans that go beyond reconfiguration of the architecture.

Quantitative verification/planning under uncertainty. Planning in partially unknown environments for robotics has already been explored in various works [20, 31, 34]. Out of these approaches, the closest to ours [31] generates mobile robot controllers with probabilistic time-bounded guarantees on successful task completion, which also try to satisfy other soft goals. Task specifications are analysed using multi-objective MDP model checking, providing an efficient mechanism for exploring the task solution space, but has limited support to capture the richer execution context in the planning, including the software running on the robot.

6 DISCUSSION AND FUTURE WORK

In this paper we have presented a framework for self-adaptation in mobile robotics that considers interdependencies among multiple domain facets (and in particular, between architecture and task plans), multiple tradeoffs, and uncertainty. To the best of our knowledge, other existing approaches sometimes cover changes only to the task plan or to the architecture, while in other cases, they are not able to consider uncertainties and dependencies that impact adaptation outcomes and quantitative guarantees (cf., Section 5). While we have focused on mobile service robots, we believe that our approach would also apply in other domains where dependencies exist between structure and behavior, e.g., patrolling drones, assisted living [33], and disaster recovery [39].

We have shown that combining formal methods with complementary strengths allows reasoning about both structural and behavioral adaptation a rich trade-off space and under uncertainty: including simple behavioural models to models with richer structure and semantics (e.g., architecture and environment).

Our results show that divide-and-conquer reconfiguration and task planning while still considering their dependencies leads to remarkable improvements over considering only one of them, or treating them as independent problems, and that the resulting implementation is tractable within reasonable time constraints for the class of application we describe.

Despite the advantages shown by our approach, our study includes the following limitations: **(L1)** Specific type of mobile robotics scenario and mission, **(L2)** Simplified architectural model, in which we assume a restricted set of valid configurations that the planner can consider, consisting of the software components, their connections, and a limited set of configuration parameters, **(L3)** Generating formal specifications from information exposed through views is view-specific and hand-coded, **(L4)** Evaluation run in simulation only. This potential limitation is mitigated somewhat in the independent evaluation (Section 4.1.1), which was run on a real mission using the robot's full software stack, under a high-fidelity simulated environment.

In future work, we will address **(L1)** by exploring our approach in other domains mentioned above to provide a broader assessment of its generalizability. We will address **(L2)** by exploring the extension of our approach using machine learning to help improve the scalability in larger configuration spaces, extending our preliminary work in this direction [26]. We will address **(L3)** by investigating how to streamline model integration using tools and languages designed for model transformations like ATL and QVT [22].

ACKNOWLEDGMENTS

This material is based on research sponsored by AFRL and DARPA under agreement number FA8750-16-2-0042. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the AFRL, DARPA or the U.S. Government.

REFERENCES

[1] A. Ahmad and M. A. Babar, "Software architectures for robotic systems: A systematic mapping study," *Journal of Systems and Software*, vol. 122, pp. 16 – 39,

2016.
[2] Autonomous Stuff, "Velodyne LiDAR PUCK^{TM}," retrieved on Mar. 7, 2020 from https://www.autonomoustuff.com/wp-content/uploads/2016/08/VLP-16-Puck.pdf.
[3] H. Bagheri and K. J. Sullivan, "Model-driven synthesis of formally precise, stylized software architectures," *Formal Asp. Comput.*, vol. 28, no. 3, 2016.
[4] J.-L. Blanco, "Contributions to localization, mapping and navigation in mobile robotics," Ph.D. dissertation, PhD. in Electrical Engineering, University of Malaga, Nov 2009.
[5] A. Borda, L. Pasquale, V. Koutavas, and B. Nuseibeh, "Compositional verification of self-adaptive cyber-physical systems," in *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '18. New York, NY, USA: ACM, 2018, pp. 1–11.
[6] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel, "MORPH: A reference architecture for configuration and behaviour self-adaptation," in *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*. New York, NY, USA: ACM, 2015, pp. 9–16.
[7] R. Calinescu, M. Češka, S. Gerasimou, M. Kwiatkowska, and N. Paoletti, "Designing robust software systems through parametric markov chain synthesis," in *2017 IEEE International Conference on Software Architecture, ICSA*. Gothenburg, Sweden: IEEE, April 3-7 2017, pp. 131–140.
[8] J. Cámara, D. Garlan, and B. Schmerl, "Synthesizing tradeoff spaces with quantitative guarantees for families of software systems," *Journal of Systems and Software*, vol. 152, pp. 33 – 49, 2019.
[9] J. Cámara, D. Garlan, and B. R. Schmerl, "Synthesis and quantitative verification of tradeoff spaces for families of software systems," in *Software Architecture - 11th European Conference, ECSA*, ser. LNCS, vol. 10475. Springer, 2017, pp. 3–21.
[10] J. Cámara, A. Lopes, D. Garlan, and B. Schmerl, "Adaptation impact and environment models for architecture-based self-adaptive systems," *Sci. Comput. Program.*, vol. 127, pp. 50–75, 2016.
[11] J. Cámara, B. R. Schmerl, G. A. Moreno, and D. Garlan, "MOSAICO: offline synthesis of adaptation strategy repertoires with flexible trade-offs," *Autom. Softw. Eng.*, vol. 25, no. 3, pp. 595–626, 2018.
[12] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. R. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. J. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovski, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. D. Schlichting, D. B. Smith, J. P. Sousa, L. Tahvildari, K. Wong, and J. Wuttke, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, ser. Lecture Notes in Computer Science, R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds., vol. 7475. Springer, 2010, pp. 1–32.
[13] V. Dwivedi, D. Garlan, J. Pfeffer, and B. Schmerl, "Model-based assistance for making time/fidelity trade-offs in component compositions," in *11th International Conference on Information Technology: New Generations, ITNG 2014*. IEEE CS, 2014.
[14] G. Edwards, J. Garcia, H. Tajalli, D. Popescu, N. Medvidovic, G. Sukhatme, and B. Petrus, "Architecture-driven self-adaptation and self-management in robotics systems," in *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, May 2009, pp. 142–151.
[15] D. Fox, "KLD-sampling: Adaptive particle filters," in *Advances in Neural Information Processing Systems 14*. MIT Press, 2001.
[16] D. Garlan, S.-W. Cheng, A. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *IEEE Computer*, vol. 37, no. 10, 2004.
[17] D. Garlan, R. T. Monroe, and D. Wile, "Acme: Architectural description of component-based systems," in *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, 2000, pp. 47–68.
[18] Gazebosim.org, "Gazebo," Retrieved on Jan. 20th, 2020 from http://gazebosim.org/.
[19] I. Gerostathopoulos, D. Skoda, F. Plasil, T. Bures, and A. Knauss, "Tuning self-adaptation in cyber-physical systems through architectural homeostasis," *Journal of Systems and Software*, vol. 148, pp. 37 – 55, 2019.
[20] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning and Acting*, 1st ed. New York, NY, USA: Cambridge University Press, 2016.
[21] F. Giunchiglia and P. Traverso, "Planning as model checking," in *Recent Advances in AI Planning*, S. Biundo and M. Fox, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1–20.
[22] R. Hebig, C. Seidl, T. Berger, J. K. Pedersen, and A. Wasowski, "Model transformation languages under a magnifying glass: A controlled experiment with Xtend, ATL, and QVT," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 445–455.
[23] R. Hilliard, "Views and Viewpoints in Software Systems Architecture," in *Proc. of the First Working IFIP Conference on Software Architecture*, San Antonio, TX, 1999, pp. 22–24.
[24] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing - degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, no. 3, 2008.
[25] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, Apr. 2002.
[26] P. Jamshidi, J. Cámara, B. Schmerl, C. Kästner, and D. Garlan, "Machine learning meets quantitative planning: Enabling self-adaptation in autonomous robots," in *Proceedings of the 14th Symposium on Software Engineering for Adaptive and Self-Managing Systems*, Montreal, Canada, 25-26 May 2019.
[27] J. Kim and D. Garlan, "Analyzing architectural styles," *J Syst Software*, vol. 83, no. 7, pp. 1216–1235, 2010.
[28] M. Z. Kwiatkowska, G. Norman, and D. Parker, "Stochastic model checking," in *FM for Performance Evaluation, 7th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems*, ser. LNCS, vol. 4486. Springer, 2007.
[29] ——, "PRISM 4.0: Verification of probabilistic real-time systems," in *Computer Aided Verification - 23rd International Conference, CAV*, vol. 6806. Springer, 2011, pp. 585–591.
[30] M. Z. Kwiatkowska and D. Parker, "Automated verification and strategy synthesis for probabilistic systems," in *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013*, ser. Lecture Notes in Computer Science, vol. 8172. Springer, 2013, pp. 5–22.
[31] B. Lacerda, D. Parker, and N. Hawes, "Optimal and dynamic planning for Markov decision processes with co-safe LTL specifications," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sep. 2014, pp. 1511–1516.
[32] S. Maoz, J. O. Ringert, and B. Rumpe, "Synthesis of component and connector models from crosscutting structural views," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE'13*. ACM, 2013.
[33] G. Mason, R. Calinescu, D. Kudenko, and A. Banks, "Assurance in reinforcement learning using quantitative verification," in *Advances in Hybridization of Intelligent Methods: Models, Systems and Applications*, ser. Smart Innovation, Systems and Technologies, I. Hatzilygeroudis and V. Palade, Eds. Springer International Publishing AG, 2018, vol. 85, pp. 71–96.
[34] C. Menghi, S. Garcia, P. Pelliccione, and J. Tumova, "Multi-robot LTL planning under uncertainty," in *Formal Methods. FM2018*, ser. Lecture Notes in Computer Science, K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, Eds., vol. 10951. Cham: Springer International Publishing, 2018, pp. 399–417.
[35] C. Mericli, S. Klee, J. Paparian, and M. Veloso, "An Interactive Approach for Situated Task Specification through Verbal Instructions," in *Proceedings of AAMAS'14, the Thirteenth International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Paris, France, May 2014.
[36] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl, "Proactive self-adaptation under uncertainty: a probabilistic model checking approach," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015, pp. 1–12.
[37] H. Muccini, M. Sharaf, and D. Weyns, "Self-adaptation for cyber-physical systems: a systematic literature review," in *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2016*. ACM, May 14-22 2016, pp. 75–81.
[38] Y.-S. Park, H.-M. Koo, and I.-Y. Ko, "A task-based and resource-aware approach to dynamically generate optimal software architecture for intelligent service robots," *Softw. Pract. Exper.*, vol. 42, no. 5, pp. 519–541, May 2012.
[39] C. Paterson, R. Calinescu, D. Wang, and S. Manandhar, "Using unstructured data to improve the continuous planning of critical processes involving humans," in *Proceedings of the 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2019*. Montreal, QC, Canada: ACM, May 25-31 2019, pp. 25–31.
[40] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.
[41] F. J. Romero-Ramirez, R. Muñoz-Salinas, and R. Medina-Carnicer, "Speeded up detection of squared fiducial markers," *Image and Vision Computing*, vol. 76, pp. 38–47, 2018.
[42] ROS.org, "Move_base - ROS Wiki," Retrieved on Jan. 20th, 2020 from http://wiki.ros.org/move_base.
[43] I. Ruchkin, S. Samuel, B. Schmerl, A. Rico, and D. Garlan, "Challenges in physical modeling for adaptation of cyber-physical systems," in *Workshop on MARTCPS Models at Runtime and Networked Control for Cyber Physical Systems at IEEE World Forum on the Internet of Things*, Reston, Virginia, 12-14 December 2016.
[44] A. Santos, A. Cunha, and N. Macedo, "Static-Time Extraction and Analysis of the ROS Computation Graph," in *2019 Third IEEE International Conference on Robotic Computing (IRC)*, Feb 2019, pp. 62–69.
[45] M. Schoppers, "Universal plans for reactive robots in unpredictable environments," in *Proceedings of the 10th International Joint Conference on Artificial Intelligence. Milan, Italy, August 1987*. Morgan Kaufmann, 1987.
[46] M. Shaw and D. Garlan, *Software architecture - perspectives on an emerging discipline*. Prentice Hall, 1996.

[47] D. Sykes, W. Heaven, J. Magee, and J. Kramer, "From goals to components: a combined approach to self-management," in *2008 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2008*, B. H. C. Cheng, R. de Lemos, D. Garlan, H. Giese, M. Litoiu, J. Magee, H. A. Müller, and R. N. Taylor, Eds. Leipzig, Germany: ACM, May 12-13 2008, pp. 1–8.

[48] ——, "Exploiting non-functional preferences in architectural adaptation for self-managed systems," in *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*. Sierre, Switzerland: ACM, March 22-26 2010.

[49] H. Tajalli, J. Garcia, G. Edwards, and N. Medvidovic, "PLASMA: a plan-based layered architecture for software model-driven adaptation," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering*. Antwerp, Belgium: ACM, September 20-24 2010.

[50] M. Veloso, J. Biswas, B. Coltin, and S. Rosenthal, "Cobots: Robust symbiotic autonomous mobile service robots," in *Proceedings of the 24th International Conference on Artificial Intelligence*, ser. IJCAI'15. AAAI Press, 2015, pp. 4423–4429.

[51] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.