Feature-Oriented Modelling and Analysis of a Self-Adaptive Robotic System

JULIANE PÄßLER, Department of Informatics, University of Oslo, Oslo, Norway
MAURICE H. TER BEEK, CNR-ISTI, Pisa, Italy
FERRUCCIO DAMIANI, University of Turin, Torino, Italy
CLEMENS DUBSLAFF, Eindhoven University of Technology, Eindhoven, Netherlands and Dresden University of Technology, Dresden, Germany
EINAR BROCH JOHNSEN, Department of Informatics, University of Oslo, Oslo, Norway
SILVIA LIZETH TAPIA TARIFA, Department of Informatics, University of Oslo, Oslo, Norway

Improved autonomy in robotic systems is needed for innovation in, e.g., the marine sector. Autonomous robots that are let loose in hazardous environments, such as underwater, need to handle uncertainties that stem from both their environment and internal state. While self-adaptation is crucial to cope with these uncertainties, bad decisions may cause the robot to get lost or even to cause severe environmental damage. Autonomous, self-adaptive robots that operate in uncontrolled environments full of uncertainties need to be reliable! Since these uncertainties are hard to replicate in test deployments, we need methods to formally analyse self-adaptive robots operating in uncontrolled environments. In this article, we show how feature-oriented techniques can be used to formally model and analyse self-adaptive robotic systems in the presence of such uncertainties. Self-adaptive systems can be organised as two-layered systems with a *managed* subsystem handling the domain concerns and a *managing* subsystem implementing the adaptation logic. We consider a case study of an Autonomous Underwater Vehicle (AUV) for pipeline inspection, in which the managed subsystem of the AUV is modelled as a family of systems, where each family member corresponds to a valid configuration of the AUV which can be seen as an operating mode of the AUV's behaviour. The managing subsystem of the AUV is modelled as a control layer that is capable of dynamically switching between such valid configurations, depending on both environmental and internal uncertainties. These uncertainties are captured in a probabilistic and highly configurable model. Our modelling approach allows us to exploit powerful formal methods for feature-oriented systems, which we illustrate by analysing safety properties, energy consumption, and multi-objective properties, as well as performing parameter synthesis to analyse to what extent environmental conditions affect the AUV. The case study is realised in the

This work was partially supported by the European Union's Horizon 2020 Framework Programme through the MSCA network REMARO (Grant Agreement No 956200), by the Italian project NODES (which has received funding from the MUR – M4C2 1.5 of PNRR with grant agreement no. ECS00000036), by the Italian MUR PRIN 2020TL3X8X project T-LADIES (Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems), by the German DFG under the projects EXC 2050/1 (CeTI, project ID 390696704, as part of Germany's Excellence Strategy) and TRR 248 (see https://perspicuous-computing.science, project ID 389792660), and by the Dutch NWO through Veni grant VI.Veni.222.431.
Authors' Contact Information: Juliane Päßler, Department of Informatics, University of Oslo, Oslo, Norway; e-mail: julipas@ifi.uio.no; Maurice H. ter Beek, CNR-ISTI, Pisa, Italy; e-mail: maurice.terbeek@isti.cnr.it; Ferruccio Damiani, University of Turin, Torino, Italy; e-mail: ferruccio.damiani@unito.it; Clemens Dubslaff, Eindhoven University of Technology, Eindhoven, Netherlands and Dresden University of Technology, Dresden, Germany; e-mail: c.dubslaff@tue.nl; Einar Broch Johnsen, Department of Informatics, University of Oslo, Oslo, Norway; e-mail: einarj@ifi.uio.no; Silvia Lizeth Tapia Tarifa, Department of Informatics, University of Oslo, Oslo, Norway; e-mail: sltarifa@ifi.uio.no.

ACM 1433-299X/2025/10-ART32
https://doi.org/10.1145/3709159

probabilistic feature-oriented modelling language and verification tool ProFeat, and in particular exploits family-based probabilistic and parametric model checking.

CCS Concepts: • **Software and its engineering → Software product lines**; **Formal methods**; **Model checking**; • **Computer systems organization → Embedded and cyber-physical systems**; **Robotics**; • **Mathematics of computing → Probabilistic representations**; • **Theory of computation → Verification by model checking;**

Additional Key Words and Phrases: Feature models, probabilistic model checking, parametric model checking, self-adaptive systems, cyber-physical systems, robotics

ACM Reference Format:
Juliane Päßler, Maurice H. ter Beek, Ferruccio Damiani, Clemens Dubslaff, Einar Broch Johnsen, and Silvia Lizeth Tapia Tarifa. 2025. Feature-Oriented Modelling and Analysis of a Self-Adaptive Robotic System. *Form. Asp. Comput.* 37, 4, Article 32 (October 2025), 39 pages. https://doi.org/10.1145/3709159

1 Introduction

More than 70 percent of the surface of Earth is covered in water. The blue economy, which includes expanding sectors such as marine energy, bio-industry and food production, water desalination, mining, underwater construction and ecosystem monitoring, is crucially dependent on underwater vehicles that are able to operate in environments that are inaccessible or hostile to humans. The sustained growth of the blue economy and control of undesirable environmental impacts essentially depends on two factors: increased *autonomy* in underwater robotics and *reliability* of underwater operations, as control shifts from human operators to the robots themselves, expanding their range of activities. However, this growth is slowed down by a gap between what the underwater robotics industry delivers today and the required level of autonomy and reliability. While there is currently a significant push towards increased autonomy, many underwater robots are still controlled by human operators at the mission level. The need for human operators significantly increases cost and restricts the activities in which underwater robots can engage. Due to low bandwidth underwater, such control goes via cable between the underwater robot and the operator on land or on ship.

For autonomous mission control in underwater robotics, a particular challenge stems from the environments in which the robots operate. Generally, these environments are not fully understood and may be subject to change, potentially causing surprises for the robot. For example, a robot performing underwater pipeline inspection may need to change how it performs its mission because of unexpected underwater currents, reduced water visibility, or even a landslide. One approach to dealing with such environments is through *self-adaptation* [86]. Self-adaptation enables the robots to autonomously adapt their mission-level control strategies to the environment in which they operate—during the mission itself. Enabling robots to adapt in this way provides several advantages. Self-adaptive mission control expands the range of scenarios in which the robots are able to operate, resulting in a higher level of autonomy. Self-adaptive mission control has been realised in, e.g., *Metacontrol* [55], in which control strategies can be manipulated by activating, deactivating or adjusting them on the fly; MROS [17] implements Metacontrol for the **Robot Operating System** (**ROS**) [74].

Let us briefly recall how a **Self-Adaptive System** (**SAS**) can be realised using a two-layered architecture which decomposes the system into a *managed* and a *managing* subsystem [61], see Figure 1. The *managed* subsystem deals with the domain concerns and tries to reach the goals set by the system's user, e.g., navigating a robot to a specific location. The *managing* subsystem handles the adaptation concerns and defines an adaptation logic that specifies a strategy on how the

system can fulfil the goals under uncertainty [86], e.g., adapting to changing environmental conditions. While the managed subsystem may affect the environment via its actions, the managing subsystem monitors the environment and the internal state of the managed subsystem. By using the adaptation logic, the managing subsystem decides whether and which changes are needed and adapts the managed subsystem accordingly.

Unless done in the right way, autonomous controllers for underwater vehicles can potentially be both damaging and expensive, as vehicles may get lost and even, in the worst case, cause harm to installations or the marine environment. However, it is cumbersome to validate new designs for autonomous mission-level control of underwater vehicles in situ: vehicles may get lost during validation and only a few real-life scenarios get tested in practice. To address this problem, it is useful to validate the mission control in simulation. This leads to the emergence of configurable simulators that target the mission control layer of robotic software stacks. Using simulation scenarios, the autonomous mission control layer of the underwater robot can be tested and validated much more efficiently. Simulators, such as the open-source exemplars UNDERSEA [45] and SUAVE [76], allow scenarios for mission-level control to be tested and compared in simulation, based on a realistic robot software stack, and enable the comparison of different strategies for autonomous mission control.

However, testing and validation might not be enough to assure the system operators that the autonomous robots will operate as desired. Therefore, in addition to testing and validation, rigorous analysis techniques such as model checking [5] are often applied in industry [7] to help provide safety guarantees for these and other (mission-critical) systems even in the presence of uncertainties. Our work considers analysis techniques for autonomous underwater robots, based on model checking, and complements simulation-based validation techniques for **Autonomous Underwater Vehicles** (**AUVs**). Specifically, our goal in this article is to model and analyse a case study of mission control for an AUV, realised as a two-layered SAS which was taken from the exemplar SUAVE. To this aim, we organise the adaptation space of the SAS as a reconfiguration problem, using feature-oriented modelling techniques [22, 23]. The functionalities of the *managed subsystem* of the AUV are modelled in a feature model, making the dependencies and requirements between the components of the AUV explicit. The behaviour of the managed subsystem is modelled as a **featured Markov Decision Process (fMDP)**, i.e., a probabilistic transition system were feature-guarded transitions can only be taken in a system configuration with matching features. Thus, the managed subsystem of the AUV is modelled as a family of systems whose family members correspond to valid feature configurations which can be understood as modes of operation of the AUV. As the behaviour of the AUV depends on environmental and internal conditions, which are both hard to control, we opted for a probabilistic model in which uncontrolled events, like a thruster failure, occur with given probabilities. We model the behaviour of the *managing subsystem* as a control layer that switches between the feature configurations of the managed subsystem according to input from the probabilistic environment model and the managed subsystem. In our case study, we consider an abstract version of an AUV, focusing on core functionalities of the AUV with limited features and variability. The resulting model can be refined and extended in several ways

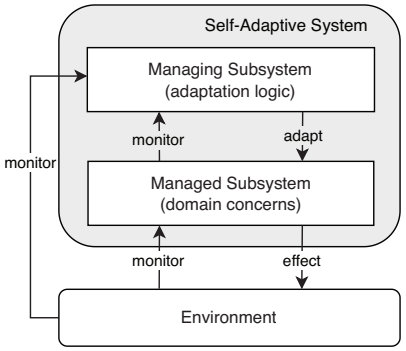


Fig. 1. The architecture of a two-layered SAS with a managed subsystem implementing the application logic and a managing subsystem implementing the adaptation logic ([69]).

depending on the necessary degree of realism to verify specific requirements. Finally, we model the *environment* as a probabilistic transition system which abstracts from the real environments in which AUVs operate, yet captures essential aspects of changes in the environment that can affect the behaviour of the AUV.

The analysis of the resulting model considers five different aspects: (1) safety guarantees concerning a mission's duration and energy usage that can be used for the deployment of the AUV; (2) safety guarantees concerning unsafe states; (3) the impact of different environments on the safety guarantees; (4) parameter synthesis to analyse to what extent environmental conditions affect the AUV; and (5) tradeoffs between mission duration and energy usage. Together, the analyses provide a good indication of the feasibility of model checking as a viable technique to analyse AUVs. The case study is modelled in ProFeat [23], a tool for probabilistic family-based model checking, which provides a means to simultaneously model check, in a single run, properties of all different configurations in a *family model* [82]. The analysis of the model was carried out in the backends for ProFeat, in PRISM [62] and Storm [54], depending on the analysed properties. We close the article with a discussion about the suitability of family-based modelling techniques for analysing SASs such as AUVs, and the relation of these techniques to dynamic ***Software Product Lines*** (**SPLs**) [14, 33, 40, 49, 80].

Contribution. This article is an extension of the conference publication by Päßler et al. [69] and the related artefact [70], recently published as an original software publication [68], which used techniques from SPLs to model and analyse the case study of a self-adaptive AUV as a two-layered system, providing the following contributions:

- a case study of an SAS from the underwater robotics domain, modelled as a probabilistic feature-guarded transition system with dynamic feature switching [69];
- automated verification of (quantitative) properties that are important for roboticists, using family-based analysis [69];
- a configurable software model of a **Dynamic SPL** (**DSPL**), reflecting the self-adaptive AUV [68, 70].

In more detail, this extended version of the work additionally provides the following contributions:

- enabling more variants of the case study which can include another environment and an AUV that can operate at more altitudes (see Section 4), enabling analysis of how the environment and the number of possible altitudes influence the analysis results;
- extending the analysis with
  - an analysis of how the number of possible altitudes influences the analysis results using PRISM (see Sections 5.2 and 5.3),
  - an analysis of how the environments influence the analysis results using PRISM experiments (see Section 5.4),
  - an analysis of how the probability of currents influences the analysis results using parametric model checking with Storm (see Section 5.5), and
  - multi-objective queries where we explore achievability, numerical and Pareto queries using Storm (see Section 5.6);
- a discussion about the suitability of using SPL techniques and tools for modelling and analysing SASs (see Section 7.1).

To the best of our knowledge, we are the first to showcase the applicability of feature-oriented verification using Storm and performing parameter synthesis.

Outline. Section 2 presents the case study of pipeline inspection with an AUV, setting up the context of the SAS analysed in this article. The analysis technique of family-based model checking

and a brief account on (dynamic) SPLs and ProFeat is provided in Section 3. Section 4 explains both the behaviour of the managed and managing subsystem of the AUV and the environment, as well as their implementation in ProFeat. Section 5 presents quantitative analyses conducted on the case study, involving the tools ProFeat, PRISM, and Storm. Section 6 summarises related work and Section 7 discusses our results and ideas for future work.

2 Case Study: Pipeline Inspection by AUV

In this section, we introduce the case study of this article, an AUV used for pipeline inspection. The case study is inspired by the case study of the exemplar SUAVE [76]. In SUAVE, the AUV has been implemented in a simulator using ROS2 [66] to enable the comparison of different self-adaptation strategies, i.e., different implementations of a managing subsystem. The full SUAVE model would clearly exceed the limits when turning towards a formal analysis. We hence abstracted the case study to its essential characteristics, aiming towards verification of the core functionalities and self-adaptivity of the AUV in various exemplary environments.

As within the SUAVE case study, an AUV has the mission to first find and then follow and inspect a pipeline located on a seabed. For moving in different directions, the AUV has several thrusters. All thrusters are needed for a reliable operation of the AUV, however, thruster failures are possible. Furthermore, the AUV can operate at different altitudes, where a higher altitude provides a wider field of view and thus increases the AUV's chances of finding the pipeline. We assume that switching between altitudes increases the energy consumption. The only aspect of the environment considered in this case study is the water visibility: it reflects the maximum distance at which the AUV can perceive objects; we assume that it changes probabilistically during runtime, e.g., due to currents that swirl up the seabed. By assuming probabilistic behaviour of the environment, we assume to have partial knowledge of the environment which is in line with the fact that it is in general difficult to understand.

The AUV's operation is influenced by thruster failures, the water visibility, and its tasks (searching for the pipeline and following it). When a thruster fails, the AUV starts drifting off its intended path and the thruster has to be restarted before the AUV can continue its mission. Thruster failures are more likely at a lower altitude because seaweed might wrap around the thrusters. If a thruster failure happens while following the pipeline and the AUV drifts too much off its path, the pipeline can be lost and the AUV needs to search for it again. The water visibility determines the maximum altitude at which the AUV can perceive the seabed, and thus the maximum altitude at which it can operate. The higher the visibility, the higher the maximum possible operational altitude. The task determines whether the AUV switches between altitudes or stays at one altitude. When following the pipeline, the AUV does not benefit from a wider field of view and should thus operate at the lowest possible altitude to keep the energy consumption low by not switching between altitudes. However, when searching for the pipeline, the AUV can operate at every altitude.

As in SUAVE, the AUV is separated into a managed and a managing subsystem. The managed subsystem is responsible for searching for and following the pipeline. It has different configurations that consist of a task and an altitude where a configuration can be understood as a mode of operation of the AUV. Thus, the managed subsystem can in fact be seen as a family of systems where each family member corresponds to a system configuration. The managing subsystem chooses a configuration of the managed subsystem depending on the current task and the water visibility. When following the pipeline, the lowest possible altitude is chosen to decrease the energy consumption. When searching for the pipeline, a higher altitude is preferred because it increases the probability of finding the pipeline and it decreases the probability of a thruster failure. However, the altitudes at which the AUV can operate are determined by the water visibility. The higher the visibility, the higher the maximum operational altitude. Considering, e.g., three possible

altitudes at which the AUV can operate, the managed subsystem consists of four family members: one family member for searching at each possible altitude and one for following the pipeline at the lowest altitude. The managing subsystem switches between the first three during the search for the pipeline depending on the current water visibility and activates the last one when the pipeline should be followed. Note that, unlike in SUAVE, we assume that thruster failures are handled by the managed subsystem and not the managing subsystem.

In this article, we present an extension of the first formal modelling of the AUV case study by Päßler et al. [69]. While the original case study only considered three possible altitudes, we additionally provide a setting with five altitudes to investigate the impact of a more fine-grained implementation of adaptation by the managing subsystem. These settings are hereafter referred to as the *three-altitudes* case study and the *five-altitudes* case study, respectively; SUAVE corresponds to the three-altitudes case study.

3 SPLs and Family-based Model Checking

SPL engineering concerns the management of *commonality* and *variability*, usually defined in terms of core and optional features, across families of software products [1, 29, 71]. There exist many different definitions of what constitutes a *feature* [27], ranging from "an increment in product functionality" [6] to the approach adopted in this article, namely, "anything users or client programs might want to control about a concept is a feature" [32, Chapter 4: Feature Modeling]. Selected features provide a *feature configuration*, from which those that give rise to an actual software product are collected in a set of *valid configurations*. Such sets of valid configurations are commonly modelled in a *variability* or *feature model*, most prominently using structures such as feature diagrams [59] (see, e.g., [12, 81] for surveys). The actual behaviour of software products to be deployed can be modelled by choosing annotative [26], compositional [79], or hybrid approaches [40, 60]. In annotative approaches, the behaviour of the whole SPL is modelled in one behavioural model where parts of behaviour are annotated by features, specifying the behaviour of the products containing this feature. Compositional approaches specify the behaviour of each single feature in an isolated feature module, and the feature modules are then composed to determine the behaviour of the software product. In this article, we opt for a hybrid approach, where the behaviour of (sets of) features is modelled in feature modules that may also contain feature-guarded behaviour to specify feature interactions or fine-grained variability [40].

Since the number of potential software products is exponential in the number of features, the analysis of SPLs is a challenging task [82]. Many different analysis approaches for SPLs have been proposed using model checking [28], static analysis [8], theorem proving [83], or testing [16]. A one-by-one analysis, investigating every product in isolation, easily becomes infeasible. Hence, when giving a behavioural model of the SPL, an all-in-one analysis is to be preferred, where the behaviour of all products is modelled in a *family model* and analysed in a single analysis step [26]. As products of an SPL typically share a large number of features and behaviours, this family model can usually be compactly represented and analysed by exploiting symbolic representations and analysis techniques [25]. In this article, we consider the *SPL model-checking problem*, which was first recognised in the seminal article of [28]. It generalises the classical model-checking problem as follows: given a logic formula φ, determine for every valid configuration whether φ is satisfied (and provide a counterexample for each configuration not satisfying φ). The straightforward one-by-one approach to solve this problem is by *product-based model checking*, applying classical model checking to each of the products in the SPL. *Family-based model checking* implements an all-in-one approach where all products of a family model, modelling the combined behaviour of all products, are simultaneously checked [82]. The answer to the SPL model-checking problem is obtained by projecting the results gained from the family model to single configurations. This approach is also

well-suited for analysing reliability or other quantitative aspects of SPLs through *family-based probabilistic model checking* [11, 42]. Another dimension where the use of family models for SPLs excels is in modelling and analysing *DSPLs* where feature configurations can be switched during runtime [49, 51]. Such reconfigurations can be modelled in a similar way as modelling actual behaviour in feature modules, leading to *feature controllers* where each feature corresponds to a variable of the module (see, e.g., [40]). As a family model contains the behaviour of all valid feature configurations, a feature controller can select the possible behaviour of the current feature configuration within the family model during runtime, and change between the behaviours of the products.

The tool ProFeat[1] [23] implements this kind of analysis for DSPLs by choosing a hybrid modelling approach combined with family-based probabilistic model checking. Thereby, all possible products and reconfigurations between products can be analysed in one run. In particular, ProFeat provides a means to model probabilistic system families in an intuitive specification language as well as perform family-based quantitative analysis on them. It extends the probabilistic model checker PRISM[2] [62] with functionalities such as family models, feature modules, and the feature controller. Technically, ProFeat performs a translation of the feature-oriented model to PRISM's input language, which is then analysed with family-based probabilistic model checking. Hence, all of the analysis capabilities of PRISM carry over to ProFeat as well. Furthermore, note that this translational approach also enables the use of other analysis tools supporting PRISM's input language, e.g., the state-of-the-art probabilistic model checker Storm [54]. The translation ensures feature-specific properties such as reconfiguration within the valid configurations and respecting the rules for reconfiguration. Analysis results from the model checkers are translated back automatically in a feature-aware representation, including compact symbolic representations through decision diagrams [23].

4 Feature-oriented Modelling of the AUV Case Study

In this section, we show that feature-oriented modelling is well-suited to formally specify the structure and behaviour of the AUV case study. First, we observe that the formal model to describe the behaviour of the AUV has to support quantitative aspects such as failure probabilities, time, and energy consumption. Hence, *probabilistic* feature-oriented modelling approaches [11, 42] have to be preferred compared to non-probabilistic ones [10, 26]. Furthermore, the managing subsystem has to be able to switch between configurations of the managed subsystem such that *dynamic reconfigurations* are supported as well [11, 23, 84]. Due to the clear separation of concerns between behaviour of features and the reconfiguration component, we decided to model the case study with the probabilistic feature-oriented engineering framework ProFeat.

Similar to an SAS, a ProFeat model can be seen as a two-layered model, as illustrated in Figure 1. The behaviour of a family of systems that differ in their features, such as the managed subsystem of an SAS, can be specified. Then a so-called *feature controller* can activate and deactivate the features during runtime, and thus change the behaviour of the system, such as the managing subsystem of an SAS that changes the configuration of the managed subsystem. Furthermore, the environment can be specified as a separate module that interacts with the managed and managing subsystem. Thus, ProFeat is well suited to model and analyse the case study described in Section 2.

Technically, a ProFeat model consists of three parts: an obligatory feature model that specifies features and their relations and constraints, obligatory modules that specify the behaviour of the features, and an optional feature controller that activates or deactivates features. This structure is also reflected in our implementation of the two case study variants in ProFeat.[3]

[1] https://pchrszon.github.io/profeat
[2] https://www.prismmodelchecker.org/manual
[3] The models are publicly available [67].

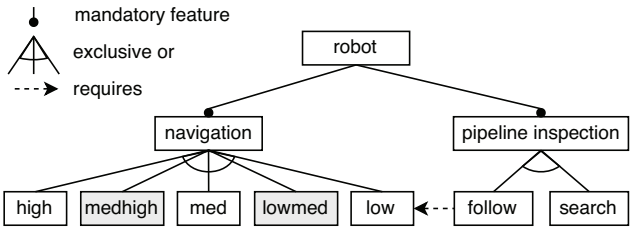


Fig. 2. Feature model of the five-altitudes model; the three-altitudes model does not contain the features with a grey background.

Both pipeline inspection case studies ultimately specify a Markov decision process, i.e., a state-based operational model that includes non-deterministic and probabilistic state transitions. The probabilities and parameters used in our model are estimates and have not been validated by experiments. This article aims at showing the practical feasibility of using feature-oriented modelling for quantitative SAS verification, not to model a realistic AUV. More realistic values can be obtained by doing experiments with a real AUV and by consulting domain experts and are left for future work.

In this section, we follow the modelling structure of the two case study variants in ProFeat. First, we introduce the feature-oriented view of the case study along with its implementation as a feature model in Section 4.1. Then we explain the behaviour of the managing subsystem, i.e., how the managing subsystem switches between valid feature configurations of the feature model, and its implementation as a feature controller in ProFeat in Section 4.2. Lastly, we describe the behaviour of the managed subsystem and the environment as well as their implementation as modules in ProFeat in Sections 4.3 and 4.4, respectively. For each of the modelling components, we describe both case studies by focusing on the three-altitudes case study and explaining the additions for the five-altitudes case study.

4.1 The Feature Model

Feature-Oriented View of the Case Study. The important characteristics of the managed subsystem for self-adaptation are the tasks of the AUV and the possible altitudes. Thus, we consider them to be the AUV's *features*. The features corresponding to the tasks of the AUV are called *search* and *follow*, and the features corresponding to the possible altitudes are called *low*, *med* (for medium), and *high* in the three-altitudes case study with additional features *lowmed* and *medhigh* in the five-altitudes case study. The basis for feature-oriented modelling is the so-called *feature model* that abstractly relates how features can be combined, i.e., which activation of features yields a valid configuration of the system. *Feature diagrams* [1, 59] constitute the standard feature model formalism. They hierarchically structure features to specify the high-level dependencies between the features. Specifically, an activation of a subfeature implies the activation of its parent feature. To create a feature model for the AUV case study, the altitude features are considered to be subfeatures of the feature *navigation*, the task features to be subfeatures of the feature *pipeline inspection*, in both cases connected by an exclusive or relation, and *navigation* and *pipeline inspection* are considered to be mandatory subfeatures of the root feature *robot*. Furthermore, since following the pipeline requires a low altitude, the feature *follow* is connected to the feature *low* with a requires relationship. The resulting feature model can be seen in Figure 2 where the feature model of the three-altitudes case study does not contain the features with the grey background.

Each configuration of the AUV contains the features *robot*, *navigation*, and *pipeline_inspection* as well as exactly one sub-feature for navigation and one for pipeline inspection, while moreover

the feature *follow* requires the feature *low*. This yields four different configurations of the managed subsystem of the AUV in the three-altitudes case study (one including the leaf features *search* and *high*, one including *search* and *med*, one including *search* and *low*, and the last one including *follow* and *low*) and two additional configurations in the five-altitudes case study (one including the leaf features *search* and *medhigh* and including one *search* and *lowmed*). Therefore, the valid configurations of the three-altitudes case study are the following (where the leaf features are displayed in bold font):

{*robot*, *navigation*, *pipeline_inspection*, **search**, **high**}, {*robot*, *navigation*, *pipeline_inspection*, **search**, **med**},
{*robot*, *navigation*, *pipeline_inspection*, **search**, **low**}, {*robot*, *navigation*, *pipeline_inspection*, **follow**, **low**}.

The five-altitudes case study includes the following configurations in addition to those of the three-altitudes case study:

{*robot*, *navigation*, *pipeline_inspection*, **search**, **medhigh**}, {*robot*, *navigation*, *pipeline_inspection*, **search**, **lowmed**}.

By differentiating between configurations depending on the selected features, distinct behaviour of the AUV can be specified depending on the configuration as described in Section 4.3.

The ProFeat Implementation of the Feature Model. We show how the feature model of the case study is expressed in ProFeat, including connections and constraints among features. Each feature is specified within a **feature** … **endfeature** block, and the declaration of the root feature is done in a **root feature** … **endfeature** block. An excerpt of the implementation of the root feature of the pipeline inspection case study according to Figure 2 is displayed in Listing 1. The root feature can be decomposed into subfeatures; in this case, in only one, the subfeature `robot`, see Line 2. The `all of` keyword indicates that all subfeatures have to be included in the feature configuration if the parent feature, in this case the root feature, is included. It is, e.g., also possible to use the `one of` keyword if exactly one subfeature has to be included, see Line 2 of Listing 2. The modules modelling the behaviour of the root feature are specified after the keyword **modules**. In our case studies, the root feature is the only feature specifying modules, thus, the behaviour of all features is modelled in the modules `auv` and `environment` described in Sections 4.3 and 4.4, respectively.

Contrary to an ordinary feature model, ProFeat allows to specify feature-specific rewards in the declaration of a feature. Reward structures allow to annotate quantitative measures to behaviour and can be interpreted in various ways. Reward structures whose aim is to be reduced are implementing costs such as energy or time. Differently, reward structures with favourable properties usually have an objective to be increased, such as utility, successfully transferred packages, points earned, or alike. Each reward structure is encapsulated in a **rewards** … **endrewards** block. In the three-altitudes case study, we consider the rewards `time` and `energy_3alt`, see Lines 4–18 of Listing 1. Additionally to these rewards, the five-altitudes case study contains the reward `energy_5alt`, see Lines 19–28. During each transition the AUV module takes, the reward `time` is increased by 1; it is a transition-based reward, see Line 5. We assume that one time step corresponds to one minute, allowing us to compute an estimate of a mission's duration. The energy rewards are state-based rewards, which are gained when entering a state. We distinguish between the one that takes all states of the three-altitudes model into account (`energy_3alt`), and the one that additionally gives rewards to the states of the five-altitudes model (`energy_5alt`).

Both energy rewards can be used to estimate the necessary battery level for mission completion. Since the reward `energy_3alt` does not take all states of the five-altitudes model into account, using it with the five-altitudes model is only done to compare the two models. In both reward structures, a reward of two energy units is given if a thruster of the AUV failed and needs to be recovered, see, e.g., Line 9. Furthermore, switching between the search altitudes requires significant energy. Since the altitude is switched if the AUV is in a search state and a navigation subfeature that does not correspond to the current search altitude is active, a higher energy reward is given in these states.

```
1  root feature
2    all of robot;
3    modules auv, environment;
4    rewards "time"
5      [step] true : 1;
6    endrewards
7    rewards "energy_3alt"
8      // Costs for being in a recovery state of the three-altitudes model
9      (s=recover_high) : 2;
10     // .. omitted code ..
11
12     // Costs for switching altitudes low-med-high
13     (s=search_high) & active(low) : 4;
14     (s=search_high) & active(med) : 2;
15     (s=found) & active(high) : 4;
16     (s=found) & active(med) : 2;
17     // .. omitted code ..
18   endrewards
19   rewards "energy_5alt"
20     // .. omitted code ..
21
22     // Costs for switching altitudes low-lowmed-med-medhigh-high
23     (s=search_high) & active(medhigh) : 1;
24     (s=search_high) & active(lowmed) : 3;
25     (s=found) & active(medhigh) : 3;
26     (s=found) & active(lowmed) : 1;
27     // .. omitted code ..
28   endfeature
```

Listing 1. An excerpt of the declaration of the root feature of the five-altitudes model; the root feature of the three-altitudes model does not contain the reward `energy_5alt`.

```
1  feature navigation
2    one of low, lowmed, med, medhigh, high;
3    initial constraint active(low);
4  endfeature
```

Listing 2. The declaration of the navigation feature of the five-altitudes model; the navigation feature in the three-altitudes model does not contain the features `lowmed` and `medhigh`.

The given energy reward corresponds to the number of switched altitude levels. For example, if the AUV needs to switch between high and low, lowmed, med, or medhigh altitude, a reward of 4, 3, 2, or 1 energy units, respectively, is given (see Lines 13, 24, 14, or 23, respectively). The reward `energy_3alt` only contains rewards for the switches between `low`, `med`, and `high`, whereas the reward `energy_5alt` contains all possible switches. Since the altitude must be changed to `low` once the pipeline has been found, these cases also receive an energy reward as explained above, see Lines 15–16 and 25–26. All other states receive an energy reward of 1, where the reward `energy_3alt` only assigns rewards to states of the three-altitudes model. We use the function `active` to determine which feature is active, i.e., included in the current feature configuration; given a feature, the function returns true if it is active and false otherwise.

The remainder of the feature model is implemented similarly to the root feature, but the features do not contain feature-specific modules or rewards. The features are implemented and named according to the feature model in Figure 2. To have only one initial state, we initialise the model

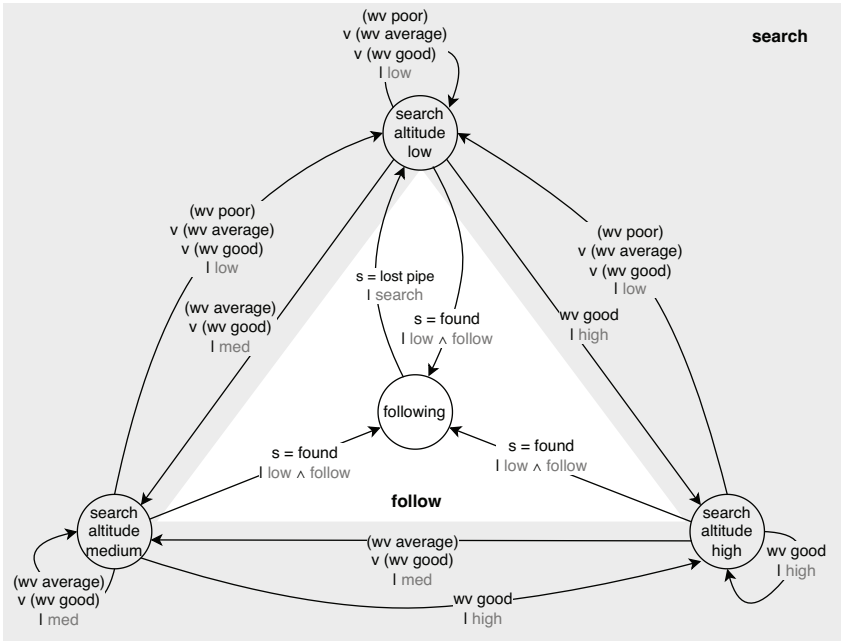


Fig. 3. The behaviour of the managing subsystem of the AUV in the three-altitudes case study ([69]); transition guards are written in black, actions in grey following a vertical bar; actions reflect the activated features, conflicting features are deactivated; by (de)activating features, transitions of the managed subsystem are (de)activated.

with the features `search` and `low` active, using the keyword `initial constraint`, see Line 3 of Listing 2. As an example of the implementation of another feature, the declaration of the feature `navigation` of the five-altitudes model can be seen in Listing 2.

4.2 The Managing Subsystem

The Behavioural Model of the Managing Subsystem. The managing subsystem implements the AUV's adaptation logic. Here, adaptation during runtime is achieved by switching between the different configurations of the managed subsystem, activating and deactivating the subfeatures of *navigation* and *pipeline inspection*. Note that adaptations have to adhere to the feature model in Figure 2, i.e., the resulting feature configuration has to be valid. By switching between the configurations of the managed subsystem, the managing subsystem enables different behaviour.

The behaviour of the managing subsystem in the three-altitudes model is displayed in Figure 3. The grey area of the figure includes the transitions that can be taken during the search for the pipeline, and the white area the transitions once the pipeline has been found. Each transition contains a guard, written in black, and an action, written in grey after a vertical bar. The managing subsystem of the five-altitudes model contains two more states, *search altitude lowmed* and *search altitude medhigh*, as well as transitions from these states to each other and to all states in Figure 3. The actions on these transitions follow the same pattern as in Figure 3 and the guards are assigned as described below and shown in Table 1.

During the search for the pipeline, i.e., when the feature *search* is active, reflected in the grey area of Figure 3 (plus the states *search altitude lowmed* and *search altitude medhigh* and their transitions in the five-altitudes model), the managing subsystem activates and deactivates the subfeatures of

Table 1. The Possible Altitudes for Different Water Visibilities in the Five-altitudes Case Study, where the Water Visibility Reflects the Maximum Distance at which the AUV Can Perceive Objects

	Possible Altitude				
water visibility (wv)	**low**	**lowmed**	**med**	**medhigh**	**high**
wv < lowmed_visib	✓	✗	✗	✗	✗
lowmed_visib ≤ wv < med_visib	✓	✓	✗	✗	✗
med_visib ≤ wv < medhigh_visib	✓	✓	✓	✗	✗
medhigh_visib ≤ wv < high_visib	✓	✓	✓	✓	✗
high_visib ≤ wv	✓	✓	✓	✓	✓

navigation according to the current water visibility (which reflects the maximum distance at which the AUV can perceive objects). To do so, thresholds *lowmed_visib*, *med_visib*, *medhigh_visib*, and *high_visib* that depend on the environment that the AUV operates in are introduced. In the three-altitudes case study, only *low* can be activated if the water visibility is below *med_visib*, both *low* and *med* can be activated if it is between *med_visib* and *high_visib*, and all of them can be activated if it is above *high_visib*. In the five-altitudes case study, the possible altitudes for different water visibilities can be found in Table 1. In Figure 3, the activated feature is displayed in grey on the transition, implicitly the other subfeatures of *navigation* and *pipeline_inspection* are deactivated. Note that the transitions in the grey area implicitly carry the guard *s != found*, i.e., the AUV is not in the state *found*, because they represent the transitions during the search for the pipeline. This guard was omitted for better readability.

Once the pipeline has been found, i.e., the managed subsystem is in the state *found*, one of the transitions in the white area (plus the transitions from *search altitude lowmed* and *search altitude medhigh* to *following* in the five-altitudes model), guarded by $s = found$, is taken. These transitions include the action of activating *low* and *follow*, and thus deactivating the other navigation subfeatures and *search*. When the AUV loses the pipeline, i.e., it is in the state *lost pipe*, the managing subsystem activates *search* and deactivates *follow*. Since the AUV is following the pipeline at a low altitude, the AUV will start searching at a low altitude.

The Implementation of the Managing Subsystem in ProFeat. As usual in the PRISM input language and thus in ProFeat as well, behaviour is specified in *modules*, containing a set of *probabilistic guarded commands* of the following form:

```
[action] guard -> prob_1: update_1 + ... + prob_n: update_n;
```

A command may have an optional label `action` to annotate it or to synchronise with other modules. In PRISM, the `guard` is a predicate over global and local variables of the model, which can also come from other modules. ProFeat extends the capability of guards to also support a predicate `active` that yields whether a given feature is active or not. If the guard is true, then the system state is changed with probability `prob_i` using `update_i` for all i. An update describes how the system should perform a transition by giving new values for variables, either directly or as a function using other variables. To model feature switches, ProFeat provides a specific module, called *feature controller*, that specifies commands over features. Through keywords `activate` and `deactivate`, the feature controller can activate and deactivate features in the `update` of a command. Several features can be activated and deactivated simultaneously. If the resulting feature configuration does not adhere to the feature model, then ProFeat blocks this transition.

Due to its close correspondence, the managing subsystem of the AUV is implemented as a feature controller in ProFeat. In the pipeline inspection case study, subfeatures of `navigation` (i.e., the different altitudes at which the AUV can operate) and subfeatures of `pipeline_inspection` (i.e., the

```
1  formula lowmed_visib = (4*min_visib+max_visib)/5;
2  formula med_visib = (3*min_visib+2*max_visib)/5;
3  formula medhigh_visib = (2*min_visib+3*max_visib)/5;
4  formula high_visib = (min_visib+4*max_visib)/5;
5
6  controller
7    // Change altitude depending on water visibility
8    [step] (s!=found) & active(search) & water_visib < med_visib
9            -> activate(low) & deactivate(high) & deactivate(med);
10   [step] (s!=found) & active(search)
11               & med_visib <= water_visib & water_visib < high_visib
12           -> activate(low) & deactivate(med) & deactivate(high);
13   [step] (s!=found) & active(search)
14               & med_visib <= water_visib & water_visib < high_visib
15           -> activate(med) & deactivate(low) & deactivate(high);
16   // .. omitted code ..
17
18   // Switch task from "search" to "follow"
19   [step] (s=found) & active(search)
20           -> deactivate(search) & activate(follow) & activate(low)
21               & deactivate(med) & deactivate(high);
22
23   // Switch task from "follow" to "search"
24   [step] (s=lost_pipe) & active(follow)
25           -> deactivate(follow) & activate(search);
26
27   // Enable transitions when following the pipeline
28   [step] (s!=lost_pipe) & active(follow) -> true;
29 endcontroller
```

Listing 3. An excerpt of the ProFeat feature controller of the three-altitudes model.

tasks the robot has to fulfil) can be switched by the feature controller during runtime, see Listing 3 for an excerpt of the three-altitudes feature controller. The feature controller of the five-altitudes model is implemented similarly, respecting the restrictions in Table 1. We will describe the implementation of the three-altitudes model here, briefly highlighting additions in the five-altitudes model in brackets.

When the feature `search` is active and the pipeline has not been found yet, the feature controller activates and deactivates the altitudes non-deterministically, but according to the current water visibility, as described before. Whether or not the pipeline has been found is reflected in the state `s` of the managed subsystem (as described in Section 4.3) which the feature controller accesses. The minimum and maximum water visibility can be set by the user during design time and influence the altitudes associated with the features `low`, `med` and `high` (plus `lowmed` and `medhigh` in the five-altitudes model); i.e., it influences when the feature controller is able to switch features. To reflect this, the variables `med_visib` and `high_visib` are declared as in Lines 2 and 4 (plus the variables `lowmed_visib` and `medhigh_visib` in Lines 1 and 3 in the five-altitudes model). A *formula* in PRISM and ProFeat can be used to assign an identifier to an expression. If the water visibility is less than `med_visib`, the feature controller activates the feature `low` (see Lines 8–9) because the AUV cannot perceive the seabed from a higher altitude. The variable `water_visib` reflects the current water visibility and is declared in the environment module which also updates this variable, see Section 4.4. If the water visibility is between `med_visib` and `high_visib`, the feature controller chooses non-deterministically between `low` and `med` (see Lines 10–15), whereas it chooses

non-deterministically between all three altitudes if the water visibility is above `high_visib`. (In the five-altitudes model, it activates the features non-deterministically according to Table 1.) Note that it is also possible to deactivate or activate a feature if it is already inactive or active, respectively. The computation of `med_visib` and `high_visib` is different from the one in our previous article to align the three and five-altitudes models.

When the pipeline is found, i.e., the managed subsystem is in state `found`, the feature controller activates the feature `follow` and deactivates `search`, see Lines 19–21. Since the AUV should be at a low altitude while following the pipeline, the feature controller also deactivates the features `high` and `med` (plus `lowmed` and `medhigh` in the five-altitudes model) and activates `low`. If the AUV lost the pipeline, i.e., the managed subsystem is in state `lost_pipe`, the feature controller deactivates `follow` and activates `search` to start the search for the pipeline, see Lines 24–25.

The feature controller synchronises with the `auv` and `environment` modules via the action label `step`. Since all transitions of the modules and feature controller have the same action label, they can only execute a transition if there is a transition with a guard evaluating to true in both modules and in the feature controller. Thus, the feature controller needs to include a transition doing nothing if the feature `follow` is active and the AUV is not in state `lost_pipe`, see Line 28.

Let us reflect on technical details how ProFeat ensures a sound encoding of the SAS into PRISM models. During the translation of the ProFeat model into the PRISM language, features are expressed by standard PRISM variables in the feature controller module, where the value of the variable reflects whether the feature is active or inactive. To ensure that the system adapts only within valid configurations, ProFeat automatically includes several constraints into guards and updates of PRISM commands. The constraints on valid feature configurations are added as guards to every command in the feature controller and to the initial state expression of the model. Furthermore, these constraints are also incorporated into the potential updates of the feature controller (for details, see [23]). For every feature module, guards of commands are extended by the corresponding feature to be active, while an additional unblocking command is added that is only enabled if the corresponding feature is inactive (for proofs of this translation to be sound, we refer to [40, 41]). These transformations ensure that the SAS is always in a valid configuration during runtime, even when adapting to changing environments.

4.3 The Managed Subsystem

The Behavioural Model of the Managed Subsystem. The behaviour of the managed subsystem of the AUV is described as an *fMDP* [41, 42]. Here, each transition results from a guarded command comprising a feature guard and a state guard on the state variables of the system. A transition can be taken if and only if both the feature guard is fulfilled in the current configuration of the managed subsystem and the state guard is fulfilled by the current values of the state variables. The possible configuration switches done by the managing subsystem can resolve the feature guards, leading to an MDP where features are encoded into the state-space of the system [42]. Figure 4 depicts such an MDP for the AUV in the three-altitudes setting, where some details have been omitted to avoid cluttering (in particular, all probabilities). The details can be obtained from the publicly available model[4]. The five-altitudes model contains four more states, *search_lowmed*, and *search_medhigh* together with their respective recovery states and the transitions following the same pattern as in the three-altitudes model, which is further explained below. The probabilistic model allows us to easily model the possibilities of, e.g., finding and losing the pipeline depending on the system configuration.

[4]The models are publicly available [67].

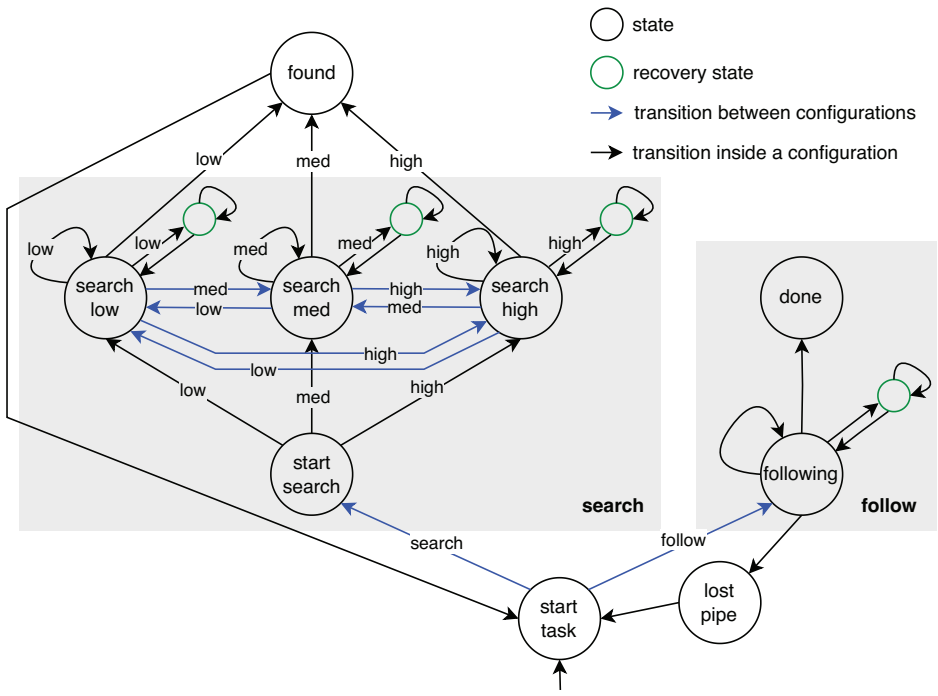


Fig. 4. The behaviour of the configurations of the managed subsystem and the switches between configurations in the three-altitudes case study ([69]); transitions with a feature annotation are guarded by the respective feature; transitions without annotation are not guarded by features; blue transitions are enabled by a feature (de)activation of the managing subsystem; black transitions are taken if the managing subsystem did not (de)activate features; probabilities are omitted for better readability.

The MDP consists of the behaviour of the configurations of the managed subsystem together with the switches between the configurations induced by the managing subsystem. Therefore, we distinguish between two kinds of transitions in Figure 4: black transitions that model the behaviour of a certain configuration of the managed subsystem and blue transitions that switch between configurations, enabled by the managing subsystem during runtime. The labels *search*, *follow*, *low*, *med*, and *high* on the transitions represent the features that have to be active to execute the respective transition, i.e., they represent the feature guard. The blue transitions between configurations implicitly carry the action to start the task or go to the altitude specified by the feature associated with the transition.

The behaviour of one configuration, i.e., the behaviour associated with a set of features F compliant with the feature model, is modelled by the projection of the MDP on the sub-MDP containing only features in F. More concretely, it is modelled by the MDP consisting of all black transitions with feature guards satisfied by F and the states that these transitions connect. Consider for example the configuration with the features *robot*, *navigation*, *pipeline_inspection*, *low*, and *search*. Then the behaviour of this configuration is represented by the states *start_task*, *search_low* with its associated recovery state, and *found* as well as the transitions between them and the self-loops of *search_low* and its recovery state. Hence, the behaviour of the MDP corresponding to this configuration consists of the AUV starting its task *search* and going to the state *search_low* where it remains until either a failure occurs and it goes to the associated failure state, or until it finds the pipeline and transitions to the state *found* and then again to *start_task*.

The managing subsystem can switch between configurations (and thus change behaviour) by activating another subfeature of *navigation* during the search for the pipeline, or by activating another subfeature of *pipeline_inspection* if the pipeline was found or lost, reflected by the states *found* and *lost pipe* of the managed subsystem. These switches between configurations are reflected by the blue transitions in Figure 4. Consider for example the same configuration as before which consists of the features *robot*, *navigation*, *pipeline_inspection*, *low*, and *search*. Then the managing subsystem can activate another subfeature of *navigation* during the search of the pipeline, i.e., in the state *search_low* which implies deactivating the feature *low*. If this happens, the black transitions within the current configuration cannot be taken anymore because the feature *low* is inactive. Only the blue transition with the newly activated *navigation* subfeature to a new configuration can be taken. If, for example, the feature *med* has been activated, the blue transition from *search_low* to *search_med* will be taken which implicitly carries the action to go to a medium altitude.

The behaviour of the AUV can then be described as follows. At deployment time, i.e., in state *start task*, the AUV can either immediately start following the pipeline if it was deployed above it, indicated by the feature *follow* in the initial configuration, or start searching for it, indicated by the feature *search* in the initial configuration. During the search for the pipeline, i.e., when the AUV is in the grey area labelled *search*, the managing subsystem might switch between the subfeatures of *navigation* during a transition as described in Section 4.2 which causes a configuration change of the managed subsystem, reflected by taking a blue transition. If the managing subsystem does not switch between configurations, then the AUV keeps searching for the pipeline, it transitions to a thruster failure state to repair its broken thrusters, or it finds the pipeline and transitions to the state *found*. Once the pipeline has been found, the AUV transitions to the state *start_task* again. As described in Section 4.2, the managing subsystem deactivates the feature *search* during this transition and activates the feature *follow*. When the AUV is following the pipeline, i.e., in the grey area labelled *follow*, it can also lose the pipeline again, transitioning to the state *lost_pipe*, e.g., because of sand covering the pipeline or because the AUV drifted off its path due to thruster failures. In this case, the managing subsystem activates the feature *search* during the transition from *lost_pipe* to *start_task* (see Section 4.2) such that the AUV will start its search again.

Additionally to the states and transitions displayed in Figure 4, the five-altitudes model contains the states *search lowmed* and *search medhigh* with their respective recovery states. The states follow the same pattern as the ones of the three-altitudes model: the search states are connected with configuration changing (blue) transitions to the other search states, guarded by the corresponding feature, and they contain a self-loop and a transition to their respective recovery state, guarded by their respective feature (black transitions). The recovery states each contain a self-loop and a (black) transition back to the corresponding search state.

The ProFeat Implementation of the Managed Subsystem. In ProFeat, each feature can be implemented by several *feature modules*, each comprising a set of guarded probabilistic commands (see Section 4.2). We encapsulate the behaviour of the managed subsystem of the AUV in a module `auv`. Feature modules in ProFeat can exhibit guards in commands that include the function `active`, evaluating to true if the corresponding feature is active. To this end, configuration-dependent behaviour can easily be specified in feature modules. Figure 4 shows the behaviour of the module together with feature switches through the feature controller for the three-altitudes model and with the additional states in the five-altitudes model as described before. In this section, we will describe the implementation of the three-altitudes model (see Listing 4 for an excerpt). As in Section 4.2, the additions for the five-altitudes model as described above are briefly introduced in brackets.

As in Figure 4, there are thirteen enumerated states (seventeen in the five-altitudes model) in the ProFeat module with names that correspond to the state labels in the figure. The recovery

```
1  module auv
2    s : [0..12] init start_task;
3    d_insp : [0..inspect] init 0;
4    t_failed : [0..infl_tf] init 0;
5
6    // To the correct task
7    [step] (s=start_task & active(search)) -> 1: (s'=start_search);
8    [step] (s=start_task & active(follow)) -> 1: (s'=following);
9
10   // .. omitted code ..
11   // From search state to another state
12   [step] (s=search_high & active(high))
13           -> 0.59:(s'=found)
14           + 0.4:(s'=search_high)
15           + 0.01:(s'=recover_high);
16   [step] (s=search_high & active(med)) -> 1:(s'=search_med);
17   [step] (s=search_high & active(low)) -> 1:(s'=search_low);
18   // .. omitted code ..
19
20   // Go to other task if pipeline is found
21   [step] (s=found) -> 1:(s'=start_task);
22
23   // Following the pipeline
24   [step] (s=following) & (d_insp<inspect) & (t_failed=0)
25           -> 0.92: (s'=following) & (d_insp'=d_insp+1)
26           + 0.05: (s'=lost_pipe)
27           + 0.03:(s'=recover_following)
28               & (t_failed'=(t_failed<infl_tf? t_failed+1 : t_failed));
29   [step] (s=following) & (d_insp<inspect) & (t_failed>0)
30           -> 0.92*(1-t_failed/infl_tf): (s'=following)
31               & (d_insp'=d_insp+1) & (t_failed'=t_failed-1)
32           + 0.05*(1+((0.92*t_failed)/(0.05*infl_tf))): (s'=lost_pipe)
33           + 0.03:(s'=recover_following)
34               & (t_failed'=(t_failed<infl_tf? t_failed+1 : t_failed));
35   [step] (s=following) & (d_insp=inspect) -> (s'=done);
36
37   // Lost the pipeline
38   [step] (s=lost_pipe) -> 1: (s'=start_task) & (t_failed'=0);
39
40   // Recovery states
41   [step] (s=recover_high) -> 0.5:(s'=recover_high) + 0.5:(s'=search_high);
42   // .. omitted code ..
43 endmodule
```

Listing 4. An excerpt of the ProFeat AUV module of the case study.

states are named according to the state they are connected to (e.g., the recovery state connected to `search_high` is called `recover_high`). The variable `s` in Line 2 in Listing 4 represents the current state of the AUV and is initialised using the keyword `init` with the state `start_task`. To record how many meters of the pipeline have already been inspected, the variable `d_insp` in Line 3 represents the distance of the pipeline that the AUV has already inspected, it is initialised with 0. The variable `inspect` represents the desired inspection length and it can be set by the user during design time. Since the number of times a thruster failed impacts how much the AUV deviates from its path, the variable `t_failed` can be increased if a thruster fails while the AUV follows the pipeline. It is

bounded by the influence a thruster failure can have on the system (`infl_tf`) that can be set by the user during design time.

As an example for the use of the `active` keyword, consider the command in Lines 12–15, which can be read as follows. If the system is in state `search_high` and the feature `high` is active, then with a probability of 0.59, the system changes its state to `found`, with a probability of 0.4 it changes to `search_high` and with a probability of 0.01 it changes to `recover_high`. These are exactly the black transitions shown in Figure 4 exiting from state *search high*. This command also has an action label, `step`. Using this action label, it synchronises with the environment module and the feature controller. The blue transitions exiting state *search high* in Figure 4 are modelled in Lines 16–17 (in the five-altitudes model, additional transitions to the states `search_lowmed` and `search_medhigh` with the respective guards are added). If the model is in state `search_high`, but the feature `low` or `med` is active, indicating that the AUV should go to the respective altitude, then the state is changed to the respective search state. The transitions exiting the states `search_med` and `search_low` are modelled similarly (as well as the transitions existing in the states `search_lowmed` and `search_medhigh` in the five-altitudes model). However, the probability of going to the state `found` is highest from state `search_high` and lowest from `search_low` because the AUV has a wider field of view when performing the search at a higher altitude. Furthermore, the probability of a thruster failure, i.e., of going to the respective `recover` state, is highest in state `search_low` and lowest in state `search_high` because the probability of seaweed getting stuck in the thrusters is higher at a lower altitude. If the AUV found the pipeline, then a transition to `start_task` is taken, see Line 21.

From the state `start_task`, a transition to either `start_search` or `following` can be taken, depending on which subfeature of `pipeline_inspection` is currently active, see Lines 7–8.

From the `following` state, the transitions that can be taken depend on the variables `d_insp` and `t_failed`. Lines 24–28 consider the case where the distance of the pipeline that has already been inspected (`d_insp`) is less than the distance the pipeline should be inspected (`inspect`) and the variable `t_failed` is 0, indicating that there were no recent thruster failures. Then the AUV stays in the `following` state and inspects the pipeline one more meter, it loses the pipeline, or a thruster fails and it transitions to the failure state and increases `t_failed` if `t_failed` is not at its maximum. Lines 29–34 consider the case where `d_insp` is less than `inspect` and `t_failed` is greater than 0. In this case, the probabilities of following and losing the pipeline depend on the value of `t_failed`. The bigger the value, the more likely it is to lose the pipeline because it indicates that the AUV's thrusters did not work for some time, causing it to drift off its path. If the already inspected distance is equal to the required inspection distance, the AUV transitions to the `done` state (see Line 35) and finishes the pipeline inspection. If the AUV lost the pipeline (see Line 38), then a transition to `start_task` is taken and the variable `t_failed` is set to 0 again.

When the AUV is in a recovery state, it can either stay there for another time step or exit it again to the state from where the recovery was triggered (see Line 41).

All commands in the module `auv` are labelled with `step`. Thus, every transition receives a `time` reward of 1, i.e., the time advances with every transition the AUV takes, see Lines 4–6 of Listing 1.

4.4 The Environment

The Behavioural Model of the Environment. The only parameter of the environment we consider in our model is the water visibility. We model it with a value that we assume to be bounded by a minimum and a maximum visibility that depend on where the AUV is deployed and can be set by the user during design time. We assume that changes in the water visibilty occur probabilistically with a probability of currents which also depends on the place of deployment of the AUV and can be set by the user during design time. Thereby, we reflect the uncertainty about how the environment will behave. We used two different behavioural models for the environment, a *symmetric* and

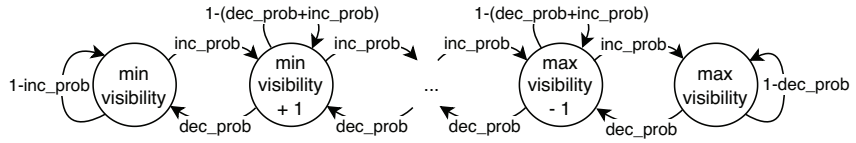


Fig. 5. The behaviour of the environment (modified from [69]); the probabilities of increasing and decreasing water visibility (inc_prob and dec_prob, respectively) can be used to model different behaviour of the environment; the minimum and maximum visibility depend on where (in which kind of water conditions) the AUV is deployed.

```
1 module environment
2   water_visib : [min_visib..max_visib] init round((max_visib-min_visib)/2);
3   [step] true -> dec_prob: (water_visib'=
4                   (water_visib=min_visib? min_visib:water_visib-1))
5                 + inc_prob: (water_visib'=
6                   (water_visib=max_visib? max_visib:water_visib+1))
7                 + (1-(dec_prob+inc_prob)): true;
8 endmodule
```

Listing 5. The ProFeat environment module of the case study, where the variables dec_prob and inc_prob have to be instantiated to model different behaviours of the environment.

an *asymmetric* one. Their behaviour is depicted in Figure 5, where *inc_prob* denotes the probability of increasing water visibility and *dec_prob* the probability of decreasing water visibility. With the asymmetric behaviour of the environment, the water visibility decreases by 1 with the probability of currents *cp*, i.e., *dec_prob*=*cp*, while it stays the same or increases by 1 with probability (1-*cp*)/2, i.e., *inc_prob*=(1-*cp*)/2. If the water visibility is already at the minimum visibility, the water visibility stays the same with probability (1+*cp*)/2 = 1-*inc_prob* and, at maximum visibility, it stays the same with probability 1-*cp* = 1-*dec_prob*. With the symmetric behaviour, we set *inc_prob*=*dec_prob*=*cp* and calculate the cases for minimum and maximum visibility as shown in Figure 5. The focus of this article is on modelling and verifying an abstract variant of the AUV, not prioritising a realistic model. Therefore, we chose two simple environment behaviours to analyse if and how the environment can influence the behaviour of the AUV. The environment model can easily be enhanced with more realistic specifications. Instantiating the values of minimal and maximal visibility as well as the probability of currents of the environment together with selecting a desired length of pipeline that should be inspected provides *scenarios* which we will use for analysis, see Section 5.1.

The Implementation of the Environment in ProFeat. The environment is modelled in a separate `environment` module, see Listing 5. The variable `water_visib` in Line 2 reflects the current water visibility and is initialised parametrically, depending on the minimum and maximum visibility. The function `round()` is pre-implemented in the PRISM language and rounds to the nearest integer. The environment module synchronises with the AUV module via the label of its action, `step`. Since the guard of the only action in the environment module is `true`, the environment executes a transition every time the AUV module does. By decoupling the environment module from the AUV module, we obtain a separation of concerns which makes it easier to change the model of the environment if needed.

For the asymmetric environment, the variables for increasing and decreasing the water visibility are instantiated as `inc_prob=(1-current_prob)/2` and `dec_prob=current_prob` whereas for the symmetric environment these variables are instantiated as `inc_prob=dec_prob=current_prob/2` where `current_prob` is the variable reflecting the probability of currents.

5 Analysis

ProFeat automatically converts models to the PRISM input language for quantitative analysis, e.g., by statistical or probabilistic model checking. To analyse a PRISM model, properties can be specified in the PRISM property specification language, which includes several probabilistic temporal logics like PCTL [53], CSL [3], or probabilistic LTL [4]. For family-based analysis, ProFeat extends this specification language to include properties depending on feature selection through the keyword `active`.[5]

The model translated into the PRISM input and property specification language can be interpreted by various (probabilistic) model checking tools. In our experiments, we report on analyses with PRISM and Storm [6] [54], depending on the properties to be investigated. Since PRISM works well together with ProFeat, we used it whenever possible. However, for parametric model checking and multi-objective reasoning, Storm is considered as the state-of-the-art [35, 58], providing in general better functionality and performance. We thus opted to use Storm for parameter synthesis and multi-objective reasoning experiments (see reporting in Sections 5.5 and 5.6).

In our analysis, we used the standard settings for both PRISM and Storm. For the parametric analysis with Storm, the chosen parameters are reported in the respective section. All experiments were conducted on a MacBook Pro with an Apple M1 chip and 16 GB memory.

The analysis of the model considered five different aspects: (1) the rewards `time`, `energy_3alt` and `energy_5alt` were used to compute safety guarantees that can be used for the deployment of the AUV using PRISM, see Section 5.2; (2) reachability of safe/unsafe states was analysed using PRISM, see Section 5.3; (3) we analysed how the different environments introduced in Section 4.4 influence the analysis results using PRISM, see Section 5.4; (4) minimal and maximal rewards with unspecified probability of currents were computed to determine the possible range of the rewards using Storm, see Section 5.5; (5) finally, we analysed tradeoff properties between the time and energy rewards using Storm in Section 5.6. Note that it is not necessary to analyse whether the model satisfies the constraints of the feature model because ProFeat automatically ensures this. Of course, a more complex analysis can be done in addition to the analyses presented in this article. Our intention in this section is to give a taste of possible analyses that demonstrate the feasibility and applicability of our approach.

5.1 Parameter Setup for the Scenarios

The operators used for analysis in this article are `P` and `R`, which reason about probabilities of events and about expected rewards, respectively. Since we use Markov decision processes (which involve non-determinism), these operators must be further specified to ask for the *minimum* or *maximum* probability and *minimum* or *maximum* expected cost, respectively, for all possible resolutions of non-determinism.

In our analyses, we used four different scenarios, two of them with an undefined value for the probability of currents. The values used in these scenarios are reported in Table 2. Scenario 1 is in the North Sea, where the minimum and maximum water visibility (in 0.5 meter units) is relatively low and the probability of currents is relatively high. In this case, only 10 meters of the pipeline have to be inspected. Scenario 2 is in the Caribbean Sea with a higher minimum and maximum visibility and a lower probability of currents compared to the North Sea, as well as 30 meters of pipeline that have to be inspected. Scenarios 3 and 4, are parametric versions of Scenarios 1 and 2, respectively. That is, they are the North Sea and the Caribbean Sea scenarios, where the value of

[5] ProFeat constructs within PRISM properties are specified using bracketing `${...}` to ensure a well-formed translation to the PRISM property specification language.
[6] https://www.stormchecker.org/index.html

Table 2. Four Different Scenarios Used for Analysis; in Scenarios 3 and 4 the Probability of Currents `current_prob` is Left as a Parameter

Scenario	min_visib	max_visib	current_prob	inspect
1 (North Sea)	1	10	0.6	10
2 (Caribbean Sea)	3	20	0.3	30
3 (Parametric North Sea)	1	10	x	10
4 (Parametric Caribbean Sea)	3	20	x	30

Table 3. The Probabilities of a Water Visibility Change in Scenarios 1 and 2

	Scenario 1			Scenario 2		
Environment	inc.	dec.	stag.	inc.	dec.	stag.
symmetric	0.30	0.30	0.40	0.15	0.15	0.70
asymmetric	0.20	0.60	0.20	0.35	0.30	0.35

Table 4. The Thresholds for Possible Altitudes for All Four Scenarios

	Scenarios	
Threshold	1&3	2&4
lowmed	2.8	6.4
med	4.6	9.8
medhigh	6.4	13.2
high	8.2	16.6

the variable `current_prob` is left unspecified. We use Scenarios 3 and 4 for the parametric analyses in Sections 5.4 and 5.5 and Scenarios 1 and 2 for the other analyses.

The probability of currents of the scenario influences how the water visibility changes. The probabilities for increasing, decreasing or stagnant water visibility in Scenarios 1 and 2 with symmetric and asymmetric environments are listed in Table 3. It can be seen that with the symmetric environment, staying at the same water visibility has by far the highest probability in Scenario 2 while the difference between increasing/decreasing and staying at the same water visibility is much smaller in Scenario 1. With the asymmetric environment, however, the situation is reversed. Here, decreasing the water visibility has a much higher probability than increasing or stagnating in Scenario 1, while the difference between the probabilities is very small in Scenario 2, with the probability of increasing or stagnant water visibility being a bit higher than the probability of decreasing.

Furthermore, the minimum and maximum visibilities of the scenarios influence the thresholds at which the managing subsystem can choose specific altitudes, they are listed in Table 4 for both scenarios. The altitude difference between the thresholds for Scenarios 1 and 3 is 1.8, while the difference is 3.4 for Scenarios 2 and 4. Therefore, increasing or decreasing water visibility can cause an increasing or decreasing number of possible altitudes of the AUV much faster in Scenarios 1 and 3 than in Scenarios 2 and 4. Consider for example a situation in the five-altitudes case study where the water visibility is 6.4 which allows all altitudes apart from `high` in Scenarios 1 and 3 and the altitudes `low` and `lowmed` in Scenarios 2 and 4. Then it is possible that `high` can be activated in two time steps in Scenarios 1 and 3 if the water visibility increases in both time steps, but it is only possible to activate `med` after at least four time steps in Scenarios 2 and 4.

We first analysed for Scenarios 1 and 2 in both models whether it is always possible to finish the pipeline inspection, i.e., to reach the state `done`. This could be confirmed since the minimum probability for all resolutions of non-determinism of eventually reaching the state `done` is 1.0.

Note that the analysis results of the three-altitudes model with the asymmetric environment reported in the following sections are different from those obtained in our previous article [69] where we also used the asymmetric environment. The difference between the model from our previous article (hereafter called the "old model") and the three-altitudes model in this article are the thresholds at which adaptation can be triggered. In this article, we divided the space in which the

```
R{"energy_3alt"}min=? [F ${s=done}];
R{"energy_3alt"}max=? [F ${s=done}];
```

Listing 6. Analysis using the rewards.

AUV can operate (between `min_visib` and `max_visib`) into five equally big parts to accommodate switches between the five different altitudes in the five-altitudes model. To enable a comparison between the analysis results of the three- and five-altitudes models, we used a subset of the thresholds defined for the five-altitudes model also for the three-altitudes model instead of the thresholds defined in the old model. Thus, the thresholds at which the managing subsystem can switch between altitudes are different, allowing different configurations for the two versions of the model at certain water visibilities. For example, using Scenario 2, the threshold `med_visib` is defined as approximately 5.7 in the old model but as 9.8 in this article's models. Similarly, the threshold `high_visib` is defined as approximately 11.3 in the old model and as 16.6 in this article's models for Scenario 2. Therefore, using the old model with Scenario 2, the managing subsystem can switch between the features `low` and `med` if the water visibility is above 5.7, increasing the energy consumption. However, in the three-altitudes model of this article, the managing subsystem has to choose the feature `low` if the water visibility is below 9.8. Similarly, the managing subsystem of the old model can switch between the three altitudes with lower water visibilities than the managing subsystem of the three-altitudes case study of this article. This enables the managing subsystem of the old model to choose a strategy that results in a higher maximum energy consumption than is possible for the three-altitudes model in this article because this strategy is not possible for the managing subsystem of this article. Therefore, the maximum energy consumption with the old model is much higher than the energy consumption with the three-altitudes model of this article.

5.2 Reward Properties

The rewards `time`, `energy_3alt`, and `energy_5alt` were used to analyse safety properties related to the execution of the AUV in the model. Since the AUV has only a limited amount of battery, an estimation of the energy needed to complete the mission is required. This ensures that the AUV is only deployed for the mission if it has sufficient battery to complete it. The commands in Listing 6 were used to compute the minimum and maximum expected `energy_3alt` reward (for all resolutions of non-determinism) to complete the mission. Since the model includes three reward structures, the name of the reward has to be specified in `{"..."}` after the `R` operator. Similarly, the minimum and maximum expected `energy_5alt` and `time` rewards to complete the mission were analysed, the latter to give the system operators an estimate of how much time the mission might take. The execution time of the analysis in PRISM for the three-altitudes case study took at most 3.8 seconds while the analysis for the five-altitudes case study took at most 15.5 seconds. For both case studies, the most time-consuming analysis was analysing the maximum time of Scenario 2 with the asymmetric environment.

The results for the two models in Scenarios 1 and 2 as well as with symmetric and asymmetric environments are reported in Table 5. For analysing the three-altitudes model, the reward `energy_5alt` was not used and therefore marked as N/A (Not Applicable) because it refers to states that are not present in the three-altitudes model. It can be seen that the variation of the parameters in the two scenarios strongly influences the expected energy and time of the mission. The difference between minimum and maximum expected energy and minimum and maximum expected time for Scenario 2 are significantly bigger than for Scenario 1. In particular, the maximum expected energy and time are much higher for Scenario 2 than for Scenario 1.

Furthermore, the behaviour of the environment has an interesting impact on the analysis results. In Scenario 1, the minimum rewards are slightly higher in the asymmetric environment

Table 5. Expected Min-/maximum Rewards for Completing the Mission for the Two Models Concerning both Scenarios and Environments

Scenario	Environment	Property		3-Altitudes Result	5-Altitudes Result
1	symmetric	energy_3alt	min	24.7839	19.3133
			max	139.6440	263.5274
		energy_5alt	min	N/A	24.5033
			max	N/A	310.3349
		time	min	23.7951	23.5909
			max	66.2743	131.4976
	asymmetric	energy_3alt	min	24.7844	21.2382
			max	31.4667	32.3222
		energy_5alt	min	N/A	24.7015
			max	N/A	37.5475
		time	min	23.8917	23.8389
			max	27.1616	32.5339
2	symmetric	energy_3alt	min	59.0823	49.8592
			max	382.1825	998.8417
		energy_5alt	min	N/A	58.5831
			max	N/A	1159.4918
		time	min	56.4871	56.1615
			max	173.2865	468.0138
	asymmetric	energy_3alt	min	59.0823	49.6258
			max	963.8585	2753.7185
		energy_5alt	min	N/A	58.5826
			max	N/A	3011.3884
		time	min	56.3243	56.0297
			max	341.9468	972.3262

while the maximum rewards are at least 2.4 times higher with the symmetric environment for all reward structures. However, in Scenario 2, the minimum rewards are slightly higher in the symmetric environment while the maximum rewards are at least 2 times higher with the asymmetric environment. Several factors influence these results and have to be consulted to explain this phenomenon:

(1) The probability of currents is higher in Scenario 1 than in Scenario 2 which leads to different probabilities for increasing or decreasing the water visibility, see Table 3;
(2) The inspection length is longer in Scenario 2 than in Scenario 1;
(3) The difference between the minimum and maximum visibility is higher in Scenario 2 than in Scenario 1, which means that the water visibility has to increase or decrease more to allow for more or less possible altitudes, respectively, see Table 4;
(4) With the symmetric environment, increasing and decreasing the water visibility has the same probability, whereas with the asymmetric environment, increasing and staying at the same altitude has the same probability.

In Scenario 1, a decrease of the water visibility leads to a decrease in the number of possible altitudes very fast because the thresholds at which different altitudes can be activated are relatively close together compared to Scenario 2, see Table 4. Therefore, the asymmetric environment, where

the probability of decreasing the water visibility is very high for Scenario 1, see Table 3, does possibly not allow as many altitude switches as the symmetric environment because the water visibility might often be low. This might lead to a higher maximum energy consumption with the symmetric environment since it allows more altitude switches. The higher maximum time for the symmetric environment can be explained similarly. Time advances with every transition of the combined transition system of the managed and managing subsystem and the environment. Switching between altitudes of the managed subsystem is one such time step during which it is impossible to find the pipeline. Therefore, increasing the switches between altitudes increases the maximum time the AUV takes to finish the pipeline inspection. As described before, altitude switches are more likely in the symmetric environment which thus leads to a higher maximum time reward.

In Scenario 2, the water visibility has to increase much more than in Scenario 1 to allow a higher altitude of the AUV, see Table 4. With the symmetric environment, the probability of the water visibility staying the same is much higher than an increase or a decrease in the water visibility, see Table 3. Therefore, it is less likely to increase the water visibility to a point where a higher altitude is allowed, which leads to a higher maximum energy and time with the asymmetric environment (the argumentation is similar to the one for Scenario 1).

It can also be seen that in all experiments, the five-altitudes model enables a mission to happen in less time and with less energy compared to the three-altitudes model, independent of the chosen energy reward. Thus, in these scenarios and with these environment behaviours, it is beneficial to enable the AUV to operate at five instead of three altitudes.

It might seem counter-intuitive that the five-altitudes model with the reward `energy_3alt` does not give the same results as the three-altitudes model with this reward; we give an explanation for this here. Considering the minimum expected energy, searching at a `lowmed` or `medhigh` altitude or switching to and from these altitudes does not cost energy with the `energy_3alt` reward because these altitudes are not part of the three-altitudes model. Therefore, the five-altitudes model, which allows these altitudes, enables a lower minimum `energy_3alt` reward than the three-altitudes model since the managing subsystem can choose a strategy with `lowmed` and `medhigh` altitudes which do not increase the `energy_3alt` reward. Concerning the maximum expected `energy_3alt` reward, we investigated the strategy for obtaining this maximum reward in both models. Interestingly, the strategy in the five-altitudes model includes going to `lowmed` altitude if the water visibility is below `med_visib`. If the water visibility is below `med_visib`, the managing subsystem can hinder the AUV from finding the pipeline by continually switching between `low` and `lowmed`. To understand this, note that there is only a probability of finding the pipeline if the current search altitude matches the active `navigation` subfeature as, e.g., in Lines 12–15 of Listing 4. If, on the other hand, the current search altitude is different from the active `navigation` subfeature, then the AUV will change with probability 1 to the search state corresponding to the active `navigation` subfeature, see, e.g., Line 16 of Listing 4, and there is no probability of finding the pipeline during this transition. Therefore, unless the water visibility drops below `lowmed_visib`, the managing subsystem in the five-altitudes model can choose a strategy that increases the `energy_3alt` reward without enabling it to find the pipeline. It can be observed that here, again, the scenario and the behaviour of the environment influence how much impact this strategy can have on the maximal `energy_3alt` reward. The highest impact can be achieved in Scenario 2 with the asymmetric environment where the maximum `energy_3alt` reward in the five-altitudes model is $\approx$2.86 times higher than in the three-altitudes model.

Note that analysing the maximum rewards is a worst-case analysis. Any reasonable adaptation strategy would not choose to, e.g., maximise the energy consumption by continually switch between altitudes if the goal of the system is to do a pipeline inspection. Nonetheless, it can be interesting to do this worst-case analysis.

```
1 label "unsafe" = s=recover_high | s=recover_medhigh | s=recover_med | s=recover_lowmed
2                  | s=recover_low | s=recover_following;
3 label "safe" = s=lost_pipe | s=start_task | s=start_search | s=search_high
4                  | s=search_medhigh | s=search_med | s=search_lowmed | s=search_low
5                  | s=found | s=following | s=done;
6
7 Pmin=? [G "safe"];
8 filter(avg, Pmax=? [ F<=k "unsafe" ], "safe");
9 filter(min, Pmin=? [ F<=k "safe" ], "unsafe");
```

Listing 7. Analysis of unsafe states.

In conclusion, the analysis revealed that the scenario and the environment behaviour can have a big impact on the expected rewards. It can therefore be beneficial to evaluate a model with different scenarios and environment behaviours. Furthermore, increasing the granularity of the model by enabling more altitudes at which the AUV can operate provided lower expected minimum rewards in our model, giving an indication that increasing the granularity of the model can be beneficial.

5.3 Unsafe States

Thruster failures, although we assume that they can be repaired, pose a threat to the AUV since unforeseen events like strong currents might cause the AUV to be damaged, e.g., by causing it to crash into a rock. Even though the possibility of, e.g., a crash is not modelled in our model, we showcase how thruster failure states can be analysed. To analyse this, the state space was partitioned into two parts, *safe* and *unsafe* states. This was achieved by using labels, see Lines 1–5 of Listing 7 for the labels of the five-altitudes model. The labels of the three-altitudes model only contain the states present in this model. These labels were then used to calculate the probability of several properties regarding reachability of safe/unsafe states. Each of the analyses in PRISM took less than half a second.

Safe States. The minimum probability of only taking safe states (see Line 7) was shown to be $\approx$0.65 for Scenario 1 and $\approx$0.32 for Scenario 2 in the two models and with both symmetric and asymmetric environments. As expected, the probability of only taking safe states is higher for a shorter pipeline inspection. It is interesting to see that the probability of only taking safe states is influenced by neither the behaviour of the environment nor the number of possible altitudes of the AUV even though the probability of going to an unsafe state depends on the altitude of the AUV, i.e., it is higher the lower the altitude of the AUV, see Section 2. The altitude at which the AUV can operate then depends on the environment. However, it seems that both of them do not influence the probability of only taking safe states as the results are always exactly the same.

Safe to Unsafe. We analysed this phenomenon further by analysing the probability of going from a safe state to an unsafe state in the two models with both scenarios and environments. In this way, we hope to find a relation between probabilities concerning safe and unsafe states and the two models and environments. This is analysed with the property in Line 8 of Listing 7. First, the maximum probability (over all possible resolutions of non-determinism) for reaching an unsafe state from a safe state is calculated, and then the average is taken. PRISM experiments allow analysing this property automatically for a specified range of k, in our experiments, the time steps. The plotted graphs for Scenarios 1 and 2 with the three- and five-altitudes models are displayed in Figure 6. The values for the symmetric and asymmetric environments were the same so they are not annotated separately.

The graphs show that the probability of reaching an unsafe state from a safe state increases with the number of considered time steps. Furthermore, in Scenario 2, the probability stabilises much later and at a higher value than in Scenario 1. While the average probability of reaching an unsafe

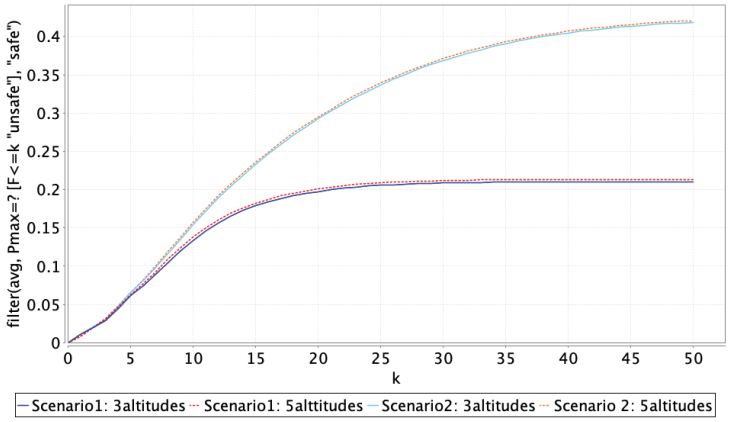


Fig. 6. Results for reaching an unsafe state from a safe state in k time steps for Scenarios 1 and 2 with the three-altitudes and five-altitudes model; more concretely, for a fixed k, the maximum probability of reaching an unsafe state from a safe state is calculated over all possible resolutions of non-determinism, after which the average value is taken.

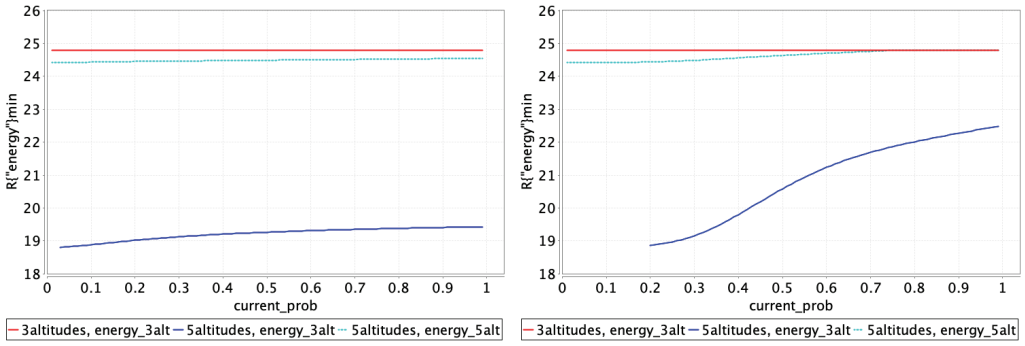
state from a safe state stabilises after about 42 time steps at ≈0.21 in Scenario 1, it stabilises after about 70 time steps at ≈0.42 in Scenario 2. As can be seen from the graphs, the differences between the three- and five-altitudes models are marginal. For example, for Scenario 2, after 50 time steps the probabilities differ by only 0.0022. In conclusion, for these experiments concerning safe and unsafe states, the environment did not influence the calculated probabilities, and the number of possible altitudes only had a minor effect on them.

Unsafe to Safe. It is also important to ensure that a safe state will be reached from an unsafe state after a short time, as, e.g., in Line 9, where k is an integer. For every unsafe state, the minimum probability (for all possible resolutions of non-determinism) of reaching a safe state within k time steps is calculated. Then the minimum over all these probabilities is taken. Thus, it gives the minimum probability of reaching a safe state from an unsafe state in k time steps. PRISM experiments showed that for the two models in both scenarios and environments, the probability of reaching a safe state from an unsafe state is above 0.95 after 5 time steps and above 0.99 after 7 time steps. Thus, for this analysis about safe and unsafe states, the behaviour of the environment and the number of altitudes again does not influence the result.

5.4 Different Environments

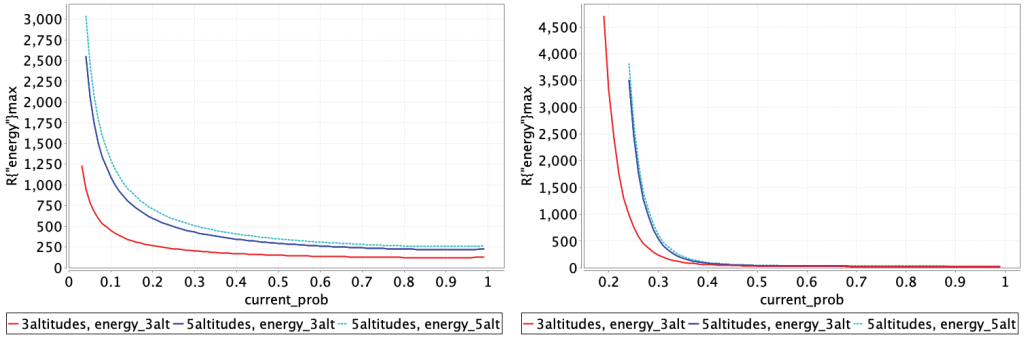
To analyse how distinct implementations of the environment, as described in Section 4.4, influence the difference between the minimum and maximum rewards for the three- and five-altitudes models, we conducted PRISM experiments with the models for Scenarios 3 and 4. That is, the probability of currents, which is a parameter in Scenarios 3 and 4, was left unspecified, and we asked PRISM to plot the graphs for minimum and maximum energy as in Listing 6. Each of the experiments in PRISM took less than 35 seconds when concerning a step size of 0.01 for the probability of currents. Considering the same model and scenario, the experiments for minimal energy took less time than the experiments for maximal time, and the experiments with the symmetric environment took less time than the ones with the asymmetric environment.

Figures 7 and 8 show the plotted graphs for minimum and maximum energy in Scenario 3, respectively, with the symmetric and asymmetric environments. The dotted lines show the values



(a) Symmetric environment (b) Asymmetric environment

Fig. 7. The graphs for minimum expected energy in Scenario 3 where the probability of currents `current_prob` is left as a parameter; they show the dependency of the minimum expected energy on the parameter `current_prob`.



(a) Symmetric environment (b) Asymmetric environment

Fig. 8. The graphs for maximum expected energy in Scenario 3 where the probability of currents `current_prob` is left as a parameter; they show the dependency of the maximum expected energy on the parameter `current_prob`.

for the five-altitudes model with `energy_5alt` which assigns rewards to all states, including the intermediate ones. The solid lines show the values for the three- and five-altitudes models with `energy_3alt` which only assigns rewards to states present in both models. The graphs for Scenario 4 look similar and are therefore not displayed here.

It can be seen that the five-altitudes model can achieve a lower expected minimal energy reward with both environments, especially with the `energy_5alt` reward. Therefore, considering the scenarios and environments described in this article, it seems that enabling the AUV to operate at more altitudes provides an advantage. However, the influence of the number of altitudes varies with the chosen environment model. With the asymmetric environment, the difference between the three- and five-altitudes model decreases much more with an increasing probability of currents compared to the symmetric environment.

For the maximum expected energy reward, the results for the five-altitudes model are again higher than the ones for the three-altitudes models. However, with increasing probability of currents, the difference between the results of the three- and five-altitudes models decreases, with the asymmetric environment they are almost the same with probabilities of currents above 0.4.

```
storm-pars --prism model.prism --prop "R{\"energy_3alt\"}min=? [F _s = done]"
    --mode feasibility --feasibility:method pla --direction min
    --region "0.3<=current_prob<=0.8" --guarantee 0.0001 abs
```

Listing 8. Analysis of rewards with parametric probability of currents using Storm.

Furthermore, for probabilities of currents less than 0.1 with the symmetric environment and less than 0.3 with the asymmetric environment, the maximum expected energy consumption is very high, independent of the chosen model. Therefore, one should consider using a different model of the AUV when employing it in scenarios with low probabilities of currents. If, however, the probability of currents is above 0.3, the experiments suggest that a higher number of possible altitudes for the AUV could provide an overall better performance: The minimum expected energy is lower and the maximum expected energy consumption is not so much higher. In future work, it would be interesting to investigate if these results can be replicated for even more possible altitudes.

5.5 Reward Properties with Varying Probability of Currents

The probability of currents as specified in our scenarios is only an estimate and can also vary depending on the season. However, the AUV should be able to operate in all different environments with all possible probabilities of currents. Furthermore, it would be useful to have information on the extreme probabilities of currents for the AUV, i.e., obtain worst- and best-case probabilities of currents, to schedule pipeline inspection missions depending on the environmental settings. In this case, it can be useful to leave the probability of currents underspecified, i.e., as a parameter, and analyse the range of minimum and maximum expected rewards depending on the instantiation of the probability of currents. *Parametric model checking* [50] provides dedicated methods to analyse Markovian models where transition probabilities may involve parameters. While PRISM supports parametric model checking through a reimplementation of the seminal parametric model checker PARAM [50], recent advancements in the field are mainly achieved within the Storm model checker [58]. In fact, already predecessors of the parametric engine of Storm clearly outperformed both, PRISM and PARAM in both functionality and speed [35]. This could also be witnessed in the following experimental studies on parametrised AUV models, where PRISM was not able to provide results within a reasonable time. With Storm, however, the analysis of all reported properties took less than 11 seconds where the analysis for the three-altitudes model took less than 4 seconds.

In the models for parametric model checking, we left the probability of currents as a single parameter. To analyse the minimum expected `energy_3alt` reward depending on the instantiation of `current_prob`, the command in Listing 8 was used. The binary `storm-pars` invokes the parametric model-checking engine of Storm that can be used with PRISM files by specifying the PRISM files to be analysed after `--prism`. The property to be analysed can be specified after `--prop`. Here, the same property as specified in Line 1 of Listing 6 is used. Since the property does not contain a bound for the reward, we have to specify with `--direction min` that the reward should be minimised. That is, for all possible instantiations of `current_prob`, Storm computes the minimal expected reward for `energy_3alt`, providing an interval of values whose lower bound is given. The upper bound can be found by using `--direction max`.

Storm supports several different engines for `storm-pars` which are called *modes* and have to be specified after `--mode`. In the experiments, we use the *feasibility* mode to find parameter values such that the specified property holds. In the feasibility mode, either *parameter lifting (pla)* [72] or *gradient decent (gd)* can be used and specified after `--feasibility:method`. The `--region` option can be used to specify the region of the parameter, and `--guarantee` to specify how close the result should be to the optimal value.

Table 6. Parametric Model Checking in Scenario 3 with Asymmetric Environment for Parametric Probability of Currents with 0.3≤current_prob≤0.8

Property		Direction	3-Altitudes		5-Altitudes	
			Result	current_prob	Result	current_prob
energy_3alt	min	min	24.7844	0.55	19.1614	0.300015259
		max	24.7844	0.55	22.0147	0.799992371
	max	min	27.9799	0.799999762	28.0336	0.799999523
		max	239.8359	0.300003815	545.114	0.300007629
energy_5alt	min	min	N/A	N/A	24.4879	0.30012207
		max	N/A	N/A	24.7843	0.76875
	max	min	N/A	N/A	30.8068	0.799998093
		max	N/A	N/A	621.3432	0.300007629
time	min	min	23.7638	0.30012207	23.5507	0.300061035
		max	23.8936	0.7375	23.8937	0.76875
	max	min	25.4847	0.799996185	28.3323	0.799996185
		max	97.5207	0.300003815	226.3386	0.300007629

We used Scenario 3 with the asymmetric environment and parametric probability of currents for parametric analysis where we restricted the probability of currents to be between 0.3 and 0.8. The used interval for the probability of currents is an estimate and can be made more realistic by consulting domain experts. Similarly to the command in Listing 8, we analysed the minimum and maximum rewards for `energy_3alt`, `energy_5alt` and `time` both in minimum and maximum direction, providing an interval of possible values for each of them.

The results of the analysis are reported in Table 6 and are two-fold: they provide an interval of possible values for each property and optimal values for the probability of currents to minimise or maximise the property. Consider for example minimising the `energy_3` reward for finishing the pipeline inspection as in Line 1 of Listing 6. The results of the analysis can be found in the first two lines of Table 6 where the "min direction" provides the lower bound of possible values, and the "max direction" an upper bound. Therefore, in the three-altitudes model, the `energy_3alt` reward is always 24.7844, whereas it is in the interval [19.1614, 22.0147] in the five-altitudes model with the lower bound achieved with `current_prob`≈0.3 and the upper bound achieved with `current_prob` ≈0.8. Thus, the lower and upper bounds in the five-altitudes model are achieved with the minimal and maximal possible value of `current_prob`.

Note that the results of Table 6 are also reflected in the graphs of Figures 7(b) and 8(b). As an example, consider the maximum `energy_3alt` reward in the three-altitudes model. Table 6 shows that the minimum of all maximal rewards (with probability of currents between 0.3 and 0.8) is achieved with a probability of currents of almost 0.8 at a value of approximately 24, and the maximum of all maximal rewards is achieved with a probability of currents of ≈0.3 at a value of ≈239. The red line in Figure 8(b) also shows these values. However, while PRISM experiments provide a way to see how one or several parameters influence the model over the whole parameter space, parametric model checking with Storm focuses on computing only the extreme values. With both techniques, it is possible to specify how close one wants to be to the optimal solution. In the case of parametric model checking with Storm, this can be done explicitly by defining a *guarantee*, while it can only be done implicitly with PRISM experiments by setting the step size for the parameter. When using PRISM experiments, the (close to) optimal result has to be determined by inspecting the calculated values which can be tedious. Parametric model checking with Storm, on the other

```
1 multi(R{\"time\"}<=23.8918 [F _s=done], R{\"energy_3alt\"}<=24.7845 [F _s = done])
2 multi(R{\"time\"}min=?[ F _s=done ], R{\"energy_3alt\"}<=24.7845 [F _s = done])
3 multi(R{\"time\"}min=?[ F _s=done ], R{\"energy_3alt\"}min=? [F _s = done]) --
      multiobjective:exportplot plot/
```

Listing 9. Multi-objective queries.

hand, provides this information by default. Therefore, depending on the required results, one or the other technique should be chosen.

We only display the results for Scenario 3 with the asymmetric environment here to give a taste of the possible analysis. Analysing the same properties with Scenario 3 and the symmetric environment or with Scenario 4 is possible but takes much more time. For example, the analysis in Listing 8 for Scenario 3 with the asymmetric environment took less than two seconds, while it took approximately 25 minutes for Scenario 4 with the same environment. Since Scenario 4 represents a longer pipeline inspection than Scenario 3, the model is much bigger which could explain this difference. However, further investigation is needed to explain why the analysis with the symmetric environment is so much slower.

It would also be interesting to use parametric model checking with a completely parametric scenario. However, the values of `min_visib` and `max_visib` are used to calculate the thresholds for changing between the search altitudes. Therefore, they cannot be left unspecified, making parametric model checking with a completely parametric scenario unfeasible for our case study.

5.6 Multi-objective Queries

The queries considered until now only considered one property at a time. For example, in Section 5.2, we asked for the minimum and maximum time and minimum and maximum energy rewards for finishing a pipeline inspection. However, intuitively, there is a tradeoff between finishing the pipeline inspection in a short time (minimising the reward `time`) and consuming less energy (minimising the rewards `energy_3alt` and `energy_5alt`). To minimise time, one wants to go to a higher altitude if possible because the probability of finding the pipeline is higher. On the other hand, switching between altitudes takes a lot of energy and should be avoided when minimising the energy rewards. Therefore, it is interesting to analyse the time and energy rewards in combination, which can be done using *multi-objective queries*.

PRISM and Storm allow three different types of multi-objective queries[7,8]: (1) *achievability* queries, asking whether it is possible to achieve two properties at the same time, (2) *numerical* queries, leaving the threshold of one property open and asking for the optimal value of the threshold, and (3) *Pareto* queries, where both thresholds are open. To analyse the tradeoff between time and energy rewards we employed the probabilistic model checker Storm, as it provided superior functionality and performance for multi-objective analysis compared to PRISM. For all analyses in this section, we used the three-altitudes model with Scenario 1 and asymmetric environment to exemplify the possible analyses. The multi-objective analysis in Storm took at most a bit more than a millisecond.

Achievability Queries. First, we wanted to know if there really is a tradeoff between minimising time and energy. To analyse this, we used the property in Line 1 of Listing 9 for the achievability query. This represents the question: "Does there exist a strategy for finishing the inspection such that the expected `time` reward is less than or equal to 23.8918 *and* the expected `energy_3alt` reward is less than or equal to 24.7845?". The values used in this query are close to the optimal values as

[7] https://www.prismmodelchecker.org/manual/PropertySpecification/Multi-objectiveProperties
[8] https://www.stormchecker.org/documentation/background/properties.html#multi-objective-model-checking

reported in Table 5. The result of this query is *false*, so it is not possible to finish the inspection with both minimal time and minimal energy. However, if we change the values to 23.9 for the reward `time` and to 24.8 for the reward `energy_3alt`, then the result is *true*. Thus, even though it is not possible to achieve minimal time and energy, it is almost possible.

Numerical Queries. Using numerical queries, we can determine how close to optimal the strategy can be. More specifically, we ask the following question: "What is the minimal expected `time` reward for finishing the inspection over all strategies for which the expected `energy_3alt` reward for finishing the inspection is less than or equal to 24.7845?", see Line 2 of Listing 9. The result is ≈23.8937 which differs from the optimal result by less than 0.002. It is also interesting to know the *maximum* `time` reward given an almost optimal `energy_3alt` reward. This is achieved by replacing `R{"time"}min` with `R{"time"}max` in Line 2. Interestingly, the result only differs from the minimal `time` reward by ≈0.00007.

Similarly, we analysed the minimal and maximal expected `energy_3alt` reward over all strategies for which the expected `time` reward is less than or equal to 23.8918, which both returned ≈24.7893 as a result. Thus, also in this case, the difference between the optimal and the expected value is minimal and the minimal and maximal expected rewards are the same.

Pareto Queries. The previous paragraphs described how to determine whether two properties can be achieved at the same time, and how well one property can be achieved given a threshold for the second property. Another interesting problem is to leave the thresholds for both properties open and ask for the *achievable* and *Pareto-optimal* points, we only consider them informally here and refer the interested reader to [43, 73] for additional explanations.

Consider the case where we want to minimise two objectives as before in this section. Informally, a point $P = (x, y)$ is *achievable*, if there exists a strategy such that the expected value for the first objective is less or equal to x and the expected value for the second objective is less or equal to y. The point P is *Pareto-optimal* if it is achievable and every point $P' = (x', y') \neq P$ with $x' \leq x$ and $y' \leq y$ is not achievable. To analyse achievable and Pareto-optimal points, we used the property in Line 3 of Listing 9 which also gives the files for plotting the achievable and Pareto-optimal points as shown in Figure 9.

The five Pareto-optimal (`time`, `energy_3alt`) points computed by Storm are:

(23.89184492, 24.78901283), (23.89174364, 24.7958376), (23.89373384, 24.78443418),
(23.89280945, 24.78621552), (23.89338681, 24.78505311).

Comparing these pairs with the (non-achievable) optimal pair (23.89170189, 24.78439125), we see that the `time` reward diverges by at most 0.00203195 and the `energy_3alt` reward by at most 0.01144635, and the sum of the pair has at most a difference of 0.0114881 to the optimal pair. Therefore, contrary to our initial intuition, it is almost possible to achieve the optimal result for both minimal time and minimal energy.

6 Related Work

The analysis of behavioural requirements is often crucial when developing an SAS that operates in the uncertainty of a physical environment. These requirements often use quantitative metrics that change during runtime. Both rule-based and goal-based adaptation logics can be used to enable the SAS to meet its behavioural requirements. Many practitioners rely on formal methods to provide evidence for the system's compliance with such requirements [65, 87], also outside the context of SASs [7], but a plethora of different methods are used [2, 56]. Recently, AUV behaviour was modelled and analysed with timed automata and UPPAAL [75].

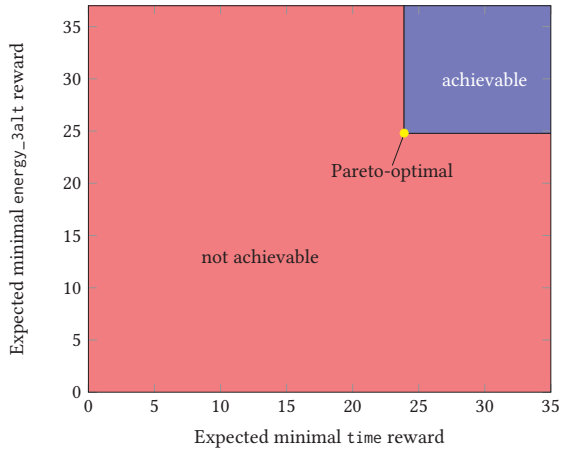


Fig. 9. Achievable points for minimising the rewards `time` and `energy_3alt` with the three-altitudes model, Scenario 1 and asymmetric environment.

6.1 SPLs for Self-adaptive and Robotic Systems

SPLs have previously been proposed to model variability for robotic systems, focusing on static variability modelled during design time [46]. In [18], it is argued that most of the costs for robotic systems come from non-reusable software. A robotic system mostly contains software tailored to the specific application and embodiment of the robot, and often even software libraries for common robotic functionalities are not reusable, so they must be re-developed all the time. Thus, they proposed a new approach for the development of robotic software using SPLs. For robotics, the authors in [46] propose the toolchain HyperFlex to model robotic systems as SPLs; it supports the design and reuse of reference architectures for robotic systems and was extended with the Robot Perception Specification Language for robotic perception systems in [20]. It allows to represent variability at different abstraction levels, and feature models from different parts of the system can be composed in several different ways. However, contrary to the approach used in this article, HyperFlex only considers design time variability. Furthermore, it is only used for modelling robotic systems, not for analysing them.

DSPLs have been proposed to manage variability during runtime [33, 49, 51, 57]. Several approaches model SASs as DSPLs to account for runtime variability such as, e.g., [15, 30, 36, 47, 52, 63, 78]. These approaches focus on modelling SASs using DSPLs and not on formal analysis. DSPLs have also been proposed to manage variability during runtime in the context of self-adaptive robots [19].

6.2 Family-based Model Checking

The benefits of family-based model checking using an annotative approach are outlined in the aforementioned seminal work on FTSs by Classen et al. [25, 26, 28]. Properties of FTSs can be verified by dedicated family-based model-checking tools such as SNIP [24], ProVeLines [31], fNuSMV [25], and FTS4VMC [8]. Based on this work on FTSs, several other approaches with various modelling languages and formalisms have been proposed [10, 16, 44, 85]. Furthermore, well-known model checkers have been made amenable to family-based model checking by suitable abstractions and encodings applied to the input languages of, e.g., PRISM [23, 42], Maude [64], SPIN [38, 39], mCRL2 [9, 13], and NuSMV [37]. A clear advantage of relying on such well-established and highly optimised model checkers is to avoid having to maintain dedicated SPL model checkers.

Also parametric model checking [34, 50] has been used for the family-based analysis of SPLs [48, 77]. While we use the techniques to synthesise optimal parameters for a SAS environment, the existing techniques use probability parameters to encode variability.

In this article, we used ProFeat [23], a software tool built on top of the de-facto standard input language of PRISM for the analysis of feature-aware probabilistic models. ProFeat hence enables all functionalities provided by the probabilistic model checkers PRISM [62] and Storm [54] also for feature-oriented systems, including analyses by statistical, parametric, and probabilistic model checking. Besides ProFeat, also QFLan [11, 84] offers a powerful modelling language and tool to analyse quantitative SPLs, focusing on probabilistic simulations to yield statistical approximations, trading 100% precision for scalability.

6.3 Feature-oriented Analysis of SASs

Dubslaff et al. [42] modelled and analysed a self-adaptive energy-aware server system using feature-oriented formalisms. There, the feature controller for dynamically switching features only sets constraints on self-adaptation and does not actively perform the adaptation actions as the managing subsystem does in our article. Chrszon et al. [21] have modelled and analysed SASs using feature-oriented approaches by means of *multi-product lines*. Specific for their approach is the concept of *roles* that correspond to features but besides activating ("admitting a role") also distinguish actively performing role actions ("role playing"). While not explicitly separating managed and managing subsystem, their *role manager* takes on the purpose of managing subsystem and allows for multiple managed subsystems (e.g., multiple robots in a production pipeline). This approach has also been extended to the quantitative setting and family-based model checking was used to analyse the reliability of a self-adaptive robotic cell [22].

7 Discussion and Future Work

In this article, we have demonstrated how a self-adaptive AUV system can be modelled and analysed using feature-oriented approaches. To this aim, we considered a two-layered SAS architecture of a self-adaptive AUV used for pipeline inspection. The variability of the managed subsystem was modelled as a feature model and the behaviour of the resulting (feature) configurations was modelled as a probabilistic (featured) transition system that also encodes the transitions between configurations. The managing subsystem was modelled as a feature controller switching between features and thereby enabling different behaviour (i.e., operation modes) of the managed subsystem.

We modelled the case study in the tool ProFeat and used it for family-based probabilistic model checking of both reward and safety properties via both PRISM and Storm. In particular, we analysed safety guarantees concerning an AUV's mission duration and energy usage as well as concerning the reachability of (un)safe states, the impact that different environments have on these safety guarantees, to what extent environmental conditions affect the AUV, and tradeoffs between mission duration and energy usage. For a more realistic analysis of an AUV, both the models of the AUV and of the environment, and in particular the probabilities, will have to be adapted to the robot and the environment with the help of real data and domain experts. The models will have to be extracted from the real system at the correct level of abstraction to verify the desired properties. We plan to further investigate this together with an industrial partner of the MSCA network REMARO (Reliable AI for Marine Robotics).

7.1 Using SPL Techniques for Modelling and Analysing SASs

In this article, we have used a feature-oriented approach for modelling and analysing the two-layered self-adaptive AUV. This allowed us to model the managed subsystem of the AUV as a family of systems (or products), where each family member corresponds to a valid configuration of the

AUV. The managing subsystem could then be considered as a control layer capable of dynamically switching between these feature configurations depending on both environmental and internal conditions. ProFeat was used for probabilistic family-based model checking, analysing reward and safety properties.

The ProFeat tool allowed us to directly model the two different layers of the SAS, the managed and managing subsystem, which also makes it easier to understand the model as well as the adaptation logic. Furthermore, it made analysing all configurations of the managed subsystem more efficient by enabling family-based (SPL) model checking. However, it remains to be seen how this scales with larger models than the ones considered in this article. Alternatively, it would also be possible to model the case study without a feature-oriented approach by encoding the features as variables. Besides others, the transitions of the managed and managing subsystem, which would then use these variables, would need to explicitly take care of staying within valid feature configurations. ProFeat conducts this approach in an automated manner, internally transforming the feature-oriented model into a PRISM model agnostic to features. The theoretical underpinning of this transformation requires several technicalities, which are difficult to ensure when done manually [41]. Even in our small case study, the number of transitions in the managed subsystem increased from 44 to 351 when transforming the ProFeat model into a PRISM model. This increase is mainly due to the required synchronisation between the managed and managing subsystems, where a correct modelling is ensured by the transformations.

A particular aspect of our work is the use of a two-layered model to analyse self-adaptation in the managing subsystem of the AUV. This allows us to not only model and analyse a family of configurations of the managed subsystem by means of family-based model checking, but also model and analyse the self-adaptation strategy by means of a feature controller which switches between these configurations depending on the so-called uncertainties of the uncontrolled environment in which the AUV operates. This feature controller, which dynamically changes between products in the configuration space of the managed subsystem, has similarities to variability management in dynamic SPLs; in our work, we explore this approach to not only model but also analyse self-adaptive behaviour in AUVs. Our case study suggests that family-based modelling and analysis of SASs using ProFeat and (probabilistic) model checking is both a very natural and feasible approach.

The case study in this article is of course an abstract model of an AUV and its mission. However, we showed through an extensive set of analyses that it is feasible to model and analyse a two-layered self-adaptive cyber-physical system as a family of configurations with a controller switching between them.

7.2 Future Work

In the future, we plan to investigate which kinds of SASs can be modelled and analysed following the overall approach applied to this case study, aiming to propose a general methodology for modelling and analysing SASs as family-based systems. Furthermore, we plan to find optimal strategies for the managing subsystem, i.e., the controller switching between features to, e.g., minimise energy consumption. We would also like to find patterns between choosing a certain feature configuration and the effect of this choice on quality criteria for the system. Finding such control patterns could help to improve the adaptation logic of the managing subsystem to be more resilient towards faults. Another interesting research direction would be to include the model into a real system and update, e.g., the probabilities using real-time information. Then the model could be run regularly to re-verify system properties upon model updates. Finally, we would like to investigate how our approach scales with larger system models than the ones considered in this article.

Acknowledgments

We would like to thank Matthias Volk and Tim Quatmann for answering various questions about Storm and multi-objective model checking.

References

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, Germany. DOI: https://doi.org/10.1007/978-3-642-37521-7

[2] Hugo Araujo, Mohammad Reza Mousavi, and Mahsa Varshosaz. 2023. Testing, validation, and verification of robotic and autonomous systems: A systematic review. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 51:1–51:61. DOI: https://doi.org/10.1145/3542945

[3] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert K. Brayton. 1996. Verifying continuous time Markov chains. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV 1996)*, Rajeev Alur and Thomas A. Henzinger (Eds.). LNCS, Vol. 1102, Springer, Germany, 269–276. DOI: https://doi.org/10.1007/3-540-61474-5_75

[4] Christel Baier. 1998. *On Algorithmic Verification Methods for Probabilistic Systems*. Universität Mannheim.

[5] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Press, USA.

[6] Don Batory, David Benavides, and Antonio Ruiz Cortés. 2006. Automated analysis of feature models: Challenges ahead. *Communcations of the ACM* 49, 12 (2006), 45–47. DOI: https://doi.org/10.1145/1183236.1183264

[7] Maurice H. ter Beek, Rod Chapman, Rance Cleaveland, Hubert Garavel, Rong Gu, Ivo ter Horst, Jeroen J. A. Keiren, Thierry Lecomte, Michael Leuschel, Kristin Yvonne Rozier, Augusto Sampaio, Cristina Seceleanu, Martyn Thomas, Tim A. C. Willemse, and Lijun Zhang. 2024. Formal methods in industry. *Formal Aspects of Computing* 37, 1 (2025), 7:1–7:38. DOI: https://doi.org/10.1145/3689374

[8] Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. 2022. Efficient static analysis and verification of featured transition systems. *Empirical Software Engineering* 22, 1 (2022), 10:1–10:43. DOI: https://doi.org/10.1007/s10664-020-09930-8

[9] Maurice H. ter Beek, Erik P. de Vink, and Tim A. C. Willemse. 2017. Family-based model checking with mCRL2. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE 2017)*. Marieke Huisman and Julia Rubin (Eds.), LNCS, Vol. 10202, Springer, Germany, 387–405. DOI: https://doi.org/10.1007/978-3-662-54494-5_23

[10] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. 2016. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *Journal of Logical and Algebraic Methods in Programming* 85, 2 (2016), 287–315. DOI: https://doi.org/10.1016/j.jlamp.2015.11.006

[11] Maurice H. ter Beek, Axel Legay, Alberto Lluch Lafuente, and Andrea Vandin. 2020. A framework for quantitative modeling and analysis of highly (re)configurable systems. *IEEE Transaction on Software Engineering* 46, 3 (2020), 321–345. DOI: https://doi.org/10.1109/TSE.2018.2853726

[12] Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. 2019. Textual variability modeling languages: An overview and considerations. In *Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC 2019)*. ACM, USA, 82:1–82:7. DOI: https://doi.org/10.1145/3307630.3342398

[13] Maurice H. ter Beek, Sjef van Loo, Erik P. de Vink, and Tim A.C. Willemse. 2020. Family-based SPL model checking using parity games with variability. In *Proceedings of the 23rd International Conference on Fundamental Approaches to Software Engineering (FASE 2020)*. Heike Wehrheim and Jordi Cabot (Eds.), LNCS, Vol. 12076, Springer, Germany, 245–265. DOI: https://doi.org/10.1007/978-3-030-45234-6_12

[14] Nelly Bencomo, Svein O. Hallsteinsen, and Eduardo Santana de Almeida. 2012. A view of the dynamic software product line landscape. *IEEE Computer* 45, 10 (2012), 36–41. DOI: https://doi.org/10.1109/MC.2012.292

[15] Nelly Bencomo, Peter Sawyer, Gordon S. Blair, and Paul Grace. 2008. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *Proceedings of the 12th International Conference on Software Product Lines (SPLC 2008)*. Steffen Thiel and Klaus Pohl (Eds.), Vol. 2, Lero, University of Limerick, Ireland, 23–32.

[16] Harsh Beohar, Mahsa Varshosaz, and Mohammad Reza Mousavi. 2016. Basic behavioral models for software product lines: Expressiveness and testing pre-orders. *Science of Computer Programming* 123 (2016), 42–60. DOI: https://doi.org/10.1016/j.scico.2015.06.005

[17] Darko Bozhinoski, Mario Garzon Oviedo, Nadia Hammoudeh Garcia, Harshavardhan Deshpande, Gijs van der Hoorn, Jon Tjerngren, Andrzej Wąsowski, and Carlos Hernández Corbato. 2022. MROS: Runtime adaptation for robot control architectures. *Advanced Robotics* 36, 11 (2022), 502–518. DOI: https://doi.org/10.1080/01691864.2022.2039761

[18] Davide Brugali. 2021. Software product line engineering for robotics. In *Software Engineering for Robotics*. Ana Cavalcanti, Brijesh Dongol, Rob Hierons, Jon Timmis, and Jim Woodcock (Eds.), Springer, Germany, 1–28. DOI: https://doi.org/10.1007/978-3-030-66494-7_1

[19] Davide Brugali, Rafael Capilla, and Mike Hinchey. 2015. Dynamic variability meets robotics. *IEEE Computer* 48, 12 (2015), 94–97. DOI: https://doi.org/10.1109/MC.2015.354

[20] Davide Brugali and Nico Hochgeschwender. 2017. Managing the functional variability of robotic perception systems. In *Proceedings of the 1st International Conference on Robotic Computing (IRC 2017)*. IEEE, USA, 277–283. DOI: https://doi.org/10.1109/IRC.2017.20

[21] Philipp Chrszon, Christel Baier, Clemens Dubslaff, and Sascha Klüppelholz. 2020. From features to roles. In *Proceedings of the 24th International Systems and Software Product Line Conference (SPLC 2020)*. ACM, USA, 19:1–19:11. DOI: https://doi.org/10.1145/3382025.3414962

[22] Philipp Chrszon, Christel Baier, Clemens Dubslaff, and Sascha Klüppelholz. 2023. Interaction detection in configurable systems – a formal approach featuring roles. *Journal of Systems and Software* 196 (2023), 111556. DOI: https://doi.org/10.1016/j.jss.2022.111556

[23] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. 2018. ProFeat: Feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects of Computing* 30, 1 (2018), 45–75. DOI: https://doi.org/10.1007/s00165-017-0432-4

[24] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. 2012. Model checking software product lines with SNIP. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 589–612. DOI: https://doi.org/10.1007/s10009-012-0234-1

[25] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. 2014. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Science of Computer Programming* 80 (2014), 416–439. DOI: https://doi.org/10.1016/j.scico.2013.09.019

[26] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. 2013. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Transaction on Software Engineering* 39, 8 (2013), 1069–1089. DOI: https://doi.org/10.1109/TSE.2012.86

[27] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. 2008. What's in a feature: A requirements engineering perspective. In *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE 2008)*. José Luiz Fiadeiro and Paola Inverardi (Eds.), LNCS, Vol. 4961, Springer, Germany, 16–30. DOI: https://doi.org/10.1007/978-3-540-78743-3_2

[28] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 2010. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010)*. ACM, USA, 335–344. DOI: https://doi.org/10.1145/1806799.1806850

[29] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley, USA.

[30] Maxime Cordy, Andreas Classen, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. 2013. Model checking adaptive software with featured transition systems. In *Assurances for Self-Adaptive Systems: Principles, Models, and Techniques*. Javier Cámara, Rogério de Lemos, Carlo Ghezzi, and Antónia Lopes (Eds.), LNCS, Vol. 7740, Springer, Germany, 1–29. DOI: https://doi.org/10.1007/978-3-642-36249-1_1

[31] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2013. ProVeLines: A product line of verifiers for software product lines. In *Proceedings of the 17th International Software Product Line Conference (SPLC 2013)*. ACM, USA, 141–146. DOI: https://doi.org/10.1145/2499777.2499781

[32] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, USA.

[33] Ferruccio Damiani and Ina Schaefer. 2011. Dynamic delta-oriented programming. In *Proceedings of the 15th Software Product Line Conference (SPLC 2011)*. ACM, USA, 34:1–34:8. DOI: https://doi.org/10.1145/2019136.2019175

[34] Conrado Daws. 2004. Symbolic and parametric model checking of discrete-time Markov chains. In *Proceedings of the Revised Selected Papers of the 1st International Colloquium on Theoretical Aspects of Computing (ICTAC 2004)*. Zhiming Liu and Keijiro Araki (Eds.), LNCS, Vol. 3407, Springer, Germany, 280–294. DOI: https://doi.org/10.1007/978-3-540-31862-0_21

[35] Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Harold Bruintjes, Joost-Pieter Katoen, and Erika Ábrahám. 2015. PROPhESY: A PRObabilistic ParamEter SYnthesis tool. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV 2015)*. Daniel Kroening and Corina S. Pasareanu (Eds.), LNCS, Vol. 9206, Springer, Germany, 214–231. DOI: https://doi.org/10.1007/978-3-319-21690-4_13

[36] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. 2007. Domain-specific adaptations of product line variability modeling. In *Proceedings of the IFIP WG 8.1 Working Conference on Situational Method Engineering: Fundamentals and Experiences (ME 2007)*. Jolita Ralyté, Sjaak Brinkkemper, and Brian Henderson-Sellers (Eds.), IFIP, Vol. 244, Springer, Germany, 238–251. DOI: https://doi.org/10.1007/978-0-387-73947-2_19

[37] Aleksandar S. Dimovski. 2020. CTL* family-based model checking using variability abstractions and modal transition systems. *International Journal on Software Tools for Technology Transfer* 22, 1 (2020), 35–55. DOI: https://doi.org/10.1007/s10009-019-00528-0

[38] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. 2017. Efficient family-based model checking via variability abstractions. *International Journal on Software Tools for Technology Transfer* 5, 19 (2017), 585–603. DOI: https://doi.org/10.1007/s10009-016-0425-2

[39] Aleksandar S. Dimovski and Andrzej Wąsowski. 2017. Variability-specific abstraction refinement for family-based model checking. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE 2017)*. Marieke Huisman and Julia Rubin (Eds.), LNCS, Vol. 10202, Springer, Germany, 406–423. DOI: https://doi.org/10.1007/978-3-662-54494-5_24

[40] Clemens Dubslaff. 2019. Compositional feature-oriented systems. In *Proceedings of the 17th International Conference on Software Engineering and Formal Methods (SEFM 2019)*. Peter Csaba Ölveczky and Gwen Salaün (Eds.), LNCS, Vol. 11724, Springer, Germany, 162–180. DOI: https://doi.org/10.1007/978-3-030-30446-1_9

[41] Clemens Dubslaff. 2021. *Quantitative Analysis of Configurable and Reconfigurable Systems*. Ph.D. Dissertation. TU Dresden. Retrieved from https://tud.qucosa.de/id/qucosa:78543

[42] Clemens Dubslaff, Christel Baier, and Sascha Klüppelholz. 2015. Probabilistic model checking for feature-oriented systems. In *Transactions on Aspect-Oriented Software Development XII*. Shigeru Chiba, Éric Tanter, Erik Ernst, and Robert Hirschfeld (Eds.), LNCS, Vol. 8989, Springer, Germany, 180–220. DOI: https://doi.org/10.1007/978-3-662-46734-3_5

[43] Vojtech Forejt, Marta Z. Kwiatkowska, and David Parker. 2012. Pareto curves for probabilistic model checking. In *Proceedings of the 10th International Symposium on Automated Technology for Verification and Analysis (ATVA 2012)*. Supratik Chakraborty and Madhavan Mukund (Eds.), LNCS, Vol. 7561, Springer, Germany, 317–332. DOI: https://doi.org/10.1007/978-3-642-33386-6_25

[44] Vanderson Hafemann Fragal, Adenilso da Silva Simão, and Mohammad Reza Mousavi. 2019. Hierarchical featured state machines. *Science of Computer Programming* 171 (2019), 67–88. DOI: https://doi.org/10.1016/J.SCICO.2018.10.001

[45] Simos Gerasimou, Radu Calinescu, Stepan Shevtsov, and Danny Weyns. 2017. UNDERSEA: An exemplar for engineering self-adaptive unmanned underwater vehicles. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2017)*. IEEE, USA, 83–89. DOI: https://doi.org/10.1109/SEAMS.2017.19

[46] Luca Gherardi and Davide Brugali. 2014. Modeling and reusing robotic software architectures: the HyperFlex toolchain. In *Proceedings of the International Conference on Robotics and Automation (ICRA 2014)*. IEEE, USA, 6414–6420. DOI: https://doi.org/10.1109/ICRA.2014.6907806

[47] Carlo Ghezzi and Amir Molzam Sharifloo. 2010. Dealing with non-functional requirements for adaptive systems via dynamic software product-lines. In *Software Engineering for Self-Adaptive Systems II*. Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw (Eds.), LNCS, Vol. 7475, Springer, Germany, 191–213. DOI: https://doi.org/10.1007/978-3-642-35813-5_8

[48] Carlo Ghezzi and Amir Molzam Sharifloo. 2011. Verifying non-functional properties of software product lines: Towards an efficient approach using parametric model checking. In *Proceedings of the 15th International Software Product Lines Conference (SPLC 2011)*. IEEE, USA, 170–174. DOI: https://doi.org/10.1109/SPLC.2011.33

[49] Hassan Gomaa and Mohamed Hussein. 2003. Dynamic software reconfiguration in software product families. In *Revised Papers of the 5th International Workshop on Software Product-Family Engineering (PFE 2003)*. Frank van der Linden (Ed.), LNCS, Vol. 3014, Springer, Germany, 435–444. DOI: https://doi.org/10.1007/978-3-540-24667-1_33

[50] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. 2010. PARAM: A model checker for parametric Markov models. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV 2010)*. Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.), LNCS, Vol. 6174, Springer, Germany, 660–664. DOI: https://doi.org/10.1007/978-3-642-14295-6_56

[51] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2013. Dynamic software product lines. In *Systems and Software Variability Management: Concepts, Tools and Experiences*. Rafael Capilla, Jan Bosch, and Kyo-Chul Kang (Eds.), Springer, Germany, 253–260. DOI: https://doi.org/10.1007/978-3-642-36583-6_16

[52] Svein Hallsteinsen, Erlend Stav, Arnor Solberg, and Jacqueline Floch. 2006. Using product line techniques to build adaptive systems. In *Proceedings of the 10th International Software Product Line Conference (SPLC 2006)*. IEEE, USA, 141–150. DOI: https://doi.org/10.1109/SPLINE.2006.1691586

[53] Hans Hansson and Bengt Jonsson. 1994. A logic for reasoning about time and reliability. *Formal Aspects of Computing* 6, 5 (1994), 512–535. DOI: https://doi.org/10.1007/BF01211866

[54] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2022. The probabilistic model checker Storm. *International Journal on Software Tools for Technology Transfer* 24, 4 (2022), 589–610. DOI: https://doi.org/10.1007/S10009-021-00633-Z

[55] Carlos Hernández Corbato. 2013. *Model-Based Self-awareness Patterns for Autonomy*. Ph.D. Dissertation. Universidad Politécnica de Madrid. DOI: https://doi.org/10.20868/UPM.thesis.23178

[56] Sara M. Hezavehi, Danny Weyns, Paris Avgeriou, Radu Calinescu, Raffaela Mirandola, and Diego Perez-Palacin. 2021. Uncertainty in self-adaptive systems: A research community perspective. *ACM Transactions on Autonomous and Adaptive Systems* 15, 4 (2021), 10:1–10:36. DOI: https://doi.org/10.1145/3487921

[57] Mike Hinchey, Sooyong Park, and Klaus Schmid. 2012. Building dynamic software product lines. *IEEE Computer* 45, 10 (2012), 22–26. DOI: https://doi.org/10.1109/MC.2012.332

[58] Sebastian Junges, Erika Ábrahám, Christian Hensel, Nils Jansen, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2024. Parameter synthesis for Markov models: Covering the parameter space. *Formal Methods in System Design* 62, 1 (2024), 181–259. DOI: https://doi.org/10.1007/s10703-023-00442-x

[59] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Carnegie Mellon University.

[60] Christian Kästner and Sven Apel. 2008. Integrating compositional and annotative approaches for product line engineering. In *Proceedings of the Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE 2008)*. Neil Loughran, Iris Groher, Roberto Lopez-Herrejon, Sven Apel, and Christa Schwanninger (Eds.), University of Passau, Germany, 35–40.

[61] Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *IEEE Computer* 36, 1 (2003), 41–50. DOI: https://doi.org/10.1109/MC.2003.1160055

[62] Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of probabilistic real-time systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*. Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), LNCS, Vol. 6806, Springer, Germany, 585–591. DOI: https://doi.org/10.1007/978-3-642-22110-1_47

[63] Yu Liu and René Meier. 2009. Resource-aware contracts for addressing feature interaction in dynamic adaptive systems. In *Proceedings of the 5th International Conference on Autonomic and Autonomous Systems (ICAS 2009)*. IEEE, USA, 346–350. DOI: https://doi.org/10.1109/ICAS.2009.24

[64] Malte Lochau, Stephan Mennicke, Hauke Baller, and Lars Ribbeck. 2016. Incremental model checking of delta-oriented software product lines. *Journal of Logical and Algebraic Methods in Programming* 85, 1 (2016), 245–267. DOI: https://doi.org/10.1016/j.jlamp.2015.09.004

[65] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. 2019. Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys* 52, 5 (2019), 100:1–100:41. DOI: https://doi.org/10.1145/3342355

[66] Steven Macenski, Tully Foote, Brian P. Gerkey, Chris Lalancette, and William Woodall. 2022. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics* 7, 66 (2022), eabm6074. DOI: https://doi.org/10.1126/SCIROBOTICS.ABM6074

[67] Juliane Päßler, Maurice H. ter Beek, Ferruccio Damiani, Clemens Dubslaff, Einar Broch Johnsen, and Silvia Lizeth Tapia Tarifa. 2024. *Feature-Oriented Modelling and Analysis of a Self-Adaptive Robotic System (Artifact)*. Zenodo. DOI: https://doi.org/10.5281/zenodo.14228090

[68] Juliane Päßler, Maurice H. ter Beek, Ferruccio Damiani, Einar Broch Johnsen, and S. Lizeth Tapia Tarifa. 2025. A configurable software model of a self-adaptive robotic system. *Science of Computer Programming* 240 (2025), 103221. DOI: https://doi.org/10.1016/j.scico.2024.103221

[69] Juliane Päßler, Maurice H. ter Beek, Ferruccio Damiani, S. Lizeth Tapia Tarifa, and Einar Broch Johnsen. 2023. Formal modelling and analysis of a self-adaptive robotic system. In *Proceedings of the 18th International Conference on integrated Formal Methods (iFM 2023)*. Paula Herber and Anton Wijs (Eds.), LNCS, Vol. 14300, Springer, Germany, 343–363. DOI: https://doi.org/10.1007/978-3-031-47705-8_18

[70] Juliane Päßler, Maurice H. ter Beek, Ferruccio Damiani, S. Lizeth Tapia Tarifa, and Einar Broch Johnsen. 2023. *Formal Modelling and Analysis of a Self-Adaptive Robotic System (Artifact)*. Zenodo. DOI: https://doi.org/10.5281/zenodo.8275533

[71] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Germany. DOI: https://doi.org/10.1007/3-540-28901-1

[72] Tim Quatmann, Christian Dehnert, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. 2016. Parameter synthesis for y models: Faster than ever. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA 2016)*. Cyrille Artho, Axel Legay, and Doron Peled (Eds.), LNCS, Vol. 9938, Springer, Germany, 50–67. DOI: https://doi.org/10.1007/978-3-319-46520-3_4

[73] Tim Quatmann, Sebastian Junges, and Joost-Pieter Katoen. 2022. Markov automata with multiple objectives. *Formal Methods in System Desisgn* 60, 1 (2022), 33–86. DOI: https://doi.org/10.1007/S10703-021-00364-6

[74] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. 2009. ROS: An open-source Robot Operating System. In *Proceedings of the Open-Source Software Workshop of the International Conference on Robotics and Automation (ICRA 2009)*. IEEE, USA, 6.

[75] Sergio Quijano, Mahsa Varshosaz, and Andrzej Wąsowski. 2024. Modeling and safety analysis of autonomous underwater vehicles behaviors. In *Proceedings of the 17th International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2024)*. IEEE, USA, 63–67. DOI: https://doi.org/10.1109/ICSTW60967.2024.00022

[76] Gustavo Rezende Silva, Juliane Päßler, Jeroen Zwanepol, Elvin Alberts, S. Lizeth Tapia Tarifa, Ilias Gerostathopoulos, Einar Broch Johnsen, and Carlos Hernández Corbato. 2023. SUAVE: An exemplar for self-adaptive underwater vehicles. In *Proceedings of the 18th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2023)*. IEEE, USA, 181–187. DOI: https://doi.org/10.1109/SEAMS59076.2023.00031

[77] Genaína N. Rodrigues, Vander Alves, Vinicius Nunes, André Lanna, Maxime Cordy, Pierre-Yves Schobbens, Amir Molzam Sharifloo, and Axel Legay. 2015. Modeling and verification for probabilistic properties in software product lines. In *Proceedings of the 16th International Symposium on High-Assurance Systems Engineering (HASE 2015)*. IEEE, USA, 173–180. DOI: https://doi.org/10.1109/HASE.2015.34

[78] Karsten Saller, Malte Lochau, and Ingo Reimund. 2013. Context-aware DSPLs: Model-based runtime adaptation for resource-constrained systems. In *Proceedings of the 17th International Software Product Line Conference (SPLC 2013)*. ACM, USA, 106–113. DOI: https://doi.org/10.1145/2499777.2500716

[79] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-oriented programming of software product lines. In *Proceedings of the 14th Software Product Lines Conference (SPLC 2010)*. Jan Bosch and Jaejoon Lee (Eds.), LNCS, Vol. 6287, Springer, Germany, 77–91. DOI: https://doi.org/10.1007/978-3-642-15579-6_6

[80] Klaus Schmid, Sooyong Park, Mike Hinchey, and Svein Hallsteinsen. 2008. Dynamic software product lines. *IEEE Computer* 41, 4 (2008), 93–95. DOI: https://doi.org/10.1109/MC.2008.123

[81] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2006. Feature diagrams: A survey and a formal semantics. In *Proceedings of the 14th International Conference on Requirements Engineering (RE 2006)*. IEEE, USA, 136–145. DOI: https://doi.org/10.1109/RE.2006.23

[82] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys* 47, 1 (2014), 6:1–6:45. DOI: https://doi.org/10.1145/2580950

[83] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. 2012. Family-based deductive verification of software product lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE 2012)*. ACM, USA, 11–20. DOI: https://doi.org/10.1145/2371401.2371404

[84] Andrea Vandin, Maurice H. ter Beek, Axel Legay, and Alberto Lluch Lafuente. 2018. QFLan: A tool for the quantitative analysis of highly reconfigurable systems. In *Proceedings of the 22nd International Symposium on Formal Methods (FM 2018)*. Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik de Vink (Eds.), LNCS, Vol. 10951, Springer, Germany, 329–337. DOI: https://doi.org/10.1007/978-3-319-95582-7_19

[85] Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi. 2018. Basic behavioral models for software product lines: Revisited. *Science of Computer Programming* 168 (2018), 171–185. DOI: https://doi.org/10.1016/J.SCICO.2018.09.001

[86] Danny Weyns. 2020. *An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective*. John Wiley and Sons, UK.

[87] Danny Weyns, M. Usman Iftikhar, Didac Gil de la Iglesia, and Tanvir Ahmad. 2012. A survey of formal methods in self-adaptive systems. In *Proceedings of the 5th International C* Conference on Computer Science and Software Engineering (C3S2E 2012)*. ACM, USA, 67–79. DOI: https://doi.org/10.1145/2347583.2347592

Received 29 August 2024; revised 4 December 2024; accepted 16 December 2024