# Analysing Self-Adaptive Systems as Software Product Lines

Juliane Päßler[a,*], Maurice H. ter Beek[b], Ferruccio Damiani[c], Einar Broch Johnsen[a], S. Lizeth Tapia Tarifa[a]

[a] *University of Oslo, Gaustadalléen 23B, NO-0373, Oslo, Norway*
[b] *CNR–ISTI, Via G. Moruzzi 1, Pisa, I-56124, Italy*
[c] *University of Turin - Department of Computer Science, Corso Svizzera 185, I-10149, Torino, Italy*

**Abstract**

Self-adaptation is a crucial feature of autonomous systems that must cope with uncertainties in, e.g., their environment and their internal state. Self-adaptive systems (SASs) can be realised as two-layered systems, introducing a separation of concerns between the domain-specific functionalities of the system (the *managed* subsystem) and the adaptation logic (the *managing* subsystem), i.e., introducing an external feedback loop for managing adaptation in the system. We present an approach to model SASs as dynamic software product lines (SPLs) and leverage existing approaches to SPL-based analysis for the analysis of SASs. To do so, the functionalities of the SAS are modelled in a feature model, capturing the SAS's variability. This allows us to model the managed subsystem of the SAS as a family of systems, where each family member corresponds to a valid feature configuration of the SAS. Thus, the managed subsystem of an SAS is modelled as an SPL model; more precisely, a probabilistic featured transition system. The managing subsystem of an SAS is modelled as a control layer capable of dynamically switching between these valid configurations, depending on both environmental and internal conditions. We demonstrate the approach on a small-scale evaluation of a self-adaptive autonomous underwater vehicle used for pipeline inspection, which we model and analyse with the feature-aware probabilistic model checker ProFeat. The approach allows us to analyse probabilistic reward and safety properties for the SAS, as well as the correctness of its adaptation logic.

*Keywords:* Dynamic software product line, Self-adaptive system, Feature model, Featured transition system, Probabilistic model checking, Robotics

## 1. Introduction

This paper proposes an approach to model and analyse self-adaptive systems (SASs) [1, 109] as dynamic software product lines (DSPLs) [65, 69]. To demonstrate the usefulness and

*Corresponding author
*Email addresses:* julipas@uio.no (Juliane Päßler), maurice.terbeek@isti.cnr.it (Maurice H. ter Beek), ferruccio.damiani@unito.it (Ferruccio Damiani), einarj@uio.no (Einar Broch Johnsen), sltarifa@uio.no (S. Lizeth Tapia Tarifa)

feasibility of the approach, we conduct a small-scale evaluation [111] using a self-adaptive autonomous underwater vehicle (AUV).

SASs form a fast-evolving field of research and range from applications like smart houses to autonomous robots. They can be realised as two-layered systems with a *managed subsystem* responsible for the domain concerns and a *managing subsystem* responsible for the adaptation logic, i.e., adapting the managed subsystem to changes in the system and its environment. Self-adaptation is extensively applied in industry, for example in the web, mobile and cloud domains and for embedded, cyber-physical and IoT systems [1]. An SAS is able to adapt its behaviour during runtime to, e.g., changes in the environment, failures of the system, or varying user requirements by exploiting existing redundancies of the system. Many SASs include design-time as well as runtime variability. Considering, e.g., self-adaptive robots, the hardware as well as the software can be configured during design time according to, e.g., customers' preferences and the application domain [59]. During runtime, variability in the robot could for example include different localisation techniques that can be selected according to the current environmental conditions. Software product lines (SPLs) have previously been proposed to model static variability of robotic systems [60], i.e., variability during design time, and DSPLs have been proposed to manage variability during runtime for self-adaptive robots [31, 61].

Although the idea of using DSPLs to manage runtime variability for self-adaptive robots is appealing [31], it is still considered an unsolved challenge [59]. Managing runtime variability for SASs is in general very difficult and "there is a need to validate the proposals, either in an industrial environment or in different test cases, expanding the application areas", according to a review on variability management in DSPLs for SASs [3], which considers 84 papers published during 2010–2021. While DSPLs have been used before to manage runtime variability of self-adaptive robots, formal verification techniques that have been applied in the past to DSPLs have not been used to analyse SASs: a recent literature review [7] on testing, validation, and verification of robotic and autonomous systems does not discuss any work using family-based analysis techniques for SASs.

One way to specify SPL models is by means of featured transition systems (FTSs) [40]. An FTS models the behaviour of all configurations of an SPL in a single transition system by associating transitions with features that condition the possibility of executing the transition in specific configurations, governed by a feature (variability) model [5, 100, 23]. Thus, an FTS realises a so-called 150% family model that contains many superimposed variants [98, 106]. When analysing the SPL, the compact structure of an FTS is exploited to reason about the whole SPL, i.e., all variants, at once. This approach was extended to DSPLs by allowing dynamic feature reconfiguration [43] and by supporting probabilistic and non-deterministic choices as well as allowing quantitative analysis [58].

*Methodology*

In this paper, we use family-based techniques from DSPLs to model and analyse a two-layered SAS focusing on runtime variability in the form of activating and deactivating features. To do so, we present a methodology that shows how two-layered SASs can be seen as DSPLs and how existing techniques for modelling and analysing DSPLs can be applied

to SASs. Thereby, we highlight the natural correspondence between SASs and family-based DSPL modelling, and show in particular how family-based analysis methods can be used for SASs. In this paper, we thus propose to use existing techniques and tools known for modelling and analysing DSPLs, to model and analyse SASs. More specifically, we highlight the correspondence between the features of a feature model and the configurations of a managed subsystem, as well as the correspondence between dynamically changing the feature configuration of the managed subsystem and the managing subsystem of the SAS. Using these correspondences, existing tools for the analysis of DSPLs can be leveraged for the analysis of SASs. To demonstrate this, we conduct a small-scale evaluation that exemplifies our approach.

In the small-scale evaluation used in this paper, an AUV is modelled as a probabilistic FTS with dynamic feature switching. As the behaviour of the AUV depends on external conditions, which are hard to control, we opted for a probabilistic model in which uncontrolled events, like a thruster failure, occur with given probabilities. Using a probabilistic model enables us to provide safety guarantees of the system even in the presence of uncertainties. The managed subsystem of the AUV, handling the domain concerns, is modelled as a family of systems whose family members correspond to valid feature configurations, and a feature model captures the domain-specific functionalities of the AUV, making the dependencies and requirements between the components of the AUV explicit. The adaptation behaviour of the AUV (the managing subsystem) is modelled as a control layer with dynamic *feature switches*, changing the feature configurations of the managed subsystem according to input from a probabilistic environment model, a probabilistic hardware failures model and the managed subsystem. As our focus in this paper is on how family-based DSPL modelling and family-based analysis can be used to analyse SASs, our small-scale evaluation presents a simplified version of an AUV, with limited variability. Although the presented model can be extended to a more realistic underwater robot, the small-scale evaluation is sufficient to illustrate the idea.

In our small-scale evaluation, the analysis can give system operators an estimate of the mission duration and of the AUV's energy consumption, provide safety guarantees, and ensure that the implementation of the adaptation logic satisfies certain correctness properties. Our analyses have been performed with ProFeat [37], a tool for probabilistic family-based model checking. Family-based model checking provides a means to simultaneously model check, in a single run, properties of a family of models, each representing a different configuration [105]. We describe families of probabilistic systems as probabilistic FTS models, augmented with *costs* and *rewards*, which are useful for quantitative analysis, i.e., the resulting FTSs are featured Markov decision processes [9].

*Contributions.* This paper is an extension of Päßler et al. [88], with related artifact [89], recently published as an original software publication [87], which introduced a small-scale evaluation of an SAS from the underwater robotics domain, modelled as a probabilistic FTS with dynamic feature switching, and used family-based analysis to verify some essential (quantitative) properties. The contributions of that paper are here extended by including:

- a background discussion and context for the proposed approach to modelling and analysing two-layered SASs as DSPLs (Section 2);

- an extended small-scale evaluation of an SAS from the underwater robotics domain, adding sensors used with different priorities as a possibility for the need to abort the mission and as another reason for adaptation (Section 3);
- an extended probabilistic FTS model with dynamic feature switching, including a new module for modelling sensor failures, an extended managed subsystem and a more complex managing subsystem (Section 4);
- new analyses, including analysing sensor failures and correctness issues of the adaptation logic (Section 5);
- an evaluation of how the proposed approach supports extensibility of the model, to what extent it supports the analysis of an SAS, and how the correctness of the adaptation logic with respect to its specification can be analysed (Section 6).

Overall, we propose to leverage existing techniques for modelling and analysing DSPLs, and apply them to the modelling and analysis of SASs instead of focusing on modelling and analysing a realistic SAS. To show the feasibility of this proposed approach by means of a proof of concept, we conduct a small-scale evaluation of a simplified AUV mission that allows us to highlight the correspondence between a two-layered self-adaptive robotic system and an FTS with its corresponding feature controller for dynamic feature switching.

*Outline.* The paper is structured as follows. Section 2 discusses the correspondence between SASs and family-based DSPL modelling. Section 3 presents the extended small-scale evaluation of a pipeline inspection with an AUV. Section 4 substantiates the discussion in terms of our small-scale evaluation and how it can be modelled as a family-based system, while Section 5 presents the analysis results. Section 6 presents an evaluation of our approach. Section 7 provides an overview of related work, and Section 8 discusses our results and some ideas for future work.

## 2. Family-Based Modelling and Analysis of SASs

In this section, we consider SASs from the perspective of runtime variability and family-based analysis. Since SPLs offer a high-level, structured view of a configuration space, self-adaptation can be understood in terms of runtime variability and DSPLs. However, taking this view on self-adaptation triggers the following question: how can we use family-based DSPL modelling and family-based analysis techniques to analyse self-adaptive behaviour in an SAS to increase its reliability? We argue that analysis techniques for family-based models like FTSs, combined with a model that captures the adaptation logic for runtime variability, moving between different valid products of the product family, can be a natural fit for the analysis of SASs.

In the remainder of this section we detail our argument by (1) giving an overview of what SASs are and the uncertainties that trigger reconfiguration, in particular, uncertainties in physical environments, a very important aspect of resilience in the context of autonomous

robots; (2) giving an overview of product variability approaches at runtime and family-based analysis techniques that mostly explore techniques to analyse a family of products under known contexts; and (3) discussing the natural fit between SASs and DSPLs and summarising our proposal for how to leverage family-based modelling and analysis techniques to explore properties in the reconfiguration space that are valid under certain uncertainties that are captured in the model.

### *2.1. SASs and Uncertainties that Trigger Reconfiguration*

*What are SASs?* Many software systems are subject to different forms of uncertainty like changes in the surrounding environment, internal failures and changing user requirements. Often, manually maintaining and adapting these systems during runtime by a system operator is prohibitively expensive and error-prone. Enabling systems to adapt themselves provides several advantages. For example, a system that is able to perform self-adaptation can also be deployed in environments where communication between an operator and the system is very limited or impossible, e.g., in space or under water. Self-adaptation gives a system a higher level of autonomy.

SASs can be realised by *internal* or *external self-adaptation* [70, 95, 109]. An internal self-adaptation mechanism embeds the adaptation logic within the application logic of the system itself using, e.g., exception handling or fault-tolerance mechanisms. Internal self-adaptation has been criticised for poor maintainability and scalability. In contrast, external self-adaptation introduces a separation of concerns between the application logic and the adaptation logic of the SAS through an external feedback loop. Using external self-adaptation has "the advantage that [it] localize[s] the adaptation concerns in sparable system elements that can be analyzed, modified, and reused across different self-adaptive systems" [109]. In this paper, we focus on external self-adaptation.

SASs with external self-adaptation can be implemented using a two-layered approach which decomposes the system into a *managed* and a *managing* subsystem [70] (see Figure 1). The *managed* subsystem deals with the domain concerns of the application logic and tries to reach the goals set by the user of the system, 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 [109], e.g., adapting to changing environmental conditions. While the managed subsystem may affect the environment with its actions, the managing subsystem monitors the environment and the internal state of the managed subsystem. Using the adaptation logic, the managing subsystem deducts whether and which reconfiguration of the managed subsystem is needed and adapts the managed subsystem accordingly, exploiting so-called hardware and software redundancies of the managed subsystem to do so. For example, for robots, this can be realised in practice [93] using, e.g., the System Modes package [83] of the Robot Operating System (ROS) [90]. Self-adaptive mission control has been realised in several studies (e.g., [67, 94]), enabling on-the-fly manipulation of control strategies (e.g., by activation or deactivation). There are also implementations of some of these systems in ROS [28, 29]. We refer to [101, Chapter 12: Robotic Systems Architectures and Programming] for further robot systems control architectures.

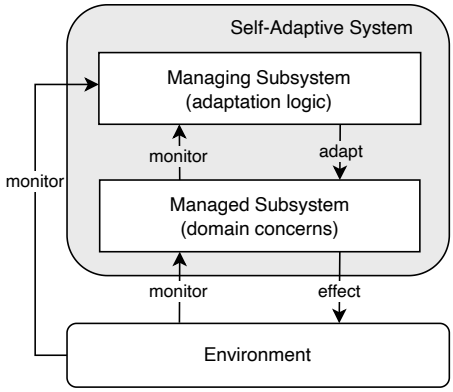


Figure 1: A two-layered architecture for an SAS ([88])

In two-layered SASs, managing subsystems can be realised using a MAPE-K feedback loop [70]. Systems implementing a MAPE-K loop *Monitor* the environment and the managed subsystem during runtime (collecting, e.g., sensor readings and information about the internal state of the managed subsystem); this information is then *Analysed* to decide whether an adaptation of the managed subsystem is needed; if this is the case, the managing subsystem *Plans* a reconfiguration of the managed subsystem, which it afterwards *Executes*. These steps are all performed in the context of a shared *Knowledge base*.

The adaptation logic of the *Plan* phase of the MAPE-K loop is typically specified using either rule-based or goal-based approaches [26], relating the monitored data about the managed subsystem and the environment (e.g., component status information and sensor data) to models in the *Knowledge base* (e.g., valid component configurations and metrics) to systematically plan how to achieve a goal (e.g., safely navigating a robot to a location). Rule-based approaches provide deterministic, predetermined actions in response to changes in the environment and the managed subsystem, whereas goal-based approaches provide more flexibility for optimisation, possibly by systematically exploring all possible configurations before making a decision.

*SAS in Physical Environments.* One of the most important challenges for an SAS that operates in a physical environment (see Figure 1) is to handle the uncertainties that are induced by this environment [68]. SASs that operate in a physical environment, such as, e.g., robots, have to face and overcome this additional uncertainty. Examples of uncertainties in the environment that a robot needs to overcome include unexpected obstacles, erosion or avalanches, and climate events like rain or wind, which affect the robot's ability to manoeuvre or to inspect an asset. While there is no established approach for how to handle uncertainty in general, many practitioners use formal techniques and learning, along with general paradigms like the MAPE-K feedback loop, to develop an adaptation logic that can handle the uncertainties of these environments [68].

To understand how changes in the environment affect the behaviour of the SAS, it is crucial to analyse behavioural requirements when developing an SAS that operates under the uncertainties of a physical environment. These requirements often use quantitative

metrics that change during runtime, such as monitored information about the managed subsystem and the environment. For instance, for behavioural requirements concerning energy consumption, the metric should be minimised, while for safety requirements, the metric should be maximised. 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 [78, 110], but many different methods are used [7, 68].

### *2.2. Towards Managing Variability for a Family of Products*

*What are DSPLs?* DSPLs are a generalisation of SPLs for developing highly-configurable, runtime-adaptive systems, i.e., they allow to switch from one product to another at runtime [26, 69, 99]. In particular, DSPLs structure the configuration space for runtime variability as an SPL. They can, e.g., be realised by means of a logic for self-adaptation that respects the structure of a feature model; different solutions to support runtime variability mechanisms have been proposed [34, 107]. DSPLs may be found in application domains such as service-oriented systems, mobile software, ecosystems and SASs [34].

A crucial distinction within the DSPL design landscape [26] is between *bounded adaptivity*, which deals with context variation that is anticipated at design time, and *open adaptivity*, which deals with context variation that is not planned at design time and requires model extension. For bounded adaptivity, DSPLs have been advocated as a means to constrain the evolution of SASs and to enable the assessment of important properties of a system even before starting its implementation [10]. For open adaptivity, dynamic feature models have been proposed to model variability during runtime [55]. More recently, DSPLs have been advocated to provide a conceptual framework for managing the variability in cyber-physical systems which need to frequently reconfigure their software components (e.g., due to the addition, update or removal of physical components) without being shut down [92].

*Analysis of Family-Based Models.* A range of techniques have been explored for the analysis of SPLs, including type checking, static analysis, theorem proving, and model checking. Thüm et al. [105] provided an overview of these analysis techniques, emphasising in particular *family-based analysis* techniques for SPLs. Family-based techniques are all-in-one techniques, according to which the behaviour of all product configurations (variants) of the SPL is examined in the same analysis, simultaneously. Family-based analysis differs from brute-force enumerative product-based analysis, in which the behaviour of every product (variant) is examined individually, one-by-one.

Family-based model checking is a prominent family-based analysis technique that provides a means to simultaneously model check, in one single run, properties of multiple models, each representing the behaviour of a different product configuration (variant) of the SPL. FTSs were introduced to model the behaviour of SPLs for family-based model checking [42]. Their action-labelled transitions are equipped with feature expressions that condition the presence of transitions in specific product configurations, which are represented by ordinary labelled transition systems obtainable by projection. The compact structure of an FTS enables reasoning on the behaviour of the whole SPL at once. The properties of

FTSs can be verified by dedicated family-based model-checking tools such as SNIP [38, 40], ProVeLines [44], (FTS4)VMC [17, 21, 22, 16] and fNuSMV [39, 53]. Statistical family-based model checking is supported by QFLan [19, 108] and probabilistic family-based model checking by ProFeat [37, 57], which is designed to analyse quantitative properties for families of stochastic systems. Next to dedicated SPL model-checking tools, suitable abstractions or encodings have made well-known classical model checkers such as SPIN [52, 51, 54], PRISM [57], Maude [77], mCRL2 [25, 20] and NuSMV [50] amenable to family-based model checking. The advantage of using well-established and typically highly optimised off-the-shelf model checkers with a broad user base is to avoid having to maintain dedicated SPL model checkers. For example, ProFeat uses PRISM as a back-end tool: after translating its input models to PRISM input models, PRISM is invoked for the actual analysis, after which the output of PRISM is post-processed by ProFeat.

The development of family-based analysis techniques for DSPLs has so far received limited attention. Model-based approaches to handle feature changes at the code level in DSPLs have been proposed [2, 47, 80], combining delta- and feature-oriented programming [5, 6, 97] with techniques for dynamic software updates such as object migration. More recently, a family-based algorithm, based on modelling the configuration space of a DSPL as an FTS, estimates the quality of service (expressed as the sum of weights) provided by each configuration of a weighted FTS over an observed execution trace [84]; the authors analyse the quality of service obtained for the trace using different configurations of the DSPL but do not analyse a self-adaptation policy as such. In contrast, *feature controllers*, which enable the dynamic activation or deactivation of features in the FTS, are supported by the family-based probabilistic model checker ProFeat [37]. Observe that such a feature controller, expressed as a transition system, is restricted to a finite state space and therefore to bounded adaptivity.

### *2.3. Family-Based Modelling and Analysis of Two-Layered SASs*

In this paper, we propose a family-based approach to model and analyse two-layered SASs. The managed subsystem of an SAS needs to include hardware and software redundancies to enable self-adaptation. These redundancies, which can, e.g., be (partially) redundant components, algorithms, and parameters, can be used by the managing subsystem to adapt the managed subsystem if a redundant part does not work as expected or cannot be used in the current (environmental) conditions. For example, an underwater robot might include different sensors that can be used to observe the environment, like a camera and a sonar. Due to energy restrictions, the managing subsystem might only want to use one sensor at a time. However, if one of the sensors breaks or the environmental conditions change, it can still adapt the managed subsystem to use the other sensor even though it might not provide optimal performance. From the SPL perspective, these redundancies can be seen as features of the managed subsystem. Each configuration of the managed subsystem then corresponds to a choice of features. Thus, a managed subsystem can be considered as a family of configurations that share the same core functionalities (features) and differ in their optional features. Consequently, the managed subsystem can be understood as an SPL where a product (variant) corresponds to a configuration of the managed subsystem. The

managing subsystem can then be understood as an adaptation logic for runtime variability for the SPL of the managed subsystem.

Note that we consider all points of variability of the managed subsystem that can be adapted by the managing subsystem to be features[1]. For example, if the managing subsystem adapts parameters of the managed subsystem, these parameters are considered to be features since we want to analyse the interaction between the managed and managing subsystem. While the more common SPL modelling approaches typically would not consider all such variability points to be features, there exist many different definitions of what constitutes a feature [41] and our approach is in line with that of [46, Chapter 4: Feature Modeling]: "anything users or client programs might want to control about a concept is a feature".

Modelling the behaviour of a managed subsystem of an SAS as a 150% SPL model, such as an FTS model, enables analysing the behaviour of all possible configurations of the managed subsystem in one run. However, this 150% SPL model is not sufficient to analyse the two-layered SAS; the model still lacks the managing subsystem. The managing subsystem of the SAS can be seen as a controller that switches between different configurations of the managed subsystem when needed. Triggered by, e.g., changes in the physical environment, the managing subsystem changes the active features of the managed subsystem. Considering an FTS model of the behaviour of a managed subsystem, the managing subsystem enables and disables transitions (i.e., behaviour) dynamically during runtime. Hence, an SAS can be modelled as a 150% SPL model together with a controller that enables and disables behaviour during runtime [58]. To capture the uncertainties in the managed subsystem, a probabilistic model can be chosen.

Therefore, we propose to model a two-layered SAS as a family-based system by modelling the managed subsystem of an SAS as a probabilistic FTS and the managing subsystem as a *non-deterministic controller* activating and deactivating features. This enables the analysis of all possible configurations and re-configurations of the managed subsystem in one single run. The family-based model of the two-layered SAS makes it possible to analyse optimal strategies for the managing subsystem with respect to different desired properties of the managed subsystem. Furthermore, since FTSs have been in use for SPL analysis for some time now, there are already existing, mature tools that can be exploited for the analysis of SASs.

In his keynote address "The 20-year journey of SPLE in Hitachi and the next" at the 2023 SPL Conference (SPLC 2023), Kentaro Yoshimura, chief researcher at Hitachi, presented the use of DSPLs for autonomous robotic systems as a new industrial challenge. He mentioned that the dynamicity is in the runtime behaviour of the autonomous robots that need to adapt and reconfigure based on input perceived from the environment without continuous human guidance. To meet this challenge, we capture the uncertainties of the environment as a probabilistic transition system that we compose with the probabilistic FTS model of the SAS; the idea is that we do not model how or why changes occur in the environment but rather the probabilities with which the SAS observes these changes.

By combining the family-based model of the two-layered SAS with a stochastic model of its environment, we obtain a Markov decision process that can be enriched with costs

[1]The opinions on what constitutes a feature differ between the SPL and the SAS communities.

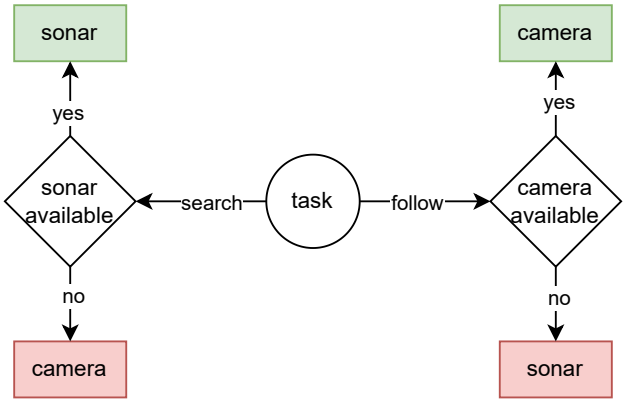


Figure 2: The priorities for choosing the vision sensors; if both vision sensors fail, the mission is aborted

and rewards, and model checked also for non-functional properties as well as for optimal adaptation strategies. It is worth remarking that this approach is restricted to bounded adaptivity in the sense that we do not extend the models. However, the underspecified (or non-deterministic) adaptation logic enables the proposed family-based analysis to find the optimal adaptation logic for a given environment model. The adaptation logic need not be fully understood or specified a priori for all environments; rather, the adaptation logic of the two-level SAS can be automatically adapted to different environment models by means of the proposed family-based analysis.

To demonstrate the proposed methodology of family-based analysis of two-level SASs, Section 3 presents a small-scale evaluation of an SAS with a rule-based managing subsystem. Goal-based systems, which may need to consider multiple possible configurations, fit equally well with family-based models by exploring multiple variants at a time. Section 3 also details how the managed subsystem of the SAS can be viewed as a family-based system, and the managing subsystem as a controller switching between configurations of the managed subsystem. Section 4 then details how the proposed methodology was implemented in the tool ProFeat [37].

## 3. Small-Scale Evaluation: Pipeline Inspection by AUV

In this section, we introduce our small-scale evaluation of an AUV used for pipeline inspection, which was inspired by the exemplar SUAVE [93]. This work extends the small-scale evaluation described and modelled in our previous work [88, 89] by including two sensors for vision, a *camera* and a *(side-scan) sonar*.

### *3.1. An Overview of the Small-Scale Evaluation*

An AUV has the mission to first find and then inspect a pipeline located on a seabed. During system operation, it can use either the camera or the sonar as a vision sensor. Both can be used for searching for the pipeline and for inspecting it. However, the sonar is preferred for searching because it can cover a wider area and operate at a higher altitude, while the camera is preferred for following and inspecting the pipeline because it is easier

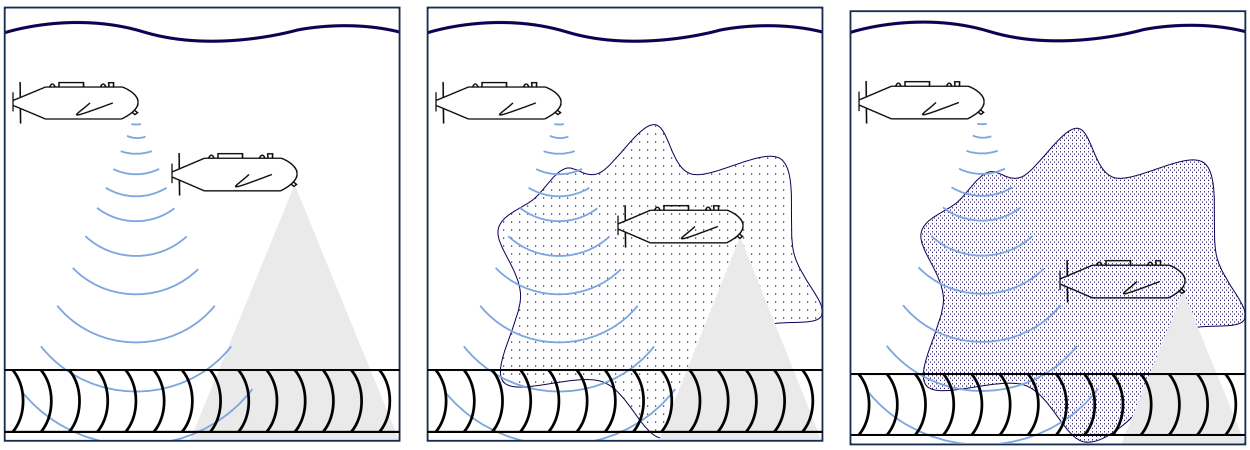

Figure 3: The maximum altitudes in different water visibilities when operating with the sonar (left AUV in each image) and with the camera (right AUV in each image). The left image shows good visibility, the middle medium visibility and the right bad visibility.

to detect faults in the pipeline with the camera, see Figure 2. The water visibility (i.e., the distance in meters within which the AUV can perceive objects with the camera) might change during runtime (e.g., due to currents that swirl up the seabed), affecting at which altitude the AUV can operate when using the camera. Furthermore, one or more of the AUV's thrusters might fail and need to be restarted before the mission can be continued. In addition, the camera and the sonar can have a permanent software or hardware failure, making it necessary to use the other vision sensor or abort the mission if both sensors fail. The camera can also get blocked, e.g., because of natural or human waste sticking to the camera. In this case, the camera is only temporarily unavailable and there is a probability of the camera getting unblocked, thus the camera becoming available again.

The AUV can choose to operate at four different altitudes, *low*, *med* (for medium), *high* and *very high*. A higher altitude allows the AUV to have a wider field of view and thus increases its chances of finding the pipeline during its search. The probability of a thruster failure is lower at a higher altitude because, e.g., seaweed typically might wrap around the thrusters at a lower altitude. However, when using the camera, the altitude at which the AUV can perceive the seabed depends on the water visibility, and the AUV can never perceive the seabed from a very high altitude. With low water visibility, the AUV cannot perceive the seabed with the camera from a high or medium altitude. Thus, it is not always possible to operate at a high or medium altitude, and the altitude of the AUV needs to be changed during the search when using the camera, depending on the current environmental conditions. However, when using the sonar for searching, the AUV can always operate at a very high altitude because the water visibility does not affect the sonar; see Figure 3 for an illustration of the maximum altitudes when operating with the sonar and the camera in different environmental conditions.

Once the pipeline is found, the AUV will follow it at a low altitude to increase the resolution of the camera and sonar images. However, the AUV can also lose the pipeline

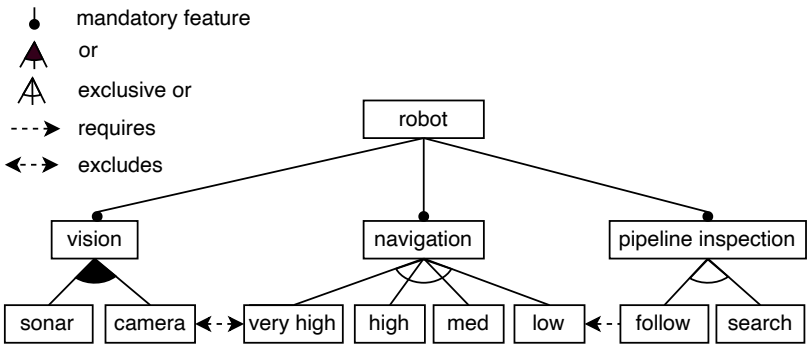


Figure 4: Feature model of the small-scale evaluation

again, e.g., when the pipeline is partly covered by sand or the AUV's thrusters failed for some time causing the AUV to drift off its path. In this case, the AUV has to search for the pipeline again, enabling all four altitudes.

### *3.2. Two-layered View of the AUV as a Family-based Model*

Considering the AUV as a two-layered SAS, the AUV's managed subsystem is responsible for the following tasks: (1) the search for the pipeline and (2) the inspection of the pipeline. Depending on the current task, altitude, and available sensors of the AUV, a different configuration of the managed subsystem must be chosen. Thus, the managed subsystem can be seen as a family of systems where each family member corresponds to a valid configuration of the AUV, i.e., it can be seen as an SPL as discussed in Section 2. To do so, the different altitudes for navigation (*low*, *med*, *high*, and *very high*), the tasks *search* and *follow*, and the vision sensors *camera* and *sonar* can be seen as *features* of the managed subsystem that adhere to the *feature model* in Figure 4, which models the dependencies and constraints among the features [5, 79]. Each configuration of the AUV contains exactly one feature for navigation and one for pipeline inspection, and feature *follow* requires feature *low*. The configuration also includes one or no feature for vision, but the feature *camera* excludes the feature *very high*. This yields nine different configurations of the managed subsystem of the AUV, i.e., nine different products of the SPL.

The managing subsystem of the AUV switches between these configurations during runtime by activating and deactivating the subfeatures of *navigation*, *pipeline inspection* and *vision*, while the resulting feature configuration has to adhere to the feature model in Figure 4, i.e., a valid product of the SPL has to be chosen. The managing subsystem chooses the feature *sonar* when searching for the pipeline if the sonar did not fail. If the sonar fails, it chooses the feature *camera* also for searching for the pipeline. When using the camera for the search, the features *low*, *med* and *high* are activated and deactivated according to the current water visibility. If the water visibility is good, the features *low*, *med*, and *high* can be activated; if the water visibility is average, *high* cannot be activated; and if the water visibility is poor, only *low* can be activated. The managing subsystem switches from the feature *search* to *follow* if the pipeline was found, and from *follow* to *search* if the pipeline

was lost. When the AUV is following the pipeline, the managing subsystem prefers to choose the feature *camera* for inspecting it if the camera did not fail and is not blocked. If the camera fails during or before the inspection or is blocked during the inspection, the managing subsystem chooses the feature *sonar* for inspecting the pipeline. It deactivates both vision subfeatures if both the camera and the sonar failed, resulting in a configuration without an active vision subfeature and in the need to abort the mission.

Thus, we do not explicitly model the MAPE-K loop discussed in Section 2.1 but abstract from it. This is because the MAPE-K loop can be used for realising a managing subsystem, but it does not impact the analysis we want to conduct with the model. In this paper, the analysis focuses on the interaction between the managed and managing subsystem, independent of the realisation of the managing subsystem.

### *3.3. Separation of Concerns between Managed and Managing Subsystem*

As described before, the managed subsystem of the AUV consists of several different hardware and software components that can be combined in predefined ways to form a valid configuration of the AUV, modelled by the feature model in Figure 4. The feature model specifies all possible valid configurations of the managed subsystem. However, the managed subsystem does not know whether these configurations make sense in certain scenarios, such knowledge is captured by a separate subsystem that implements how to adapt the AUV. The managing subsystem implements such an adaptation logic. Thus, it chooses configurations of the managed subsystem depending on the current environmental and internal conditions. Depending on the non-functional requirements of the system, a different managing subsystem can be implemented, e.g., minimising the energy consumption or the time taken for the mission. Therefore, a managing subsystem might not choose certain configurations of the managed subsystem that are possible since they are not useful for the strategy of the managing subsystem. Thus, there is a separation of concerns between the managed and the managing subsystem.

In this small-scale evaluation, the configurations using the sonar for the search at a high, medium or low altitude are permitted by the managed subsystem because they do not cause problems. However, the managing subsystem implemented here does not use these configurations because it prefers searching for the pipeline with the sonar at a very high altitude. Nonetheless, another managing subsystem, implementing a different strategy, might choose them.

## 4. Modelling the AUV Small-Scale Evaluation with ProFeat

In this section, we describe the behavioural models of the managed and managing subsystems, of the environment, and the hardware failures, i.e., we describe the behaviour of how the camera and the sonar may run into hardware and software failures and of how the camera may get blocked. Furthermore, we model the small-scale evaluation with the family-based model checker ProFeat.

ProFeat[2][3] [37] provides a means to both specify probabilistic system families and perform family-based (quantitative) analysis on them. Thus, different from classical FTSs, it also caters for probabilistic models as well as costs and rewards for quantitative analysis. ProFeat extends the probabilistic model checker PRISM[4] [72] with functionalities such as family models, features and feature switches. Thereby, it enables family-based modelling and (quantitative) analysis of probabilistic systems in which feature configurations may dynamically change during runtime. Thus, ProFeat fits well for modelling DSPLs as considered in this paper. The whole model can be analysed with probabilistic family-based model checking using PRISM. The probabilities used in our model are estimates and have not been validated by experiments, since in this paper our goal is not to make a model that is as realistic as possible, but rather to show the feasibility of our methodology.

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, or more generally a DSPL, 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 and other parts of the system like, e.g., the behaviour of the hardware, e.g., failures, can be specified as separate modules that interact with the managed and managing subsystem. Thus, ProFeat is well-suited to model and analyse the small-scale evaluation described in Section 3.

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.

The pipeline inspection small-scale evaluation was modelled as a Markov decision process in ProFeat[5]. The ProFeat model consists of (1) the implementation of the feature model of Figure 4, which is explained in Section 4.1; (2) modules describing the behaviour of the managed subsystem of the AUV, called *AUV module* (see Figure 6), that of the environment (see Figure 7) and that of the hardware failures (see Table 1), which are explained in Sections 4.2, 4.3, and 4.4, respectively; and (3) the feature controller that switches between features during runtime, corresponding to the managing subsystem of the AUV (see Figure 8), whose behaviour is explained in Section 4.5.

An overview of the different components, and how they fit into the two-layered view of an SAS as presented in Figure 1, is shown in Figure 5. The managing subsystem is modelled by the feature controller; the managed subsystem is modelled by the feature model, the AUV module, and the hardware failure module; and the environment is modelled by the environment module. The feature controller monitors the environment by receiving information about the current water visibility, and it monitors the managed subsystem by receiving information about sensor failures (which sensors failed or are blocked) from the

[2] https://pchrszon.github.io/profeat

[3] https://github.com/pchrszon/profeat

[4] https://www.prismmodelchecker.org/manual

[5] The complete ProFeat model of the small-scale evaluation is publicly available [86].

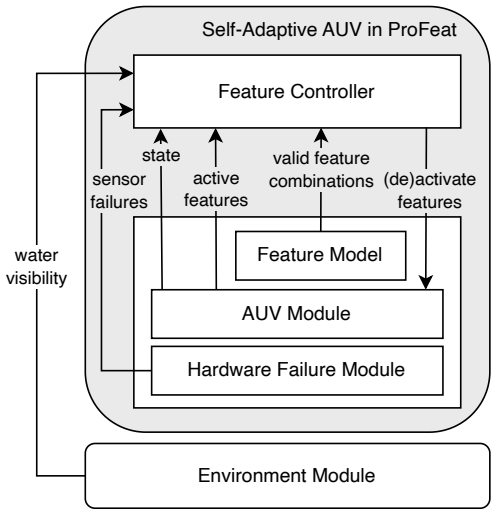


Figure 5: The two-layered view (cf. Figure 1) of the self-adaptive AUV modelled in ProFeat, including the components of the ProFeat model and how information concerning features, environmental conditions, failures and states flows between them

hardware failure module; information about the state of the AUV and the active features from the AUV module; and information about valid feature combinations from the feature model. Using this information, it decides non-deterministically which features to activate and deactivate and updates the information about active and inactive features in the AUV module, corresponding to an adaptation of the managed subsystem. This updated information determines which transition the AUV module can take. In our model, there is no direct information flow between the environment and the managed subsystem.

Starting from Section 4.2, the subsequent sections are all structured in the same way: first, we describe the behaviour of the module and, second, we describe its ProFeat implementation. Therefore, a reader only interested in the high-level behaviour can easily navigate the sections without reading the implementation details.

### *4.1. The Feature Model*

This section explains how the feature model of the small-scale evaluation is expressed in ProFeat, including connections and constraints among features. Each feature is specified within a **feature** ... **endfeature** block, whereas the declaration of the root feature is done in a (unique) **root feature** ... **endfeature** block.

*The Root Feature.* An excerpt of the implementation of the root feature of the pipeline inspection small-scale evaluation according to Figure 4 is displayed in Listing 1. The root feature can be decomposed into subfeatures; in this case 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, or [m..n] if between m and n features have to be included, see Line 7 of Listing 2. It

```
1  root feature
2    all of robot;
3    modules auv, environment, hardware;
4    rewards "time"
5      [step] true : 1;
6    endrewards
7    rewards "energy"
8      // Costs for being in a recovery state
9      (s=recover_very_high) : 2;
10     // .. omitted code ..
11
12     // Costs for switching altitudes
13     (s=search_very_high) & active(high) : 2;
14     (s=search_very_high) & active(med) : 4;
15     (s=search_very_high) & active(low) : 6;
16     // .. omitted code ..
17
18     // Costs for going to low altitude when the pipeline is found
19     (s=found) & active(very_high) : 6;
20     (s=found) & active(high) : 4;
21     (s=found) & active(med) : 2;
22     // .. omitted code ..
23
24     // Additional costs for sonar because it takes more energy than camera
25     active(sonar) : 3;
26   endrewards
27 endfeature
```

Listing 1: An excerpt of the declaration of the root feature of the small-scale evaluation

is possible to specify the behaviour of a (set of) features in so-called *modules*. The modules implementing a feature's behaviour can be specified following the keyword **modules**. In this small-scale evaluation, the root feature is the only feature specifying modules, thus the behaviour of all features is modelled in the modules auv, environment and hardware described later. Using separate modules enables to introduce a separation of concerns between the different parts of the model.

Unlike common feature models, ProFeat allows to specify feature-specific rewards in the declaration of a feature. Rewards are real values that can be attached to transitions and states to analyse quantitative properties of the system, e.g., the expected energy consumption or mission time. Even though they are called "rewards" in ProFeat and PRISM, they can also be interpreted as costs, i.e., they can be used both to motivate and to penalise going to a state or taking a transition. Each reward is encapsulated in a **rewards** ... **endrewards** block. In the small-scale evaluation, we consider the rewards *time* and *energy*, see Lines 4–26 of Listing 1. During each transition that 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 reward energy is a state-based reward and can be used to estimate the necessary battery level for a mission completion. If a thruster of the AUV fails and needs to be recovered, a reward of 2 is given, see Line 9. The model also reflects that 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

```
1 feature navigation
2   one of low, med, high, very_high;
3   initial constraint active(very_high);
4 endfeature
5
6 feature vision
7   [0..1] of sonar, camera;
8   initial constraint active(sonar);
9 endfeature
```

Listing 2: The declaration of the navigation and vision features of the small-scale evaluation

altitude that is active, a higher energy reward is given in these states. If the AUV needs to switch between low and very high altitudes, such as, e.g., in Line 15, an energy reward of 6 is given; if it needs to switch between very high and medium or between high and low altitude, such as, e.g., in Line 14, an energy reward of 4 is given; while all other altitude switches receive a reward of 2, see, e.g., Line 13. Since the altitude must be changed to *low* once the pipeline is found, these cases also receive an energy reward as explained above, see Lines 19–21. All other states, apart from the state in which the mission is aborted, receive an energy reward of 1. Since the sonar uses more energy than the camera, an additional energy reward of 3 is given if the sonar is used, see Line 25. 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. Note that both time and energy rewards are interpreted as costs.

*Ordinary Features.* 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 4. To have only one initial state, we initialise the model with the features search, very_high and sonar active, using the keyword initial constraint, see for example Lines 3 and 8 of Listing 2. As examples of the implementation of other features, the declarations of the features navigation and vision are given in Listing 2.

### *4.2. The Managed Subsystem*

*The Behavioural Model of the Managed Subsystem.* The behaviour of the managed subsystem of the AUV can be described by a probabilistic transition system equipped with features that guard transitions (a probabilistic FTS). Only if the feature guarding a transition is included in the current configuration of the managed subsystem of the AUV, the transition can be taken. This transition system adheres to the feature model in Figure 4 and is depicted in Figure 6, where some details have been omitted to avoid cluttering (in particular, all probabilities and some guards such as camera or sonar). The details can be obtained from the publicly available model [86]. The probabilistic model allows to easily model the possibilities of, e.g., finding and losing the pipeline depending on the system configuration.

The transition system can roughly be divided into two parts, one concerning the search for and one the following of the pipeline, as shown by the grey boxes in Figure 6. When the AUV starts its mission, i.e., in state *start task*, the AUV can either immediately start

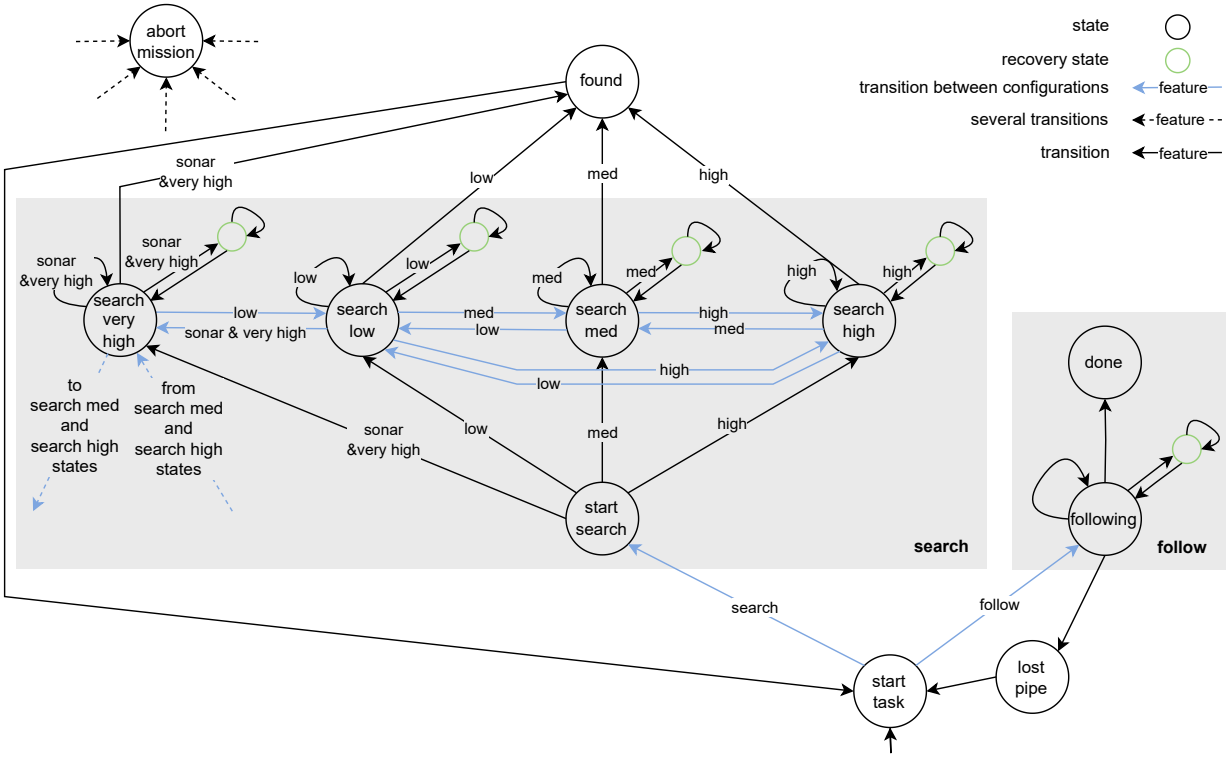


Figure 6: The managed subsystem of the AUV, focusing on the tasks and the search altitude

following the pipeline if it was deployed right above it, or start searching for it. During the search for the pipeline, i.e., when the AUV is in the grey area labelled *search*, the feature *search* should be active and remain active until the state *found* is reached. The managing subsystem can switch between the features *low*, *med* *high* and *very high* during every transition, but the state *search very high* can only be reached when the feature *sonar* is active, not with the feature *camera*. Note that this probabilistic FTS does not reflect that the altitude should only be changed according to the water visibility if the feature *camera* is active, as described in Section 3, since this logic is encoded in the managing subsystem, see Section 4.5.

Once the pipeline is found, the managing subsystem has to deactivate the feature *search* and activate the feature *follow*, which also implies activating the feature *low* and deactivating *med*, *high* and *very high* due to the feature constraints in Figure 4. We assume that the managing subsystem activates and deactivates features during transitions, so the features *follow* and *low* should be activated during the transition from the state *found* to the state *start task*. When the AUV is following the pipeline, i.e., in the grey area labelled *follow*, it can lose the pipeline again, e.g., because of sand covering it or because it drifted off its path due to thruster failures. Then the managing subsystem has to activate the feature *search* during the transition from *lost pipe* to *start task*.

Observe that the feature model does not require to have a vision feature active, see Line 7 of Listing 2, capturing that both vision sensors can have a hardware or software failure and

can no longer be used. If both vision sensors fail, then the pipeline inspection cannot be accomplished and the managed subsystem goes to the state *abort mission*, which can be reached from every state of the probabilistic FTS.

Additionally, observe that apart from the state *search very high* and its recovery state, which can only be reached when the feature *sonar* is active, the probabilistic FTS does not reflect which vision feature is currently active. It would also be possible to duplicate the search states to, e.g., have a *search low camera* and a *search low sonar* state, but this would increase the state space considerably. Since we are not interested in analysing properties associated with these different states, we opt for a probabilistic FTS with no explicit representation of the vision features.

In the model, we distinguish between two kinds of transitions: transitions that model the behaviour of a certain configuration of the managed subsystem (black transitions) and (featured) transitions that switch between configurations, enabled by the managing subsystem during runtime (blue transitions). The labels *search*, *follow*, *low*, *med*, *high*, *very high*, *sonar* and implicitly *camera* on the transitions represent the features that have to be active to execute the respective transition. The transitions between configurations (blue) implicitly carry the action to start the task or go to the altitude specified by the feature associated with the transition. For instance, the transitions from *search low* to *search medium* can be taken if the feature *med* is active because the transition has the guard *med*. When taking this transition, the AUV should perform the action of going to a medium altitude. The transitions inside a configuration (black) with a feature label contain the implicit action to stay at the current altitude because the navigation subfeature has not been changed during the previous transition. However, as described above, the blue transitions do not represent all transitions between configurations. If the configuration is changed from the *camera* to the *sonar* or the other way around, this is not reflected in the probabilistic FTS in Figure 6.

Whether a transition inside the configuration or between configurations is executed in the search states *search low*, *search medium*, *search high* and *search very high* depends on the managing subsystem, i.e., the controller switching between features (see Section 4.5). If the managing subsystem switched between the features *low*, *med*, *high* and *very high* during the last transition, a transition to the search state corresponding to the new feature will be executed, i.e., the configuration will be changed (for switching to the *search very high* state, also the feature *sonar* has to be active). Otherwise, a transition inside the configuration will be executed. For instance, consider the state *search low*. If the feature *low* is active, a black transition will be executed. If, however, the managing subsystem deactivated the feature *low* during the last transition and activated *med*, *high* or *very high*, then the AUV will perform a transition to the state *search medium*, *search high* or *search very high*, respectively.

*The Implementation of the Managed Subsystem in ProFeat.* The module auv models the behaviour of the managed subsystem of the AUV as displayed in Figure 6, see Listing 3 for an excerpt of the model. As depicted in Figure 6, there are sixteen enumerated states in the ProFeat module with names that correspond to the state labels in the figure. The recovery 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 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 the AUV has already inspected the pipeline, it is initialised by 0. The variable inspect represents the desired inspection length and 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 in Line 4 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.

The behaviour of the module is specified with *guarded commands*, corresponding to possible, probabilistic transitions 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 guards by, e.g., enabling the use of the function active. 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.

For instance, consider the command in Lines 14–17, which can be read as follows. If the system is in state search_high, the feature high and at least one of the features sonar and camera 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 6 exiting from state *search high*. This command also has an action label, step. Using this action label, the managed subsystem synchronises with the environment and hardware failures modules, and with the feature controller, as described later.

The blue transitions exiting from the state *search high* in Figure 6 are modelled in Lines 18–23, where the transition to state search_very_high is only represented implicitly in Figure 6. If the model is in state search_high, but the feature low, med or very_high 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_very_high, search_med and search_low are modelled similarly. However, the probability of going to the state found is highest from state search_very_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 states search_high and search_very_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 27.

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 7–9.

From the following state, the transitions that can be taken depend on the variables d_insp and t_failed. Lines 30–34 consider the case where the distance of the pipeline that

```
module auv
  s : [0..15] init start_task;
  d_insp : [0..inspect] init 0;
  t_failed : [0..infl_tf] init 0;

  // To the correct task
  [step] (s=start_task & active(search) & (active(camera) | active(sonar)))
         -> 1: (s'=start_search);
  [step] (s=start_task & active(follow) & (active(camera) | active(sonar)))
         -> 1: (s'=following);

  // .. omitted code ..
  // From search state to another state
  [step] (s=search_high & active(high) & (active(camera) | active(sonar)))
         -> 0.59:(s'=found)
         + 0.4:(s'=search_high)
         + 0.01:(s'=recover_high);
  [step] (s=search_high & active(very_high) & active(sonar))
         -> 1:(s'=search_very_high);
  [step] (s=search_high & active(med) & (active(camera) | active(sonar)))
         -> 1:(s'=search_med);
  [step] (s=search_high & active(low) & (active(camera) | active(sonar)))
         -> 1:(s'=search_low);
  // .. omitted code ..

  // Go to other task if pipeline is found
  [step] (s=found) & (active(camera) | active(sonar)) -> 1:(s'=start_task);

  // Following the pipeline
  [step] (s=following) & (d_insp<inspect) & (t_failed=0) & (active(camera) | active(sonar))
         -> 0.92: (s'=following) & (d_insp'=d_insp+1)
         + 0.05: (s'=lost_pipe)
         + 0.03:(s'=recover_following)
             & (t_failed'=(t_failed<infl_tf? t_failed+1 : t_failed));
  [step] (s=following) & (d_insp<inspect) & (t_failed>0) & (active(camera) | active(sonar))
         -> 0.92*(1-t_failed/infl_tf): (s'=following)
             & (d_insp'=d_insp+1) & (t_failed'=t_failed-1)
         + 0.05*(1+((0.92*t_failed)/(0.05*infl_tf))): (s'=lost_pipe)
         + 0.03:(s'=recover_following)
             & (t_failed'=(t_failed<infl_tf? t_failed+1 : t_failed));
  [step] (s=following) & (d_insp=inspect) & (active(camera) | active(sonar)) -> (s'=done);

  // Lost the pipeline
  [step] (s=lost_pipe) & (active(camera) | active(sonar))
         -> 1: (s'=start_task) & (t_failed'=0);

  // Recovery states
  [step] (s=recover_high) & (active(camera) | active(sonar))
         -> 0.5:(s'=recover_high) + 0.5:(s'=search_high);
  // .. omitted code ..

  // Abort mission if both sensors failed (if the mission has not been finished yet)
  [step] (s!=done) & (s!=abort_mission) & (camera_failed & sonar_failed)
         -> 1: (s'=abort_mission);

  // Wait that the camera gets unblocked if it is blocked but didn't fail, sonar failed
  [step] (s!=done) & (s!=abort_mission)
            & !active(sonar) & !active(camera) & !camera_failed -> 1: true;
  // .. omitted code ..
endmodule
```

Listing 3: An excerpt of the ProFeat AUV module of the small-scale evaluation

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 35–40 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 of 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 41) and finishes the pipeline inspection. If the AUV lost the pipeline (see Line 44), 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 Lines 48–49.

Note that all transitions have a guard active(camera) | active(sonar) (apart from the transitions to the search_very_high state which have a guard active(sonar)). Thus, these transitions can only be taken if one of the two features is active, i.e., they cannot be taken if both the sonar and camera failed or if the sonar failed and the camera is blocked. If both vision sensors failed, indicated by the variables camera_failed and sensor_failed, then the pipeline inspection has to be aborted, modelled in Lines 53–54. Note that the check camera_failed & sonar_failed cannot be replaced by !active(camera) & !active(sonar) because the feature camera is also inactive if the camera is blocked, which is not a permanent fault but can be unblocked. Thus, if both features camera and sonar are inactive but the camera did not fail, then the managed subsystem does nothing and waits for the camera to get unblocked, see Lines 57–58.

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.

### *4.3. The Environment*

*The Behavioural Model of the Environment.* The only parameter of the environment considered in this small-scale evaluation is the water visibility, which influences at which altitude the AUV can operate when using the camera as a vision sensor. We assume that there is a minimum and maximum visibility of the environment, depending on where the AUV is deployed and set by the user during design time. Furthermore, different environments also have different probabilities of currents that influence the water visibility. This can also be set during design time. The behaviour of the environment is then modelled as depicted in Figure 7, where *cp* represents the *current probability*. With the probability *cp* of currents, the water visibility decreases by 1, while it stays the same or increases by 1 with probability $(1\text{-}cp)/2$. If the water visibility is already at minimum visibility, the water visibility stays the same with probability $(1+cp)/2$ and, at maximum visibility, it stays the same with probability $(1\text{-}cp)$.

*The Implementation of the Environment in ProFeat.* The environment is modelled in a separate environment module, see Listing 4. The variable water_visib in Line 2 reflects the

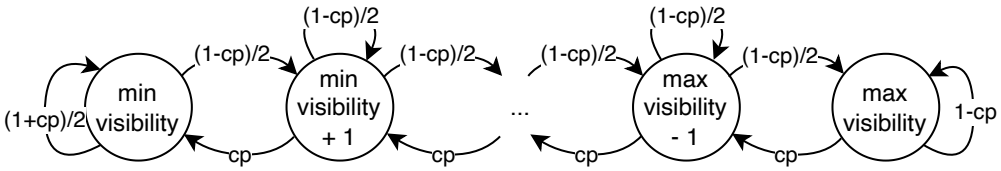


Figure 7: The behaviour of the environment ([88])

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

Listing 4: The ProFeat environment module of the small-scale evaluation

current water visibility and is initialised parametrically, depending on the minimum and maximum visibility, see Line 2. The function round() is pre-implemented in the PRISM language and rounds to the nearest integer. The environment module synchronises with the other modules and the feature controller via the label of its command, step. Since the guard of the only command in the environment module is true, the environment executes a transition every time the other modules and the feature controller do. 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.

### *4.4. The Hardware Failures*

*The Behavioural Model of the Hardware Failures.* The hardware failures module models how the vision sensors, the camera and the sonar, can fail and get blocked (in case of the camera) during runtime, causing the need for switching between vision sensors or aborting the mission. In our model, we assume to know probabilities for the failure of the sonar and the camera that could be provided by the sensor manufacturers in practice. The probability of the camera getting blocked can be determined depending on the environment the AUV is deployed in, e.g., depending on the amount of natural and human waste, while the probability of the camera getting unblocked can be determined depending on the probability of currents as the currents might take away the blockage of the camera.

Every time step, the camera and sonar can fail and the camera can get blocked or unblocked (if it was blocked). In fact, between none and all of these things can happen simultaneously. Table 1 shows the states and transitions of the hardware failures module. It is possible that all sensors are working ($aw$), that the camera or sonar failed ($cf$ and $sf$, respectively), that the camera is blocked ($cb$), or a combination of the last three. The table contains a 1 if the respective transition exists and a 0 otherwise. For example, there exists a transition from sonar failed and camera blocked ($sf$, $cb$) to sonar failed ($sf$), namely, if the camera gets unblocked, but no transition to camera blocked ($cb$) because the sonar cannot recover once it failed.

| from \ to | aw | cf | sf | cb | cf sf | cf cb | sf cb | cf sf cb |
|---|---|---|---|---|---|---|---|---|
| aw | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| cf | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| sf | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| cb | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| cf, sf | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| cf, cb | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| sf, cb | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| cf, sf, cb | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |

Table 1: Transition matrix, *aw* stands for "all working", *cf* for "camera failure", *sf* for "sonar failure", and *cb* for "camera blocked"; a 1 indicates that there is a transition, a 0 that there is none

Note that the thruster failures are not modelled in this module since they are embedded in the managed subsystem. This is because we only consider internal self-adaptation for thruster failures, i.e., exception handling; if a thruster failure occurs, the system goes to the corresponding recovery state and tries to restart the thruster. On the other hand, adapting to failed sensors (or changes in the water visibility) is realised with external self-adaptation and is thus not included in the managed subsystem but in separate modules to create a separation of concerns between the behaviour of the managed subsystem and the uncertainties it is subject to. This also makes it easier to include more uncertainties, requiring external adaptation, to the small-scale evaluation.

*The Implementation of the Hardware Failures in ProFeat.* The hardware failures are modelled in a separate hardware module, see Listing 5. The Boolean variables sonar_failed, camera_failed and camera_blocked in Lines 2–4 indicate whether the sonar failed, the camera failed or the camera is blocked, respectively. They are initialised to be false.

If both the sonar and the camera did not fail and the camera is not blocked, one of the sensors can fail or get blocked (see Lines 8–13), two sensors can fail or get blocked at the same time (see Lines 16–25) or both sensors fail and the camera gets blocked at the same time (see Lines 28–31). These commands reflect the transitions in the first line of Table 1. Similar commands exist for the other transitions indicated in the table. Like the environment module, the hardware failures module synchronises with the other modules and the feature controller via the label of its commands, step.

### *4.5. The Managing Subsystem*

*The Behavioural Model of the Managing Subsystem.* As described in Section 3, the managing subsystem of the AUV implements the AUV's adaptation logic, which corresponds to activating and deactivating the features of the managed subsystem. The behaviour of the managing subsystem of the AUV is depicted in Figure 8. As in the figure of the managed subsystem (Figure 6), the transition system is divided into two parts, one part for *searching* for the pipeline and one for *following* it, indicated by the grey boxes. For better readability, both parts are again divided into a part for using the *camera* and one for using the *sonar*,

```
module hardware
  sonar_failed : bool init false;
  camera_failed : bool init false;
  camera_blocked : bool init false;

  /////// Everything is working
  // One of the sensors fails/gets blocked
  [step] (!sonar_failed & !camera_failed & !camera_blocked)
         -> sonar_fail_prob: (sonar_failed' = true) + (1-sonar_fail_prob): true;
  [step] (!sonar_failed & !camera_failed & !camera_blocked)
         -> camera_fail_prob: (camera_failed' = true) + (1-camera_fail_prob): true;
  [step] (!sonar_failed & !camera_failed & !camera_blocked)
         -> camera_block_prob: (camera_blocked' = true) + (1-camera_block_prob): true;

  // Two sensors fail/get blocked at the same time
  [step] (!sonar_failed & !camera_failed & !camera_blocked)
         -> camera_fail_prob*sonar_fail_prob: (camera_failed' = true)
             & (sonar_failed' = true) + (1-camera_fail_prob*sonar_fail_prob): true;
  [step] (!sonar_failed & !camera_blocked & !camera_blocked)
         -> sonar_fail_prob*camera_block_prob: (camera_blocked' = true)
             & (sonar_failed' = true) + (1-sonar_fail_prob*camera_block_prob): true;
  [step] (!camera_failed & !camera_blocked & !camera_blocked)
         -> camera_fail_prob*camera_block_prob: (camera_blocked' = true)
             & (camera_failed' = true)
         + (1-camera_fail_prob*camera_block_prob): true;

  // Everything fails/gets blocked at the same time
  [step] (!camera_failed & !camera_blocked & !sonar_failed)
         -> camera_fail_prob*camera_block_prob*sonar_fail_prob: (camera_blocked' = true)
             & (camera_failed' = true) & (sonar_failed' = true)
         + (1-camera_fail_prob*camera_block_prob*sonar_fail_prob): true;
    // .. omitted code ..
endmodule
```

Listing 5: An excerpt of the ProFeat hardware failures module of the small-scale evaluation

indicated by the black-framed boxes. Each transition contains a guard, written in black, and an action, written in grey separated by a vertical bar. The guards of transitions in our model are Boolean operations over the states of the managed subsystem, checks over active features and values of variables. The action means activating the indicated feature and implicitly deactivating the other subfeatures of the same category, e.g., activating the feature *camera* and deactivating the feature *sonar*. The dashed transitions connect every state of the box/part where they start to the box where they end. For example, the dashed transition between the *search* part and the *camera* box in the *following* part connects the states *search altitude low*, *search altitude medium*, *search altitude high* and *search altitude very high* to the state *follow camera*.

All transitions in the *search* part of the figure contain the guards *s!=abort mission*, *s!=found* and *active(search)*, where *s* refers to the current state of the managed subsystem. Since the sonar is the preferred sensor for searching for the pipeline, see Figure 2, the *camera* part of *search* also contains the guard *sonar failed*, so transitions can only be taken if the sonar failed, and there are only transitions from the *sonar* part of *search* to the *camera* part of *search*. Furthermore, all transitions in the *camera* part of *search* contain the guard *!camera unavailable* because the camera cannot be used if it is unavailable.

When searching with the sonar, the managing subsystem always chooses to activate

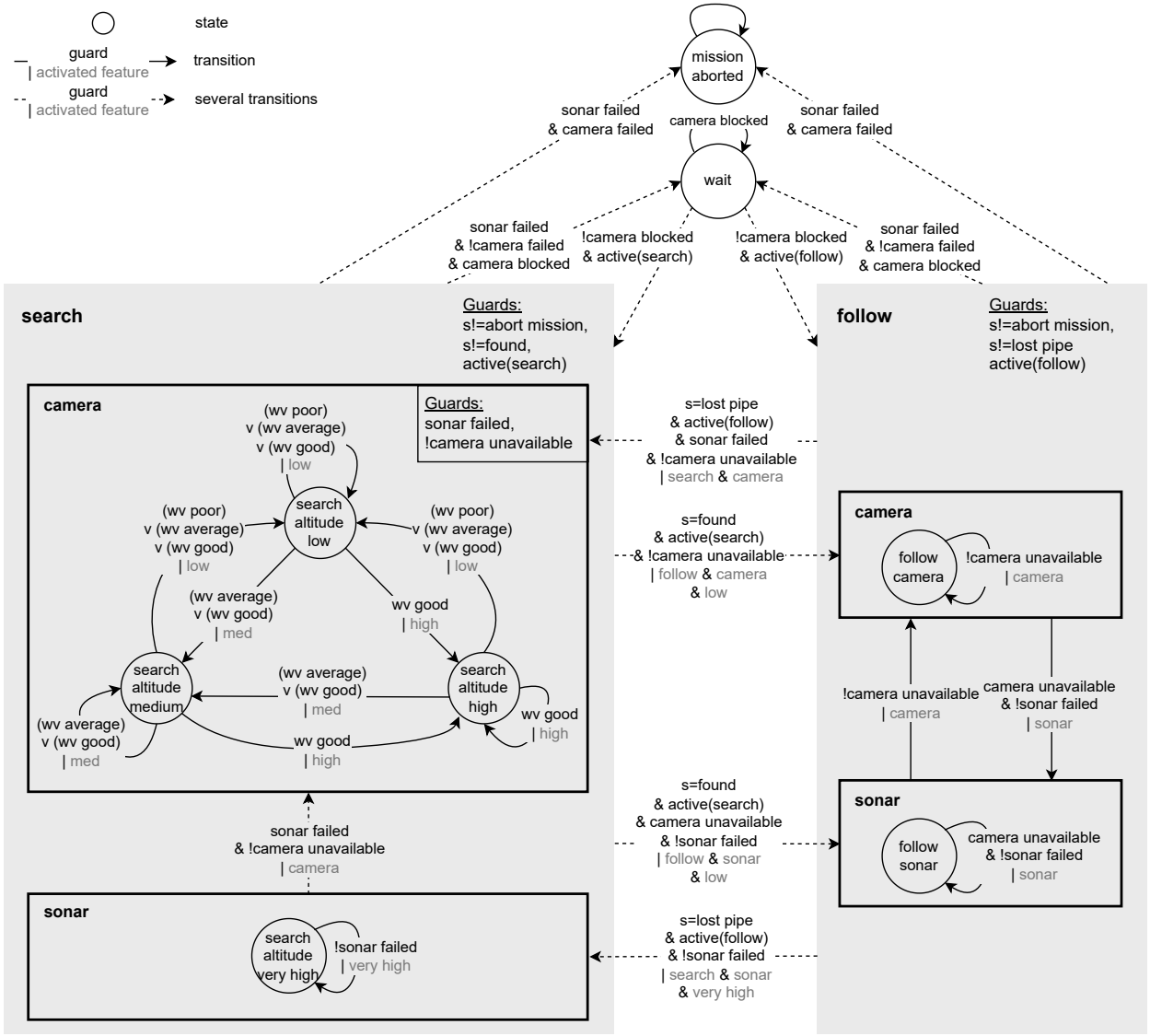


Figure 8: The managing subsystem of the AUV, showing the transitions between feature configurations; the dashed transitions connect every state of the box where the transition starts to every state of the box where the transition ends

the feature *very high*, as indicated in grey on the transition, since this gives the biggest field of view, and thus the highest probability of finding the pipeline. Implicitly, the other subfeatures of *navigation* are deactivated. Since the transition in the *sonar* part contains the guard *!sonar failed*, it can only be taken if the sonar did not fail. If the sonar failed but the camera is available, i.e., it did not fail and is not blocked, a transition to the *camera* part is taken, depending on the current water visibility. When searching with the camera, the managing subsystem activates and deactivates the features *low*, *med*, and *high* according to the current water visibility as described in Section 3. Both when searching with the sonar and the camera, a transition to the state *wait* is taken if the sonar failed and the camera did not fail but is blocked, and a transition to the state *mission aborted* is taken if both the

sonar and the camera failed.

Once the pipeline has been found, i.e., the managed subsystem is in state *found*, one of the transitions between the *search* and *follow* part is taken. These transitions include the action of activating *follow*, *low*, and either *camera* or *sonar* (and deactivating the other features). The camera is preferred for following the pipeline, see Figure 2, so a transition to the *camera* part of *follow* is taken if the camera is available. Otherwise, a transition to the *sonar* part is taken. In the *follow* part, every transition contains the guards *s!=abort mission*, *s!=lost pipe* and *active(follow)*, where *s* again refers to the current state of the managed subsystem. When following the pipeline, the managing subsystem switches to the feature *sonar* if the camera becomes unavailable, and back to the feature *camera* if the camera becomes available again. As in the *search* part, a transition to the state *wait* or *mission aborted* is taken if the respective guards are satisfied.

When the AUV loses the pipeline, i.e., the managed subsystem is in state *lost pipe*, the managing subsystem activates *search* and either *sonar* or *camera*, depending on whether the sonar failed or not. If the sonar did not fail, the managing subsystem also activates the feature *very high* to search at a very high altitude. Otherwise, the AUV will start searching for the pipeline with the camera at a low altitude.

From the state *wait*, the managing subsystem will transition to the *search* or *follow* part, depending on the active feature, if the camera gets unblocked.

*The Implementation of the Managing Subsystem in ProFeat.* The managing subsystem of the AUV is implemented as a feature controller in ProFeat. The feature controller can use *commands* to change the state of the system. Such commands are similar to those used in a module; they are mostly of the form [action] guard −> update. Each command can have an optional label action to synchronise with the modules, and its guard is a predicate of global and local variables of the model and can also contain the function active. In contrast to the commands in the modules, the feature controller can activate and deactivate features in the update of a command. Several features can be activated and deactivated at the same time, but this cannot be done probabilistically and the resulting feature configuration has to adhere to the feature model.

In the pipeline inspection small-scale evaluation, subfeatures of navigation (i.e., the different altitudes at which the AUV can operate), subfeatures of pipeline_inspection (i.e., the tasks the robot has to fulfil), and the subfeatures of vision (i.e., the vision sensors of the AUV) can be switched by the feature controller while the AUV is operating (at runtime), see Listing 6.

When the feature search is active and the pipeline has not been found yet, the feature controller prefers activating the sonar and only activates the camera if the sonar failed. This is achieved by checking if the Boolean sonar_failed, which can be changed probabilistically by the hardware module, is true, see Line 9. If it is false, the features very high and sonar are activated and the other navigation and vision subfeatures are deactivated, see Lines 10–11. Note that it is also possible to activate or deactivate a feature if it is already active or inactive, respectively.

If sonar_failed is true, the feature controller checks if the camera is available, i.e., it did not fail and is not blocked, enabled by the variable camera_unavailable, see Line 3 (in PRISM

```
formula med_visib = (max_visib-min_visib)/3;
formula high_visib = 2*(max_visib-min_visib)/3;
formula camera_unavailable = camera_failed | camera_blocked;

controller
  /////// Searching
  // Prefer searching with the sonar if it is available
  // Sonar should search on a very high altitude
  [step] (s!=abort_mission) & (s!=found) & active(search) & !sonar_failed
         -> activate(very_high) & deactivate(high) & deactivate(med) & deactivate(low)
             & activate(sonar) & deactivate(camera);

  // Use camera if sonar is not available and camera is available
  // Change altitude depending on water visibility (not possible at very high altitude)
  [step] (s!=abort_mission) & (s!=found) & active(search) & sonar_failed
             & !camera_unavailable & med_visib<=water_visib & water_visib<high_visib
         -> activate(low) & deactivate(med) & deactivate(high) & deactivate(very_high)
             & activate(camera) & deactivate(sonar);
  [step] (s!=abort_mission) & (s!=found) & active(search) & sonar_failed
             & !camera_unavailable & med_visib<=water_visib & water_visib<high_visib
         -> activate(med) & deactivate(low) & deactivate(high) & deactivate(very_high)
             & activate(camera) & deactivate(sonar);
  // .. omitted code ..

  /////// Switch task from "search" to "follow"
  // Prefer camera for following if it is available
  [step] (s=found) & active(search) & !camera_unavailable
         -> deactivate(search) & activate(follow) & activate(camera) & deactivate(sonar)
             & activate(low) & deactivate(med) & deactivate(high) & deactivate(very_high);
  [step] (s=found) & active(search) & camera_unavailable & !sonar_failed
         -> deactivate(search) & activate(follow) & activate(sonar) & deactivate(camera)
             & activate(low) & deactivate(med) & deactivate(high) & deactivate(very_high);

  /////// Switch task from "follow" to "search"
  // Prefer sonar for searching if it is available
  [step] (s=lost_pipe) & active(follow) & !sonar_failed
         -> deactivate(follow) & activate(search) & activate(sonar) & deactivate(camera)
             & activate(very_high) & deactivate(high) & deactivate(med) & deactivate(low);
  [step] (s=lost_pipe) & active(follow) & sonar_failed & !camera_unavailable
         -> deactivate(follow) & activate(search) & activate(camera) & deactivate(sonar);

  /////// Follow the pipeline
  // Use the camera for following/inspecting if it is available
  [step] (s!=abort_mission) & (s!=lost_pipe) & active(follow) & !camera_unavailable
         -> activate(camera) & deactivate(sonar);
  [step] (s!=abort_mission) & (s!=lost_pipe) & active(follow)
             & camera_unavailable & !sonar_failed
         -> activate(sonar) & deactivate(camera);

  // Deactivate vision sensors when they are broken
  [step] (s!=abort_mission) & camera_failed & sonar_failed
         -> deactivate(camera) & deactivate(sonar);

  // Wait for camera to get unblocked (sonar failed and camera is blocked but did not fail)
  [step] sonar_failed & camera_blocked & !camera_failed -> 1: true;

  // Avoid deadlock with broken sensors
  [step] (s=abort_mission) -> 1: true;
endcontroller
```

Listing 6: An excerpt of the ProFeat feature controller of the small-scale evaluation

and ProFeat, a *formula* can be used to assign an identifier to an expression). If the camera is available, the feature camera is activated and the feature controller activates and deactivates the altitude features low, med, and high non-deterministically, but according to the current water visibility, as described before. 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; i.e., it influences when the feature controller can switch features (the altitude associated with the feature very high does not depend on the water visibility). To reflect this, the variables med_visib and high_visib are declared as in Lines 1–2. If the water visibility is less than med_visib, the feature controller activates the feature low because the AUV cannot perceive the seabed from a higher altitude. If the water visibility is between med_visib and high_visib, it chooses non-deterministically between low and med (see Lines 15–21), whereas it chooses non-deterministically between all three altitudes if the water visibility is above high_visib.

When the pipeline is found, i.e., the managed subsystem is in state found, the feature controller activates the features follow and low and deactivates the other pipeline inspection and navigation subfeatures since the AUV should follow the pipeline at a low altitude. The camera is the preferred sensor for following the pipeline, so it is activated if it is available, and otherwise the sonar is activated, see Lines 27–32. The same holds when the AUV is following the pipeline, see Lines 44–48.

If the AUV loses the pipeline, i.e., the managed subsystem is in state lost_pipe, the feature controller activates search and deactivates follow to start the search for the pipeline. Since the sonar is the preferred sensor for searching for the pipeline, the feature controller activates the feature sonar if the sonar is available and camera otherwise, see Lines 36–40. Note that the feature controller also activates the feature very_high and deactivates the other navigation subfeatures if the sonar is activated, see Line 38, because the AUV should search for the pipeline at a very high altitude with the sonar. However, if the sonar is not available and the feature controller activates the feature camera, it does not change the navigation subfeature, i.e., the AUV will start searching for the pipeline at a low altitude with the camera.

All commands described so far have guards to ensure that it is not the case that both the sonar and the camera are unavailable. If both of them failed, the features sonar and camera should both be deactivated, see Lines 51–52, leading to the managed subsystem going to state abort_mission. If the sonar failed and the camera is blocked but did not fail, then the feature controller waits for the camera to get unblocked, i.e., it does nothing, see Line 55.

The feature controller synchronises with the auv, environment, and hardware modules via 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 all modules and in the feature controller. Thus, the feature controller needs to include a transition doing nothing if the managed subsystem is in state abort_mission, see Line 58.

## 5. Analysis

ProFeat automatically converts models to PRISM for probabilistic model checking. To analyse a PRISM model, properties can be specified in the PRISM property specification lan-

| Scenario | min_visib | max_visib | current_prob | inspect | c_block_prob |
|---|---|---|---|---|---|
| 1 (North Sea) | 1 | 10 | 0.6 | 10 | 0.05 |
| 2 (Caribbean Sea) | 3 | 20 | 0.3 | 30 | 0.03 |

Table 2: Two different scenarios used for analysis

| | Finish inspection | | Abort mission | |
|---|---|---|---|---|
| **Scenario** | **min** | **max** | **min** | **max** |
| 1 (North Sea) | 0.962 | 1.0 | 0.0 | 0.038 |
| 2 (Caribbean Sea) | 0.747 | 1.0 | 0.0 | 0.253 |

Table 3: The probabilities for finishing the inspection and aborting the mission for both scenarios

guage, which includes several probabilistic temporal logics like PCTL, CSL, and probabilistic LTL. For family-based analysis, ProFeat extends this specification language to include, e.g., the function active. The properties specified with ProFeat are automatically translated into the PRISM property specification language such that the PRISM engine can be used for probabilistic, family-based model checking. When specifying properties in ProFeat, ProFeat constructs, including variables and features, have to be specified in `${...}` to be correctly translated to the PRISM property specification language.

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

The analysis of the model considers four different aspects. First, we analyse properties related to sensor failures in Section 5.1. Second, the rewards energy and time are used to compute some safety guarantees that can be used for the deployment of the AUV, which is detailed in Section 5.2. We analyse safety properties concerning unsafe states in Section 5.3. Lastly, correctness issues of the adaptation logic with respect to its specification described in Section 3 are analysed in Section 5.4. Note that it is not necessary to analyse whether the model satisfies the constraints of the feature model because this is automatically ensured by ProFeat. In addition to the analyses described in this section, complementary complex analyses could be performed, e.g., comparing different implementations of the feature controller. In this paper, we just give a taste of possible analyses to demonstrate the feasibility of our methodology.

*Deployment Scenarios.* We analyse two different scenarios, the values used in these scenarios are reported in Table 2. Scenario 1 is intended to capture a deployment of the AUV in the North Sea, and Scenario 2 in the Caribbean Sea. In Scenario 1, the minimum and maximum water visibility (in 0.5 meter units) are relatively low and the probability of currents that decrease the water visibility is relatively high. In this case, only 10 meters of the pipeline have to be inspected, and the probability of the camera getting blocked is higher than in the Caribbean Sea since we assume that in this scenario, there is more natural and human

```
1 // The minimum and maximum probability that the sonar does not fail until
2 // the pipeline inspection is finished
3 Pmin=? [${!sonar_failed} U ${s=done}];
4 Pmax=? [${!sonar_failed} U ${s=done}];
5
6 // The minimum and maximum probability that the camera does not fail until
7 // the pipeline inspection is finished
8 Pmin=? [${!camera_failed} U ${s=done}];
9 Pmax=? [${!camera_failed} U ${s=done}];
```

Listing 7: Analysis of finishing the inspection without a sensor failure

| | **No sonar failure** | | **No camera failure** | |
|---|---|---|---|---|
| **Scenario** | **min** | **max** | **min** | **max** |
| 1 (North Sea) | 0.809 | 1.0 | 0.899 | 1.0 |
| 2 (Caribbean Sea) | 0.592 | 1.0 | 0.721 | 1.0 |

Table 4: The probabilities for finishing an inspection without a sonar failure or a camera failure

waste in the water than in the other scenario. Scenario 2, in the Caribbean Sea, has a higher minimum and maximum visibility and a lower probability of currents compared to the North Sea. In this scenario, 30 meters of pipeline have to be inspected, and it has a lower probability of the camera getting blocked than in the North Sea.

For both scenarios, we first analyse whether it is always possible to finish the pipeline inspection, i.e., reach the state done, and how likely it is to abort the mission. The results are reported in Table 3. As expected, it holds that the minimum probability of aborting the mission is equal to 1 minus the maximum probability of finishing the inspection, and similarly for the maximum probability of aborting the mission.

### *5.1. Sensor Failures*

Sensor failures can lessen the results the end-user gets from a pipeline inspection. If the sonar fails and the AUV has to search for the pipeline with the camera, then it will probably take more time because the search cannot be done at a very high altitude and it will probably take more energy since the AUV has to change altitudes depending on the water visibility. Thus, since the battery of the AUV is limited, less pipeline can be inspected compared to when both sensors work. If, on the other hand, the camera fails, then the inspection results when using the sonar will not be as good because fewer details can be seen when inspecting with the sonar. For example, detecting tiny holes in the pipeline which could be found when inspecting with the camera because of bubbles rising from the pipeline, cannot be discovered when inspecting with the sonar. Thus, in both cases, the results from the pipeline inspection will probably not be as good as with working sensors.

Therefore, it is interesting to know how likely it is for the camera and sonar to fail before the inspection is finished. The commands in Listing 7 give the minimum and maximum probability of reaching the state done without a sonar failure (Lines 3–4) and without a camera failure (Lines 8–9). The results for Scenarios 1 and 2 are reported in Table 4. It can be seen that the length of pipeline that has to be inspected (a longer pipeline in Scenario 2)

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

Listing 8: Analysis using the rewards

| | Energy | | Time | |
|---|---|---|---|---|
| **Scenario** | **min** | **max** | **min** | **max** |
| 1 (North Sea) | 49.28 | $\infty$ | 22.29 | $\infty$ |
| 2 (Caribbean Sea) | 97.33 | $\infty$ | 53.78 | $\infty$ |

Table 5: Expected minimum/maximum rewards for completing the mission for both scenarios

has an impact on the minimum probability of finishing the inspection without a camera or sonar failure.

### *5.2. Reward Properties*

The rewards time and energy are used to analyse some safety properties related to the execution of the AUV. 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 8 are used to compute the minimum and maximum expected energy (for all resolutions of non-determinism) to complete the mission. Since the model includes two reward structures, the name of the reward has to be specified in {"..."} after the R operator. Similarly, the minimum and maximum expected time to complete the mission is analysed to give the system operators an estimate of how long the mission might last. The results for Scenarios 1 and 2 are reported in Table 5. It can be seen that the variation of the parameters in the two scenarios strongly influences the expected energy consumption and time duration of the mission, leading to an almost doubled minimum energy consumption and a more than doubled minimum time duration for Scenario 2. The maximum rewards for completing the mission are infinite. In PRISM, if a reachability reward is computed and "the probability of satisfying the formula is less than 1, the expected reward is defined to be infinite"[6]. Thus, the maximum rewards are infinite because the minimum probability for completing the pipeline inspection is less than 1, see Table 3, i.e., there are cases in which the pipeline inspection cannot be completed. Note that this is different from the results reported in previous work [88] because the mission can be aborted in the extended small-scale evaluation considered here.

### *5.3. Unsafe States*

Thruster failures, although we assume that they can be repaired, pose a threat to the AUV. Unforeseen events like strong currents might cause the AUV to be damaged, e.g., by causing it to crash into a rock. To analyse this, the states are grouped into *safe*, *unsafe* and *thruster failure* states, where the unsafe states contain all thruster failure states and the *abort mission* state, and all other states are *safe*. The grouping of states is achieved by using

[6] https://www.prismmodelchecker.org/manual/PropertySpecification/Reward-basedProperties

```
1  label "unsafe" = s=recover_very_high | s=recover_high | s=recover_med | s=recover_low
2                 | s=recover_following | s=abort_mission;
3  label "safe" = s=lost_pipe | s=start_task | s=start_search
4                | s=search_very_high| s=search_high | s=search_med | s=search_low
5                | s=found | s=following | s=done;
6  label "thruster_failure" = s=recover_very_high | s=recover_high | s=recover_med
7                | s=recover_low | s=recover_following;
8  Pmin=? [G "safe"];
9  Pmax=? [F "unsafe"];
10 filter(max, Pmax=? [ F<=k "thruster_failure" ], "safe");
11 filter(avg, Pmax=? [ F<=k "thruster_failure" ], "safe");
```

Listing 9: Analysis of unsafe states

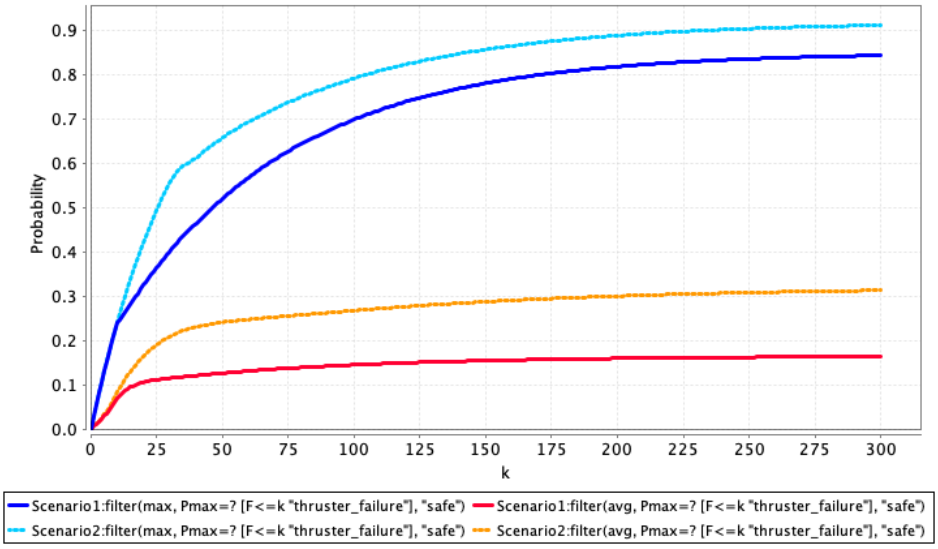


Figure 9: Results for reaching a thruster failure state from a safe state in k time steps

labels, see Lines 1–7 of Listing 9. These labels are then used to calculate the probability of several properties. The minimum probability of only taking safe states (see Line 8) is shown to be 0.686 for Scenario 1 and 0.29 for Scenario 2. As expected, the probability of only taking safe states is higher for a shorter pipeline inspection. Furthermore, the maximum probability of finally reaching an unsafe state (see Line 9), i.e., a state where either a thruster failed or the mission was aborted, is reported to be 0.31 in Scenario 1 and 0.71 in Scenario 2. Again, the fact that the probability is much higher for Scenario 2 probably comes from the fact that the length of pipeline that needs to be inspected is much higher in Scenario 2 than in Scenario 1.

The probability of going to a thruster failure state from a safe state should be as small as possible. This is analysed with the properties in Lines 10–11. First, the maximum probability (over all possible resolutions of non-determinism) for reaching a thruster failure state from a safe state in k time steps is calculated, and then the maximum (or average) over all these probabilities is taken. PRISM experiments allow analysing this property automatically for a specified range of k; the plotted graphs for Scenarios 1 and 2 are displayed in Figure 9. They show that the probability of reaching a thruster failure state from a safe state increases with the number of considered time steps. Furthermore, the maximum

```
1  // The features camera and sonar are deactivated if the respective sensor failed
2  P>=1.0 [G (${camera_failed} => (F G ${!active(camera)}))];
3  P>=1.0 [G (${sonar_failed} => (F G ${!active(sonar)}))];
4
5  // The feature camera is deactivated if the camera is blocked
6  // (unless the camera got unblocked again)
7  P>=1.0 [G (${camera_blocked} => (F (${!active(camera)} | ${!camera_blocked})) )];
8
9  // If the camera is blocked, it will eventually be unblocked
10 P>=1.0 [G (${camera_blocked} => (F ${!camera_blocked}))];
11
12 // When using the camera for searching, the managing subsystem will eventually choose a
13 // correct altitude according to the water visibility
14 // (unless the the water visibility changed again)
15 P>=1.0 [G ((${active(camera)} & ${active(search)} & ${water_visib<med_visib})
16             => (F (${active(low)} | ${water_visib>=med_visib}))))];
17 P>=1.0 [G ((${active(camera)} & ${active(search)}
18             & ${water_visib>=med_visib} & ${water_visib<high_visib})
19             => (F (${active(med)} | ${active(low)} | ${water_visib>=high_visib}))))];
20
21 // The sonar is used for searching for the pipeline if it did not fail
22 P>=1.0 [G ((${active(search)} & ${!sonar_failed})
23             => (F (${active(sonar)} | ${sonar_failed}))))];
24
25 // The camera is used for following the pipeline if it did not fail and is not blocked
26 P>=1.0 [G ((${active(follow)} & ${!camera_unavailable})
27             => (F (${active(camera)} | ${camera_unavailable}))))];
```

Listing 10: Analysis of the correctness of the adaptation logic

probability of reaching a thruster failure state from a safe state stabilises much later and at a higher value than the average probability (for both scenarios). While the maximum probability of reaching a thruster failure state from a safe state stabilises after about 270 time steps at ≈0.84 in Scenario 1 and after about 280 time steps at ≈0.91 in Scenario 2, the average probability stabilises after about 180 time steps at ≈0.16 in Scenario 1 and after about 250 time steps at ≈0.31 in Scenario 2.

### *5.4. Correctness Issues of the Adaptation Logic*

Since adaptation is a crucial part of the small-scale evaluation, it is important to check that (1) the adaptation logic described in Section 3 was correctly implemented, and that (2) the managing subsystem is able to adapt the managed subsystem correctly. Both are analysed with the properties in Listing 10.

First, we want to ensure that, if the camera or sonar failed, the features camera and sonar, respectively, are deactivated. For the camera, this was achieved by checking that it always holds that if the camera failed, i.e., camera_failed is true, eventually the feature camera will always be inactive, see Line 2. By writing P>=1.0 in front of this property, ProFeat will return true, if the property always holds, i.e., if its minimum probability is 1.0, and false otherwise. The same is analysed for the sonar, see Line 3.

Similarly, it is important to ensure that, if the camera is blocked, the feature camera will eventually be inactive or the camera will get unblocked, see Line 7, and that the camera will eventually be unblocked if it was blocked, see Line 10.

Furthermore, the properties in Lines 15–16 and Lines 17–19 ensure that, when searching for the pipeline with the camera, the managing subsystem will eventually choose a correct altitude according to the water visibility. To do so, we ask whether the probability is greater than or equal to 1 that it is always the case that if the features camera and search are active (i.e., the AUV is searching for the pipeline with the camera) and the water visibility is low (i.e., water_visib < med_visib), then eventually the feature low will be active (i.e., the AUV goes to a low altitude), or the water visibility will have increased (i.e., water_visib >= med_visib), see Lines 15–16. A similar property should hold if the water visibility is medium, see Lines 17–19, except that in this case we need to ensure that either the feature low or med will be active. Note that for the case of the water visibility being high, i.e., water_visib >= high_visib, no analysis is needed since all three altitudes can be chosen, depending on the strategy of the managing subsystem.

Lastly, we need to check whether the managing subsystem maintains the priorities defined in Figure 2, i.e., the sonar is used for searching for the pipeline if it did not fail and the camera is used for following the pipeline if it did not fail. In Lines 22–23, this is analysed by checking if the probability is greater than or equal to 1 that it is always the case that, if the feature search is active and the sonar did not fail, then eventually the feature sonar will be active or the sonar will fail. This is analysed similarly for the camera, see Lines 26–27, except that the variable camera_failed is not used, but instead the variable camera_unavailable is used, since the camera should only be used if it did not fail and is not blocked, i.e., is not unavailable.

The analysis showed that the properties described in this subsection are true for both scenarios.

## 6. Evaluation

To evaluate our methodology with the help of the small-scale evaluation detailed in Section 4 and analysed in Section 5, we consider the following research questions (RQs):

**RQ1** How does the proposed methodology of modelling an SAS as a DSPL support extensibility?

**RQ2** To what extent does the proposed methodology enable the analysis of the reliability of SASs?

**RQ3** How can one analyse whether the adaptation logic has been realised according to its specification?

### *6.1. RQ1: How Does the Proposed Methodology of Modelling an SAS As a DSPL Support Extensibility?*

We consider different kinds of model extensions:

1. Adding variability to the managed subsystem;
2. Adding behaviour of the managed subsystem;
3. Adding uncertainties;

4. Extending the adaptation logic (to account for new uncertainties).

All of these extensions have been covered by the extension of the small-scale evaluation in comparison to the model in our previous paper [88], hereafter called the "old model".

*Adding Variability.* Adding variability to the managed subsystem is done by extending the feature model with new features. The variability can be restricted by adding new constraints to the feature model. Both of these changes have been done in the extension presented in this paper by adding four new features (*very high*, *vision*, *sonar* and *camera*) where the camera excludes the feature *very high*, adding a new constraint. Adding these features and constraints to the ProFeat model was straightforward, see Section 4.1.

*Adding Behaviour.* New features are introduced to model new behaviour and new constraints can restrict the already existing behaviour. These behavioural changes have to be included in the behavioural model of the managed subsystem where the behaviour of all valid configurations and the possible re-configurations are modelled. All changes mentioned here have to be done in the model of the behaviour of the managed subsystem (the *auv* module of our small-scale evaluation).

If a new constraint has been added to the previous feature model, this is reflected by including additional guards to some of the transitions of the module modelling the behaviour of the system. If new features (and constraints relating to these features) are included, this can also be reflected in the guards of transitions. For example, in this paper, we added the features *camera* and *sonar* to the feature model of the old model in [88], where either of them has to be active to be able to perform the mission. If none is active, then the mission should be aborted. Thus, all transitions present in the old model need to receive a guard stating that either the camera or the sonar is active (see, e.g., Line 7 of Listing 3).

Furthermore, introducing new behaviour can also require introducing new states and transitions. In our model, we had to include the states *search very high* and its corresponding recovery state, as well as the state *abort mission*, see Figure 6. The new states and the states from the old model had to be connected with transitions whose feature guards may include new features. For example, from each of the search states of the old model, a transition guarded by the features *very high* and *sonar* was added to reflect going to a very high altitude if the features *very high* and *sonar* are active (see, e.g., Lines 18–19 of Listing 3).

Extending our model with new behaviour was straightforward. It mostly required to keep track of the states, transitions, and constraints that have already been added and the ones that still need to be added.

*Adding Uncertainties.* Uncertainties in the managed subsystem can be added by including probabilistic updates in transitions or in the same way as uncertainties were added in the environment, by including a module that models these uncertainties and synchronises with the other modules. In this paper, we added the uncertainty of failures (and blockages) of the camera and sonar. These failures can occur with given probabilities and are modelled in a separate module that only models the sensor failures and blockages but synchronises with the other modules. Thus, uncertainties can be added to the model without changing the already existing model.

*Extending the Adaptation Logic.* When uncertainties or variability in the managed subsystem are introduced, the adaptation logic (modelled in the feature controller) will often have to be updated too. As for adding behaviour to the managed subsystem, this can include an extension of the guards of already existing transitions, restricting when the adaptation rule is applicable, and adding new transitions, corresponding to new adaptation rules. In our model extension, the transitions (adaptation rules) were modified to reflect the priorities for searching with the sonar and inspecting with the camera. This included extending the guards with the variables changed by the hardware failures module that reflect whether the sonar or camera failed or the camera is blocked (see Listing 6). Extending the adaptation logic was straightforward and, similar to adding behaviour to the managed subsystem, it required keeping track of which adaptations have already been implemented.

In conclusion, the proposed methodology supports extensibility of the variability model by adding features to the feature model; it caters for an extension of the behaviour of the managed subsystem by adding new states and transitions; it supports an extension of uncertainties by including new modules that synchronise with the already existing ones; and it provides extensibility of the adaptation logic by including new adaptation rules in the form of guarded commands. All of the mentioned extensions were implemented into the model in a straightforward manner.

### *6.2. RQ2: To What Extent Does the Proposed Methodology Enable the Analysis of SASs?*

In our small-scale evaluation, we consider different kinds of reliability analysis:

1. Analysing how likely it is for a mission to be successful;
2. Analysing sensor failures;
3. Analysing rewards;
4. Analysing unsafe states and thruster failures.

*Successful Mission.* We analysed both best- and worst-case scenarios for finishing the pipeline inspection and aborting the mission (see Table 3). It can be seen that, depending on the chosen scenario, the probability of finishing the inspection and the probability of aborting the mission vary. While the minimum probability of a successful mission is 0.962 in Scenario 1, it is only 0.747 in Scenario 2. Therefore, the chosen scenario can have a big impact on the probability of a successful mission and it can thus be useful to analyse the impact of the scenario on a successful mission beforehand.

*Sensor Failures.* In Section 5.1, we analyse the probability of sonar and camera failures in the different scenarios. Table 4 reports the results of this analysis. It can be seen that the scenario has a significant impact on the probability of achieving a mission without a sensor failure. Even though the maximum probability of finishing the mission without a camera or sonar failure is always 1.0, the minimum probability of finishing without a sensor failure is much lower in Scenario 2 compared to Scenario 1 (for the sonar, 0.592 in Scenario 2 vs. 0.809 in Scenario 1). Furthermore, the probability of a camera failure during the mission is less than the probability of a sonar failure, where the difference varies again with the scenarios.

*Rewards.* In Section 5.2, we analysed the minimum and the maximum expected energy consumption as well as the minimum and the maximum expected mission time for finishing the mission. As in the previously described analyses, the scenario has an impact on the analysis results such that the minimum rewards for both energy and time are approximately doubled in Scenario 2 compared to Scenario 1. Since the probability of finishing the pipeline inspection is not 1.0 (because the mission can be aborted if both the camera and the sonar failed), the maximum expected energy and time are defined to be infinite.

*Unsafe States and Thruster Failures.* By grouping the set of states into different parts, it can be analysed how likely it is to go to these states. In Section 5.3, we analysed how likely it is to eventually reach an unsafe state, i.e., a thruster failure state or the mission aborted state, and how likely it is to only take safe states. As in the previously presented analyses, the scenario has a big impact on the analysis results, making unsafe states in a mission execution more likely in Scenario 2 than in Scenario 1. Furthermore, PRISM experiments could be exploited to show how the probability of going to a thruster failure state increases with the amount of considered time steps.

To conclude, these reliability analyses are enabled by the two-layered modelling approach that provides a separation of concerns between the managed and the managing subsystem, i.e., between the application logic and the adaptation logic. Of course, the analysis presented here is not exhaustive, we just give a taste of the possible analyses. In Section 8, we provide further ideas of analyses possible for the model.

### *6.3. RQ3: How Can One Analyse Whether the Adaptation Logic Has Been Realised According to Its Specification?*

To analyse whether the adaptation logic has been implemented according to its specification, we considered two aspects in our small-scale evaluation (see Section 5.4): we analysed whether changes in the system like sensor failures and changing water visibility triggered the correct adaptations and whether the managing subsystem is able to maintain the desired sensor priorities. Using this analysis, we could determine that the adaptation logic was implemented according to its specification. Depending on the implemented adaptation logic, other properties could be analysed to determine that the adaptation logic was implemented according to its specification, such as, e.g., that an adaptation happens within a certain amount of time.

Again, the two-layered approach enabled this analysis and also makes it straightforward to modify the adaptation logic in case the analysis reveals that it has not been realised according to its specification.

### *6.4. Threats to Validity*

Formal methods research typically ignores the main empirical research strategies as defined in software engineering [103]. A summary of these strategies, specifically adapted for their application in formal methods, including guidelines for selecting the most appropriate research strategy in light of the peculiarity of formal methods research, is reported in [18].

Typical threats to validity of research involving case studies or small-scale evaluations are related to the representativeness of the data and analysis (*external validity*), i.e., to what extent can the results be applied to contexts other than the one of the study (*scope of validity*), the soundness of the design (*internal validity*), including, e.g., the expectations or inclinations of the researcher that may have impacted the design or model (*researcher bias*), and the definition of variables and associated measures (*construct validity*), i.e., to what extent the abstract constructs of interest are well-defined as variables that can effectively be measured quantitatively or evaluated qualitatively. There are many well-known trade-offs between internal and external validity, between the knowledge depth that can be achieved, and concerning the generalisability of the results (e.g., a model may have realistic elements, yet its results are typically hardly applicable to real-world cases).

Considering the small-scale evaluation presented in this paper, there might be other kinds of model extensions that we did not consider and that are more complicated or not possible to realise, limiting the scope of validity. It also remains to be seen how well our methodology scales. One limitation is that the different cases that have to be considered are sometimes hard to track and cases can easily be forgotten. For the adaptation logic, such kind of mistakes can be caught using the analysis presented in Section 5.4. However, catching mistakes in the managed subsystem might require more effort. Further construct validity follows from the fact that more complicated requirements for the adaptation logic might be harder to analyse because they are not expressible in one of the logics supported by PRISM. Finally, since the development of the modelling and analysis methodology as well as its evaluation have been performed by the same group of researchers, there might also be a researcher bias.

## 7. Related Work

A comprehensive literature study of state-of-the-art techniques for testing, validation, and verification of robotic and autonomous systems is given in a recent survey [7] and recently AUV behaviour has been modelled and analysed with timed automata and UPPAAL [91]; however, work using family-based analysis and DSPLs models was not found in this survey, which indicates the novelty of the general approach taken in our work. In this section, complementing the related work already discussed throughout the paper, we provide further references and context and consider more detail of work related to the family-based analysis of SPL models, the modelling of DSPLs, using SPLs for robotics, and the analysis of SASs as DSPLs.

*Family-Based Analysis of SPL Models.* As already discussed in Section 2, the properties of FTSs can be verified with dedicated SPL model-checking tools. Most of these tools accept FTS-like input models, but VMC for instance accepts as input either an FTS through its front-end tool FTS4VMC for static analysis and family-based model checking, or a Modal Transition System (MTS) with a set of logical variability constraints (MTS$\upsilon$), akin to an FTS' feature expressions. MTSs were introduced to capture the refinement of partial descriptions into more detailed ones [74, 71]. MTS$\upsilon$s were introduced to compactly model

product family behaviour, whose individual variant (product) behaviour can be obtained through a special-purpose refinement relation or by an equivalent operational derivation procedure [17]. Such MTSs are equally expressive as FTSs [15]. Also other well-known formalisms have been extended with features for SPL modelling and analysis, including variable (modal) I/O automata [73, 75], feature (Petri) nets [81, 82], and featured team automata [14]. Yet, to the best of our knowledge, none of these can natively deal with DSPLs. While featured modal contract automata [13, 12] can be used to synthesise the dynamic composition and orchestration of SPLs valid products, the associated toolset [11] does not offer family-based model checking nor quantitative analysis.

*Modelling of DSPLs.* Cordy et al. [43] showed how to model DSPLs with Adaptive FTSs, which are FTSs whose set of features is partitioned into (adaptable) system features and environment features, and which implicitly capture the environment by means of macrostates that are triples formed of the system's state and the configurations of both system and environment. No specific tooling is available and failures are modelled as subtypes of environment features, which allows one to describe failure modes and effects but not their probabilities. In this paper, we therefore used ProFeat [37], a software tool built on top of PRISM for the analysis of feature-aware probabilistic models. It provides a guarded-command language to model SPL models of probabilistic systems as well as an automatic translation of SPL models to the input language of PRISM (i.e., featureless models). ProFeat can deal with probabilistic DSPLs by offering dynamic feature switching (i.e., activation and deactivation of features at runtime), as we have seen in this paper, and with feature attributes. Moreover, a separate feature controller, i.e., orthogonal to the behavioural model, is responsible for feature switching. We used this in this paper, as it neatly fits the concept adopted here to split an SAS into a managed and a managing subsystem (see Figure 1). Alternatively, QFLan [19, 108] offers probabilistic simulations to yield statistical approximations, thus trading 100% precision for scalability. However, in QFLan features can be (un)installed or replaced by operational store actions, i.e., as part of the behavioural model, which interact with a declarative store of constraints. Also approaches based on dynamic Delta-Oriented Programming [48, 47] or reconfiguration automata [33, 76] allow the installation of new features as part of (staged) reconfigurations.

Saller et al. [96] proposed a model-based approach for designing context-aware DSPLs, i.e., DSPLs with a feature model enriched during design time with context information that is exploited during runtime. The focus is on mobile devices and resource constraint systems in a non-probabilistic setting. Model-based approaches for engineering supervisory controllers for DSPLs in non-probabilistic settings have been proposed in [13, 12, 104]. Supervisory controllers control (i.e., manage) a system by guaranteeing the maximally permissive behaviour allowed under a given set of constraints.

*Using SPLs for Robotics.* There are several approaches that model, but do not analyse, SASs as (dynamic) SPLs (e.g., [8, 27, 49, 66]). For robotics, Gherardi et al. [60] proposed their toolchain HyperFlex to model robotic systems as SPLs. HyperFlex supports the design and reuse of reference architectures for robotic systems and was extended with the

Robot Perception Specification Language for robotic perception systems by Brugali and Hochgeschwender [32]. It supports the representation of 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 paper, HyperFlex only considers design time variability. Furthermore, it is only used for modelling robotic systems, not for analysing them. Gherardi and Hochgeschwender [61] provide an approach to model both design time and runtime variability of robotic systems based on (D)SPLs. They also use features to model the functionalities of the robot and change features during runtime depending on the environment. However, the approach is focused on achieving adaptation in a real robot and does not consider analysis of the system.

The relevance of SPLs for robotic systems is underlined by Brugali [30]. He argues that most of the costs for robotic systems come from non-reusable software. A robotic system mostly contains software that is tailored to the specific application and embodiment of the robot, and often even software libraries for common robotic functionalities are not reusable. Therefore, they have to be re-developed each and every time. The author thus proposes a new approach for the development of robotic software using SPLs.

*Analysis of SASs as DSPLs.* Chrszon et al. [35, 36] model and analyse configurable systems as role-based systems, an extension of feature-oriented systems, with a focus on feature interaction; in contrast to our paper, a separation between managed and managing subsystem is not considered.

A number of studies [45, 63, 64, 85] address the analysis of SASs specified as DSPLs along a dimension orthogonal to the one considered in our work, namely, they focus on reconfiguration costs and real-time constraints of reconfiguration decisions. Sousa et al. [102] proposed the use of temporal constraints and reconfiguration operations to model the reconfiguration lifecycle of a DSPL. They address modelling the variability of cloud systems and identifying reconfigurations that meet given criteria.

## 8. Discussion and Future Work

The work presented in this paper shows the natural correspondence between an SAS that is implemented using external self-adaptation with a managed and a managing subsystem, and a DSPL in the form of a 150% SPL family model with a controller switching between features during runtime. We exploit this correspondence to analyse a two-level SAS operating in an uncontrolled environment, using family-based analysis techniques.

To showcase how an SAS in an uncontrolled environment can be modelled as an FTS model, we used a feature model together with a probabilistic FTS to model the managed subystem of an AUV used for pipeline inspection, and a controller switching between these features to model the managing subsystem of the AUV. This allowed modelling the managed subsystem of the AUV as a family of systems, where each family member corresponds to a valid feature 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. Furthermore, we showed that

these kinds of models can be analysed by means of probabilistic family-based model checking. In our small-scale evaluation, the tool ProFeat was used for this, analysing reward and safety properties.

ProFeat allowed to model the two different layers of abstraction of an SAS, the managed and managing subsystem, which also makes it easier to understand the model and the adaptation logic. As shown in Section 6, this type of modelling enables extensions of the model in a straightforward manner (see RQ1). Furthermore, it makes analysing all configurations of the managed subsystem more efficient by enabling family-based model checking. It also enabled us to analyse not only whether the adaptation logic was implemented correctly (see RQ3), but also enabled an analysis of important system properties (see RQ2). We are unaware of other work that exploits the family-based modelling and analysis capabilities of ProFeat for SASs, but we believe this is a natural fit.

The family-based analysis proposed in this paper is subject to the usual restrictions of automata-based analyses, and it remains to study the degree to which these can be circumvented. In particular, automata modelling of complex systems is known to suffer from the state-space explosion (i.e., the number of states needed to accurately model a system may exceed the amount of available computer memory). Techniques to mitigate the state-space explosion can be based on compositionality or abstraction. Compositional analysis techniques for FTSs have not been extensively studied so far (but see, e.g., [24, 56]). In our work, it has so far been sufficient to use finite representations of the managed subsystem and of the environment. Techniques to discretise continuous systems exist [4, 62], but, to the best of our knowledge, it is an open problem to what extent these techniques reduce the precision of optimised strategies produced by the family-based analysis.

In addition to the analyses showcased in this paper, the models allow for many other kinds of analysis, some of which we list here. By implementing different environment and hardware failure modules, it can be analysed how the behaviour of the environment and the hardware failures influences the behaviour of the AUV. Furthermore, parametric model checking, where certain variables are left unspecified, can be used to, e.g., analyse in which scenarios the AUV can be deployed in while still ensuring certain safety properties. Lastly, it is possible to use multi-objective analysis to analyse properties where several objectives have to be satisfied, e.g., minimising both time and energy.

It remains to be seen how the modelling and analysis scale with more complex systems. Concerning the modelling, we believe that the compositional approach we took makes it easier to develop models of complex systems compared to a monolithic approach, because different concerns can be modelled in distinct, synchronising modules. This makes both developing and modifying the model easier. One problem for modelling might be that the adaptation logic of a complex system can get very complicated and introducing errors when modelling becomes more likely. Here, the proposed analysis for analysing the adaptation logic can help to find mistakes that were introduced when modelling the adaptation logic. As described above, the analysis of complex systems often suffers from a state-space explosion, and developing a finite representation of the different parts of the system might become a challenge. It would be interesting to investigate if the proposed techniques for mitigating these risks can be applied to our methodology.

The small-scale evaluation in this paper is of course a highly simplified model of an AUV and its mission. However, we showed 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. There are many ways of extending the model. First, more functionalities and variability points of the AUV as well as new tasks can be incorporated in the model by including more features in the feature model. This also necessitates adapting the model of the managed subsystem to include the new behaviour, as well as adapting the model of the managing subsystem to include the new variability points in the adaptation logic. Furthermore, including new uncertainties that lead to adaptation needs (like the environment and the hardware failures in this paper) can be done by including new modules that model the uncertainties and including them in the adaptation logic of the managing subsystem. It would also be possible to consider a swarm of AUVs working together by leveraging ProFeat's functionality of module parametrisation. However, of the extensions mentioned here, this is probably the most difficult one and likely requires remodelling of most parts of the small-scale evaluation. Furthermore, the model of the environment could easily be exchanged with a more realistic one. All these changes would require some modelling effort, however, there should not be challenges concerning the analysis of the extended models. To analyse a real AUV, the models of the AUV and of the environment, and in particular the probabilities, have to be adapted to the robot and the environment with the help of real data and domain experts. We plan to investigate this together with an industrial partner of the MSCA network REMARO (Reliable AI for Marine Robotics).

In the future, we plan to find optimal strategies for the managing subsystem, i.e., the controller switching between features, e.g., to minimise energy consumption. We would also like to find patterns between choosing a certain feature configuration and the effect of this on quality criteria of the system. Finding such control patterns could help to improve the adaptation logic of the managing subsystem to be more resilient towards faults. Furthermore, it would be interesting to investigate more complex ways of representing relations between features, for example, representing in the feature model that the AUV has to follow the pipeline at a low altitude when using the camera and at a medium altitude when using the sonar. Finally, analysing a system with and without adaptation to determine both the gains and the costs of using self-adaptation would be interesting.

**Acknowledgements**

We would like to thank Clemens Dubslaff for explaining ProFeat and its usage to us, and for answering numerous questions. This work was 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) and by the Italian MUR PRIN 2020TL3X8X project T-LADIES (Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems).

## References


[1] Abbas, N., Andersson, J., Gerostathopoulos, I., Lago, P., Biffl, S., Musil, J., Brada, P., Bures, T., Salle, A.D., Galster, M., Lewis, G., Litoiu, M., Patros, P., Pelliccione, P., 2023. Self-Adaptation in Industry: A Survey. ACM Transactions on Autonomous and Adaptive Systems 18, 5:1–5:44. doi:10.1145/3589227.

[2] Adelsberger, S., Sobernig, S., Neumann, G., 2014. Towards Assessing the Complexity of Object Migrationin Dynamic, Feature-oriented Software Product Lines, in: Proceedings of the 8th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2014), ACM. pp. 17:1–17:8. doi:10.1145/2556624.2556645.

[3] Aguayo, O., Sepúlveda, S., 2022. Variability Management in Dynamic Software Product Lines for Self-Adaptive Systems—A Systematic Mapping. Applied Sciences 12. doi:10.3390/app122010240.

[4] Alur, R., Henzinger, T.A., Lafferriere, G., Pappas, G.J., 2000. Discrete Abstractions of Hybrid Systems. Proceedings of the IEEE 88, 971–984. doi:10.1109/5.871304.

[5] Apel, S., Batory, D., Kästner, C., Saake, G., 2013. Feature-Oriented Software Product Lines: Concepts and Implementation. Springer. doi:10.1007/978-3-642-37521-7.

[6] Apel, S., Kolesnikov, S.S., Liebig, J., Kästner, C., Kuhlemann, M., Leich, T., 2012. Access control in feature-oriented programming. Science of Computer Programming 77, 174–187. doi:10.1016/J.SCICO.2010.07.005.

[7] Araujo, H., Mousavi, M.R., Varshosaz, M., 2023. Testing, Validation, and Verification of Robotic and Autonomous Systems: A Systematic Review. ACM Transactions on Software Engineering and Methodology 32, 51:1–51:61. doi:10.1145/3542945.

[8] Ayala, I., Papadopoulos, A.V., Amor, M., Fuentes, L., 2021. ProDSPL: Proactive self-adaptation based on Dynamic Software Product Lines. Journal of Systems and Software 175. doi:10.1016/J.JSS.2021.110909.

[9] Baier, C., Katoen, J.P., 2008. Principles of Model Checking. MIT Press.

[10] Baresi, L., 2014. Self-Adaptive Systems, Services, and Product Lines, in: Proceedings of the 18th International Software Product Line Conference (SPLC 2014), ACM. pp. 2–4. doi:10.1145/2648511.2648512.

[11] Basile, D., ter Beek, M.H., 2022. Contract Automata Library. Science of Computer Programming 221. doi:10.1016/j.scico.2022.102841.

[12] Basile, D., ter Beek, M.H., Degano, P., Legay, A., Ferrari, G.L., Gnesi, S., Di Giandomenico, F., 2020. Controller synthesis of service contracts with variability. Science of Computer Programming 187. doi:10.1016/j.scico.2019.102344.

[13] Basile, D., ter Beek, M.H., Di Giandomenico, F., Gnesi, S., 2017. Orchestration of Dynamic Service Product Lines with Featured Modal Contract Automata, in: Proceedings of the 21st International Systems and Software Product Line Conference (SPLC 2017), ACM. pp. 117–122. doi:10.1145/3109729.3109741.

[14] ter Beek, M.H., Cledou, G., Hennicker, R., Proença, J., 2021. Featured Team Automata, in: Huisman, M., Pasareanu, C.S., Zhan, N. (Eds.), Proceedings of the 24th International Symposium on Formal Methods (FM 2021), Springer. pp. 483–502. doi:10.1007/978-3-030-90870-6_26.

[15] ter Beek, M.H., Damiani, F., Gnesi, S., Mazzanti, F., Paolini, L., 2019a. On the expressiveness of modal transition systems with variability constraints. Science of Computer Programming 169, 1–17. doi:10.1016/j.scico.2018.09.006.

[16] ter Beek, M.H., Damiani, F., Lienhardt, M., Mazzanti, F., Paolini, L., 2022. Efficient static analysis and verification of featured transition systems. Empirical Software Engineering 22, 10:1–10:43. doi:10.1007/s10664-020-09930-8.

[17] ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F., 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, 287–315. doi:10.1016/j.jlamp.2015.11.006.

[18] ter Beek, M.H., Ferrari, A., 2022. Empirical Formal Methods: Guidelines for Performing Empirical Studies on Formal Methods. Software 1, 381–416. doi:10.3390/software1040017.

[19] ter Beek, M.H., Legay, A., Lluch Lafuente, A., Vandin, A., 2020a. A Framework for Quantitative Modeling and Analysis of Highly (Re)configurable Systems. IEEE Transaction on Software Engineering 46, 321–345. doi:10.1109/TSE.2018.2853726.

[20] ter Beek, M.H., van Loo, S., de Vink, E.P., Willemse, T.A., 2020b. Family-Based SPL Model Checking Using Parity Games with Variability, in: Wehrheim, H., Cabot, J. (Eds.), Proceedings of the 23rd International Conference on Fundamental Approaches to Software Engineering (FASE 2020). Springer. volume 12076 of *Lecture Notes in Computer Science*, pp. 245–265. doi:10.1007/978-3-030-45234-6_12.

[21] ter Beek, M.H., Mazzanti, F., 2014. VMC: Recent Advances and Challenges Ahead, in: Proceedings of the 18th International Software Product Line Conference (SPLC 2014), ACM. pp. 70–77. doi:10.1145/2647908.2655969.

[22] ter Beek, M.H., Mazzanti, F., Sulova, A., 2012. VMC: A Tool for Product Variability Analysis, in: Giannakopoulou, D., Méry, D. (Eds.), Proceedings of the 18th International Symposium on Formal Methods (FM 2012). Springer. volume 7436 of *Lecture Notes in Computer Science*, pp. 450–454. doi:10.1007/978-3-642-32759-9_36.

[23] ter Beek, M.H., Schmid, K., Eichelberger, H., 2019b. Textual Variability Modeling Languages: An Overview and Considerations, in: Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC 2019), ACM. pp. 82:1–82:7. doi:10.1145/3307630.3342398.

[24] ter Beek, M.H., de Vink, E.P., 2014. Towards Modular Verification of Software Product Lines with mCRL2, in: Margaria, T., Steffen, B. (Eds.), Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods: Technologies for Mastering Change (ISoLA 2014). Springer. volume 8802 of *Lecture Notes in Computer Science*, pp. 368–385. doi:10.1007/978-3-662-45234-9_26.

[25] ter Beek, M.H., de Vink, E.P., Willemse, T.A.C., 2017. Family-Based Model Checking with mCRL2, in: Huisman, M., Rubin, J. (Eds.), Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE 2017). Springer. volume 10202 of *Lecture Notes in Computer Science*, pp. 387–405. doi:10.1007/978-3-662-54494-5_23.

[26] Bencomo, N., Hallsteinsen, S.O., de Almeida, E.S., 2012. A View of the Dynamic Software Product Line Landscape. IEEE Computer 45, 36–41. doi:10.1109/MC.2012.292.

[27] Bencomo, N., Sawyer, P., Blair, G.S., Grace, P., 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 Software Product Line Conference (SPLC 2008), Lero, University of Limerick. pp. 23–32.

[28] Bozhinoski, D., Oviedo, M.G., Garcia, N.H., Deshpande, H., van der Hoorn, G., Tjerngren, J., Wąsowski, A., Corbato, C.H., 2022. MROS: runtime adaptation for robot control architectures. Advanced Robotics 36, 502–518. doi:10.1080/01691864.2022.2039761.

[29] Brugali, D., 2020. Runtime reconfiguration of robot control systems: a ROS-based case study, in: Proceedings of the 4th International Conference on Robotic Computing (IRC 2020), IEEE. pp. 256–262. doi:10.1109/IRC.2020.00047.

[30] Brugali, D., 2021. Software Product Line Engineering for Robotics, in: Cavalcanti, A., Dongol, B., Hierons, R., Timmis, J., Woodcock, J. (Eds.), Software Engineering for Robotics. Springer, pp. 1–28. doi:10.1007/978-3-030-66494-7_1.

[31] Brugali, D., Capilla, R., Hinchey, M., 2015. Dynamic Variability Meets Robotics. IEEE Computer 48, 94–97. doi:10.1109/MC.2015.354.

[32] Brugali, D., Hochgeschwender, N., 2017. Managing the Functional Variability of Robotic Perception Systems, in: Proceedings of the 1st International Conference on Robotic Computing (IRC 2017), IEEE. pp. 277–283. doi:10.1109/IRC.2017.20.

[33] Bürdek, J., Lity, S., Lochau, M., Berens, M., Goltz, U., Schürr, A., 2014. Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints, in: Proceedings of the 8th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2014), ACM. pp. 16:1–16:8. doi:10.1145/2556624.2556627.

[34] Capilla, R., Bosch, J., Trinidad, P., Ruiz-Cortés, A., Hinchey, M., 2014. An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. Journal of Systems and Software 91, 3–23. doi:10.1016/j.jss.2013.12.038.

[35] Chrszon, P., Baier, C., Dubslaff, C., Klüppelholz, S., 2020. From Features to Roles, in: Proceedings of the 24th International Systems and Software Product Line Conference (SPLC 2020), ACM. pp. 19:1–19:11. doi:10.1145/3382025.3414962.

[36] Chrszon, P., Baier, C., Dubslaff, C., Klüppelholz, S., 2023. Interaction detection in configurable systems – A formal approach featuring roles. Journal of Systems and Software 196. doi:10.1016/j.jss.2022.111556.

[37] Chrszon, P., Dubslaff, C., Klüppelholz, S., Baier, C., 2018. ProFeat: Feature-Oriented Engineering for Family-Based Probabilistic Model Checking. Formal Aspects of Computing 30, 45–75. doi:10.1007/s00165-017-0432-4.

[38] Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y., 2012. Model checking software product lines with SNIP. International Journal on Software Tools for Technology Transfer 14, 589–612. doi:10.1007/s10009-012-0234-1.

[39] Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y., 2014. Formal semantics, modular specification, and symbolic verification of product-line behaviour. Science of Computer Programming 80, 416–439. doi:10.1145/2499777.2499781.

[40] Classen, A., Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., Raskin, J.F., 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. IEEE Transaction on Software Engineering 39, 1069–1089. doi:10.1109/TSE.2012.86.

[41] Classen, A., Heymans, P., Schobbens, P.Y., 2008. What's in a Feature: A Requirements Engineering Perspective, in: Fiadeiro, J.L., Inverardi, P. (Eds.), Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE 2008). Springer. volume 4961 of *Lecture Notes in Computer Science*, pp. 16–30. doi:10.1007/978-3-540-78743-3_2.

[42] Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F., 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. pp. 335–344. doi:10.1145/1806799.1806850.

[43] Cordy, M., Classen, A., Heymans, P., Legay, A., Schobbens, P.Y., 2013a. Model Checking Adaptive Software with Featured Transition Systems, in: Cámara, J., de Lemos, R., Ghezzi, C., Lopes, A. (Eds.), Assurances for Self-Adaptive Systems: Principles, Models, and Techniques. Springer. volume 7740 of *Lecture Notes in Computer Science*, pp. 1–29. doi:10.1007/978-3-642-36249-1_1.

[44] Cordy, M., Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., 2013b. ProVeLines: a product line of verifiers for software product lines, in: Proceedings of the 17th International Software Product Line Conference (SPLC 2013), ACM. pp. 141–146. doi:10.1145/2499777.2499781.

[45] Cordy, M., Legay, A., Schobbens, P.Y., Traonouez, L.M., 2013c. A Framework for the Rigorous Design of Highly Adaptive Timed Systems, in: Proceedings of the 1st FME Workshop on Formal Methods in Software Engineering (FormaliSE 2013), IEEE. pp. 64–70. doi:10.1109/FormaliSE.2013.6612279.

[46] Czarnecki, K., Eisenecker, U.W., 2000. Generative Programming: Methods, Tools, and Applications. Addison-Wesley.

[47] Damiani, F., Padovani, L., Schaefer, I., Seidl, C., 2018. A core calculus for dynamic delta-oriented programming. Acta Informatica 55, 269–307. doi:10.1007/S00236-017-0293-6.

[48] Damiani, F., Schaefer, I., 2011. Dynamic Delta-Oriented Programming, in: Proceedings of the 15th International Software Product Line Conference (SPLC 2011), ACM. doi:10.1145/2019136.2019175.

[49] Dhungana, D., Grünbacher, P., Rabiser, R., 2007. Domain-Specific Adaptations of Product Line Variability Modeling, in: Ralyté, J., Brinkkemper, S., Henderson-Sellers, B. (Eds.), Proceedings of the IFIP WG 8.1 Working Conference on Situational Method Engineering: Fundamentals and Experiences (ME 2007). Springer. volume 244 of *IFIP Advances in Information and Communication Technology*, pp. 238–251. doi:10.1007/978-0-387-73947-2_19.

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

[51] Dimovski, A.S., Al-Sibahi, A.S., Brabrand, C., Wąsowski, A., 2017. Efficient family-based model checking via variability abstractions. International Journal on Software Tools for Technology Transfer 5, 585–603. doi:10.1007/s10009-016-0425-2.

[52] Dimovski, A.S., Al-Sibahi, A.S., Brabrand, C., Wąsowski, A., 2015. Family-Based Model Checking Without a Family-Based Model Checker, in: Fischer, B., Geldenhuys, J. (Eds.), Proceedings of the 22nd International Symposium on Model Checking Software (SPIN 2015). Springer. volume 9232 of *Lecture Notes in Computer Science*, pp. 282–299. doi:10.1007/978-3-319-23404-5_18.

[53] Dimovski, A.S., Legay, A., Wąsowski, A., 2019. Variability Abstraction and Refinement for Game-Based Lifted Model Checking of Full CTL, in: Hähnle, R., van der Aalst, W. (Eds.), Proceedings of the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE 2019). Springer. volume 11424 of *Lecture Notes in Computer Science*, pp. 192–209. doi:10.1007/978-3-030-16722-6_11.

[54] Dimovski, A.S., Wąsowski, A., 2017. Variability-Specific Abstraction Refinement for Family-Based Model Checking, in: Huisman, M., Rubin, J. (Eds.), Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE 2017). Springer. volume 10202 of *Lecture Notes in Computer Science*, pp. 406–423. doi:10.1007/978-3-662-54494-5_24.

[55] Dinkelaker, T., Mitschke, R., Fetzer, K., Mezini, M., 2010. A Dynamic Software Product Line Approach Using Aspect Models at Runtime, in: Lahire, P., Georg, G., Oussalah, M., Whittle, J., Moha, N., Van Baelen, S. (Eds.), Proceedings of the 1st International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines (Composition&Variability 2010). volume 564 of *CEUR Workshop Proceedings*, pp. 1:1–1:8. URL: https://ceur-ws.org/Vol-564/compositionvariability2010_submission_4.pdf.

[56] Dubslaff, C., 2019. Compositional Feature-Oriented Systems, in: Ölveczky, P.C., Salaün, G. (Eds.), Proceedings of the 17th International Conference on Software Engineering and Formal Methods (SEFM 2019). Springer. volume 11724 of *Lecture Notes in Computer Science*, pp. 162–180. doi:10.1007/978-3-030-30446-1_9.

[57] Dubslaff, C., Baier, C., Klüppelholz, S., 2015. Probabilistic Model Checking for Feature-Oriented Systems, in: Chiba, S., Tanter, E., Ernst, E., Hirschfeld, R. (Eds.), Transactions on Aspect-Oriented Software Development XII. Springer. volume 8989 of *Lecture Notes in Computer Science*, pp. 180–220. doi:10.1007/978-3-662-46734-3_5.

[58] Dubslaff, C., Klüppelholz, S., Baier, C., 2014. Probabilistic Model Checking for Energy Analysis in Software Product Lines, in: Proceedings of the 13th International Conference on Modularity (MODULARITY 2014), ACM. pp. 169–180. doi:10.1145/2577080.2577095.

[59] García, S., Strüber, D., Brugali, D., Di Fava, A., Schillinger, P., Pelliccione, P., Berger, T., 2019. Variability Modeling of Service Robots: Experiences and Challenges, in: Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2019), ACM. pp. 8:1–8:6. doi:10.1145/3302333.3302350.

[60] Gherardi, L., Brugali, D., 2014. Modeling and Reusing Robotic Software Architectures: the HyperFlex Toolchain, in: Proceedings of the International Conference on Robotics and Automation (ICRA 2014), IEEE. pp. 6414–6420. doi:10.1109/ICRA.2014.6907806.

[61] Gherardi, L., Hochgeschwender, N., 2015. RRA: Models and Tools for Robotics Run-time Adaptation, in: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2015), IEEE. pp. 1777–1784. doi:10.1109/IROS.2015.7353608.

[62] Girard, A., Pappas, G.J., 2007. Approximation Metrics for Discrete and Continuous Systems. IEEE Transactions on Automatic Control 52, 782–798. doi:10.1109/TAC.2007.895849.

[63] Göttmann, H., Bacher, I., Gottwald, N., Lochau, M., 2021. Static Analysis Techniques for Efficient Consistency Checking of Real-Time-Aware DSPL Specifications, in: Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2021),

ACM. pp. 17:1–17:9. doi:10.1145/3442391.3442409.
[64] Göttmann, H., Luthmann, L., Lochau, M., Schürr, A., 2020. Real-Time-Aware Reconfiguration Decisions for Dynamic Software Product Lines, in: Proceedings of the 24th International Systems and Software Product Line Conference (SPLC 2020), ACM. pp. 13:1–13:11. doi:10.1145/3382025.3414945.
[65] Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K., 2013. Dynamic Software Product Lines, in: Capilla, R., Bosch, J., Kang, K.C. (Eds.), Systems and Software Variability Management: Concepts, Tools and Experiences. Springer, pp. 253–260. doi:10.1007/978-3-642-36583-6_16.
[66] Hallsteinsen, S., Stav, E., Solberg, A., Floch, J., 2006. Using Product Line Techniques to Build Adaptive Systems, in: Proceedings of the 10th International Software Product Line Conference (SPLC 2006), IEEE. pp. 141–150. doi:10.1109/SPLINE.2006.1691586.
[67] Hernández Corbato, C., 2013. Model-Based Self-awareness Patterns for Autonomy. Ph.D. thesis. Universidad Politécnica de Madrid. doi:10.20868/UPM.thesis.23178.
[68] Hezavehi, S.M., Weyns, D., Avgeriou, P., Calinescu, R., Mirandola, R., Perez-Palacin, D., 2021. Uncertainty in Self-adaptive Systems: A Research Community Perspective. ACM Transactions on Autonomous and Adaptive Systems 15, 10:1–10:36. doi:10.1145/3487921.
[69] Hinchey, M., Park, S., Schmid, K., 2012. Building Dynamic Software Product Lines. IEEE Computer 45, 22–26. doi:10.1109/MC.2012.332.
[70] Kephart, J.O., Chess, D.M., 2003. The Vision of Autonomic Computing. IEEE Computer 36, 41–50. doi:10.1109/MC.2003.1160055.
[71] Křetínský, J., 2017. 30 Years of Modal Transition Systems: Survey of Extensions and Analysis, in: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (Eds.), Models, Algorithms, Logics and Tools. Springer. volume 10460 of *Lecture Notes in Computer Science*, pp. 36–74. doi:10.1007/978-3-319-63121-9_3.
[72] Kwiatkowska, M., Norman, G., Parker, D., 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems, in: Gopalakrishnan, G., Qadeer, S. (Eds.), Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011). Springer. volume 6806 of *Lecture Notes in Computer Science*, pp. 585–591. doi:10.1007/978-3-642-22110-1_47.
[73] Larsen, K.G., Nyman, U., Wąsowski, A., 2007. Modal I/O Automata for Interface and Product Line Theories, in: De Nicola, R. (Ed.), Proceedings of the 16th European Symposium on Programming (ESOP 2007). Springer. volume 4421 of *Lecture Notes in Computer Science*, pp. 64–79. doi:10.1007/978-3-540-71316-6_6.
[74] Larsen, K.G., Thomsen, B., 1988. A Modal Process Logic, in: Proceedings of the 3rd Symposium on Logic in Computer Science (LICS 1988), IEEE. pp. 203–210. doi:10.1109/LICS.1988.5119.
[75] Lauenroth, K., Pohl, K., Töhning, S., 2009. Model Checking of Domain Artifacts in Product Line Engineering, in: Proceedings of the 24th International Conference on Automated Software Engineering (ASE 2009), IEEE. pp. 269–280. doi:10.1109/ASE.2009.16.
[76] Lochau, M., Bürdek, J., Hölzle, S., Schürr, A., 2017. Specification and automated validation of staged reconfiguration processes for dynamic software product lines. Software & Systems Modeling 16, 125–152. doi:10.1007/S10270-015-0470-4.
[77] Lochau, M., Mennicke, S., Baller, H., Ribbeck, L., 2016. Incremental model checking of delta-oriented software product lines. Journal of Logical and Algebraic Methods in Programming 85, 245–267. doi:10.1016/j.jlamp.2015.09.004.
[78] Luckcuck, M., Farrell, M., Dennis, L.A., Dixon, C., Fisher, M., 2019. Formal Specification and Verification of Autonomous Robotic Systems: A Survey. ACM Computing Surveys 52, 100:1–100:41. doi:10.1145/3342355.
[79] Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., Saake, G., 2017. Mastering Software Variability with FeatureIDE. Springer. doi:10.1007/978-3-319-61443-4.
[80] Muschevici, R., Clarke, D., Proença, J., 2013. Executable Modelling of Dynamic Software Product Lines in the ABS Language, in: Proceedings of the 5th International Workshop on Feature-Oriented Software Development (FOSD 2013), ACM. pp. 17–24. doi:10.1145/2528265.2528266.
[81] Muschevici, R., Clarke, D., Proença, J., 2010. Feature Petri Nets, in: Proceedings of the 1st In-

ternational Workshop on Formal Methods in Software Product Line Engineering (FMSPLE 2010), Technical Report, University of Lancaster.

[82] Muschevici, R., Proença, J., Clarke, D., 2016. Feature Nets: behavioural modelling of software product lines. Software & Systems Modeling 15, 1181–1206. doi:10.1007/s10270-015-0475-z.

[83] Nordmann, A., Lange, R., Rico, F.M., 2021. System modes – digestible system (re-)configuration for robotics, in: Proceedings of the 3rd IEEE/ACM International Workshop on Robotics Software Engineering (RoSE 2021), IEEE. pp. 19–24. doi:10.1109/ROSE52553.2021.00010.

[84] Olaechea, R., Atlee, J., Legay, A., Fahrenberg, U., 2018. Trace Checking for Dynamic Software Product Lines, in: Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2018), ACM. pp. 69–75. doi:10.1145/3194133.3194143.

[85] Pfannemuller, M., Krupitzer, C., Weckesser, M., Becker, C., 2017. A Dynamic Software Product Line Approach for Adaptation Planning in Autonomic Computing Systems, in: Proceedings of the International Conference on Autonomic Computing (ICAC 2017), IEEE. pp. 247–254. doi:10.1109/ICAC.2017.18.

[86] Päßler, J., ter Beek, M.H., Damiani, F., Johnsen, E.B., Tapia Tarifa, S.L., 2024. Analysing Self-Adaptive Systems as Software Product Lines (Artifact). Zenodo. doi:10.5281/zenodo.14230735.

[87] Päßler, J., ter Beek, M.H., Damiani, F., Johnsen, E.B., Tapia Tarifa, S.L., 2025. A Configurable Software Model of a Self-Adaptive Robotic System. Science of Computer Programming 240. doi:10.1016/j.scico.2024.103221.

[88] Päßler, J., ter Beek, M.H., Damiani, F., Tapia Tarifa, S.L., Johnsen, E.B., 2023a. Formal Modelling and Analysis of a Self-Adaptive Robotic System, in: Herber, P., Wijs, A. (Eds.), Proceedings of the 18th International Conference on integrated Formal Methods (iFM 2023). Springer. volume 14300 of *Lecture Notes in Computer Science*, pp. 343–363. doi:10.1007/978-3-031-47705-8_18.

[89] Päßler, J., ter Beek, M.H., Damiani, F., Tapia Tarifa, S.L., Johnsen, E.B., 2023b. Formal Modelling and Analysis of a Self-Adaptive Robotic System (Artifact). Zenodo. doi:10.5281/zenodo.8275533.

[90] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A., 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).

[91] Quijano, S., Varshosaz, M., Wąsowski, A., 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. pp. 63–67. doi:10.1109/ICSTW60967.2024.00022.

[92] Quinton, C., Vierhauser, M., Rabiser, R., Baresi, L., Grünbacher, P., Schuhmayer, C., 2021. Evolution in dynamic software product lines. Journal of Software: Evolution and Process 33, e2293:1–e2293:25. doi:10.1002/SMR.2293.

[93] Rezende Silva, G., Päßler, J., Zwanepol, J., Alberts, E., Tapia Tarifa, S.L., Gerostathopoulos, I., Johnsen, E.B., Hernández Corbato, C., 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. pp. 181–187. doi:10.1109/SEAMS59076.2023.00031.

[94] Romero-Garcés, A., Freitas, R.S.D., Marfil, R., Vicente-Chicote, C., Martínez, J., Inglés-Romero, J.F., Bandera, A., 2022. QoS metrics-in-the-loop for endowing runtime self-adaptation to robotic software architectures. Multimedia Tools and Applications 81, 3603–3628. doi:10.1007/S11042-021-11603-7.

[95] Salehie, M., Tahvildari, L., 2009. Self-Adaptive Software: Landscape and Research Challenges. ACM Transactions on Autonomous and Adaptive Systems 4, 14:1–14:42. doi:10.1145/1516533.1516538.

[96] Saller, K., Lochau, M., Reimund, I., 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. pp. 106–113. doi:10.1145/2499777.2500716.

[97] Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N., 2010. Delta-Oriented Programming of Software Product Lines, in: Bosch, J., Lee, J. (Eds.), Proceedings of the 14th International Software Product Line Conference (SPLC 2010). Springer. volume 6287 of *Lecture Notes in Computer Science*,

pp. 77–91. doi:10.1007/978-3-642-15579-6_6.
[98] Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S., Villela, K., 2012. Software diversity: state of the art and perspectives. International Journal on Software Tools for Technology Transfer 14, 477–495. doi:10.1007/S10009-012-0253-Y.
[99] Schmid, K., Park, S., Hinchey, M., Hallsteinsen, S., 2008. Dynamic Software Product Lines. IEEE Computer 41, 93–95. doi:10.1109/MC.2008.123.
[100] Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y., 2006. Feature Diagrams: A Survey and a Formal Semantics, in: Proceedings of the 14th International Conference on Requirements Engineering (RE 2006), IEEE. pp. 136–145. doi:10.1109/RE.2006.23.
[101] Siciliano, B., Khatib, O. (Eds.), 2016. Springer Handbook of Robotics. Springer Handbooks, Springer. doi:10.1007/978-3-319-32552-1.
[102] Sousa, G., Rudametkin, W., Duchien, L., 2017. Extending Dynamic Software Product Lines with Temporal Constraints, in: Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2017), IEEE. pp. 129–139. doi:10.1109/SEAMS.2017.6.
[103] Stol, K.J., Fitzgerald, B., 2018. The ABC of software engineering research. ACM Transactions on Software Engineering and Methodology 27, 1–51. doi:10.1145/3241743.
[104] Thuijsman, S., Reniers, M., 2024. Supervisory Control for Dynamic Feature Configuration in Product Lines. ACM Transactions on Embedded Computing Systems 23, 71:1–71:25. doi:10.1145/3579644.
[105] Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G., 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. ACM Computing Surveys 47, 6:1–6:45. doi:10.1145/2580950.
[106] Thüm, T., van Hoorn, A., Apel, S., Bürdek, J., Getir, S., Heinrich, R., Jung, R., Kowal, M., Lochau, M., Schaefer, I., Walter, J., 2019. Performance Analysis Strategies for Software Variants and Versions, in: Reussner, R.H., Goedicke, M., Hasselbring, W., Vogel-Heuser, B., Keim, J., Märtin, L. (Eds.), Managed Software Evolution. Springer. chapter 8, pp. 175–206. doi:10.1007/978-3-030-13499-0_8.
[107] Valdezate, A., Capilla, R., Crespo, J., Barber, R., 2022. RuVa: A Runtime Software Variability Algorithm. IEEE Access 10, 52525–52536. doi:10.1109/ACCESS.2022.3175505.
[108] Vandin, A., ter Beek, M.H., Legay, A., Lluch Lafuente, A., 2018. QFLan: A Tool for the Quantitative Analysis of Highly Reconfigurable Systems, in: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (Eds.), Proceedings of the 22nd International Symposium on Formal Methods (FM 2018). Springer. volume 10951 of *Lecture Notes in Computer Science*, pp. 329–337. doi:10.1007/978-3-319-95582-7_19.
[109] Weyns, D., 2020. An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective. John Wiley & Sons.
[110] Weyns, D., Iftikhar, M.U., de la Iglesia, D.G., Ahmad, T., 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. pp. 67–79. doi:10.1145/2347583.2347592.
[111] Wohlin, C., Rainer, A., 2022. Is it a case study?—A critical analysis and guidance. Journal of Systems and Software 192. doi:10.1016/j.jss.2022.111395.