Chapter 1

Background

1.1 Configurable Software Systems

Almost all software systems in use today are configurable. This means, almost all software systems allow a user to customize the software system to their specific requirements. To allow for such customizability, software systems provide a number of configuration options, also called features, to specify the desired behaviour of the software system. A feature is an attribute of a system that directly affects end-users [**?**]. Each combination of configuration options describes a variant of the software system, making configurable software systems similar to Software Product Lines (SPL) [**?**]. For example, a web server like the Apache HTTP Server, allows the user to select features like compression, encryption or the use of specific transport protocols like HTTP/2.

Features usually satisfy functional requirements [**?**] of users. Functional requirement as described by **?** "[...] capture the intended behaviour of the system - or what the system will do". Meaning, for the example of the webserver, a functional requirement would be the use of a specific compression algorithm like Brotli[1]. On the other hand, software systems also have non-functional requirements. **?** describes non-functional requirements as "[...] requirements on its development or operational cost, performance, reliability, maintainability, portability, robustness, and the like."

In this work, non-functional requirements, like performance or memory usage, of a software system are analyzed in regard to the configurations used. Usually, the effect of configuration options on functional properties of the software are well documented and understood, but the effect on non-functional properties are unknown to the user and more often than not, to the developer.

[1]Brotli is a compression algorithm developed by Google primarily used by web servers to compress HTTP content.

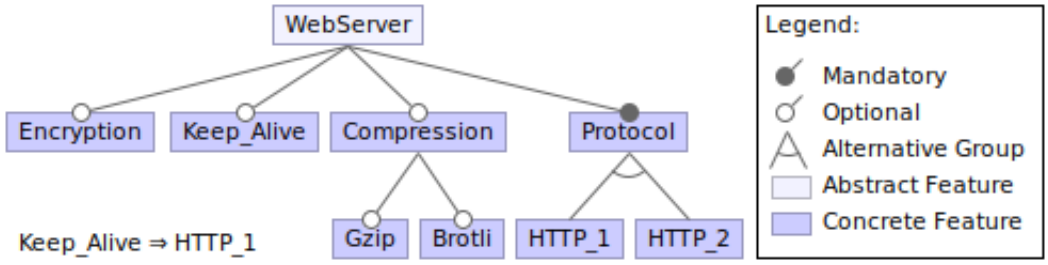

Figure 1.1: Feature diagram of an exemplary web server created with FeatureIDE

Feature models

Feature models, introduced by **?**], serve as a mean to convey information about the features of a system. Which features can be chosen and how those features can be combined. For this, the structural relationship between features is described. Each feature can either be optional or mandatory and each feature can have child features that are either in an or group or an alternative group. In an alternative group, only one of the child features can be selected, in an or group, at least one of the child features must be selected. There are also composition rules, also known as cross-tree constraints, which define constraints not expressible in the hierarchical structure of the features.

Feature models can be represented visually by feature diagrams or textually by formats like SXFM [**?**]. Feature diagrams are an intuitive way for a user to work with feature models, they visualize the feature model in a tree diagram. An example of a feature diagram can be seen in Figure 1.1. There, the features of an exemplary web server are described. The web server has 4 features, of which 3 are optional and one is mandatory. It also includes two cross-tree constraints. The feature *Encryption* is an optional feature, without any constraints on it. This feature can be enabled or disabled however the user likes. In this work, this kind of feature without any constraints on it is also referenced as an independent feature. Even though the *Keep_alive* feature has no differences in the hierarchical structure to the *Encryption* feature, it has a cross-tree constrain. This constraint mandates, that if *Keep_alive* is selected, the *HTTP/1* feature is selected too, since the *Keep_alive* functionality is only available in the HTTP/1 protocol. *Compression* and *Protocol* are both features with child features. Since *HTTP/1* and *HTTP/2* are in an alternate group, only one of these two protocols can be chosen for a valid configuration. With *Compression*, one or both of the child features can be chosen, since they are in an or group.

Interactions

Features in a software system directly influence the way a system behaves with respect to its functional and non-functional properties. If we turned on the feature *Encryption* in our exemplary web server, the webserver would need to encrypt the sent data. This not only has impacts on the functional properties of the web server, but it also impacts the non-functional property. The web server now has to further process the data before sending it, which results in more computational power needed. The same goes for the feature *Compression*, or more specifically for the features *Brotli* and *Gzip*. If the compression is enabled in the webserver, the webserver needs to compress the data before sending it, which requires more computational power.

One could now assume, that if we enable both features at the same time, the computational power needed to process a request is the sum of the power needed if the encryption is enabled and if the compression is enabled. A more likely scenario would be, that the computational power needed would be less than that. If we first compress the data, the encryption part of the webserver has to encrypt fewer data. The two features *Encryption* and *Compression* interact.

1.2 Performance-Influence Models

The number of possible configurations of modern software systems and the complex constraints between them can be overwhelming. Making it difficult to find an optimal configuration, that performs as desired. Performance influence models are meant to ease understanding, debugging and optimization of configurable software systems [**?**].

A performance influence model consists of several terms that describe the performance of a configuration based on the values of configuration options [**?**]. In this context, performance can be measurable quality attributes such as execution time, memory size, or energy consumption. The model describes the influence of several independent variables X, our configuration, on a dependent variable y, our measurable quality attribute. While there are several approaches to predict performance, in general, they all work similarly. They sample a subset of configurations - this is done because it is infeasible to measure the performance of all configurations if the configuration space is too big - and learn a model with the sampled configurations.

?] introduce a variability-aware approach to predict a configuration's performance based on random sampling. They use a Classification-And-Regression-Tree [**?**] to recursively partition the configuration space into smaller segments until they can fit a simple local prediction model into each segment.

?] describe how to create human understandable models based on previous work [**?**]. They combined binary sampling strategies such as option-wise, negative option-wise, and pair-wise, with numerical sampling strategies, such as the Placket-Burman-Design. They then used stepwise linear regression to learn the influence model.

1.2.1 Sampling

The selection of a subset of configurations plays an integral role in almost all methods to predict the performance of a software system. If a configuration option is not present in the sampled subset of configuration a model can not learn the influence of this option. The random selection, random sampling, as used by most machine learning applications proves to be difficult with configurations. Mainly due to the constraints on the configuration options. Near uniform random sampling in the presence of constraints, although possible, is infeasible [**?**]. This resulted in developing dedicated sampling strategies for configuration spaces.

Distance based sampling

?] describes a way to randomly sample a configuration space based on a distance metric and a probability distribution, called distance-based sampling. For this, they rely on a distance metric, like the Manhattan-Distance, to assign each configuration a distance value. By selecting a distance value through a discrete probability distribution and then picking a configuration with the corresponding distance value, they achieve a spread over the configurations resembling the given probability distribution. This allows for uniform random like sampling if the chosen probability distribution is the uniform distribution.

Binary decision diagram-sampling

Although **?**] do not create a model to predict the performance of a system, they implement a way of random sampling configuration spaces through binary decision diagrams (BDD)[**?**]. They transform a given feature model into BDD, this makes it easy to count the number of valid configurations and thus easy to randomly sample from them. While a BDD allows for random sampling, a major drawback is the creation of it, which may exceed time or memory constraints for some use cases.

1.3 SAT-Solver

The satisfiability problem in propositional logic (SAT) is the problem of determining the existence of any solution, that satisfies a given boolean formula. A boolean formula is called satisfiable, if we can assign the variables in the formula true or false values in such a way, that the formula evaluates to true. As an example, the formula $A \vee B$ is satisfiable. We can prove this by assigning $A = true$ and $B = false$. The resulting formula would be $true \vee false$, which evaluates to true, meaning the formula is satisfiable. If a formula is not satisfiable, the formula is called unsatisfiable. An example of this would be the formula $A \wedge \neg A$. No possible assignment of A would result in the formula being evaluated to true. SAT-problems arise in many application domains and, most notably for this work, can be used in validating configurations for SPLs.

SAT-Solvers are programs, which aim to solve the SAT-problem. Even though the SAT-problem is NP-complete [**?**], modern SAT-Solvers can often handle problems with hundreds of thousands of variables and millions of constraints [**?**]. They often use algorithms such as DPLL [**?**] or variations of it at their core and their optimization is a large field of research on its own. Many free and open-source implementations of SAT-Solvers exist [**?** **?** **?**]. In this work, we use Z3 [**?**], which is an SMT-solver. Satisfiability modulo theories (SMT) generalize the SAT-problem to formulas involving real numbers and various data structures. The use of an SMT-solver, instead of an SAT-Solver, allows us to define cost functions to optimize the solution of the SMT-solver.

In this work, we make use of the SMT-solver Z3 to generate valid configurations for a configurable software system. Like **?**] defined it, we define our feature model as a set of boolean features with a set of constraints over them. A configuration is a set of features, where all features in that set are selected and features that are not in the set are unselected. A configuration is valid if it satisfies all the constraints of the feature model, otherwise, it is called invalid. **?**] shows, that a feature model can be defined as a propositional formula. This not only allows the feature model to be stored in file formats like DIMACS, but it also allows the use of of-the-shelf SAT-Solvers for feature models.

1.4 Sensitivity analysis

? describes sensitivity analysis as "[...] the study of how the uncertainty in the output of a model (numerical or otherwise) can be apportioned to different sources of uncertainty in the model input" [**?**]. To understand this definition, we need to understand, what a model is. A model, in this context, is a mathematical description of a system and its rules. Such models could be weather models to predict future rainfall or financial models to predict the

stock market. By defining the input of the model, we can get a prediction from the model. In this work, we view a configurable software system as a black-box model on which we can perform sensitivity analysis. Our input variables are the features the systems have, and our output variable is the measurable quality attribute. Such models, no matter what they predict or are, can be highly complex, and it can be hard to understand the relationship between the inputs and the outputs of the models. The degree to which an output variable is affected by the change in an input variable is called sensitivity. There are many approaches to performing sensitivity analysis. One common and simple approach to sensitivity analysis is the one-at-a-time (OAT) approach. In this experimental design, one input variable is changed, while all other values are kept at a baseline value. By observing the changes in the output variable, one can determine the influence an input variable has. Sensitivity can then be described, for example, by linear regression or partial derivatives.

Performing sensitivity analysis on a configurable software with the OAT-design would mean, enabling one feature at a time while all other features are disabled and observing the change in the quality attribute under analysis. We assume here, that the software system has no constraints among the configuration options. By fitting a regression model we can determine the sensitivity of the quality attribute to the features enabled.