Predicting and Learning the Performance of Configurable Software Systems

Lion Wagner

University of Stuttgart
Institute of Software Technology (ISTE)
70569 Stuttgart, Germany

Abstract. This is my abstract.

1 Introduction

Todays programs are mostly highly customizable. Whilst buying or downloading a program a user already decides on which version of program (s)he wants to have. On Installation there are mostly multiple Options reaching from 'Language' to a selection of software features. Once a program is installed there are usually multiple startup, customization and configuration options provided for a customer. But having such a large amount of options brings some problems into the development of modern applications. We need to have a stable development strategy that can offer the creation of multiple similar products with little redundant code or components.In practice software product lines are used for this case
Furthermore there is the problem of performance prediction. For this we introduce some Definitions. This paper will use the definition of *feature* and *configuration* as they are described in [**?**] "[...], where a feature is a stakeholder-visible behavior or characteristic of a program." and "a specific set of features, [is] called a configuration". The amount of possible configurations naturally lies in $\mathcal{O}(2^n)$. This scaling makes it hard to test each singular configuration for its performance or correctness. Especially if the configuration under test is unpredictably chosen by a user [**?**]. Nevertheless this paper will only look at the non-functional performance properties of an application. Consequently efficiently analyzing a product's performance is only partially possible and behavior beyond that has to be predicted. This opens up the challenge of efficiently and accurately predicting performance. Over time a lot of methods in different disciplines have been prosed.
This paper aims to give a short introduction into the importance of software product lines (SPL) in regards to performance engineering and an overview over solutions for predicting and learning the performance of a configurable software system. mention references TODO

2 Software Product Lines Performance Influence

TODO Intro tarditionell präprözessoren oder compiler optionen oder startup optionen

2.1 Introduction into Software Product Lines

As mentioned above modern software systems are often highly configurable and customizable. To be able to provide such a large amount of customization, software engineers adopted the concept of *mass customization* inside a product line. A technique mostly known from the automotive industry[5].[5] further describes how a general product-line functions: There is a *common platform*, which is in itself expandable and holds all basic functionalities. It also provides interfaces for adding on parts (features). Some parts are mandatory and some are optional. For each part there might be different versions or variations that are interchangeable.

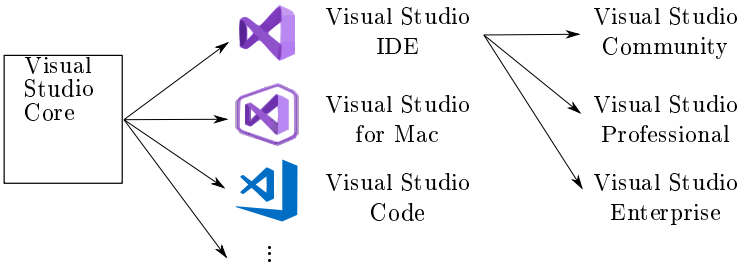

Fig. 1: Partial view of the Visual Studio Product Line [1]. (Assuming a Microsoft is using core libaries for its product. Which is likely since .Net Core runs on all major operating systems.)

Software product lines (SPL) inherit this behavior of reusing existing components to enhance the development process [3]. They are used to create a software product that is highly configurable and customizable. Products inside a SPL can even serve as a common platform for further customization. A more detailed look at the economical thought behind SPLs is given by Díaz-Herrera et al. [3]. An example of a SPL would be the Visual Studio product line from Microsoft as shown in figure 1. The common platform are some core libraries that provide shared code to all products of this line. The products themselves add platform or usage specific features (parts) to the core to offer a broad variation of products. The Visual Studio IDE even splits up further into 3 products that all serve a different purpose, but use same code base. Note that figure 1 only displays static variations of the product line. Almost all versions are additionally dynamically expendable via plug-ins and customizable via options and preferences.

2.2 Performance Influences of Feature Binding in SPLs

The following section is based on Combining Static and Dynamic Feature Binding in Software Product Lines by Rosenmüller et al. [6].
When designing a SPL one has a choice for each feature to either bind it statically or dynamically. A statically bound feature is compiled directly into the binary of a program. Its always available to the environment when needed but uses up binary space to do so. When dynamically binding a feature it is not a part of the core binary, yet a part of an external *feature library*. Once such a feature is needed the core loads the according library from an external source. This leads to the core binary being smaller but the size needed for each feature is increased by some meta-data (size, location, format, etc. of the feature library). Multiple features can be combined into one feature library for optimization. Feature libraries are also called *binding units*. Note that [6] uses the *decorator pattern* for the realization of dynamic feature binding.
To describe the effects of binding types we use the terms *functional* and *compositional overhead*. A functional overhead originates form features (or binding units) that are loaded inside a program yet never used. In turn a compositional overhead is found as meta data that is needed to describe dynamic behavior.
An abstract example: A program core consisting out of 3 features and uses up 2MB of (hard-drive) memory. We extract 1 feature that is rarely used into a dynamic link library with the size of 1.1MB. Afterwards our core is only 1MB large. We can already conclude that our original program had a functional overhead of 1MB by subtracting the old and the new binary size. When using the extracted feature our program uses up 2.1MB of memory. So we can say that our compositional overhead for extracting the feature is 0.1MB. That is the data needed to describe the location, format, etc. of the feature library.

daten aus paper einfügen TODO

Fig 18 einbauen TODO Coming back to binding types we have the problem of finding an optimal combination of static and dynamic bindings. Rosenmüller et al. [6] suggest to prefer static binding if no dynamic extensibility is required and resources are not limited. A full static bound program has no compositional overhead and thus only requires additional (persistent and transient) memory. However programs that are more limited in their resources may need to be designed to minimize the overall overhead. This is basically the same task of finding an optimal configuration of a program. Techniques regarding the analysis and prediction will come up later in the paper reference einfügen. TODO The compositional overhead of binding units can be predicted partially since feature libaries have a constant size header (e.g. 5KB per Dynamic Link Library {DLL}). A general guideline is to avoid a large number and a large size of binding units. Overlapping binding units as well as dynamic splitting and merging of binding units are possible optimizations. Rosenmüller et al. [6] also mention that "[...] there is no optimal size for a binding unit, a domain expert can define binding units per application scenario.".

3 Measuring and Predicting the Performance of Highly Configurable Systems

With such a large amount of customization options a

3.1 Automated Feature Interaction Detection

Automated feature detection is the most straight forward approach to predicting the performance of a highly configurable system. It was developed by Siegmund et al. [7]. Unlike other methods it does not depend on machine learning but rather tires to identify the performance impact of each feature and each features interactions. This method reaches a precision of 95% in predicting a systems performance [7].

Formulars The composition of using two (or more) units/features is denoted by $\cdot$. The interaction of two features is denoted by $a\#b$. By combining both we get a feature interaction:

$$a \times b = a\#b \cdot a \cdot b \tag{1}$$

This equation expresses, that when using both a and b we also need to consider their interaction $a \cdot b$.
Further Sigmund defines a performance function Π that is used to represent the performance value of a configuration:

$$\Pi(a \cdot b) = \Pi(a) + \Pi(b) \tag{2}$$
$$\Pi(a\#b) = \Pi(a \times b) - (\Pi(a) + \Pi(b)) \tag{3}$$
$$\Pi(a \times b) = \Pi(a\#b) + (\Pi(a) + \Pi(b)) \tag{4}$$

Following that the performance of a program $P = a \times b \times c$ can be written down as

$$\Pi(P) = \Pi(a) + \Pi(b) + \Pi(c) + \Pi(a\#b) + \Pi(a\#c) + \Pi(b\#c) + \Pi(a\#b\#c). \tag{5}$$

The Problem with the equations (2)-(5) is that they assumes that we can measure the performance of a feature in isolation. This is in general not possible [7]. Also we are still in the space of $\mathcal{O}(2^n)$ of possible configurations that we need to measure. To reduce this Sigmund et al. uses a *delta*.

$$\begin{aligned} \Delta a_C &= \Pi(C \times a) - \Pi(C) \\ &= \Pi(a\#C) + \Pi(a) \end{aligned} \tag{6}$$

Where C is a base configuration. This formula describes how the performance influence (Δ) of a in a configuration C can be calculated. Its either the performance difference between using C with and without a, or the performance influence of a itself plus the influence of the interaction between C and a. For a general approach Automatic Feature Interaction Detection (AFID) looks at

$$\Delta a_{min} = \Pi(a \times min(a)) - \Pi(min(a)) \tag{7}$$

and

$$\Delta a_{max} = \Pi(a \times max(a)) - \Pi(max(a)) \tag{8}$$

Where $min(a)$ is a valid minimal configuration not containing a but to which a can be added to create another valid configuration. $max(a)$ is a valid maximal configuration not containing a but to which a can be added to create another valid configuration.

For **automated detection of feature interaction** one first needs to define when a feature is interacting. For this Siegmund et al. [7] use the definition of

$$a \text{ interacts} \Leftrightarrow \exists C, D | C \neq D \wedge \Delta a_C \neq \Delta a_D. \tag{9}$$

$C = min(a)$ and $D = max(a)$ are chosen to find interacting features and to reduce the search space for C or D from $\mathcal{O}(2^n)$ to $\mathcal{O}(n)$. By measuring $\Delta a_{min(a)} = \Delta a_C$ and $\Delta a_{max(a)} = \Delta a_D$ for each feature some first information about their behavior can be obtained. If both values for a feature a are similar it does not interact with the features of $max(a) \backslash min(a)$. Otherwise a is marked as interacting. In both cases it can still interact with the features of $min(a)$. In total 4 measurements per feature are required ($\Pi(a \times min(a))$, $\Pi(min(a))$, $\Pi(a \times max(a))$, $\Pi(max(a))$).
Since most of the interacting features are known by now one can look for the groups of features whose interaction does have an influence on performance. Again the problem arises that there is an exponential number of possible combinations. Three heuristics are used to simplify the finding of these groups.

Pair-Wise Heuristik (PW): Most groups of interacting features appear in the size of two[7, 4]. So it makes sense to look for pair interaction first.

Composition of Higher-Order Interactions Heuristic (HO): Siegmund et al. [7] only look at higher order interactions of the rank of three. These can be relatively easily found by looking hat the results of the PW-Heuristik. Three features that interact pair-wise are likely to interact in a 3rd order interaction. For example, looking at features a, b and c, $\{a\#b, b\#c, a\#c\}$ all have to be non zero. Higher order interactions are not considered to prevent too many measurements.

Hot-Spot Features (HS): Based on [2, 8] Siegmund et al. [7] assume that hot spot features exist. " [...] There are usually a few features that interact with many features and there are many features that interact only with few features"

4 Measuring Option and Problems

5 Prediction

References

[1] https://visualstudio.microsoft.com/de/products/, Accessed: 10.05.2019.

[2] S. Apel and D. Beyer. Feature cohesion in software product lines: An exploratory study. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 421–430, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985851. URL http://doi.acm.org/10.1145/1985793.1985851.

[3] J. L. Díaz-Herrera, P. Knauber, and G. Succi. Issues and models in software product lines. *International Journal of Software Engineering and Knowledge Engineering*, 10(4):527–539, 2000. doi: 10.1142/S0218194000000286. URL https://doi.org/10.1142/S0218194000000286.

[4] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines, 2010. URL http://doi.acm.org/10.1145/1806799.1806819.

[5] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag Berlin Heidelberg, Tiergartenstraße 17, 69121 Heidelberg, 1 edition, 2005. ISBN 3540243720. doi: 10.1007/3-540-28901-1.

[6] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Combining static and dynamic feature binding in software product lines. technical report 13, 2009.

[7] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *In Proc. of ICSE*, pages 167–177. IEEE, 2012.

[8] C. Taube-Schock, R. Walker, and I. Witten. Can we avoid high coupling? pages 204–228, 07 2011. doi: 10.1007/978-3-642-22655-7_10.