Performance variability in software product lines: A case study in the telecommunication domain

Varvana Myllärniemi  
Aalto University  
Finland  
varvana.myllarniemi@aalto.fi

Juha Savolainen  
Danfoss  
Denmark  
juhaerik.savolainen@danfoss.com

Tomi Männistö  
University of Helsinki  
Finland  
tomi.mannisto@cs.helsinki.fi

ABSTRACT

In the research on software product lines, product variants typically differ by their functionality and quality attributes are more or less similar across products. To accumulate empirical evidence, this paper presents a descriptive case study of performance variability in a software product line of mobile network base stations. The goal is to study the motivation to vary performance, and the strategy for realizing performance variability in the product line architecture. The results highlight that the evolution of customer needs motivates performance variability; performance variability can be realized either with software or hardware variability strategy, with the latter often being prevailing; and the software strategy can be kept focused by downgrading performance.

Categories and Subject Descriptors

D.2.13 [Software Engineering]: Reusable Software

General Terms

Performance, Design

Keywords

case study, software product line, variability, architecture

1. INTRODUCTION

Heterogeneous customer segments and varying customer needs make it more challenging to develop software products. In order to cater for these varying needs, software product lines have emerged as an important paradigm for efficiently developing varying software products. In order to cope with this, software product line assets must explicitly manage and handle variability. Variability is the ability of a system to be efficiently extended, changed, customized or configured for use in a particular context [29].

Typically, products in a software product line differentiate from each other through their functional capabilities. In contrast, quality attributes, such as performance, security, reliability, are kept more or less similar, or at least their variability is not intentional and explicitly managed. Despite this, it is possible that different customer segments have different quality needs.

From the research point of view, the literature on quality attribute variability in software product lines remains rather scarce, and mostly lacks empirical evidence [23]. In particular, there is a lack of thoroughly reported industrial case studies of quality variability in software product lines [23].

This paper presents a post-mortem, descriptive case study [33] of performance variability within a 3G base station software product line. The goal is to study the intention and motivation to vary performance, and the realization in the architecture design. The research questions are as follows:

RQ1: Which characteristic of performance is decided to be varying?

RQ2: Why is performance decided to be varied?

RQ3: What is the strategy to realize performance variability within the product line architecture?

RQ4: Why is the strategy chosen?

A key contribution of this paper is to report on an industrial case study of intentional performance variability, thus accumulating the empirical evidence in the literature. Another key contribution is characterizing the relation of software and hardware. Performance variability can be realized either with software or hardware variability strategy, although several factors favour the latter. If there is a need to vary performance with software, downgrading provides a simple and effective mechanism. Finally, evolution of customer needs acts as one driver for performance variability and motivates the use of software variability strategy to cope with high rebinding costs. A lesson learned to the research community is that in case of intentional, differentiating performance variability, the differences between variants can be efficiently realized with hardware and by downgrading software. This is to contrast many research approaches, which focus on realizing differences in performance by managing the impact of software features to product performance.

The rest of this paper is organised as follows. Section 2 discusses previous work. Section 3 describes the research method. Section 4 describes the results of the case study, whereas Section 5 discusses the results. Section 6 covers the validity of the results, while Section 7 concludes.

2. PREVIOUS WORK

The products in the telecommunication domain are often developed and sold as product lines. Consequently, the
research has described several case studies in the telecommunication domain, e.g. [10].

Quality attributes can be defined as characteristics that affect an item's quality [7]. Quality attributes are often defined via attribute taxonomies [9, 2]. In the context of software product lines, special focus is on external quality attributes [9], which are visible in the product externally, to the customer or to other stakeholders. The software architecture is critical to their realization of many quality attributes: they should be designed in and evaluated at the architectural level [3].

Variability in software product lines has been a focus of intense research activity during recent years. It has been acknowledged that also non-functional properties of software can vary in a product line [14]. However, the research has mostly concentrated on variability of functional product characteristics, and variability of quality attributes has received less attention.

There are two literature studies that cover quality attribute variability in software product lines. A recent systematically conducted literature review [23] classified the way quality attribute variability can be represented, and compared these classes to the specific quality attributes that were proposed to be varied. Another literature survey [4] compared six different methods for modelling quality variability in software product lines.

There are studies that address performance or memory consumption variability in software product lines. Many approaches utilize feature modelling as a basis, for example, by calculating and optimizing response time [28], CPU consumption [32], memory consumption [31], or speed [1] out of feature configurations. However, it has been argued that performance and response time can be evaluated and optimized only variant-wise, whereas memory footprint is feature-wise quantifiable and thus optimizable over feature configurations [26]. Further, some approaches augment feature models with softgoals to find feature configurations that satisfy performance softgoals [13]. Finally, there are approaches that study performance variability at the architecture level and through architectural tactics. For example, variability of response time and data search space requirements are analyzed to evaluate the feasibility of implementing the variants under one architecture design [15, 16]. As another example, data and transaction volumes in enterprise software product lines may vary between clients and evolve over time, yet response times should be kept similar [8].

The empirical evidence on variability of performance and quality attributes in general is scarce, as noted in the literature review [23]. Firstly, three Finnish companies that have performance, security, and accuracy variability are described [24], but the analysis does not go into details, e.g., into how quality attribute variability is varied or realized in the software product line. Secondly, a case study of performance variability in Futhammer 3D mobile games is presented [22]. Since the capabilities of different mobile devices vary considerably, the performance and memory consumption of the game software is varied to maximize the gaming experience on all devices; this happens mainly via tuning the graphics of the games. Thirdly, a case study of performance and memory consumption variability in Intrada product line is presented [27]: the dependencies and impact of functional variation points to performance are used to guide the derivation process. Fourthly, there seems to be evidence about varying quality attributes in Japanese Intelligent Transport Services [16]; however, the varying quality attributes described include accuracy and coverage, which are more related to functionality. Fifthly, a brief description of BRIX reference architecture by DNC Software gives an example of how quality variability may cause architectural variation in a product line [6]: optional architectural drivers include adaptability, availability, suitability and interoperability. Finally, there are several studies, e.g. [19, 13, 31, 8, 18], that utilize an example of varying quality attributes, and mention or imply an industrial software product line. However, it is not known which parts of the examples are from the industrial cases.

To summarize, all empirical evidence in the literature has been quite brief, and no thorough treatment on the characteristics of industrial quality variability, or performance variability in particular, has been given. Further, all papers except [22] do not describe the method using which empirical evidence was gained, which means that the validity of the results cannot be assessed.

3. RESEARCH METHOD

The methodology used in this study was a descriptive case study [33, 25]. Qualitative methods permit the evaluator to study selected issues in depth and detail [25]. Hence a case study is a suitable approach for situations in which the phenomenon of interest is complex and the understanding of the topic is still lacking. This is exactly the situation with quality variability. Further, the research questions fall into the category of "why" and "how", which implies the suitability of case studies [33].

The unit of analysis [33] was performance variability within the domain of 3G mobile phone base stations, and as an embedded unit of analysis [33], decisions on performance variability design and requirements.

The case study was performed post mortem: the case product line was designed, but was discontinued before the product stage, mainly due to changes in the market situation. The post-mortem nature of this study made it possible to access confidential project documentation, thus making this single case an information-rich special case. At the same time, this single case was also representative in regard to the units of analysis, since similar findings apply to other base stations in the case company portfolio. Thus we are confident that the results can be generalized beyond this single case.

3.1 Sampling

The sampling method used for selecting the cases combined snowball sampling and convenience sampling. Snowball sampling [25] involves asking well-informed people of suitable information-rich cases. In this study, the first author asked the second author of his knowledge on cases that exhibit quality variability in a software product line. Convenience sampling [25] includes those cases that are easily accessible. Initially, three product lines were considered, but only one was included in the final version of this case study. In this remaining case, rich information was available, and it was possible to validate and publish the results.

3.2 Data collection

In general, there are six sources for data collection [33]: documentation, archival records, interviews, direct observa-
The main responsibility of a base station is to transmit and receive data from the mobile device. A base station typically contains a cabinet, an antenna mast and the actual antenna. Due to the limited functionality in base stations, a separate component called Radio Network Controller (RNC) is responsible for controlling several base stations. When a mobile phone moves out of the range of one base station, the corresponding RNC is responsible for handling soft handover to another base station. Together Node B base stations and RNCs form a Radio Access Network, which is responsible for handling traffic and signaling between a mobile phone and the Core Network in Figure 1.

IP-BTS base stations had functionality in three different planes: user plane, control plane, and management plane. Firstly, user plane is about carrying speech and packet data, which involves functionality such as the air interface transmission and receiving, physical layer processing, channel coding, encoding and interleaving, rate adaptation, spreading, and so forth. Secondly, control plane is about controlling and signalling, which involves functionality such as cell management and channel management. Finally, management plane is about administration and network management, which involves functionality such as start up, reporting, local management user interface, and management and updating of software and hardware configuration.

IP-BTS was an embedded product line with both software and hardware: some functionality was implemented by hardware and some by software. From the software point of view, IP-BTS was planned to be a reusable platform that worked on top of all IP-BTS hardware configurations. Furthermore, customers were supposed to create their tailored IP-BTS products by building own client applications on top of IP-BTS software platform.

The customers of the IP-BTS product line were mobile phone operators that invested in 3G infrastructure. The motivation that initiated the design of IP-BTS was the anticipated introduction of All-IP radio access networks. Because
of this, IP-BTS was designed to support various technologies, including traditional 3G, All-IP, and HSDPA, so that IP-BTS could be configured to support various kinds of radio access networks. In this respect, IP-BTS was designed to replace the first-generation Node Bs.

IP-BTS was discontinued before it ever reached full production stage. This is because the promise of All-IP never took off with operators, after they had made the initial investment to the first 3G networks. At the time IP-BTS was discontinued, it was at a prototype stage; this happened approximately ten years ago. In total, the timeline of the product was approximately two years from the initial planning to the termination of the project. Despite the fact that IP-BTS was terminated, it was designed to a detailed level, and some implementation existed. Since the architecture was designed and evaluated, there was documentation available for collecting data on the units of analysis.

### 4.2 Performance variability and its motivation

Research question RQ1 aims to define the particular quality characteristic that was decided to be varied. This is because quality attributes are often too general to be addressed without refining them into subattributes and characteristics, like is done in [9]. Further, the measure for the varying attribute is also used to communicate differences between product variants to the customers.

In the case study, the characteristic of performance that was decided to be varied was capacity. As one subconcern of performance, capacity is a measure of the amount of work a system can perform, and it is usually defined in terms of throughput [2]. For example, capacity could be defined as the maximum throughput the system can achieve, or as the maximum workload a system can handle. For IP-BTS, the capacity was defined as the maximum number of phone calls that the base station can serve at a time. Capacity could also have been defined as the amount of data one base station can route in a given time, but at the time of IP-BTS design, 3G packet data transfer was not that common.

Research question RQ2 aims to study the motivation to intentionally vary performance. This is because all explicitly managed variability adds to the complexity and cost of a software product line, and therefore there has to be a business reason to intentionally vary quality attributes.

In the case study, the motivation for capacity variation was twofold. Firstly, the customers, that is, operators, had different capacity needs for different product instances. Secondly, the vendors had an incentive to invest in realizing performance variability in the case study product line. In the following, we discuss these aspects.

From the operator point of view, the different capacity needs stemmed from the need to align the capacity of the base station with the traffic estimations. The operators aim at optimizing the number and capacity of base stations to cover a certain geographical area. When purchasing a new base station, the operators will have to estimate the number of phone calls within the area of each base station, and match the capacity of each base station to these estimates. Depending on the customer base and geographical location, these estimates are different between operators and individual base stations.

From the operator point of view, evolution created another source of variation in needs. The products in this domain are very long-lived, and they have to be able to cope with evolving needs after installation. This is also the case with capacity: although the usage of 3G networks was modest in the beginning, the traffic has exploded with the deployment of new 3G-enabled devices and mobile applications. If the usage, that is, the number of phone calls made, exceeds the available capacity of a base station, users will experience unacceptable congestion. To ensure customer satisfaction and leave room for future usage growth, operators typically leave some extra capacity compared to average usage, perhaps excluding some peak situations. But operators do not want to over-invest in capacity either, since the initial investment of setting up the 3G network is in any case high. Instead, operators wish that the capacity can be adjusted to follow the usage (Figure 3): as the usage of 3G networks evolves, operators can upgrade the base station capacity. Hence, the installed base station products should support evolution of customer needs by enabling rebinding of capacity.

From the vendor point of view, the reason to invest in realizing performance variability was that the capacity variants could differentiated in pricing. Firstly, the vendors can communicate their product characteristics to the customers precisely: the quality attributes of interest in this domain (e.g., throughput, latency, availability) can be described with explicit and unambiguous metrics that customers are able to

<table>
<thead>
<tr>
<th>RQ</th>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ1</td>
<td>Which characteristic of performance is decided to be varying?</td>
<td>Capacity, that is, maximum number of phone calls one product can serve at a time</td>
</tr>
<tr>
<td>RQ2</td>
<td>Why is performance decided to be varying?</td>
<td>Motivation from customer side: Initial differences in the estimated traffic for base stations, evolution of usage. Motivation from vendor side: Price differentiation, due to differentiating quality variability</td>
</tr>
<tr>
<td>RQ3</td>
<td>What is the strategy to realize performance variability within the product line architecture?</td>
<td>Strategy for realizing variability: Software variability strategy and hardware variability strategy. Software variability strategy: Licenses, down-grading by restricting capacity-related resources, such as channel elements. Hardware variability strategy: Different hardware in products, software utilizes abstraction and layers.</td>
</tr>
<tr>
<td>RQ4</td>
<td>Why is the strategy chosen?</td>
<td>Motivation for software variability strategy: Quick and efficient run-time rebinding, compared to the cost and difficulty of on-site maintenance for hardware upgrades. Flexibility of rebinding: more frequent variant re-binding, smaller initial investments for operators. Downgrading was architecturally simple and focused. Motivation for hardware variability strategy: Bill of Material (BOM), well-understood, ease of implementation and testing.</td>
</tr>
</tbody>
</table>
understand. Secondly, the customer can estimate the business value or the return of investment (ROI) of various quality levels, and a higher quality can be justified financially, for example, there may be a significant difference between availability 0.99995 and 0.99999. Because of this, the capacity variability was differentiating: different levels of capacity were of different value to the customer, and the customer was willing to pay more for better capacity. Hence, the vendor had an incentive in realizing performance variability.

4.3 Strategy for performance variability

Research question RQ3 studies the impact of performance variability on the product line architecture. This is because of the link between quality attributes and architecture [3]: a decision to intentionally vary performance may potentially cause architecture-wide variation [6].

In the case study, the strategy for realizing performance variability in the product line architecture was a combination of hardware and software variability strategy (see Figure 2). Like within all embedded domains, the base station products consist of both hardware and software, and the system performance is dependent on both. Traditionally, different levels of performance in the telecommunication domain have been achieved by having different hardware in the product variants. The hardware variability strategy is depicted in Figure 2: a variant with more efficient hardware yields better capacity. However, when the usage has grown over time and the capacity variant needs to be rebound, the hardware configuration of the base station must be updated, which requires on-site maintenance and is costly. Therefore, IP-BTS design used also software to vary capacity. It was decided that operators can buy capacity licenses to the base stations: when a need for a new capacity variant emerged, the base station capacity could be programmatical upgrade by purchasing a new license.

To summarize, the realization strategy for performance variability can be classified as being either hardware variability strategy or software variability strategy. In the hardware variability strategy, different product instances have different hardware, which causes the products to have different performance. In the software variability strategy, different product instances have the same hardware, but the differences in performance is achieved with software. Figure 2 illustrates these strategies in the case study.

Because of the evolving capacity needs, rebound considerations were an important aspect in the realization strategy for capacity variability. All variants in Figure 2 were available at the time of setting up new base stations. Smaller differences between variants were achieved with software variability strategy, while larger differences between variants were achieved with hardware variability strategy. After deployment, capacity variant rebinding happened mainly via software, and hardware upgrades were done only when the maximum license capacity was not enough. Thus, there were two possible rebounding times for capacity variability, and they depended on the strategy. If the rebound occurred between software strategy variants (see Figure 2), the binding time was runtime: licenses could be changed and applied on-the-fly. If the rebound occurred between hardware strategy variants (see Figure 2), the binding time was installation-time. In particular, the on-site hardware upgrades of base stations are typically handled by the network roll-out departments of operators, and they involve installing new hardware units, start-up, testing, and integration to network and taking into use. Because of this, hardware rebinding happens quite seldom: on average, the lifetime of a base station hardware is about eight years, although certain hot-spot areas may require more frequent hardware upgrades.

In the following, we discuss how these strategies were designed in the software product line architecture.

For the software variability strategy, the software product line architecture had two responsibilities.

Firstly, the software architecture was designed to support different license keys that can be purchased at runtime. Table 2 lists the components that were responsible for implementing license key and option management. When an operator wishes to purchase new license, she enters her license key to a network management system that manages base stations. Network management system then connects to the base station and invokes BTS O&M and its License Key

<table>
<thead>
<tr>
<th>Component</th>
<th>Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTS O&amp;M</td>
<td>System component in IP-BTS that is responsible of license keys and capacity variability. Part of control and management functionality in IP-BTS.</td>
</tr>
<tr>
<td>Option Manager</td>
<td>Supports runtime variability and options. Offers a database and corresponding operations for application level parameters.</td>
</tr>
<tr>
<td>License Key Manager</td>
<td>Uses Option Manager to set correct variability.</td>
</tr>
<tr>
<td>Resource Manager</td>
<td>Responsible for managing and restricting capacity resources, e.g., channel elements.</td>
</tr>
</tbody>
</table>
Manager, which utilizes Option Manager and its database of available options and current licenses.

Secondly, to match different licenses, the software architecture design utilized downgrading. That is, lower capacity variants were achieved by throttling down the maximum system capabilities via software. At the time of designing IP-BTS architecture, the exact mechanism for throttling down capacity was not decided. There were several alternative design decision for achieving this, but the most potential candidate was restricting the number of channel elements. A channel element is a resource that is needed to provide capacity for one voice channel; a voice channel can carry dozens of voice calls simultaneously. Channel elements are often used as a metric in the base station capacity planning, and they can thus be directly used as units of selling more capacity to the customers. A base station software can monitor the exact channel element consumption, and the consumption can be limited by the software. In IP-BTS software architecture, component Resource Manager (Table 2) was responsible for resizing channel element resources, since it supervised the operation of channel elements and related hardware resources. Further, \( N \) channel elements corresponded to a certain set of hardware resources, which meant that it was straightforward to add more channel elements by enabling the corresponding hardware resources.

For the hardware variability strategy, the product line architecture was designed to support varying hardware in the base station, that is, to support different numbers of processing units and memory, or different processor types. In order to support varying hardware, software utilized a layered architecture and abstraction of the software from hardware. Layered architecture style was used in the product line architecture to limit hardware visibility of software components. Property files were used for abstracting the properties of the hardware, so that the software components could use this abstraction without needing to access detailed hardware parameters.

### 4.4 Motivation for the realization strategy

Research question RQ4 addresses the motivation behind the realization strategy discussed in Section 4.3. The realization strategy directly affects the cost and difficulty of performance variability: a cost-efficient strategy acts as one motivation to intentionally vary performance (c.f. RQ2). Table 3 identifies some factors that motivated the decisions on the realization strategy, and in particular, decisions between the software and hardware variability strategies. Table 3 is not necessarily an exhaustive list, but other factors may also be applicable.

For the software variability strategy, the main motivation was related to the cost and difficulty of variability rebinding. Software variability rebinding happened via the operator purchasing a new license key and entering it to a network management system, and the new capacity variant was immediately available. In contrast, hardware variability rebinding required physical on-site maintenance. But the number of base stations is typically high (hundreds or even thousands), they may be geographically very scattered, and their accessibility is sometimes poor. Further, after the base station has been reached, the actual hardware reconfiguration requires considerable maintenance effort. Thus, the cost of on-site hardware upgrade was high, it took more time, and required that compatible hardware components would be available even after several years of installation.

Another benefit of the software variability strategy was the flexibility of rebinding depicted in Figure 3. Because of this flexibility, performance rebindings could be made more often, and operators could start with smaller initial investments. In Figure 3, the lines represent the evolution path and capacity rebindings of a base station. With software variability strategy, that is, the black line in Figure 3, the capacity upgrades could be made more often, and the capacity could be more adjusted with the usage. With software variability strategy, the operators could start with a lower capacity variant and upgrade as needed, which resulted in the smaller initial investment in Figure 3.

For the hardware variability strategy, the main motivation was the Bill of Materials (BOM): hardware played a major role in the cost of the products. In general, the hardware variability strategy means that different product variants have different BOM, whereas in software variability strategy, all product variants have the same BOM. If there is price differentiation, meaning that better quality variants have higher price, software variability strategy causes the price-to-production-cost ratio to be worse in cheaper, lower quality variants. Hence, in hardware variability strategy the vendor does not have to subsidize the lower quality variants by providing "too good" hardware and then downgrading it.

Another benefit of the hardware variability strategy was its efficiency: the performance of the product is largely affected by the hardware of the base station. Further, the effect of hardware on performance is straightforward, and the

![Figure 3: For operators, software variability strategy provides additional rebinding flexibility and smaller initial investments.](image-url)
Table 3: Some factors that motivated the decisions on the software and hardware variability strategies in the case study.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of design and implementation</td>
<td>The strategy should involve an easy and effective mechanism for increasing/decreasing performance. Scaling hardware is often straightforward to design and implement.</td>
</tr>
<tr>
<td>Ease of testing</td>
<td>The strategy affects the number of variants that need to be tested; the effect of hardware to performance is typically more predictable.</td>
</tr>
<tr>
<td>BOM (Bill of Material)</td>
<td>With hardware strategy, lower-quality variants have lesser hardware and smaller BOM compared to higher-quality variants. With software strategy, hardware is selected to support variant with the best quality, which implies high BOM for all products.</td>
</tr>
<tr>
<td>Cost of variant rebinding</td>
<td>For hardware strategy, the cost includes the maintenance task it requires to go to the site of the product instance and upgrade the hardware. For software strategy, this is negligible.</td>
</tr>
<tr>
<td>Cost of supporting variant rebinding</td>
<td>For hardware strategy, this cost includes ensuring a stock of spare parts, even for decades. For software strategy, this is negligible or relatively small.</td>
</tr>
<tr>
<td>The time it takes to rebind a variant</td>
<td>For hardware strategy, takes longer due to manual maintenance task involved. For software strategy, automatic, which means it can be practically instantaneous.</td>
</tr>
<tr>
<td>Flexibility of variant rebinding</td>
<td>Software variant rebinding is more flexible, and it can be done more often, which implies smaller initial investments for the customer.</td>
</tr>
</tbody>
</table>

principles of building scalable products are well-understood. Thus, hardware variability strategy is relatively easy to implement. This efficiency of hardware strategy is highlighted by the fact that even the software variability strategy relied partly on hardware to affect capacity: adding more channel elements involved that component Resource Manager enabled related hardware resources.

Another factor that affected the decisions on strategy was testing of performance variants. Due to large sums of money involved, the base stations must deliver the quality that the customer pays for, that is, the quality must be guaranteed. Further, the operator typically tests the products herself, possibly subjecting several competing products to test bench before making the purchasing decision. Therefore, the variants must be thoroughly tested. In the case study, the testing complexity of software variability strategy was eased by using a simple and predictable mechanism, which did not interfere with other software variability in the product. Thus, it was sufficient to test only the maximum, minimum, and selected throttled-down variants per one hardware configuration, instead of testing all possible license keys against all possible feature configurations of the base stations.

In general, several important factors in Table 3, such as BOM and ease of design, seem to favour hardware variability strategy. In the case study, the specific reasons to utilize the software variability strategy were the rebinding cost and difficulty, and flexibility of rebinding. Further, the testing and implementation effort of the software strategy was reduced by downgrading the capacity via channel elements.

5. DISCUSSION

In the following, we discuss the results and their general implications, and compare to related research.

5.1 Performance variability and its motivation

In general, software product lines have three different approaches for answering customers’ different quality needs:

1. No intentional quality variability: a software product line with homogenized quality. This approach, all product variants in a product line exhibit more or less similar quality attributes. For example, the product line architecture is designed so that all product variants exhibit the “the worst case” quality level [6], e.g., have the highest required capacity. Impact of other variability to quality attributes is not explicitly managed or optimized; consequently, any quality differences between products are unintentional. From the variability management point of view, this approach is easy, but it may be difficult to find an architecture design that satisfies all worst-case quality requirements at once [6]. Further, the design for the worst case customer segment may be expensive [6], especially if hardware costs are taken into account: customer segments that would be satisfied with lower quality may find the resulting price unacceptable.

2. Products with different quality, but not developed as a software product line. The different products have been intentionally designed to exhibit different quality attributes. However, the products are not developed as a product line, for example, under one software product line architecture. This kind of approach serves the needs of the different customer segments well, since architectures and design solutions can be tailored to specific needs, but at the same time is costly, since the level of reuse will be lower. This may be an option if the quality needs are very different or conflicting: no single architecture can accommodate the quality attribute needs of all product variants. Whether one architecture can accommodate varying performance requirements of the products is also analyzed in [15, 16].

3. Intentional quality attribute variability between products, products developed as a software product line. Product variants have intentionally different quality attributes, and the variability of quality attributes is explicitly designed and managed. This approach serves well differentiating quality needs, and gains economics by reuse, but the realization and variability management requires more effort. If the realization strategy for different quality attributes has larger architectural impact, must be able to handle architectural variation [6], which is challenging.

The case study was an example of the third approach: a software product line with intentional and managed capacity variability. Because of the explicit design for variability, different variants exhibited different capacity, and were even priced differently. This is because the capacity variability was differentiating, that is, of different value to the customers.

In general, the motivation to intentionally vary quality attributes as a product line depends on the specific quality...
attribute and the domain. In the mobile network domain, capacity is a good candidate for variability: it is tied to the business need and to the mechanisms that produce revenue to the customers. In contrast, the domain has quality attributes that are not good candidates for variability, for example, security is mostly dictated by the application domain standards. Further, within this domain, the price differentiation between quality attributes variants is possible: the customer is able to understand and characterize her quality needs using quantifiable metrics, and the customer can evaluate the return of investment (ROI) of various quality levels. This is in contrast to consumer domains, where the notion of "quality" is often described in imprecise and vague terms, and the customer is not able to precisely put value in different quality levels. Further, in many domains, the customer cannot verify the quality level, e.g., by testing the quality herself, and the quality of the products is not guaranteed. For example, the products in the internet domain typically do not guarantee a certain quality of service, but product quality is characterized as a best effort to a certain level. Because of all these issues, controlled value and price differentiation between quality variants may be more difficult in other domains.

The literature acknowledges that quality attribute variability may be driven by the evolution of customer needs. For example, in enterprise software applications, which similarly are long-lived products, non-functional requirements related to data and transaction volumes evolve over time [8]. Because of this, the concept of "variation point" has been suggested to be named as "evolution point" [17]. The evolution of quality attribute requirements may cause the need to do technological restructuring [8] or widen the scope of a product line into a product population [6]. Thus, it may be that certain quality attributes, such as performance, are good candidates for evolution-driven variability: as customer expectations grow over time, product lines with long-lived products have to support this with explicit variability.

In general, evolution has its drawbacks as a driver for quality attribute variability. When planning for the evolution, it must be decided which quality characteristics are implemented as variation points, and which are assumed to be fixed. However, it is difficult to predict how needs will evolve in the future: for example, the estimates on the number of mobile subscribers that use packet data and the estimates of packet data traffic might become outdated over the years. But adding variation points for every possible characteristic that can evolve is not feasible, since each variation point will add to the complexity of the system. The problem of forecasting which quality attributes the product line will have to support in the future is also mentioned in [30].

5.2 Realization strategy for performance variability and its motivation

The case study identified the software and hardware variability strategies for realizing performance variability. In particular, the results highlighted the importance of the hardware: several factors seemed to favour the hardware strategy, the software strategy was kept simple by downgrading performance, and even the software strategy was dependent on hardware. In the following, we discuss these aspects.

Firstly, there are several important factors that motivate the use of the hardware variability strategy in realizing performance variability. In the case study, the specific reason to utilize the software variability strategy was a combination of long-lived products, evolution of customer needs, and re-binding difficulty. In many consumer domains, new products are purchased as old ones are not powerful enough, instead of re-binding to a better quality variant. Further, the difficulty of re-binding the hardware may not be so evident in all domains. With cloudified services and infrastructures emerging, it may be possible to purchase more hardware resources on-the-fly, which again benefits the hardware variability strategy. In contrast, in many domains, Bill of Material (BOM) is of utmost importance, which favours hardware variability strategy: the vendor does not have to provide "too good" hardware for lower quality variants. This is especially relevant if the hardware is specialized and expensive, or if the products are mass-market consumer products with tight profit margins. However, a specific reason to utilize a software variability strategy may be the scope of the product line, that is, if the product line scope consists of software only. For example, in the product line of mobile phone games [22], the only way to adjust performance was to vary game software.

Secondly, to keep things simple, the case study utilized a downgrading software variability strategy. In general, it is beneficial to keep the software variability strategy simple and focused. For example, the difficulty and cost of architecture-wide variation [6] was avoided in the case study, since the downgrading mechanism did not create cross-cutting dependencies to other variation points in the product line architecture. Further, downgrading also kept testing effort relatively simple. In general, a more complicated software variability strategy may be difficult to test: the effect of software on system performance is not as predictable as with hardware, performance variability realized with software may interfere with other software variability, and consequently, all performance variants achieved with software should be tested against all other software variants.

Thirdly, the software variability strategy also relied on hardware: adding more channel elements involved enabling more hardware resources. To follow the classification in [11, 12], this kind of strategy can be called hardware dependent software variability strategy: the software variability is either dependent on or required to enable the hardware. Another example of hardware dependent software variability strategy is the use of clinical science keys that could enable newly develop hardware in [12]. To follow the classification in [11, 12], a software strategy could also be hardware neutral software variability strategy. For example, varying the type of the algorithm to vary performance [1, 26] utilizes purely software means to create differences in performance. At first glance, hardware dependent software variability strategy may seem to be quite similar to hardware variability strategy: does it matter whether hardware is physically installed, or programatically enabled? However, these two strategies have different implications and different motivations, for example, concerning the BOM and re-binding issues in Table 3.

To contrast these findings to the literature on performance variability in software product lines, there seem to be some differences. A large part of the literature concentrates on performance variability in feature models, for example, by optimizing performance over feature configurations [28, 31, 32]. In such approaches, performance variants are realized...
by selecting different feature configurations. This is in contrast to the case study, which utilized dedicated mechanisms, both in hardware and software, that were mostly independent on other variants. There are some proposals to vary performance in a controlled way, including varying architectural patterns and transformations [21, 5], but they often imply architecture-wide variation, which creates additional complexity.

To synthesize, at least two different kinds of software strategies for performance variability can be identified: downgrading and impact management. Downgrading means that the capabilities of the best possible variant are somehow limited via software. Examples of downgrading include this case, and the downgrading of the software product line [12]. Impact management means that different feature configurations are selected to match different performance requirements: this requires having some knowledge on the impact of the selected features to the overall software performance. An example of impact management involves optimizing response time by summarizing response times of leaf features, augmented with a tool support [28]. Another example relies on tacit knowledge (or explicit dependency modeling): during derivation, the selection between no matcher, normal matcher and large matcher algorithm is used to accommodate available memory [27].

Whether to use downgrading or impact management depends on the case. The benefit of downgrading is that it is more controlled, since there is one known and separate mechanism that can be adjusted to vary performance, and this mechanism can be tested in isolation. Impact management may be difficult in the case where the performance requirements are strict and must be guaranteed, e.g., due to price differentiation. Without tool support, impact management may result in a complicated derivation process as is described for Intrada product line [27]. Even with tool support, it may be difficult to define the impact of one feature on overall response time [26]. Further, if there are no functionally similar but performance-wise different feature alternatives, such as different versions of the same algorithm, it may be difficult to find a feature configuration that meets both strict performance requirements and is functionally acceptable. Finally, the resulting feature configuration must be tested per variant, as is described in [26]; this testing may be impossible for run-time rebinding. However, downgrading may not be applicable in quality attribute trade-off situations: the downgrading mechanism has to also improve the other conflicting quality attribute.

Finally, the case study utilized license keys to control software variability. In general, this practice is called a license key driven configuration [20]: the same product is shipped to all customers, and the code base configures itself based on the provided license key, allowing access to certain parts of the functionality while blocking others. For example, license keys are used for enabling and disabling features in trunked mobile radio product line [10] and in medical MRI scanner product line [12].

6. VALIDITY

The post-mortem nature of this case study has some implications on validity. One threat to validity is that the case study product line never reached production phase, and the product line was designed approximately ten years ago. However, the units of the analysis, that is, the performance variability and related design decisions, were established before the product line was terminated. Further, these threats are mitigated by the representativeness of the case study: even today the case company produces similar kinds of mobile telecommunication base stations, which utilize both software and hardware strategies for handling capacity licenses. The issue of being able to accommodate increasing traffic in mobile networks is even more crucial today. Thus, the results can be generalized beyond this single case study.

The data collection may pose another threat to validity. Typically, descriptive case studies collect data through interviews, which was not the case in this study. However, the lacking richness of open-ended interviews was alleviated by having an involved participant as an author, as well as through asking clarifying questions from the chief architects. The threat of biased observations from the participating author was mitigated by comparing to the documents and by confirming the findings with the chief architects.

Whether and how the results can be generalized to other domains or to other quality attributes is yet unknown. Firstly, many aspects of the analysis are specific to the domain, for example, high maintenance costs of hardware upgrades. However, there are similarities to other domains, such as embedded, long-lived products, business criticality to the customer, and the evolution of customer needs. Secondly, due to the different nature of quality attributes, the results may only be applicable to capacity and other performance-related quality attributes.

7. CONCLUSIONS

This paper presented a descriptive case study of performance variability and its design into a product line in the domain of 3G mobile base stations. The case exhibited intentional capacity variability that was designed into a software product line. Several important lessons can be learned from the case. Together, the evolution of the needs and long-lived products, as well as differences in how customers value and are willing to pay for the performance, constitute a good motivation to vary performance intentionally. To realize performance variability, either hardware and software variability strategy can be used. The realization strategies highlighted the importance of the hardware in performance variability: many important factors seemed to favour the hardware strategy, the selected software strategy was dependent on hardware, and the software strategy utilized simple and controlled downgrading. To contrast, the existing literature often concentrates on software in realizing performance variability, and especially, on studying the impact of different feature configurations on performance.

As a future work item, case studies of other quality attributes than performance are needed to understand the characteristics of quality attribute variability in software product lines. Further, the relation of software and hardware in realizing quality variability should be studied further. Also, identifying other software variability strategies in real industrial software product lines may prove to be beneficial. In this respect, a post-mortem case study like this has its benefits, since it is easier to get access to data and publish the results: another case that was considered for this study was dropped out due to the inability to publish the confidential results. To summarize, further empirical evidence on industrial quality attribute variability is needed to
understand the phenomenon better and lay out ground for more theoretical work.

8. ACKNOWLEDGEMENTS
Aki Nyyssönen, Jukka Peltola, Ari Evisalmi, and Juha Timonen from NSN are acknowledged for evaluating the validity of and commenting on our results. Anssi Karhinen and Juha Kuusela are acknowledged for ideas and comments.

9. REFERENCES