Scalability of Decision Models for Dynamic Product Lines

Gunnar Brataas, Svein Hallsteinsen
SINTEF ICT
7465 Trondheim, Norway
Gunnar.Brataas@sintef.no
Svein.Hallsteinsen@sintef.no

Romain Rouvoy, Frank Eliassen
University of Oslo
0316 Oslo, Norway
Rouvoy@ifi.uio.no
Frank@ifi.uio.no

Abstract

Product lines need decision models that guide the derivation of product variants satisfying specific requirements. In dynamic product lines, whose requirements vary during runtime, these decision models are also required to support automatic product reconfigurations in response to changing requirements. However, because of the combinatorial explosion of variants, such automatic decision making suffers from poor performance with many variants. In this paper, we introduce a mathematical formulation of this scalability problem. Based on the formulation, we discuss the limitations of existing approaches to support variants scalability. These limitations are addressed by our modular approach whose decision model combines (1) the use of utility functions, and (2) various optimisations of the search space. The analysis is supported by experience with the QuA and MADAM middleware, which apply this approach to supports self-adaptation based on dynamic product lines.

1. Introduction

Variation points and decision models are central concepts in software product lines. Variation points encode the variability of the product line and make possible the derivation of different variants from a common code base. The decision model guides the derivation of a variant that meets given requirements. The derivation of a product variant can be seen as a design effort within a constrained design space given by the product line architecture. The decision model encodes design knowledge relevant to this constrained design space. Basically, the process has three main steps: *i)* decide the requirements, *ii)* resolve the variation points, and *iii)* build the variant from the selected parts.

In classical applications of software product lines, the derivation of product variants has typically been done by engineers before delivery. Automation in the form of tool support has been leveraged to cut cost and calendar time, but one could always rely on human beings to agree with the customer on the requirements and to handle possible conflicts, ambiguities or lack of completeness in the decision model and to sort out build problems.

In our research we attempt to apply product lines to support the development of self-adapting applications for mobile use. Mobile settings are characterized by dynamic variation in both user needs and resource constraints and therefore need applications that are capable of adapting to such changes at runtime. The idea is to build self adapting applications as product lines with explicit variability and to achieve self adaptation by automating the product derivation process.

Utilizing product lines in the way explained above poses particular requirements both on the kind of variation points that can be used and on the decision model.

Firstly, the decision model must lend itself to automatic resolution of variation points that represents a good solution for a given set of requirements. “Good” in this connection means respecting the resource constraints and satisfying the user needs.

Secondly, we must be able to find a solution in sufficiently short time that the operation of the application is not disturbed in an unacceptable way.

Finally, on a mobile device we must be able to handle the coordinated adaptation of a dynamically varying set of applications competing about the available computing and communication resources. Although mobile devices are typically resource poor, most of them are multitasking units that supports the use of several applications concurrently and the user will stop and start applications to suit the task at hand.

This means that the decision models must be automatically combinable.

In this paper, we focus on the decision model and the automation of the process of finding a resolution of variation points that gives a variant that best fits a given set of requirements. Ideally, we want the best solution, but we may be willing to trade goodness for speed to some extent. The contribution of this paper is threefold:

- A precise description of the variant scalability problem illustrated with realistic scenarios
- An set of heuristics for reducing the search space of variants
- A modular decision model framework that combines the above heuristics achieving a greater reduction of the search space than each of them can achieve alone

The contributions in this paper represent ongoing research work, and we plan to do more extensive validation later on.

The rest of the paper is organized as follows. In Section 2, we present alternative approaches to representing decision models for dynamic product lines and discuss their merits. Then, in Section 3, we examine the approach based on utility functions in more detail focusing on the scalability issue. Section 4 reports experiences with a middleware platform for dynamic product lines based on utility functions. Section 5 describes some new propels and finally Section 6 offers some conclusions and suggestions for further work.

2. Alternative approaches

There are basically three main approaches which have been proposed for the description of decision models for self-adaptation: (i) situation-action approaches where adaptation rules specify exactly what to do in certain situations [1], (ii) goal-based approaches where goals describe high-level objectives that the self-adapting system should attempt to fulfill [2], and (iii) utility functions-based approaches where utility functions assign a utility value to each application variant as a function of application properties, context and goals [3][4].

We have chosen to apply utility functions in our MADAM approach [5] for several reasons. Firstly, the analysis of mobile scenarios shows that the selection of the "best configuration" is complex as it requires reasoning on dependencies between context elements, adaptation forms and concurrent forms. Utility functions enable us to express such dependencies. Secondly, unlike situation-action approaches, the actions needed to reconfigure to a new configuration are not explicitly described, but derived at runtime by the middleware. Thirdly, as will be discussed in more detail in the next section, a decision model for a set of applications competing for shared resources can be built from the model fragments associated with the components making up the applications at runtime.

The drawback however, is that the computational complexity of finding a variant is exponential in the number of variation points. In the following we investigate possible approaches to handle this.

3. Scalability of utility based decision models

3.1. The MADAM decision model

In MADAM [5], four types of variation points are supported:

- Several variants of each software component.
- Configuration parameters for software components, e.g. buffer size.
- Distribution or the deployment of software components on different computers in a distributed computing environment.
- Hardware variation points, e.g. *(a)* Alternative processor clock frequencies, affecting response times and power consumption. *(b)* Alternative display colour schemes for different light conditions. *(c)* Alternative communication technologies (*e.g.,* UMTS uses more power compared to GSM, but will also offer more bandwidth).

In addition, the middleware may choose to stop one or more applications if this increases the combined utility of the remaining applications.

Variation points are characterized by properties which may vary between their variants. These properties express functional and/or QoS properties of the provided service. In addition, software variants differ in terms of the computing and communication resources they need in order to execute. Hardware variants differ in terms of battery consumption.

The properties and the resource needs of an application variant are computed from the properties and resource needs of the component variants. The properties and resource needs of component variants are built by means of associated property predictor functions.

More formally, the basic concepts in the MADAM approach are represented in Figure 1.

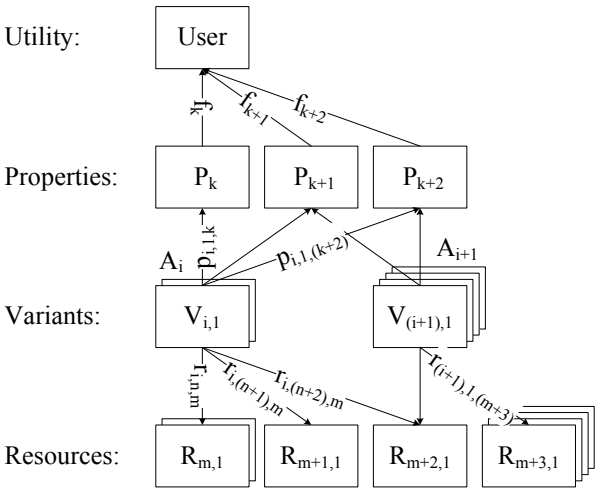

Figure 1. Relation between basic concepts.

In Figure 1, there are I applications $A_i \ldots, A_{i+I}$. Each application A_i has $N(i)$ variants $V_{i,1}, \ldots, V_{i,N(i)}$. Each variant $V_{i,j}$ requires $r_{i,j,m}$ resources of resource type m. In total, there are M resource types $\sigma_1, \ldots, \sigma_M$ For each resource type σ_m there are $Q(m)$ hardware variants. Each hardware variant q of resource type σ_m has its resource limitation $R_{m,q}$.

Examples of resource dimensions are processing requirements in terms of average instructions per second, disk and memory storage requirements, and network bandwidth requirements. For each resource type σ_m, the total resources consumed must be less than or equal to the available resources R_m:

$$\left[\sum_{i=1}^{I} \sum_{n=1}^{N(i)} r_{i,n,m} x_{in} \right] \le R_m, \quad m = 1, \ldots M$$

In this equation R_m, is the total amount of available resources of resource type σ_m for the selected hardware variant. $x_{i,n}$ is a decision variable which is set to 1 if this application variant n is used for application type i. At most one variant should be selected for each application type. However, we may not use *any* application types for an application:

$$\underset{i=1}{\overset{I}{\forall}} \left[\left(\sum_{n=1}^{N(i)} x_{i,n} \right) \in \{0, 1\} \right]$$

Each of the application variants satisfies one or more varying properties, but must in addition also satisfy fixed properties. There are K varying properties P_k, and each application variant $V_{i,n}$ has the properties $p_{i,n,k}$. The user weights the importance of the varying properties according to the preference function f_k.

Based on the amount of varying properties and the user preferences, a utility $u_{i,n,k}$ is calculated for each property for each application variant. This utility is a number between 0 and 1. The total utility for all application variants selected shall be maximized:

$$\max \sum_{i=1}^{I} e_i \sum_{n=1}^{N(i)} x_{i,n} \sum_{k=1}^{K} u_{i,n,k}$$

This sum is normalised based on the user preferences for each application, $e_i, \ldots, e_I$, where $\sum_{i=1}^{I} e_i = 1$. This last equation should be optimised, using the first two equations as constraints.

The weights on properties will in general differ between applications, and the functional properties will of course be different. If we take also this into consideration, then the equations will become more complex, but they will basically be similar.

3.2. Scalability

The algorithm for automating the selection of the best combination of application and resource variants works by going through all possibilities and selecting the one with the highest utility which also respect the resource constraints. If all variants could be combined freely, the number of possibilities S will be:

$$S = \left(\prod_{i=1}^{I} \left[N(i) + 1 \right] - 1 \right) \prod_{m=1}^{M} Q(m)$$

The possibility to stop applications is accounted for by adding one variant to the $N(i)$ combinations of components for each application. We have to run at least *one* application, accounted for by "- 1".

We also need to express the number of variants per application based on the number of variants per component $C(z)$ for all the $Z(i)$ components for application I, to get an expression for $N(i)$ $(1 \le z \le Z(i))$:

$$\underset{i=1}{\overset{I}{\forall}} \left[N(i) = \prod_{z}^{Z(i)} C(z) \right]$$

Combining the last two formulas we see how S grows exponentially with the total number of variation points.

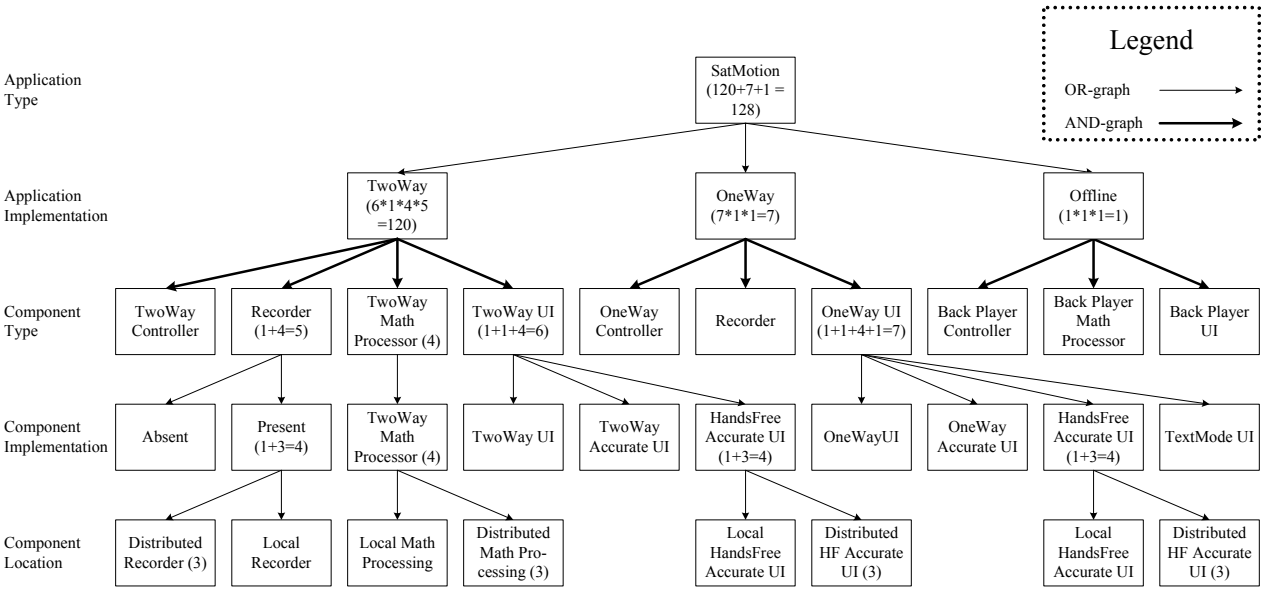

Figure 2. Variants in SatMotion using architectural constraints.

These two formulas are the worst case. In practice, there will typically be dependencies between variation points constraining allowable combinations. Within an application, the developer will introduce *architectural constraints*, effectively limiting the feasible combinations of component variants within one application, because some variants of a component depend on specific variants of other components. This effectively reduces the potential permutation of component variants. Also not all sets of application variants will depend on all resource types. Nevertheless, we still have an exponential growth of the number of possibilities with the total number of variation points.

Although this is not a pleasant property of our solution, it will only be a problem if the number of possibilities to consider exceeds what can be processed in the available time in typical cases. To get an idea about the size, in the next section we analyse two scenarios based on our experience with the MADAM middleware [5].

3.3. Scenario 1: SatMotion

The SatMotion application is used to align satellite antennas for satellite-based wireless communication. It consists of a client part running on a handheld device used by the installer during the alignment procedure, and a server part running on a server located at the installer company headquarters and controlling some signal analysis equipment connected to an antenna. The handheld sends a test signal through the antenna being aligned. The test signal is picked up by the antenna at the headquarters and analysed. The analysis result is sent back to the handheld via a separate internet connection. This application can be used in three different modes, OneWay, TwoWay and Offline, with different requirements regarding the internet connection to the headquarter. Furthermore, some components can be deployed either on the handheld or on other computers to which the handheld has network connections. For this scenario, let us assume that the four alternatives are the installers own laptop, a server at the customer site, in addition to an even more powerful server.

Assuming SatMotion is the only application of interest, and taking dependencies between variation points into account, there are 128 alternatives to consider as shown in the AND/OR graph in Figure 2. In this graph, AND means that all elements below an element in the tree have to be selected. In this case we multiply the number of variants. With OR in the graph, only one element needs to be chosen and we add the number of variants in the tree.

Without architectural constraints we will have 1260 variants, as each component has the number of variants as shown in Table 1. With architectural constraints we are then able to reduce the number of variants from 1260 to 128.

The total adaptation time consists of three elements:

- Detection of context changes: depending on the complexity of the context and on the filtering of context changes.
- Reasoning about changes and making adaptation decisions.

- Implementing the selected alternative — *i.e.*, to reconfigure. This time will depend on the number of components necessary to change, with increasing time for more component changes.

Of the above factors, only the middle one, the decision algorithm, depends on the number of variants. Under some conditions, the first and final may be considered to be fixed.

An ideal response time for the complete adaptation is less than 1 second. Less than 4 - 5 seconds is acceptable, and more than 10 seconds is in general unacceptable, but this depends on the utility increase. If the system is currently not usable, the user may be willing to wait for as long as one minute. On the other hand, when he only gets a marginal improvement, even 3 seconds may be too much (under the assumption that the user is not able to work simultaneously, in which case the adaptation time does not matter that much).

Experience with the MADAM middleware indicates that the capacity of the decision algorithm is in the order of 1000 alternatives per second.

Considering also that handheld devices on the market today do not have capacity to run much more than one application of this size at the time it may seem that we do not have a problem. However, the capacity of handheld devices increases and in the future more complex scenarios are likely.

Table 1 Component permutations in SatMotion.

Component	Variants
UI	6+7+1=14
Controller	1+1+1=3
Math processing	4+0+1=5
Recorder	5+1+0=6
PRODUCT	$14 \times 3 \times 5 \times 6 = 1260$

3.4. Scenario 2: Several applications

In this scenario the satellite installer in addition to using SatMotion also uses an IP telephony application and an email application to communicate with headquarters about future assignments, and also listens to music provided by a player on the device. Finally, a virus checking program is running in the background. If we assume that the additional applications have 10 variants each, we will in total get $129 \times 11 \times 11 \times 11 \times 11 - 1 = 1.888.688$ permutations of the application variants. We may assume that the CPU has five different clock frequencies and that we can choose between four types of internet connection: WLAN, UMTS, GSM and Bluetooth. The grand total of variants is then $1.888.688 \times 5 \times 4 = 37.773.760$.

3.5. Concluding on problem size

With in the order of tens of million alternatives and a reasoning capacity of in the order of 1,000 alternatives per second we are not able to complete within a reasonable time. We therefore need ways of dealing with the scalability problem. Our experience from the MADAM project seems to indicate that it is the number of simultaneous running applications which primarily causes the scaling problems, and not that each application has very many variants.

4. Existing solutions

Before we combine all possible variants for all variation points, we are in some ways in a *linear domain*. In the linear domain, we have some explicit information, which is lost when we do the expansion into the *exponential domain*. A clever approach will try to use this explicit information in the linear domain. As a simple example, suppose we have altogether 1,000,000 variants. If by using explicit information we are able to eliminate one variation point in the linear domain having 10 variants, then we reduce the problem to 100,000 variants.

4.1. Brute force approach

The brute force approach simply traverses the tree of all alternative variants and always keeps the best variant visited. The execution time is proportional to the number of variants, and grows exponentially with the number of variation points. It maintains indexed collections of variants for each variation point while recursively traversing the tree. The memory requirement will therefore be proportional to the number of variation points, i.e. it will not grow exponentially with the number of variants points, but will grow linearly.

4.2. Greedy approach

In addition to the brute force approach, MADAM also provides a greedy approach were the utility for all application variants of all applications are stored in a sorted table [6]. The applications with the highest potential utility are selected first. However, this choice can influence the available resources, which means that the variant utilities of the remaining applications need to be updated. Assuming that we have 3

applications containing 10 variants, we may have to evaluate the utility of all the 10+10+10 variants when selecting the application variant with the best utility for the first application. In the worst case, we have to update the application variant utilities for the remaining 10+10 application variants when selecting the variant for the second application and finally we may have to evaluate the best variant for the third application. Altogether, we will need to evaluate 60 application variant utilities in the worst case. Using the brute force approach, we will need to evaluate $(10+1)^3-1 = 1330$ variants, when running at least one of the three applications.

In this greedy approach, the selected application variants may quickly exhaust the available resources. Thus, the user will generally be able to run fewer applications than with an optimal brute force approach. A potential refinement of this approach is to select variants which have the highest ratio of utility to resource consumption [7].

4.3. Bellman-Ford algorithm

The Bellman-Ford planning heuristic in [7] considers components which have both a fixed utility and a fixed resource consumption, regardless of its relation to other components and their resource consumption. We may therefore replace their concept of components with our application concept.

A digraph is generated when searching for the combination of components with the best utility. In the x-axis, each variation point has its own position so that each possible component variant for the first variation point is placed at x-position 1, and each variant for both variation point 1 and 2 are placed at x-position 2. The y-position is the resource consumption for all combinations of variants. For each vertex, we find the edge with the best utility. With no resource constraints, there may theoretically be as many edges leading to the vertices with the highest x-coordinate as the possible permutation of all component variants, and the complexity will be even worse than using brute force. However, if there are resource constraints making many combinations invalid, then this approach becomes advantageous. Moreover, the digraph can be reused and updated when the components resource consumption and utility is unchanged or when only the available resources change.

We have to store the complete digraph for the permutations of all component variants. Hence, the memory requirement for this approach grows *exponentially* with the number of variation points. If a server is available, then this can be used as a digraph repository, but in a mobile setting, we cannot always assume network connection to a server. Besides, to access a server always consumes both energy and time.

4.4. Aura and Q-CAD

Compared to MADAM, components are not explicit in Aura [8], which focuses on selecting completely different variants of applications for each service. A natural consequence of the absence of components is a static application without any internal adaptation.

The utility function in Aura consists of two parts, the *functional preferences* and the *QoS preferences*. Using functional preferences, the user ranks all applications offering a particular service (*e.g.*, for editing, I prefer MS Word slightly more than Emacs, while Wordpad is close to unacceptable, modelled by the utility values 1.0, 0.9 and 0.2, respectively). In Aura, we know the application variant preferences before we need to evaluate the QoS utility functions. The application variant preferences then become an upper bound on the combined utility function. If we have two possible variants for each of the applications *a* and *b*, we may term four resulting application variant combinations: (1) $a_1 + b_1$, (2) $a_1 + b_2$, (3) $a_2 + b_1$ and (4) $a_2 + b_2$. Let us assume that the application variant preferences for these four variants are 0.8, 0.6, 0.4 and 0.2. If we find a combined application utility which is better than 0.6 when evaluating the QoS utility functions for the configuration $a_1 + b_1$, then we do not need to search the QoS utility functions for the other configuration variants, since the overall utility cannot be better than 0.6 in any of them.

In addition to functional preferences, Aura also gives a special treatment to warm-up time (see Section 5.5). The MADAM framework provides a more uniform and conceptually simpler treatment of these concerns, but then looses some flexibility.

The two-step approach in Q-CAD [9] resembles the two-step approach in Aura, but where the first step in Aura focuses on functional requirements only, additionaly non-functional requirements are used in the first step in Q-CAD (*e.g.*, when there is more than 30% battery left, then the resource must provide encryption, but when there is less battery, no encryption should be used).

4.5. QuO

QuO [10] models the functional relationship between a given context and the best variant. A good variant will in general be best for an area in the multi-dimensional space of all context properties. Some variants will never be best for any context properties

and can simply be discarded. To generate this mapping will require extensive resources, and may also consume much storage space. To use it will however be inexpensive. Nevertheless, this approach represents something to strive for, but may be difficult to implement fully.

4.6. Divide and conquer

The Divide and Conquer (D&C) approach [11] is aimed both at adaptation scalability and at variability. The main goal is the immediate and quick breakdown of an adaptation scenario with many independent users, machines and applications into smaller, logical and physical independent sub-problems that can be handled separately and in parallel.

D&C does not divide up until reaching single variation points. Conditions for stopping D&C can be that a sub-problem is small enough to be solved quickly by another method or that it cannot be divided into almost independent parts.

D&C works if the adaptation problems of the parts are small and independent enough to be solved, and if the combined solution to these parts is good enough to satisfy the users. In other words, D&C does not aim at finding a global optimal solution to the adaptation problem but at a scalable one that is considered good enough.

4.7. Genetic algorithms

Genetic algorithms are another kind of algorithms that are concerned about planning heuristics performance and scalability [12]. In these approaches, the search space is explored randomly during a fixed number of steps (called *generations*). For each generation, the weakest configurations of the population (a constant subset of relevant configurations in the adaptation space) are replaced by stronger configurations using *crossover* and *mutation* operators. The strength of configurations is based on the computation of their associated utility function. Thus, depending on *(1)* the number of generations and *(2)* the size of the initial population considered by the algorithm, the quality of the result varies in terms of optimality since the exploration of the adaptation space is unpredictable and mostly depends on random (or probability) operators.

5. New proposals

As the existing approaches are far from optimal, we explore some novel approaches in this section. Our proposal is to reduce the scalability problem by limiting the search space of variants. We dynamically combine several heuristics to filter irrelevant variants for a given adaptation.

5.1. Early filtering

Early filtering approaches aim at reducing complexity in the linear domain. This means that they exploit the knowledge of variants to discard as early as possible those that are unlikely to be selected by the heuristic. This filtering process can be achieved by considering *(1)* static properties defined by variants, and *(2)* dynamic properties related to context changes.

Filtering based on static properties

Variants can be early filtered by comparing their static properties (*e.g.*, memory consumption, network bandwidth required) to contextual information. This means that if a variant requires resources which are not available, it can be excluded from the search space. When considering the SatMotion application from Section 3.3, the loss of network connectivity implies the discarding of 127 useless variants to keep only 1 variant supporting a disconnected execution mode.

Early filtering can be implemented by using a trading service where look-up is based on required service properties for retrieving variants [13]. The variants are registered in the trading service based on their static properties, while the filtering of variants is ensured by the trading query used by the adaptation process to retrieve compatible variants.

Filtering based on dynamic properties

Variants can also be early filtered on properties of variants that are not registered in the trading service. Such filtering can be realized by associating to variants predicates whose variables range over context elements [13]. This solution complements trading based filtering described above and allows filtering on both static and dynamic properties.

A context dependency could *e.g.*, express a minimum memory requirement. Context dependencies of component compositions must be derived from the context dependencies of its constituent components, and it may not be statically known which components that will be selected.

In scalable video streaming, bandwidth requirements depend on the configuration of the video encoder. In many cases, it is not feasible to measure the bandwidth for every possible configuration. One alternative approach is to estimate the bandwidth

requirement using a suitable mathematical function fitted to a smaller number of measured points. Consult [14] for several examples of context dependencies applied to adaptable video streaming which effectively reduces the search space and ensures only feasible variants for the current context are considered.

5.2. Utility function analysis

The utility for each application variant and for each property k is a function S_k of the varying property and the weight, f_k, on each property:

$$u_{i,n,k} = f_k S_k(p_{i,n,k})$$

We then get this function for the total user utility:

$$U = \sum_{i=1}^{I} e_i \sum_{n=1}^{N(i)} x_{i,n} \sum_{k=1}^{K} u_{i,n,k}$$

$$= \sum_{i=1}^{I} e_i \sum_{n=1}^{N(i)} x_{i,n} \sum_{k=1}^{K} f_k S_k(p_{i,n,k})$$

$$= \sum_{i=1}^{I} \sum_{n=1}^{N(i)} \sum_{k=1}^{K} e_i x_{i,n} f_k S_k(p_{i,n,k})$$

The more combinations of variants and properties we are able to exclude, the better. Less important applications will contribute very little to the overall utility and can therefore be discarded in the scalability exploration. Similarly, less important properties can also be discarded. We can therefore look at weights of the product of application importance and quality utility functions, and focus on some of them. In the table below, we have four applications with four properties.

Table 2 Combing applications with properties

	App *1*, $e_1 = 0.5$	App *2*, $e_2 = 0.3$	App *3*, $e_3 = 0.1$	App *4*, $e_4 = 0.1$
Property *a*, $f_a = 0.4$	0.2	0.12	0.04	0.04
Property *b*, $f_b = 0.3$	0.15	0.09	0.03	0.03
Property *c*, $f_c = 0.2$	0.1	0.06	0.02	0.02
Property *d*, $f_d = 0.1$	0.05	0.03	0.01	0.01

From Table 2, we observe that the product of App *1* and Property *a* is 20 times larger than the product of App *4* and Property *d*. One way of using this information, is to only focus on App *1* and *2*, with properties *a*, *b* and *c*, in a first approximation. Reducing the number of applications to look at, is in the linear domain as defined in the taxonomy in Section 4, and will exponentially reduce the number of application variants. Inspired by the Aura heuristic, if we find a utility above a certain threshold, we can stop the search. Otherwise, we must also consider both property *d* and App *3* and *4*, in a step-wise fashion.

Issues for further work in this context are:

- Priorities of applications and properties will change depending on the context, but it may also be considered as stable for some time. They will probably change based on the situation the user is in *e.g.*, he/she is sitting quietly in a waiting room of the train station, or he/she is operating some quick tasks when walking to the train.
- We should define how many applications and properties to consider for each step. When we only consider some properties, several variants will differ in utility only with the properties which are not considered, and may be combined.

Even if we are not able to consider low priority applications together with high priority applications, it is still possible to consider them *after* the high priority applications. As an example, if we have 9 applications with 10 variants each, and there are 3 high priority applications and 6 low priority applications, then the number of variants to consider in the first iteration will not be 1 billion, but 1,000. In the second iteration, we consider only 1,000 variants, and in the third iteration we may consider the remaining 1,000 variants, if at all necessary.

In some ways, focusing on the most important applications first is a generalisation of the greedy approach in Section 4.2, because in the greedy approach we always consider one application at a time, whereas we here may consider several applications simultaneously. The utility function analysis may produce a better utility for the user, but will also cost more in terms of computing resources.

5.3. Utility function decomposition

To generalise the Aura heuristic [8] two conditions have to be met: *(1)* The application utility function itself must be decomposable, and *(2)* it must be possible to estimate one part of the utility function in advance (also known as *early binding/fixing*).

Thus, this heuristic decomposes the utility function into a *stable* part and an *unknown* part. The stable part of the utility function may be used as a basis for sorting variants, and may be also used to ignore many variants. Consequently, the unknown part of the utility function does not need to be evaluated for the excluded variants. For example, the variants whose stable part is

below the utility of the variant currently deployed can be automatically discarded to keep only the promising variants. Then, when evaluating these promising variants, the resulting utility is computed as the product of the stable part and the unknown part. If the unknown part of the utility function is 0.8 and its stable part is 0.9, then the total utility of the variant V_1 is 0.72. Since the unknown part of the utility function cannot exceed 1.0, no other variant can give a total utility above 0.7. Consequently, we can select the variant V_1 and stop the search for the best utility.

	Variants	V_1	V_2	V_3	V_4	V_5	V_6	V_7
Utility	Stable part	0.9	0.7	0.6	0.5	0.3	0.2	0.1
	Unknown part	x_1	x_2	x_3	x_4	x_5	x_6	x_7

There is a trade-off between how often the first part of the utility function changes and how efficient this heuristic is. This means that the stability of the first part of the utility function is the keystone of the heuristic. We must then know the frequency of adaptations —*i.e.*, changes in the execution context, or changes in the set of applications— because some of these changes can affect the first part of the utility function.

Aura uses the user's perceived utility of the functional properties as the stable part of the utility function. However, in a mobile environment, a user may have different functional requirements depending on his situation. For some situations, the user will benefit from more sophisticated functionalities.

The stable part of the utility function can also be adjusted dynamically by balancing it with the history of the unknown parts computed during the previous adaptations of the system. In this case, the early ranking of variants can evolve dynamically, based on the experience collected during the life cycle of the system. Thus, the adaptation heuristic will learn and memorise the best variants.

5.4. Distribution

Part of the planning may be delegated to more powerful servers, which may increase the feasible number of variants for example by a factor of 100. These servers may in the extreme case be Grid infrastructures, which may increase the factor by an additional factor of 1,000. The cost for the latter option will of course make this infeasible in the general case. Delegation to a more powerful server is only relevant if we have a network connection, but as described above, the number of variants is higher with network connection than without network connection.

To achieve the distribution of the planning process, the adaptation mechanism should deploy a planning engine on a remote server (or delegate to a running one) together with the variants and the parameters required to operate the adaptation heuristic.

If the current utility is fairly acceptable, the planning may be spread out in time and run as a background process, holding memory resources for a longer time.

5.5. Warm-up time

The time to reconfigure after deciding which variant to go for is also termed warm-up time [8]. In a distributed environment, the warm-up time will not be constant. If the new variant requires components currently residing on a server, it will take some time to acquire and install these components. Similarly, a configuration where only a few components are changed may be easier to configure than when all components are new. Also the user's tolerance for warm-up time will differ. If the current application variant can no longer run (*e.g.*, because the required network connection is no longer available), then the user is simply unable to do anything useful. In this situation the warm-up time should of course be as short as possible. On the other hand, in this case there is no real alternative to adaptation.

The cost of warm-up time could be modelled as one varying property. This serves to deflate the utility of a new potential configuration compared to the utility of the existing application. With only a marginal utility improvement the user is not willing to wait —*i.e.*, if the utility improvement is below a certain threshold, then we will not trigger an adaptation. In this way, warm-up time may be treated as a stable part of the utility function of a variant.

The inflation of the utility for the running application variants means that under some circumstances it is not needed to look for potential new application variants.

- If the utility of the running application variants have a quite high utility, then many alternative variants can simply be discarded. In the extreme case, no alternatives will manage to compete with the existing application variants, given the warm-up time.
- We know using one of the two decomposition heuristics that the utility of some application variants may never climb above the level required to beat the existing utility value giving the warm-up time deflation.

5.6. Decision model framework

The MADAM middleware defines a decision model framework (left part of Figure 3) that can be extended with the five heuristics described above that all in combination reduce the variants search space.

The heuristics introduced (right part of Figure 3) operate at different steps of the decision model to gradually reduce the size of the search space. By exploiting different forms of meta-data (*e.g.*, static/dynamic properties, dominating factors), these heuristics discard the variants that are not likely to be selected by the decision model.

The meta-data variety exploited by our approach does not imply that the application variants have to exhibit all the necessary meta-data to be efficient. Nevertheless, the performance of our approach directly depends on the amount and the quality of meta-data available.

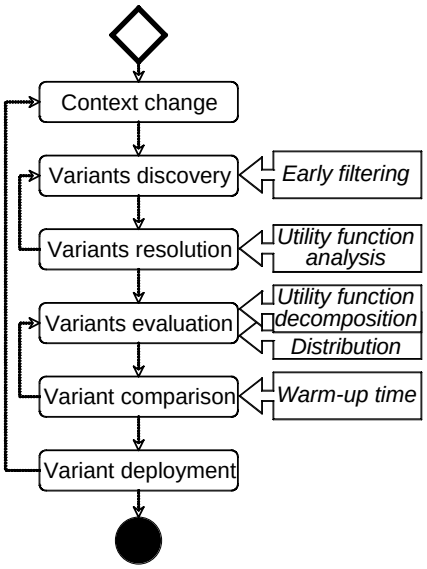

Figure 3: Combination of proposed heuristics

6. Conclusion and further work

This paper has suggested a set of heuristics to reduce the scalability issues related to the implementation of decision models in dynamic software product lines. In this domain, the number of product variants to consider during a dynamic reconfiguration can quickly exceed the capacity of the reconfiguration system due to the phenomenon of combinatorial explosion of variants. We also reported on some experience of applying some of these heuristics in the QuA and MADAM middlewares.

These are preliminary results that require further validation. In our future work we plan to implement and experiment with the proposed approaches in the context of the MUSIC project and further investigate the modeling and performance issues. This will include gaining experience with real-life commercial applications.

Acknowledgements

Thanks to partners of the MADAM and MUSIC projects for valuable comments; and in particular Ulrich Scholz, EML. This work was partly funded by the European Commission through the project MUSIC (EU IST 035166), see http://www.ist-music.eu/ .

References

[1] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure", *IEEE Computer*, vol. 37, pp. 46-54, 2004.

[2] J. O. Kephart and D. M. Chess, "The vision of autonomic computing", *IEEE Computer*, vol. 36, pp. 41-52, 2003.

[3] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility Functions in Autonomic Systems," ICAC'04.

[4] T. Kelly, "Utility-directed allocation", *Proc. Algorithms & Architectures for Self-Managing Systems*, 2003.

[5] Floch, J., et al. Using architecture models for runtime adaptability, IEEE Software, 23(2), pp. 62-70, 2006.

[6] MADAM, "*Theory of Adaptation - Specification of the MADAM Core Architecture and Middleware Services*". Deliverable 2.1, 2005. http://ist-madam.org

[7] M. Alia, *et al*., *A Component-Based Planning Framework for Adaptive Systems*, LNCS 4276, pp. 1686 – 1704, Springer, 2006.

[8] Sousa, J.P. *et al*. *Task-Based Adaptation for Ubiquitous Computing,* IEEE Trans. on Systems, Man, and Cybernetics, Part C, Vol 36, No 3, May 2006

[9] Capra L., Zachariadis, S. and Mascolo, C. *Q-CAD: QoS and Context Aware Discovery Framework for Mobile Systems*, IEEE Proc. of International Conference on Pervasive Services (ICPS'05), July 2005.

[10] Sharma, P.K. et al. Component-Based Dynamic QoS Adaptations in Distributed Real-Time and Embedded Systems, DOA'04, LNCS 3291, Springer, 2004.

[11] Scholz, U. and Rouvoy, R.. *Divide and Conquer – Scalability and Variability for Adaptive Middleware*. ESSPE'07, ACM Press, September 2007.

[12] Cao, L. , Li M., Cao, J. Using genetic algorithms to implement cost-driven web service selection, Multi-agent and Grid Systems 3(9):9-17, 2007.

[13] Amundsen Lundesgaard, S., et. al. "*Utilising Alternative Application Configurations in Context- and QoS-aware Mobile Middleware*" LNCS 4025, Springer, June 2006.

[14] Eliassen, F. "*Evolving Self-Adaptive Services using Planning-Based Reflective Middleware*". ARM'06, ACM Press, 2006.