Co-ordinated Utility-based Adaptation of Multiple Applications on Resource-constrained Mobile Devices*

Ulrich Scholz and Stephan Mehlhase

European Media Laboratory GmbH
Schloß-Wolfsbrunnenweg 33, 69118 Heidelberg, Germany
{Ulrich.Scholz,Stephan.Mehlhase}@eml-development.de

Abstract. Running several applications on a small, mobile device requires their constant adjustment to changing environments, user preferences, and resources. The decision upon this adjustment has to regard various factors of which the optimality of the result is only one: Further non-functional aspects including user distraction and the smoothness of operation have to be taken into account, too. This paper explains various events causing adaptation and details several non-functional aspects to be considered. It then presents Serene Greedy, a pragmatic approach for deciding upon adaptation and non-adaptation of simultaneously running applications in resource constrained, mobile settings. Finally, this paper discusses Serene Greedy by comparing it against other adaptation reasoning techniques for performance and the mentioned non-functional properties.

1 Introduction

With the emergence of ubiquitous computing, common future scenarios will consist in people moving around carrying general-purpose mobile devices, which they use extensively to assist both leisure and business related tasks. Naturally, the users expect their devices to run powerful applications as well as to run several of them simultaneously, serving different purposes at the same time.

For developers of mobile applications this scenario is very challenging. Users expect applications on these devices to have capabilities close to those of contemporary laptop PCs. But on top of that, such applications have to cope with various additional restrictions, such as sudden context changes, scarce resources, and limited device capabilities. Applications that meet these complex requirements have to provide the variability to adjust to the varying environment as well as a reasoning mechanism that selects the best fitting variant for every specific situation. Implementing these capabilities in addition to the application functionality is indeed a demanding task.

Developers can meet the challenges of a mobile setting by building on dedicated middleware platforms that provide reasoning and variability modeling support [5,9]. For example, utility-based adaptation reasoning allows to factor out the optimization mechanism from the business logic: The developer provides a function as measure for

* This work was partly funded by the European Commission through the project MUSIC (EU IST 035166) as well as by the Klaus Tschira foundation.

the usefulness of a particular application variant in a given situation; a reasoning mechanism then selects the optimal variant. While utility-based adaptation reasoning has been demonstrated to work well for individual applications, handling multiple simultaneous applications poses additional challenges that currently receive little attention.

The contributions of this paper are as follows: First, it describes issues arising when handling multiple applications, in particular performance decrease through excessive adaptations as well as non-functional problems such as stalling and user distraction caused by low-yielding re-configurations. It then presents Serene Greedy, a utility-based adaptation reasoning technique suitable for co-ordinated adaptation of multiple applications. Finally, it discusses the results of applying this technique in the context of the MUSIC middleware [9].

The next section introduces terms and concepts related to utility-based adaptation of multiple applications, while Sect. 3 describes non-functional aspects of adapting them. Section 4 gives a detailed analysis of different adaptation reasons as well as their influence and importance for maintaining the optimal usefulness. Section 5 presents Serene Greedy, a pragmatic approach to the adaptation of sets of simultaneously running applications. Section 6 demonstrates and discusses Serene Greedy. Section 7 reviews related work and gives further directions. Section 8 concludes the paper.

2 Utility-based Adaptation of Multiple Applications

Non-functional adaptation of multiple applications poses various challenges for the application developer. We describe these problems in the context of an execution environment for applications which facilitates adaptation to varying context [4]. We assume such an environment to follow an externalized approach to the implementation of self-adaptation where the adaptation logic is delegated to generic middleware working on the basis of models of the software and its context represented at runtime. We also presuppose the use of utilities as a means to specify the objectives that guide the adaptation logic [7].

2.1 Components, Variants, Context, and Resources

Applications are assembled of *components*, i.e., pieces of code, and several different collections of components, each called a *variant*, can realize the same application. At runtime, the knowledge required for adaptation is represented by *plans*, where each plan contains the code of a component and information how to assemble this component with others. Plans can be installed and removed at runtime, even those used by a running variant, so the set of available variants of an application can change dynamically.

Context is a set of values that describes the world from the view of the middleware (but not properties of the middleware itself). Applications state which context they depend on and the middleware provides the corresponding values on request. Context values change in accordance with changes in the world and in principle such changes are outside of the control of the middleware, the applications, and the user.

Resources, e.g., memory and CPU, are specific context values whose availability determines whether a variant can be executed in a specific situation. Each variant announces a specific, fixed amount for each resource that it requires. Resources are being

assigned to variants by the middleware and only the middleware can take them away. Consequently, running variants can continue to run regardless of resource changes.

An application variant is *valid* if it is given enough resources, i.e., if for each resource the required amount is smaller than what is available, and if it uses installed plans only; otherwise, it is *invalid*. In addition, a valid, running variant becomes invalid if the user removes a plan that is used by that variant.

2.2 Utility and the Utility Function

Many variants provide the same function to the user (e.g., participation in a picture sharing community), but often with different quality (e.g., with respect to reliability and bandwidth). The degree to which a particular variant has the potential to satisfy the user's needs is called the *utility* of that variant, which is a real number between zero and one (worst and best).

Each application variant has an associated *utility function* and the utility of a running application at a specific time point is given by evaluating the current variant's utility function on the current context. Formally, the utility function is a mapping $f_u : V \times C \mapsto [0, 1]$, where V is the set of variants and C is the set of possible contexts. As shorthand, we say *utility function of variant v* to refer to the utility function where the variant is held constant to v. Because the function f_u is arbitrary, the utility values of a variant under two different contexts are unrelated in the general case; likewise, the utilities of two different variants under the same context are unrelated.

When working with multiple applications, the user takes interest in them to varying degree. To allow the user to indicate his preferences to the middleware, it is possible to assign a *priority* to applications, which is a real number between zero and one (lowest and highest). We call an unprioritized utility a *raw utility*. Priorities enable the user to weight the relevance of applications according to his/her real needs: Giving low priority to an application with high utility indicates that this application does not help the user much despite it provides optimal service on an absolute scale.

The product of priority and raw utility of an application is the application's *weighted utility* and the sum of the weighted utilities of all running applications, normalized by the sum of their weights, is the *overall weighted utility* $u_{ow} = {\Sigma^n p_i u_i}/{\Sigma^n p_i}$, or simply overall utility. Utility-based adaptation assumes that u_{ow} equals user satisfaction and has the aim to keep this number high at all times.

2.3 Application States and Adaptation

Depending on the interest of the user, an installed application can be in use or not. Consequently, applications can be in two different states called *installed* and *running*; users can *start* and *stop* them. Figure 1 gives a state diagram of the possible transitions.

On starting an application, the middleware selects and configures its initial variant. After an event that might render the current variant sub-optimal, the middleware has to *adapt*, i.e., to re-consider all currently valid variants of all running applications together with their priorities. If necessary, it then has to exchange the current variant of some applications with another variant. The first step in this process is called *adaptation reasoning*, the second *re-configuration*. The latter step handles state transfer transparently.

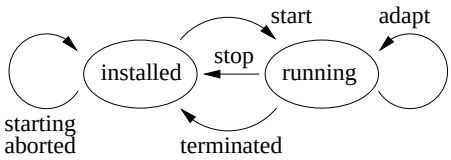

Fig. 1: State diagram of application states

Besides re-configuring a running application and letting it untouched, adaptation reasoning can also decide to *terminate* an application, i.e., to stop it without user request: If your running applications consume all available resources then running another one does not work. In the component based approach, termination is always necessary if an application does not have longer a valid variant as well as if a set of applications does not have a set of variants that is valid in combination (i.e., all sets contain an invalid variant). In the latter case, one or more applications of the set have to be terminated. For the same reasons as with termination, adaptation reasoning can *abort* the start of an application. The next section shows another possible reason for termination.

2.4 Indirect Dependencies between Applications

In this work, we consider the adaptation of multiple, independent applications running on the same device. Although such applications do not functionally depend on each other, there is an *indirect dependency* among them via their shared use of system resources. In resource-constrained settings, giving more resources to one application requires to take them away from another. Therefore, distributing the available resources is part of finding a valid variant set. The same is true if the weighted utility of an application changes. For this reason, finding the variant set with the highest overall utility in general requires considering all running applications.

Because of indirect dependencies, the maintenance of the optimal overall utility can cause termination: Consider the case of two applications that can run simultaneously, i.e., which have variants that are valid in combination. If a variant of one application has a high weighted utility but uses so much resources that no variant of the second can run then this variant alone might yield a higher overall utility than running any valid pair of variants. In this case, the middleware might stop the second application.

3 Non-functional Aspects of Adaptation

The utility-based adaptation approach takes the overall weighted utility as sole measure for the quality of its result. In other words, utility is thought to equal user satisfaction. Ideally, the middleware constantly adapts the running applications such that their current variants provide optimal overall utility at all times. Changes in the variant set remain unnoticed by the user except for modifications in application functionality. The user is expected to approve these user-perceivable changes because they are essential for maintaining high utility, i.e., user satisfaction.

Obviously, this approach is based on strong assumptions: It requires the application designer to encode the user's perceived application quality into a real value between

zero and one. But there are additional user-visible effects that influence user satisfaction, especially when considering multiple applications. Even if each adaptation constantly upholds an optimal utility, the side-effects of this mechanism are noticeable by the user. In the following we discuss the side-effects performance decrease, stalling of applications, fidgetiness, and application termination.

Performance decrease: Adaptation requires processing resources and if the reasoning runs on a machine that is shared with applications then these resources are not available for the latter. Because in the worst case, finding an optimal set of variants is exponential in the number of variants, the user might experience a significant performance decrease. For multiple applications this problem is particular prominent because it results in long search times even if each application has a moderate number of variants.

Application stalling: After the reasoning mechanism has decided upon the new variants of applications, they are re-configured by the configuration middleware. This process requires to suspend and to re-start applications, which the user experiences as stalling. In general, we can assume that the middleware limits the negative effects of re-configuration by detecting and discarding requests for unchanged applications.

Fidgetiness: User-visible changes, e.g., of the GUI and of application functionality, have the potential of annoying the user. In case of explicit user actions (e.g., disconnection of a device and change of location), the user "understands" and endorses a resulting adaptation. Drastic changes of unimportant applications that yield only slight improvements are less accepted. Systems that exhibit such behavior are fidget.

Consider the case of an application with two, visually distinct variants. The utility functions of all variants are almost the same, except for the dependency on a binary context property: Each variant is slightly better than the other for one of the property values. If the property quickly oscillates between these two values, the application changes accordingly, although the absolute utility improvement is negligible for each change. In such cases, suppressing re-configurations for small improvements reduces the fidgetiness of the system.

Application termination: The user can be displeased by applications that terminate on their own, i.e., without explicit user request. The same alienation can occur if the user starts an application the middleware decides it is better not to and aborts. As detailed in Sect. 2.3, this drastic measure must be taken if an application cannot be started or cannot continue to run and it might be the result of maintaining the optimal overall utility.

4 Adaptation Events and Affected Applications

Mobile applications have to react to changes in their environment. If such *adaptation events* can affect the utility of the currently running applications then the middleware may have to adapt. Adaptation events can occur at any time and in any number. An application can be affected by one or more events or it can be unaffected. For each of the various combinations, the consequences for the adaptation and the overall system differ. In the following, we first examine the different events and then classify applications according to the events they are affected by.

4.1 Adaptation Events

For our notion of utility-based adaptation, we can distinguish five classes of events:

Changes in Application Status: A user request to start or to stop an application results in an event of this kind.

Plan Changes: Plans can be installed and uninstalled by the user at any time. Adding a plan can affect the utility of an application because it possibly allows new variants that improve utility. Removing a used plan renders the using application invalid; removing an unused one has no effect.

Context Changes: If an application depends on a particular context element then changes in that element can cause a change of the utilities of all variants of that application. Because the mapping from context to utility is arbitrary and cannot be foreseen, finding the new best variant requires examining all variants. On the other hand, a running application can continue to run regardless of changes in context, although its utility might no longer be optimal.

Priority Changes: The priority of an application scales the raw application utility. Consequently, a change in priority does not affect the validity of a variant nor the ordering of the variant of an application regarding utility. Of course, changing application priority can render the current variant set sub-optimal via indirect dependencies.

Resource Changes: Variants of an application differ in their use of system resources. Because of indirect dependencies between applications, the amount of available resources determines the set of valid variants.

4.2 Classification of Applications Affected by Adaptation Events

An adaptation event can affect the running applications in different ways: For example, if the user stops one application, the others can continue to run without change. On the other hand, preserving the optimal utility requires to consider all applications in combination: If one application adapts, the others have to adapt, too.

In the following we define four classes in which we group the running applications in case of an adaptation event. To which class an application belongs depends on the kind of event and whether the application is directly affected or not. Loosely spoken, the classes are ordered top-down according to the “seriousness” of not adapting their applications. The classes are mutually exclusive and if an application could belong to several classes then it is included in the one mentioned first. For example, if an application is affected by a change of context and of resources, it is in class “Utilities Changed”.

Adaptation Required: Contains applications that are started or stopped and that use a removed plan. Applications in this class must be adapted by the middleware.

Applications not in this class do not require adaptation, i.e., all valid variants before the event are valid afterwards, although they might yield a low utility. For these applications the middleware can decide to skip adaptation reasoning and re-configuration.

Utilities Changed: Contains applications affected by a context change. Because the utilities of the affected variants change arbitrarily, it is unknown without adaptation whether there is a better valid variant.

Utilities Similar: Contains applications with new plans and with a new priority. Furthermore, it contains any application in case of a resource increase. The utility functions previously valid variants of an application with new plans are unchanged while

new variants might be available. The same is true for any application in case of a resource increase. If an application has a new priority, its variant set is unchanged but its utility function is scaled by a constant factor.

Unaffected: Contains all applications not directly affected by any adaptation event. Adapting an unaffected application is least likely to improve overall utility: Provided it is given the same amount of resources then adaptation reasoning will decide for the currently running variant again. Because of indirect dependencies between applications, it might still be useful to adapt unaffected applications along with affected ones.

5 The Serene Greedy Adaptation Reasoning Technique

This section presents an adaptation reasoning technique designed for resource-constrained mobile devices that pragmatically balances optimality versus the non-functional aspects of adaptation presented in Sect. 3. In principle, these aspects are relevant for the adaptation of multiple applications in general. As different settings require different adaptation mechanisms, we state several properties that we assume to be present in resource-constrained platforms. We continue with detailing two adaptation mechanisms that have been demonstrated applicable to solve the adaptation problem. Finally, we present the Serene Greedy adaptation technique.

5.1 Adaptation in Resource-constrained, Mobile Settings

Adaptation in a mobile setting is assumed to be performed by a single algorithm running as part of the middleware; there are no resources to perform extensive negotiations and to wait for external consultancy. The adaptation process is atomic from the viewpoint of the applications and potentially affects all applications controlled by the middleware.

Application adaptation has two parts: Adaptation reasoning and re-configuration. Changing a running variant as well as starting an application requires both steps. Adaptation reasoning is computationally expensive but does not stop applications that are reasoned about. Re-configuration is cheaper than reasoning but requires suspending and re-starting a running application. Performing adaptation reasoning for an application yields a list of all valid variants, sorted by utility. It always considers all variants of an application, caching and pre-processing between different adaptations are not used. Nevertheless, adaptation reasoning has to be performed at most once for an application during one adaptation process.

In the following, we refer with “applications” to those only that can potentially run after adaptation, i.e., to non-stopped running applications as well as started ones.

5.2 Brute Force and Greedy

The Brute Force adaptation technique [1] can serve as baseline for adaptation reasoning. It searches through all sets of variants of all applications. In particular, it always performs adaptation reasoning for all applications and it does not distinguish between adaptation events. Termination handling is taken into account by applying two optimization criteria: The first prefers large valid variant sets, the second optimizes overall

```
sereneGreedy
  c_sig := 0.1  /* Double value in the range [0,1] */
  A := set of all applications; sumP := 0
  while(|A| > 0)
    S := {t | a in A, t:=getSereneGuess(a, c_sig), t!=null}
    if(S == {})
      terminateOrAbortStarting(A)
      return
    else
      (p_a, u_a, v_a) := tuple in S with highest p_a*u_a
      A := A\{a}; sumP := sumP + p_a
      if(p_a*u_a/sumP >= c_sig || cannotContinueToRun(a))
        establishVariant(v_a)
      else
        continue(a)
```

Fig. 2: The Serene Greedy reasoning method

weighted utility. Therefore, Brute Force will prefer a large, low-yielding variant set over a single variant with high utility. If all applications can run, i.e., in a resource-rich setting, Brute Force will yield the optimal utility. On the down side, it is exponential in the number of variants: If it is applied to p applications with q variants each then it considers p^q variant sets.

The Greedy adaptation technique [1] performs adaptation reasoning on each application individually. It then selects applications one by one, preferring those that provide a valid variant yielding the highest weighted utility. If the resources are used up then it stops the remaining applications or aborts their start. Usually, Greedy evaluates much fewer variant combinations than Brute Force, i.e., only up to $p \times q$. A drawback of Greedy is that the selected application variants may quickly exhaust the available resources. Thus often, the user will be able to run fewer applications than with an optimal Brute Force approach.

5.3 Serene Greedy

The simplest way to prevent the non-functional downsides of adaptation is to not adapt. Reasoning techniques, such as Brute Force and Greedy, that always reason about all applications and that re-configure indiscriminately are prone to waste resources, stall applications, and annoy the user by being fidget. But obviously, not adapting – if possible at all – will likely result in a sub-optimal overall utility.

Serene Greedy tries to reach a pragmatic balance between optimality versus the non-functional aspects of adaptation in two ways: (i) It uses a notion of significance, i.e., it tries to make unforced change to the system only if the improvement is deemed to be significant and (ii) it tries to guess whether an adaptation is significant or not, based on the classification of applications affected by adaptation events (cf. Sect. 4.2).

Figures 2 to 4 present the Serene Greedy algorithm. Figure 2 shows the main loop, which collects guesses of achievable weighted utility, chooses the application with the best guess, and then either re-configures it or keeps it running unchanged. The latter option is taken if it is available and if a change would not yield a significant improvement.

```
getSereneGuess(a, c_sig): (p, u, v)
  if(adaptationRequired(a) || cannotContinueToRun(a))
    return adaptationReasoning(a)
  else
    v_a := running variant of a
    u_a := raw utility of v_a under the current context
    p_a := current priority of a
    if(!unaffected(a) && u_a<1-c_sig && p_a>=c_sig)
      return adaptationReasoning(a)
    else
      return (p_a, u_a, v_a)
```

Fig. 3: Making serene guesses

`cannotContinueToRun(a): Boolean`
Yields true if either application `a` is currently not running or if there is no valid variant of `a` that can run with the available resources; false otherwise.

`adaptationReasoning(a): Tuple of priority, utility, and variant`
Performs adaptation reasoning on application `a`. Returns the tuple `(p_a, u_a, v_a)`, where `v_a` is a valid variant that provides the optimal raw utility `u_a` and `p_a` is the priority of `a`. It returns `null` if there is no valid variant. On the first call to this function, all variants of `a` are considered; subsequent calls simply return a cached result.

`establishVariant(v_a)` and `continue(a)`
After applying the first method, `v_a` is the running variant of application `a`: If `v_a` is currently running then it remains untouched; otherwise, the method re-configures or starts `v_a`. The second method keeps the running variant of application `a` unchanged.

`adaptationRequired(a): Boolean` and `unaffected(a): Boolean`
These functions yield true if application `a` is classified in the respective class according to Sect. 4.2; false otherwise.

`terminateOrAbortStarting(A)`
All applications in set `A` are terminated if running; otherwise starting them is aborted.

Fig. 4: Additional functions and methods of Serene Greedy

Figure 3 supplies the serene guesses: If an adaptation is required, it reports the resulting variant and its utility. For applications in class “Unaffected” it hands back the running variant and its current utility. For applications in classes “Utilities Changed” and “Utilities Similar” adaptation reasoning is performed only if their current raw utility is less than excellent ($u_a < 1 - c_{\text{sig}}$) and their priority allows for a significant weighted utility ($p_a \geq c_{\text{sig}}$). Figure 4 describes the functions and methods used by the algorithm.

6 Discussion

In this section we demonstrate and discuss the Serene Greedy adaptation mechanism regarding the non-functional aspects detailed in Sect. 3. The demonstration uses artificial applications that clearly exhibit the effects under discussion. Of course, an evaluation with real applications and users would be preferable. But because the impact of, e.g., fidgetiness is subjective, such studies require large samples and are out of scope of this

paper. The applied algorithms were implemented as part of the MUSIC framework [9], which provides an externalized approach as it was described in Sect. 2.1. The source code of MUSIC and of the applications used in this section is available online.[1]

For pragmatic reasons, MUSIC does not consider resource changes as adaptation causing event, i.e., it categorizes an application that is only affected by a resource change as "Unaffected" and not as "Utilities Similar". MUSIC currently disregards these events because they happen frequently and considering them would result in constant adaptation reasoning. Note however that the arguments given in Sects. 3 and 4 remain valid despite the change, Serene Greedy is demonstrated according to Sect. 5, and resources are heeded during adaptation reasoning. Section 7 details an extension that would allow taking advantage of resource changes.

Serene Greedy requires as input a value for significance. The higher this value, the more applications remain unchanged during adaptation. Certainly, this value should be chosen in accordance with the applications under consideration: Important changes should yield a significant utility increase. For the following demonstration we have decided for a significance value of $c_{\text{sig}} = 0.1$.

6.1 Performance

The performance of an adaptation mechanism strongly depends on the number of evaluated variants. A multi-application setting allows reducing this number in an easy way by avoiding the adaptation of unaffected applications. We demonstrate this effect with four applications in a resource-rich setting, where only two of them depend on a specific context element. Each of the applications has 10 different variants. After a context change event, we measure the total number of evaluated variants and the total number of applications that were subject to adaptation reasoning. We also measure the average adaptation time on a normal PC. The results of this scenario are summarized in Table 1.

Table 1: Performance of different reasoning algorithms

Algorithm	Avg. time (ms)	# Variants	# Apps reasoned about
Brute Force	117.72	10 000	4
Greedy	15.70	40	4
Serene Greedy	9.41	20	2

The numbers clearly show the performance differences of the three algorithms: Brute Forces evaluates an exponential number of variants, which results in a large run time. Furthermore, Brute Force and Greedy reason about all applications, while Serene Greedy reasons only about two. Note that all three algorithms re-configure only the two applications that are affected by the context change.

Although certainly synthetic, the given scenario is realistic. The number of applications is kept small, as on small devices most users do not use too many applications at the same time. Also, applications usually differ in their context dependencies. Note that using a resource-constrained setting, e.g., a small mobile device, would show an even more significant performance gain of Serene Greedy over the other two.

[1] `http://developer.berlios.de/projects/ist-music`

Table 2: Priorities, utilities, and memory requirements of applications A and B

Variant	Priority	Utility (context "1")	Utility (context "2")	Memory requirement (kB)
A_1	0.5	0.9	0.8	100
A_2		0.8	0.7	40
B_1	0.15	0.8	0.1	120
B_2		0.3	0.4	70
B_3		0.4	0.3	80

6.2 Fidgetiness and Stalling

Fidget applications waste resources for reasoning and often stall. The following experiment shows how Serene Greedy improves on both effects while loosing only little in utility. The set up consists of two applications A and B in a resource-constrained setting that does not allow all variant combinations to be valid. The two applications depend on a context element that oscillates between the values "1" and "2". Furthermore, application A has a higher priority as application B. Table 2 gives the priorities of the applications and the utility values of their variants, as well as their memory requirements. The available memory is 190 kB.

Scene 1 in Fig. 5 shows the initial situation of the experiment, using context value "1". The variant set with optimal overall utility $\{A_1, B_1\}$ is not valid because of resource constraints. Therefore, Brute Force selects the set $\{A_2, B_1\}$. Greedy and Serene Greedy select both A_1 because it has the highest weighted utility. We also assume that they both select B_3, so that they initially yield the same overall utility.

After changing the context to "2" (scene 2), all reasoners change to set $\{A_1, B_2\}$. Brute Force adapts both applications, the others only one. On the following context change back to "1" (scene 3), their behavior differs: Brute Force selects its initial variant set, thus re-configuring both applications; Greedy re-configures application B; while Serene Greedy does not re-configures at all. The additional change from B_2 to B_3 yields an increase of overall utility of about 0.04, which is insignificant. Regarding overall utility, Brute Force always yields the best values: 0.8, 0.71, and 0.8, while the others are sub-optimal under context "1". After reaching context "1" the second time, Greedy is slightly better than Serene Greedy, because the latter waives the re-configuration to B_3.

The experiment shows that Serene Greedy re-configures, i.e., stalls less than the other two while yielding sub-optimal but comparable utility compared to Greedy for applications in classes other than Unaffected . The latter observation is supported by an analysis of the maximal difference between the two reasoners in case of a sub-optimal decision of Serene Greedy for a single application a after a sequence of identical ones. This difference is at most the overall utility using the optimal variant minus the one using the current, i.e., $u_{ow}^{\max} - u_{ow}^{\text{cur}} = p_a \times (u_a^{\max} - u_a^{\text{cur}}) / (p_a + p^{\text{sum}})$, where $u_a^{\max}$, u_a^{cur}, and p^{sum} are the current maximal raw utility of a, the current utility of the running variant of a, and the sum of priorities of all applications accepted prior to a, respectively. Because of $c_{\text{sig}} > p_a$ and $u_a^{\max} \geq u_a^{\text{cur}} \geq 1 - c_{\text{sig}}$ (cf. Fig. 3), each individual decision of Serene Greedy is at most by $c_{\text{sig}} / (1 + c_{\text{sig}}^{-1} \times p_{\text{sum}}) \leq c_{\text{sig}}$ below the optimum achievable in this situation. Multiple fidget applications are uncritical, too, because with an increasing number

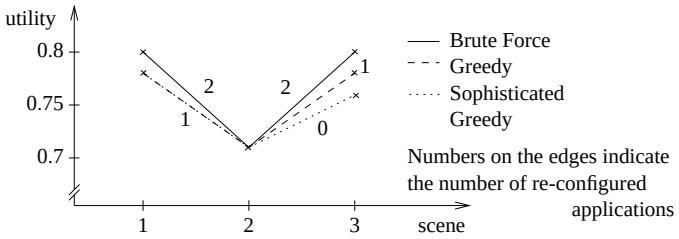

Fig. 5: Behavior of the different adaptation mechanisms

of running applications their individual contributions decrease. The error term reflects this correlation by an increasing denominator.

As demonstrated, Serene Greedy can reduce the fidgetiness when adapting multiple applications. Nevertheless, this problem cannot be solved by a reasoning technique alone: A fidget system disturbs the user, but how much it does so is subjective and also depends on the applications. For one user, a perceivable change can be important while another does not care. Likewise, a slight improvement in utility can make the difference for one user but not for another. In the end, overcoming fidgetiness will require coordinating the applications as well as foresight by the developer and user involvement.

6.3 Application Termination

In a resource-scarce setting, the involuntary termination of applications can be unavoidable. The same can result from maintaining the optimal utility. As such behavior will likely annoy the user, it has to be minimized. Unfortunately, the problem of keeping a maximal number of applications running is a variant of the Knapsack problem and thus $\mathcal{NP}$-complete [6]. Brute Force, which is optimal in this respect, evaluates an exponential number of variant combinations. Greedy and Serene Greedy perform better; consequently they are sub-optimal regarding utility and stop more applications.

In general, the problem of termination becomes more prominent for systems which run adaptation techniques that limit the search effort in favor of performance: They might miss variant sets that keep applications running and therefore terminate more applications than necessary. Thus, performance and susceptibility to termination have to be considered in combination when choosing an adaptation algorithm.

At least, an adaptation mechanism should keep important applications running, i.e., those with high weighted utility, and notify the user for those it decides to stop. Serene Greedy adheres to the first requirement by preferring high yielding applications. Clearly, the second one is a task for the middleware as a whole.

7 Related Work and Further Directions

According to the roadmap presented in [2], our work on non-functional effects addresses challenges for the engineering of self-adaptive systems, among which are understanding the various aspects of self-adaptation, such as user needs and system properties, as well as classifying the modeling dimensions available. Serene Greedy falls in the category of effects of adaptation: It prefers the optimality and operationally of applications

with high utility over those with low. Its decisions are predictable because high value applications are likely to continue in case some applications must be stopped, it reduces overhead by not adapting low value applications, and it increases resilience because remaining operational is preferred over an insignificant performance gain.

Systems that take into account non-functional aspects when adapting usually aim at more "high-level" aspects as the ones covered by this paper: Cheng et al. [3] present a language to describe non-functional objectives and information about the system, which allows an adaptation mechanism to take the described aspects into account. Aura [5] aims at selecting optimal providers for resources and other characteristics (e.g., security) to keep the user undistributed. Pladian et al. [8] try to improve user experience by anticipating future resource needs of multiple running applicationsand takes into account the overhead of adapting. Serene Greedy tries to limit this overhead by deciding to not adapt and not re-configure.

While these methods address important non-functional aspects of adaptation, they are nevertheless susceptible to the "low-level" ones related to the actual search (or non-search) through the available set of variants. On small mobile devices – and for future demanding applications – we still consider these search-related issues of high importance because the optimality of a decision can be easily outweighed by the effort required to search for it.

Sykes et al. [10] regard the frequency of adaptation and the related delay when adapting single applications, thus improving on stalling and user distraction. Our approach considers the adaptation of multiple applications, thus taking into account the fidgetiness caused by adapting unimportant applications.

The remainder of this section presents directions in which we plan to extend Serene Greedy. As explained in Sect. 6, its implementation as presented in this paper does not use resource change events, so it misses the chance to gain valuable increases in utility on resource-constrained devices. We plan to remove this limitation as follows: Define an adaptation delay and, for each resource, a significant amount. For each resource, when reasoning about an application, record the need of the optimal variant. If more than this amount is available then disregard changes. Otherwise, on significant changes, adapt after the given delay. With this strategy, resource changes are taken into account but the undesired non-functional effects are limited.

A way to limit stalling by re-configuration is to stop and re-start only those parts of an application that differ between the variants; unaffected parts can continue to run. Imagine an application with a GUI and a business component. If adaptation decides to exchange the latter part but leaves the GUI unchanged then the user might not notice the change. Realizing this technique transparently requires the application designer to provide information about which parts of an application can be adapted independently, new reasoning techniques that penalize the perpetuation of components, and administration code that buffers communication between re-configured and unchanged components.

As mentioned before, the utility function can be kept (in part) in memory, which allows filtering variants beforehand and thus saves re-evaluating them again. Consequently, the time to decide upon a good variant set can be reduced. On the down side, the memory used for storing this information is not available to the applications such

that this approach might yield sub-optimal utility. We plan to explore the use of more structured utility functions, where the same effect can be reached with less memory.

8 Conclusions

The operation of multiple adaptive applications on small, mobile devices requires handling them in a co-ordinated way. Otherwise, non-functional effects of the adaptation process can obstruct and annoy the user. This paper improves towards this end in three ways: (i) We discuss the problem of indirect dependencies between applications in resource-constrained settings and identify resulting non-functional aspects of adaptation, in particular the problem of application termination and of fidgetiness, i.e., disturbing adaptations of unimportant applications. (ii) We then analyze different events that can cause the applications to adapt. This analysis allows to classify running applications according to the consequences to the system and to the utility of not adapting them. Finally, (iii) we present Serene Greedy, an adaptation method based on the given classification. We compare an implementation of this method with two other adaptation techniques for the MUSIC middleware. The results show that Serene Greedy reduces stalling and fidgetiness of adapting multiple applications while providing improved performance and a utility comparable to the Greedy reasoning technique.

References

1. Gunnar Brataas, Svein Hallsteinsen, Romain Rouvoy, and Frank Eliassen. Scalability of decision models for dynamic product lines. In *Proceedings of the International Workshop on Dynamic Software Product Line (DSPL 07)*, September 2007.
2. Betty H.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee et al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, LNCS 5525, pages 1–26. 2009.
3. Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2006.
4. Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjørven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.
5. Davic Garlan, Daniel P. Siewiorec, Asim Smailagic, and Peter Steenkiste. Project Aura: Towards distraction-free pervasive computing. *IEEE Pervasive Computing*, 21(2), 2002.
6. Richard M. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
7. Jeffrey O. Kephart and Rajarshi Das. Achieving self-management via utility functions. *IEEE Internet Computing*, 11(1):40–48, 2007.
8. Vahe Pladian, David Garlan, Mary Shaw, M. Satyanarayanan, Bradley Schmerl, and Joao Sousa. Leveraging resource prediction for anticipatory dynamic configuration. In *SASO'07 Conference on Self-Adaptive and Self-Organizing Systems*, 2007.
9. Romain Rouvoy, et al. MUSIC: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In *Software Engineering for Self-Adaptive Systems*, LNCS 5525, pages 164–182. 2009.
10. Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Exploiting non-functional preferences in architectural adaptation for self-managed systems. *ACM Symposium on Applied Computing, Track on Dependable and Adaptive Distributed Systems*, 2010.