Meeting the Software Engineering Challenges of Adaptive Mobile Applications

Rajesh Krishna Balan João Pedro Sousa
Mahadev Satyanarayanan

February 2003

CMU-CS-03-111

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was supported by the National Science Foundation (NSF) under contracts CCR-9901696 and ANI-0081396, the Defense Advanced Projects Research Agency (DARPA) and the U.S. Navy (USN) under contract N660019928918. Rajesh Balan was additionally supported by a USENIX student research grant. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, DARPA, USN, USENIX, nor the U.S. government.

Keywords: Software Engineering, Mobile Systems

Abstract

A critical factor for the commercial success of mobile and task-specific devices is the fast turnaround time of software development. However, developing software for mobile devices is especially hard since applications need to be aware of and adapt to changing resources such as bandwidth and battery.

In this paper we validate that the idea of stub generation can successfully address the complexity introduced by resource adaptation. Our approach is based on factoring generic resource-adaptation mechanisms out of the applications and into operating system extensions. Rather than having to deal with system-specific details, an application writer provides a high-level description of the adaptation needs for each application. The generation of code stubs bridges such high-level descriptions to the adaptation mechanisms specific to each platform.

We validated this approach against three representative applications: a video streaming application, a natural language translator and an augmented reality application. In all three cases, the effort for the application writer was reduced by orders of magnitude. The cost of writing the operating system extensions and the stub generator is amortized over the many applications that can share the generic resource-adaptation mechanisms.

1 Introduction

The proliferation of task-specific mobile and wearable devices with short lifetimes places severe stress on the development and maintenance of adaptive mobile applications. A critical factor limiting the commercial success of such a device is the software development time needed to create useful applications for it. The longer this development time, the shorter the useful life of the device in the marketplace. Slow software development can make the device obsolete by the time it emerges as a product. Business opportunities are measured in months rather than years in this fast-paced field.

Developing mobile computing applications is especially difficult because they have to be adaptive [7, 11, 17]. The resource constraints of mobile devices, the uncertainty of wireless communication quality, the concern for battery life, and the lowered trust typical of mobile environments all combine to complicate the design of mobile applications. Only through dynamic adaptation in response to varying runtime conditions can applications provide a satisfactory user experience. Unfortunately, the complexity of writing and debugging adaptive code adds to the application software development time.

How can we reduce the software development time of adaptive mobile applications? In this paper, we describe our approach to solving this problem. It is based on three observations that are derived from our first-hand experience with building adaptive mobile applications.

- First, most applications for mobile devices can be created by modifying existing applications rather than writing new applications from scratch.
- Second, the modifications for adaptation typically affect only a small fraction of total application code size. Much of the complexity of implementing adaptation lies in understanding the base code well enough to be confident of the changes to make.
- Third, the changes for adaptation can be factored out cleanly and expressed in a platform-neutral manner.

Our approach can be summarized as follows:

- We provide a lightweight semi-automatic process for customizing the adaptation API used by the application. Such customization is targeted to the specific adaptation needs of each application.
- We provide a tool for automatic generation of code stubs that map the customized API to the specific adaptation features of the underlying mobile computing platform.
- We factor the run-time support for monitoring resource levels and triggering adaptation out of applications and into a set of operating system extensions for resource adaptation.

Each of these components plays an important role in the overall effectiveness of our approach. The first component (semi-automatic process) amortizes the effort of understanding an application and extending it for adaptation. The second component (stub generator) insulates application code from frequent changes of the underlying mobile computing platform.

The third component (OS support) allows a clean separation of policy and mechanism — the OS monitors resource levels and triggers adaptation, but it is the individual applications that decide how to adapt. OS support also helps ensure that the adaptations of multiple concurrently executing applications do not interfere with each other.

Our approach complements traditional software engineering techniques such as code modularity. In addition, our approach takes into account the context-sensitive nature of adaptation policies. In other words, high level attributes such as a user's location, physiological state, and cognitive load are often important factors in determining how a low-level adaptation decision should be made [19]. This implies a bridging of system layers that is not common in non-mobile applications.

The rest of this paper is organized as follows. We describe our approach in Section 2. Then, in Section 3, we illustrate how our approach can be applied to three representative applications in mobile computing: data streaming, natural language translation, and augmented reality. Next, in Section 4, we describe how we incorporate context senstivity into the choice of adaptation policy. We conclude with a discussion of future work and related work.

2 Reducing Development Cost

Much of the cost of building and maintaining adaptive applications comes from the low-level at which adaptation enhancements are captured. Understanding the required adaptation features and implementing them over the APIs offered by the underlying platform is a costly process. Currently, there is no effective way to preserve such investment except in the form of embedded code modifications. These are hard to maintain in the face of the fast rate of release of new platforms.

We propose to use a high-level declarative language to describe the adaptation aspects of an application. That description is then compiled and a code stub is generated. This code stub creates a *customized* API for the application, which is derived from the high-level description of the adaptation requirements. This customized API is much closer to the application's needs than a generic low-level adaptation API, and thus makes it much easier to integrate the adaptation aspects with the bulk of the application code.

Furthermore, applications in the same domain, say video players, are likely to have very similar adaptation requirements. Hence, adaptation descriptions can be reused among such applications. For example, it would be easier to extend the next video player for adaptation once we've completed the first one.

By having a compiler-based approach, we are able to amortize the effort of retargeting a set of adaptive applications to a new platform. After all, it is easier to retarget the code generation of a compiler than to modify each application manually.

The hypothesis here is that it will be easier to retarget the code generation for a new platform, and recompile all the applications, than retargeting every application.

The specific runtime targeted by our stub generator is *Chroma*: a resource management and adaptation layer that we are developing. Chroma provides generic support for adaptation in applications, including remote execution: the ability to dynamically run portions of an application's functionality on a fast compute server [6]. Chroma is part of the Aura framework [18] for pervasive computing. This paper does not describe Chroma or Aura in any detail; instead we focus on our platform-independent approach to building adaptive applications. The stub generator allows us to potentially use any other OS or adaptation middleware [1, 2, 10, 14], without changes to the application source or description files.

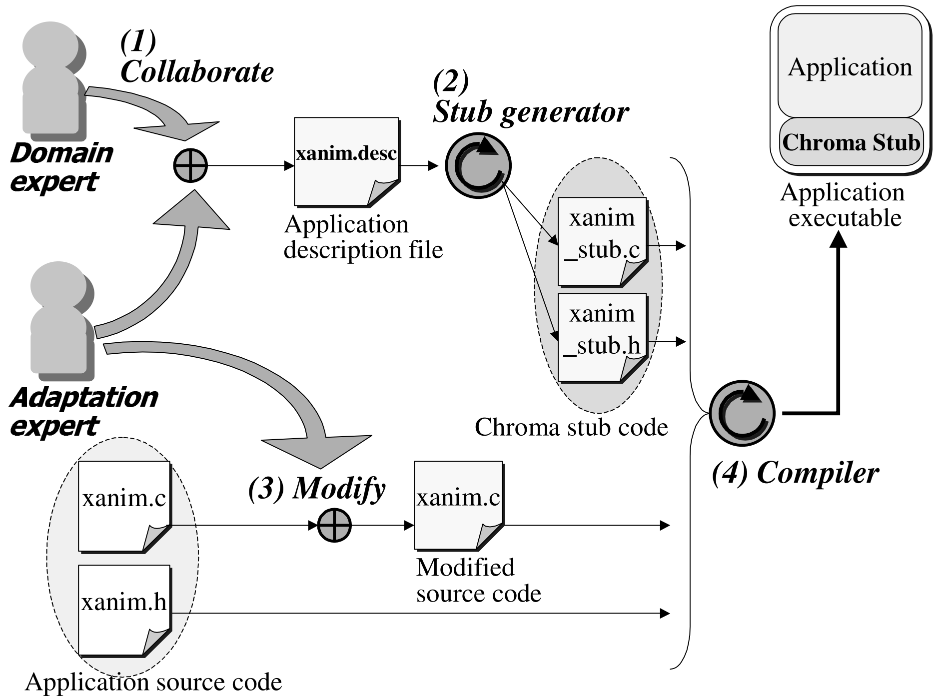

Figure 1: Process for adding adaptation to an application

We now describe how application developers can create the application descriptions and the corresponding application stubs. Figure 1 illustrates our 4-step process:

1. The adaptation expert collaborates with a domain expert to produce an *application description* that captures the information necessary for the application to be adaptive. For instance, the description for XAnim contains the adaptive variables relevant to adaptive video playing: frame rate, encoding, frame quality, height, and width. This description is platform-independent, and can be reused for other applications that provide adaptive video playing capabilities. I.e., it applies equally to XAnim and to MediaPlayer, to Linux and to Windows.

2. A *stub generator* compiles the application description into a set of stubs that interface between the application and the underlying runtime support.

3. The application is modified to invoke the functions provided by the stub layer. This step is manual, and must be done for each application. However, these changes are small and localized as demonstrated in our case studies, and this fact makes it easy to preserve the adaptation enhancements in new releases of the applications, as described in Section 1.

4. The application source code and stub are compiled, and linked together to form the application binary. When executed, this binary invokes the runtime support layer to make adaptive decisions.

3 Case studies

To validate our solution we applied it in three representative case studies. We chose applications which are representative of the unique computational needs of mobile users. As such, instead of typical desktop applications like Word or Powerpoint, we have a video player (XAnim), natural language translator (Pangloss-Lite) and an augmented reality application (GLVU).

3.1 XAnim

The first case study is XAnim. XAnim is a video player that can be used to play AVI format video files. It represents the class of applications that handle streaming media and for whom bandwidth is a critical resource. This class of applications is important for mobile users as mobile users would like the ability to play video files while moving from place to place. In this section, we will show how the process described in Section 2 can be used to make XAnim adaptive.

3.1.1 Creating the description file

The version of XAnim that we are using receives video streams from a server. The server can provide different quality levels of the same video stream, which differ in their frame rate and compression level. XAnim, to be adaptive, should be extended to automatically change the quality requested from the server according to the current resource availability. This decision is made periodically every few video segments.

```
APPLICATION XAnim;

OUT DOUBLE frame_rate  FROM 0 TO 60;
OUT DOUBLE compression FROM 0 TO 100;

IN  STRING  video_name;
IN  ENUM    encoding MPEG, MPEG2, QTCinepak;
IN  INTEGER video_height;
IN  INTEGER video_width;
```

Figure 2: Description file for XAnim

Figure 2 shows the description file for XAnim. XAnim has two “OUT” variables and four “IN” variables. An OUT variable is a parameter that can be adapted by the runtime. To make good adaptive decisions, the runtime needs additional information from the application.

```
.
params = xanim_playsegment_initialize ();
.
/* Main loop of video playback
  This loop retrieves n segments of the video at a time
  from the video server. */
while (video_needs_to_be_played) {
   .
   xanim_playsegment_set_video_height (params, height);
   xanim_playsegment_set_...
   .
   xanim_playsegment_find_fidelity (params);
   frame_rate = xanim_playsegment_get_frame_rate (params);
   .
   /* Retrieve video from video server using frame_rate */
   .
} /* Exiting video playing loop */
.
xanim_playsegment_cleanup (params);
.
```

Figure 3: Source code for the modified XAnim

For example, the runtime will need to know the size of the video before it can decide what frame rate is appropriate given the current bandwidth. IN parameters are used to specify this necessary information.

Even though the description file shown in Figure 2 was created for XAnim, it can also be used for other video players. This is because the description file contains just the adaptation behaviour of XAnim and this is similar for other video players as well. Every other video players will also have inputs consisting of the video name, the encoding and the video dimensions and they will also require a frame rate and a compression level for the video stream. Hence, our method of extracting the adaptation behaviour of an application into a description file allows us to reuse description files between similar applications.

3.1.2 Modifying the application

Figure 3 shows the modifications made to XAnim. Note that all needed to be done was to place these calls in the correct places in XAnim (shaded lines).

The bulk of the modifications takes place in the part of the application that does the work that can be adapted. In the case of XAnim, this is the video playing loop.The methodology used for the modifications is as follows:

- An `initialize` function is called at the start of the application to create and initialize all necessary variables for interfacing with Chroma. The initialize call returns an opaque data structure, that contains all the information relevant to XAnim, which is provided as an input to all subsequent stub generated function calls.

- The `find_fidelity` function is called within the video playing loop. This function queries Chroma and figures out the fidelity level that the application should use, given the current application settings (the IN parameter values) and the current resource availability.

- The application sets all the IN parameters via `set` function calls before calling `find_fidelity`.

- After calling `find_fidelity`, the application reads the values of all the OUT parameters via `get` function calls. Using these values, the application performs a chunk of work at the appropriate fidelity level.

- This process of setting the IN parameters, calling `find_fidelity`, reading the OUT parameters and then doing a chunk of work at the appropriate fidelity level continues until the application exits.

The stub generator automatically generates the `initialize`, `cleanup` and `find_fidelity` functions and the application specific params data structure. It also automatically generates all the set and get functions required to manipulate the IN and OUT parameters. This greatly reduces the amount of work involved in modifying an application to be adaptive.

3.2 Pangloss-Lite

The second class of applications we consider is natural language translation as characterized by Pangloss-Lite. Language translation is important for mobile users as their mobility brings them into contact with documents and speech composed in non native languages. The critical resource for this class of applications is computional power.

One important way that applications can adapt is to run pieces of code on remote servers [6], taking advantage of computational resources in pervasive computing environments. Natural language translation applications are well suited for remote execution as they are CPU and memory intensive. In this second case study, we show how to extend Pangloss-Lite, a natural language translator, to adapt using remote execution. Remote execution services are accessed through an RPC [4] interface.

3.2.1 Creating the Description File

Pangloss-Lite [8] translates text from one natural language to another. It can use multiple *translation engines* with varying degrees of accuracy and speed — and correspondingly, different resource consumptions. Each engine returns a set of potential translations for phrases contained within the input text. A *language modeler* combines the output of the engines to generate the final translation. Since each translation engine consumes different amounts of resources, Pangloss-Lite is enhanced for adaptation by choosing the translation engines to use depending on the available resources. In addition, the translation engines and the language modeler can also be remotely executed. The translation engines can also be executed in parallel. For the purpose of this case study, we will use just two engines: EBMT (example-based machine translation) and GBT (glossary-based translation).

Describing how an application can use remote execution requires two components: enumerating the functions that can be remotely executed (keyword RPC) and the permitted execution *tactics*. Each execution tactic specifies a way of executing a set of functions in some parallel or sequential order. Naturally, each of these tactics will have different resource requirements, corresponding to the subset of functions that gets executed. Furthermore, the adaptation run-time will also select whether to run each function locally on the mobile platform, or remotely on some previously configured set of servers. This decision is based on comparing the resource requirements of each tactic against the available CPU cycles, battery charge, bandwidth to the remote servers, etc. It is up to the adaptive system to pick the most appropriate tactic for each operation, given the current resources and the constraints of the users task — see Section 4.

```
APPLICATION panlite;

IN INTEGER nwords FROM 0 TO infinity DEFAULT 1;

// RPC spec. for the glossary engine
RPC server_gbt   (IN STRING line, OUT STRING gbt_out);

// RPC spec. for the ebmt engine
RPC server_ebmt  (IN STRING line, OUT STRING ebmt_out);

// RPC spec. for the language modeler
RPC server_lm    (IN STRING gbt_out, IN STRING ebmt_out,
                  OUT STRING translation);

TACTICS           = gbt OR ebmt OR gbt_ebmt;

// glossary engine followed by language modeler
DEFINE gbt        = server_gbt & server_lm;

// ebmt engine followed by language modeler
DEFINE ebmt       = server_ebmt & server_lm;

// both engines run in parallel
DEFINE gbt_ebmt   = (server_gbt, server_ebmt) & server_lm;
```

Figure 4: Description file for Pangloss-Lite

The description file for adaptive Pangloss-Lite is shown in Figure 4. There is one IN variable that specifies the number of words in the input string. Chroma uses this value to decide how much resources the translation will require. The RPC definitions for Pangloss-Lite correspond to the GBT engine, EBMT engine and the language modeler. As shown, Pangloss-Lite has three tactics for remote execution: *gbt*, *ebmt* and *gbt_ebmt*. The *gbt* tactic executes just the GBT engine and sends the output to the language modeler. The *ebmt* tactic

```
.
params = panlite_translate_initialize_params ();
.
while (do_translation) {

   /* read input into "line" and do other processing */
   .
   panlite_translate_set_nwords (params, value);
   panlite_translate_find_fidelity (params);
   .
   panlite_translate_do_tactics (params, line, translation);
   .
   /* display translation and do other processing */
}
.
panlite_translate_cleanup_params (params);
.
```

Figure 5: Modifications to Pangloss-Lite

executes only the EBMT engine and sends the output to the language modeler. Finally, the *gbt_ebmt* tactic executes both of the engines in parallel and sends the output to the language modeler.

3.2.2 Modifying the Application

Figure 5 shows the modifications that were made to the Pangloss-Lite source. The methodology used to modify Pangloss-Lite to make it adaptive is similar to XAnim.

- An `initialize` call is made at the start of the application with a corresponding `cleanup` call at the end of the application.
- The single IN variable for Pangloss-Lite is set via a `set` function call before calling `find_fidelity`.
- A call to `find_fidelity` is made to determine which tactic to use. This choice is made by checking the resource availability of the local and remote servers and the value of the IN parameter.
- The main difference is a `do_tactics` function call which is inserted after the `find_fidelity` call. The `do_tactics` function call (this function is also automatically generated by the stub generator) performs the remote execution of Pangloss-Lite using the tactic decided by `find_fidelity`.

By separating the decision making of which tactic to use (done in `find_fidelity`) from the actual execution of the tactic (done in `do_tactics`), we allow the application to cache the selected tactic. Deciding which tactic to use can be potentially expensive as Chroma needs to search through all possible tactics and decide on the optimal one given the values of all the IN variables and the resource availability on the local and remote machines. Caching the result thus allows the application to tradeoff the overhead of computing a new tactic for every translation against the agility of adaptation to changing resource conditions.

3.3 GLVU

Our third case study looks at GLVU which represents the augmented reality application class. Augmented reality applications allow a mobile user to access information about his current environment on his mobile device or even via a head mounted display. This information is superimposed over the current viewing environment; hence the name augmented reality. This class is characterized by strict performance constraints as large jitter or delays can have nauseating effects on the user.

GLVU is a 3D graphics rendering application that uses the OpenGL library to display 3D models of buildings. GLVU computes the image to display by factoring in the current position of the viewer (in 3D space) and the current maximum and minimum display co-ordinates. The quality of the final image, the latency and the computational requirements of GLVU is highly dependent on the number of polygons used to create the 3D model.

3.3.1 Creating the Description File

```
APPLICATION    glvu
OPERATION      draw

OUT  double   polygons    FROM 0  TO infinity
OUT  double   resolution  FROM 0  TO 1

IN   double   min_x       FROM 0  TO infinity
IN   double   min_y       FROM 0  TO infinity
IN   double   min_z       FROM 0  TO infinity

IN   double   max_x       FROM 0  TO infinity
IN   double   max_y       FROM 0  TO infinity
IN   double   max_z       FROM 0  TO infinity

IN   double   eye_x       FROM 0  TO infinity
IN   double   eye_y       FROM 0  TO infinity
IN   double   eye_z       FROM 0  TO infinity
```

Figure 6: Description file for GLVU

The description file for GLVU is shown in Figure 6. As shown, GLVU has nine IN variables and two OUT variables. The nine IN variables are used to provide additional information about the current state of GLVU to the runtime. In this case, the nine IN variables specify the current minimum and maximum display co-ordinates as well as the current co-ordinates

```
.
params = glvu_render_initialize ();
.
/* Main loop of 3D rendering
  This loop renders the 3D using the specified resolution
  and number of polygons    */
while (model_needs_to_be_rendered) {
   .
   glvu_render_set_eye_x (params, user_x_position);
   glvu_render_set_…
   .
   glvu_render_set_polygons (params, num_polygons);
   glvu_render_find_fidelity (params);
   resolution = glvu_render_get_resolution (params);
   polygons = glvu_render_get_polygons (params);
   .
   /* Render model using resolution and number of
      polygons */
   .
} /* Exiting 3D rendering loop loop */
.
xanim_render_cleanup (params);
.
```

Figure 7: Source code for the modified GLVU

of the viewer. The OUT variables are used by the runtime to tell GLVU what resolution it should use to render the image and how polygons should be used to construct the 3D model.

3.3.2 Modifying the Application

The modifications that were made to GLVU are shown in Figure 7. GLVU was modifed using a methodology similar to XAnim.

- An `initialize` call is made at the start of the application with a `cleanup` call made at the end of the application.
- The IN variables are assigned values using `set` functions.
- The `find_fidelity` function is called. This call invokes the runtime which uses the values of the IN variables to determine the optimal fidelity for the application given the current resource availability. The values of the OUT variables are set by the runtime to reflect the optimal fidelity settings.
- The values of the OUT variables are read via `get` function calls. These values are then used by GLVU to render the image.

All the calls used above are generated automatically by the stub generator.

4 Adaptation policy

In Section 3, we presented three case studies to demonstrate that it is possible to extract the adaptive behavior of applications in a platform independent manner. However, in reality, the exact decision of how to adapt an application is frequently context sensitive and thus dynamic. For instance, would the user of a language translator prefer accurate translations or snappy response times? Should an application running on a mobile device use power-save modes to preserve battery charge, or should it use resources liberally in order to complete the user's task before he or she runs off to board their plane? That knowledge is very hard to obtain at the application level.

Does this dynamism mean that our static descriptions of adaptation are inapplicable in real environments? We claim that this is not the case. Our architecture allows us to specify various static policies for different contexts. The exact policy to use is determined by the runtime based on the current environmental conditions and user specified preferences. In general, we would need an infinite number of policies to handle every possible context. However, in practice, the situation is not so bad.

We claim that we can statically define a *family* of adaptation policies that covers a satisfactory dynamic range. Choosing one particular instance from such a family of policies is achieved by parameterization.

The key observations here are that, first, *user expectations* ultimately determine which adaptation policies are appropriate. Second, these expectations change as a function of the nature of the user's task and of the physical context around the user. Although describing how user expectations are captured is beyond the scope of this paper, we briefly describe our approach to this problem, and give some detail on how user expectations are represented and used to determine the adaptation policy enacted by the application.

The novelty in our approach is threefold:

- User expectations are captured outside the adaptive application, in a layer that is aware of the user's task and surrounding context. That layer builds models of user expectations that can be passed to adaptive applications [19].
- User expectations are represented in an application- independent way, making it easy to reuse models of user expectation across multiple applications. For instance, a model of the expectations of the user when watching a video can be used to drive adaptation in every video playing application equipped to work in this framework.
- The representation we adopt is easy to pass to a running application, making it easy to adjust adaptation policies on the fly to changes in user expectations.

4.1 Defining the adaptation policy

We use a simple model of user expectations based on *utility functions.* These functions take the user-perceived quality attributes as inputs and return a value indicating their appropriateness. The higher the value the more appropriate the combination is relative to the user's expectations. For instance, utility functions for watching a video would take frame-update rate and video quality as inputs. Now, if the user is watching a sports video, an appropriate utility function is one that is more sensitive to the frame-update rate than the video quality. However, if the user is watching a tour of a museum, an appropriate utility function is one that is more sensitive to the video quality, and not as sensitive to frame-update rates.

The adaptation policy is implicitly defined by maximizing utility functions. The values that maximize these utility functions give the fidelities that the application should run at. These values are returned by the calls to find_fidelity (in Figures 3 , 5 and 7) as values for the OUT parameters. Naturally, the maximization of the utility functions is constrained by the available resources.

```
APPLICATION XAnim;

OUT DOUBLE frame_rate  FROM 0 TO 60;
OUT DOUBLE compression FROM 0 TO 100;

 UTILITY =  WSIGMOID(frame_rate) *
            WSIGMOID(compression);

IN  ...
```

Figure 8: Utility function description for XAnim

The description for XAnim in Figure 8 extends the description in Figure 2 by defining the generic form of the utility functions driving the adaptation in XAnim (and in fact in any video playing application that follows this model of user expectations.) Each of the user-perceived quality attributes has a model of utility: in this case a weighted *sigmoid* function. Sigmoid functions are step-like functions that have a “bad” threshold, below which the function is exponentially close to zero, and a “good” threshold, above which the function is exponentially close to one. Between the “good” and “bad” thresholds the function grows smoothly (and is roughly linear). A weighted sigmoid is raised to a power, its weight, between 0 and 1. The overall utility is obtained by multiplying the two weighted sigmoids. Note that assigning a small weight to a sigmoid tends to make it flat, and hence reduces the sensitivity of the overall utility to the corresponding quality attribute.

This representation allows utility functions to be encoded in a totally parametric way, which is a big advantage. For example, in Figure 8, a utility function is encoded by six numeric parameters: the “good” and “bad” thresholds and the weight for each of the sigmoids.

4.2 Enacting the adaptation policy

The stub generator takes the description of utility in the application description file and generates an interface that allows an external source to set the corresponding parameters. In our work, the information exchanged between the layer modeling the user and the applications is encoded in XML [19]. Therefore, the interface produced by the stub generator includes a parser for the specific XML format we are using. Note that the implicit assumption here is that the language to build utility functions in the application description file is expressive enough to represent the possible forms of user expectation for the relevant quality attributes. In the case studies we analyzed so far we had no difficulty expressing the form of utility functions using sigmoids for continuous attributes and simple tables for discrete attributes.

For example, suppose that the user is watching a sports video and that the layer in charge of capturing the user expectations has empirically determined the range of quality attributes that makes the user happy in those circumstances. Suppose that the range is as follows: the user is happy as long as the frame-update rate is above 20 frames per second, and really unhappy if it drops below 5 frames per second. Video quality is expressed by the "compression" parameter in Figure 8. Although higher quality is better, it is of secondary importance.

```
<utility combine="mult">
<wSigmoid attr="frame_rate" weight="0.8"
          bad="5" good="20"/>
<wSigmoid attr="compression" weight="0.2"
          bad="0" good="100"/>
</utility>
```

Figure 9: XML encoding of utility function

This knowledge is encoded in a utility function composed of two weighted sigmoids with the following parameters: for the frame rate sigmoid, set "bad" to 5, "good" to 20 and weight to 0.8. For the compression sigmoid set "bad" to 0, "good" to 100 and weight to 0.2. Note that the sigmoid for the compression attribute degenerates into a linear function by placing the thresholds at the extremes of the scale for the attribute. Note also, that the relative weights of the two sigmoids are empirically set by observing what makes the user happy. Figure 9 shows the encoding of this utility function.

5 Future work

We are developing a new runtime system called Chroma that builds on our past experience with Odyssey [14]. Currently we are still using Odyssey as our adaptive runtime system as Chroma is still being developed. Chroma will have many features currently not found in Odyssey. These include:

- The ability to easily enhance applications for adaptation using the process described in this paper.
- The ability to accept application-specific stubs. These stubs will be automatically generated by the stub generator based on the application's description file. They will provide Chroma with the necessary logic to handle the application's resource requirements.
- Integration with *Prism* (see below).

- Better handling of global constraints like battery power.

One key observation of our work is that determining appropriate adaptation policies is critically dependent on the ability to capture user expectations. Capturing user expectations is a hard problem that we plan to address in a layer called *Prism*. Prism treats user tasks as first class entities and interacts with context-aware components to assess the physical context around the user. It determinines the most accurate models of user expectations using stochastic techniques to correlate the current user context to past experiences. By capturing user expectations outside of applications, we enable the reuse of user expectation models. This allows the migration of user tasks in pervasive computing environments [19].

Chroma and Prism are being developed as part of the Aura framework [18]. Aura aims to provide a complete pervasive computing environment ranging from better user interfaces to low level intelligent networking.

Additionally, we plan to do more case studies using our process to evaluate its effectiveness for a larger class of applications. This will allow us to refine our process and tools where necessary.

6 Related work

As mentioned in Section 1, our current work builds on previous experience with Odyssey [14]. Odyssey provides support for mobile information access through *application-aware adaptation*, a collaborative partnership between the operating system and applications.

The technique of using stubs and a stub generator is derived from RPC [4]. RPC has shown the effectiveness of stubs in insulating system details from applications and the usefulness of a stub generator for automated code generation. We have simply applied these techniques to the realm of adaptation in pervasive computing.

The application description language addresses some of the same issues as 4GLs [12] and "little languages" [3]. The latter are task-specific languages that allow developers to express higher level semantics without worrying about low level details. Our description language is similar as it allows application developers to specify the adaptation capabilities of their applications at a higher level without needing to worry about low level system integration details. Our stub generator converts this high level description into low level code for interfacing the application with the runtime. Another system that uses this method is CORBA [15, 20]. However, our approach is focused towards adaptive systems.

Initial research [5] on adaptive multimedia applications concentrated on low-level system parameters, while concern for user-perceived quality attributes appeared later [13]. Expressing user satisfaction took an econometric slant, and new expressive power, with the introduction of utility functions in resource allocation systems in [16]. Capturing user goals and using that knowledge to drive systems is a cornerstone of recent work on expert systems that provide assistance to computer users. For example, Horvitz [9] uses Bayesian networks to perform inference on user goals and utility functions to evaluate the relative merit of alternative system actions.

7 Conclusions

In this paper we have shown an effective approach for reducing the cost of developing and maintaining mobile adaptive applications. Specifically, our approach is:

- A *description language* for representing the adaptation features of applications in a platform and implementation-independent fashion. The description language is rich enough to describe features for adaptation by remote execution and for driving the adaptation policies based on user expectations.

- A *stub generator* that produces an interface between the application and the underlying runtime support for adaptation. Although the design of such interfaces is applicable to a broad class of adaptive applications, the stub generator tailors each generated interface to the specific adaptation features of the application, thus making it easier to extend each application.

- A *methodology* for extending applications for adaptation.

We have implementated this approach for a video player (Xanim), a language translator (Pangloss-Lite), a speech recognizer (Janus) and a 3-D viewer (GLVU). We have reported three of these experiments as case studies in this paper. From Figures 3, 5 and 7, we see that a small amount of manual effort had to be done to modify XAnim, Pangloss-Lite and GLVU. These changes were also systematic and very similar across all the three applications. This provides preliminary evidence that our process minimizes the amount of work needed to modify the application.

Although more case studies are needed to further validate our approach, we are confident that the mechanisms that we have created can be used to extend a broad class of applications for adaptability.

We have also shown how adaptive mobile applications can deal effectively with the problem of adjusting adaptation policies to cope with dynamically changing user expectations. We recognized that the appropriate policy is best determined outside the application and designed an interface that allows an adaptive application to receive a representation of that policy at runtime, as often as required by the changes in user expectations.

References

[1] Amiri, K., Petrou, D., Ganger, G., and Gibson, G. Dynamic function placement for data-intensive cluster computing. *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, CA, June 2000.

[2] Basney, J. and Livny, M. Improving goodput by co-scheduling CPU and network capacity. *Intl. Journal of High Performance Computing Applications*, 13(3), Fall 1999.

[3] Bentley, J. Little languages. *Communications of the ACM*, 29(8):711–21, 1986.

[4] Birrell, A. D. and Nelson, B. J. Implementing remote procedure call. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[5] Clark, D. D., Shenker, S., and Lixia, Z. Supporting real-time applications in an integrated services packet network; architecture and mechanism. *ACM SIGCOMM '92*, 22(4):14–26, aug 1992.

[6] Flinn, J., Narayanan, D., and Satyanarayanan, M. Self-tuned remote execution for pervasive computing. *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany, May 2001.

[7] Forman, G. and Zahorjan, J. Survey: The challenges of mobile computing. *IEEE Computer*, 27(4):38–47, April 1994.

[8] Frederking, R. and Brown, R. D. The Pangloss-Lite machine translation system. *Expanding MT Horizons: Proceedings of the Second Conference of the Association for Machine Translation in the Americas*, pages 268–272, Montreal, Canada, 1996.

[9] Horvitz, E. Principles of mixed-initiative user interfaces. *Proceedings of CHI '99, ACM SIGCHI Conference on Human Factors in Computing Systems*, Pittsburgh, PA, May 1999.

[10] Hunt, G. C. and Scott, M. L. The Coign automatic distributed partitioning system. *Proceedings of the 3rd Symposium on Operating System Design and Implemetation (OSDI)*, pages 187–200, New Orleans, LA, Feb. 1999.

[11] Katz, R. H. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1):611–17, 1994.

[12] Martin, J. *Fourth-Generation Languages*, volume 1: Principles. Prentice-Hall, 1985.

[13] McCanne, S. and Jacobson, V. Vic: A flexible framework for packet video. *ACM Multimedia*, pages 511–522, Nov. 1995.

[14] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. Agile application-aware adaptation for mobility. *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 276–287, Saint-Malo, France, October 1997.

[15] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1999. Revision 2.3.1, ftp://ftp.omg.org/pub/docs/formal/99-10-07.ps.

[16] Rajkumar, R., Lee, C., Lehoczky, J., and Siewiorek, D. Practical solutions for QoS-based resource allocation. *The 19th IEEE Real-Time Systems Symposium (RTSS'98)*, pages 296–306, Dec. 1998.

[17] Satyanarayanan, M. Mobile Information Access. *IEEE Personal Communications*, 3(1), February 1996.

[18] Satyanarayanan, M. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, Aug. 2001.

[19] Sousa, J. and Garlan, D. Aura: An architectural framework for user mobility in ubiquitous computing environments. In Jan Bosch, Morven Gentleman, C. H. and Kuusela, J., editors, *Software Architecture, System Design, Development and Maintenance*, pages 29–43. Kluwer Academic Publishers, Aug. 2002.

[20] Vinoski, S. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications*, 35(2):46–55, Feb. 1997.