Adaptation: This Won't Hurt a Bit!

Rajesh Krishna Balan[†] João Pedro Sousa[†], SoYoung Park[†], Tadashi Okoshi[†],
Jason Flinn[†‡], Dushyanth Narayanan[†], Takahide Matsutsuka[†], Mahadev Satyanarayanan[†‡]
[†]School of Computer Science, Carnegie Mellon University
[‡]Intel Research Pittsburgh
{rajesh,jpsousa}@cs.cmu.edu

Abstract

With the prevalance of handheld devices, it is increasingly necessary to build systems that adapt applications according to the available resources. These systems offer new and constantly changing interfaces for adaptation. Integrating applications with these systems requires extensive modification and maintainence: this is painful! In this paper, we present an approach that allows application writers to easily integrate their applications with an adaptive runtime system. This approach involves the use of a high level description language coupled with a runtime-specific stub generator. We present two case studies of enhancing applications for adaptation using our approach. Our case studies show that our approach is viable and can help to ease the integration of applications into adaptive runtime systems.

1 Introduction

Building a successful mobile application is a daunting task. While adaptation is now widely recognized as essential for mobility [7, 11, 17], there is no consensus yet on how best to build an adaptive mobile application. A clean sheet design is often impractical — one cannot ignore the enormous prior investment by developers in authoring a popular application, and by users in learning to use it effectively. Modifying the application to support adaptation is often the only feasible solution.

Our first-hand experience with modifying existing applications for mobility in Odyssey [14] has revealed many difficulties. First, it is necessary to gain an understanding of the application from the viewpoint of adaptation. Acquiring this knowledge is time consuming, and there is no effective way to preserve it except in the form of embedded code modifications. Second, the underlying platform for adaptation may evolve over time as its support for mobility is enhanced. This typically requires changes to applications as well because the API provided by the platform changes. Since research in mobile computing is still very active, it would be unwise to freeze this API and forego the benefits of future research advances. Third, applications also evolve and each new release of an application requires re-examination and reintegration of changes for mobility. Thus each major release of the application or underlying platform requires the changes for adaptation to be revisited and reconsidered. In a long-lived system where the identity of developers changes over time, these difficulties result in a major software engineering headache. Can we alleviate this pain?

We propose a solution that effectively amortizes the effort involved in gaining a deep understanding of applications from the viewpoint of adaptation. The insight behind our solution is that it is typically possible to capture this understanding in a compact external description that is invariant across application and platform releases. To conform to each such release, we use a stub generator that uses the external description to produce release-specific code modules that are linked into the application and the adaptation platform.

The benefits of our approach are threefold:

- It becomes much easier to preserve the adaptation enhancements in new releases of the applications. Since the changes made to the applications are small and localized, it becomes feasible to use automatic tools to insert the modifications at the appropriate places in the applications code base. Alternatively, the application's original developer can insert stubs for the modifications in the appropriate places, thus making the application *adaptation enabled*. The adaptation capabilities will become active when the application is linked with the appropriate adaptation libraries.
- The adaptation enhancements are protected, by the stub generation process, from changes in the interfaces supporting adaptation at operating system level.
- It becomes easy to drive the adaptation mechanisms inside the application by policies determined outside the application. Determining the appropriate

policies often involves detailed knowledge of the nature of the user's task and physical context, and that knowledge is very hard to obtain at application level.

The rest of this paper is organized as follows. In Section 2, we discuss the difficulties in building adaptive applications and describe the main components of our approach. Section 3 details our process for building adaptive applications. Section 4 and Section 5 illustrate our process through the use of case studies. The first case study involves XAnim, a video playing application. The second case study demonstrates adaptation by remote execution in Pangloss-Lite, a natural language translator. Section 6 describes how we drive adaptation policies with an explicit representation of *user expectation*. Section 7 describes our future work while Section 8 and Section 9 present related work and conclusions, respectively.

2 Adaptation: heaven and hell

Consider the following scenario:

Joe is a new hire at his company. His manager proclaims him an "adaptation expert" and wants him to develop an adaptive video player for handheld devices running Linux and X. Not being a video expert, Joe collaborates with domain experts to understand how to make video players adaptive. As a result, he realizes that adapting video playing can be done by adjusting image quality and/or frame rate.

Joe decides to use the XAnim application as the base. He modifies XAnim to adjust its image quality and frame rate to the available resources. This is a tedious and iterative process as Joe has to make extensive changes to the XAnim source, while periodically reconsulting with the video experts.

Finally, Joe achieves the desired modification: XAnim now uses an underlying resource management layer to adapt its behavior to the available resources. His manager is delighted and demands a similar modification of MediaPlayer, a video player for Windows CE. The functionality of MediaPlayer is similar to that of XAnim; so are the adaptive extensions to be added. However, the code base is completely different, and so is the underlying OS. Joe must redo for MediaPlayer all the painstaking work that he put into XAnim.

Since his company uses five other OSs — Macintosh, Solaris, IRIX, BSD and Windows 2000 — Joe anticipates that he will have to enhance the video players in all of them! His hell is just beginning!!

Why should modifying MediaPlayer and the other video players be so difficult, given Joe's previous experience with XAnim? First, Joe lacks a method that allows him to reuse the mechanisms used in XAnim to adapt image quality and frame rate. Second, since MediaPlayer runs on WinCE, Joe has to port his modifications to interface with this new OS. Furthermore, every time the OS or runtime layer changes, Joe has to modify the application to use the new interface.

In this paper we describe an approach that addresses these problems. Our solution is based on:

- a *description language* for adaptive applications: a platform-independent and implementation-independent representation of an application's adaptation capabilities. This allows us to reuse XAnim's description when modifying MediaPlayer or any other adaptive video player.
- a *stub generator* that converts descriptions in this language into code stubs tailored to the application and to the underlying runtime system. This stub insulates the application from the details of the runtime system.

The specific runtime targeted by our stub generator is *Chroma*: a resource management and adaptation layer that we are building. 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 [14, 1, 10, 2], with no changes to the application source code or description files.

3 Adaptation in 4 easy steps

How can one minimize the pain involved in adding adaptive capabilities to applications? Figure 1 illustrates our 4-step process:

1. The adaptation expert collaborates with a domain expert to produce a *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

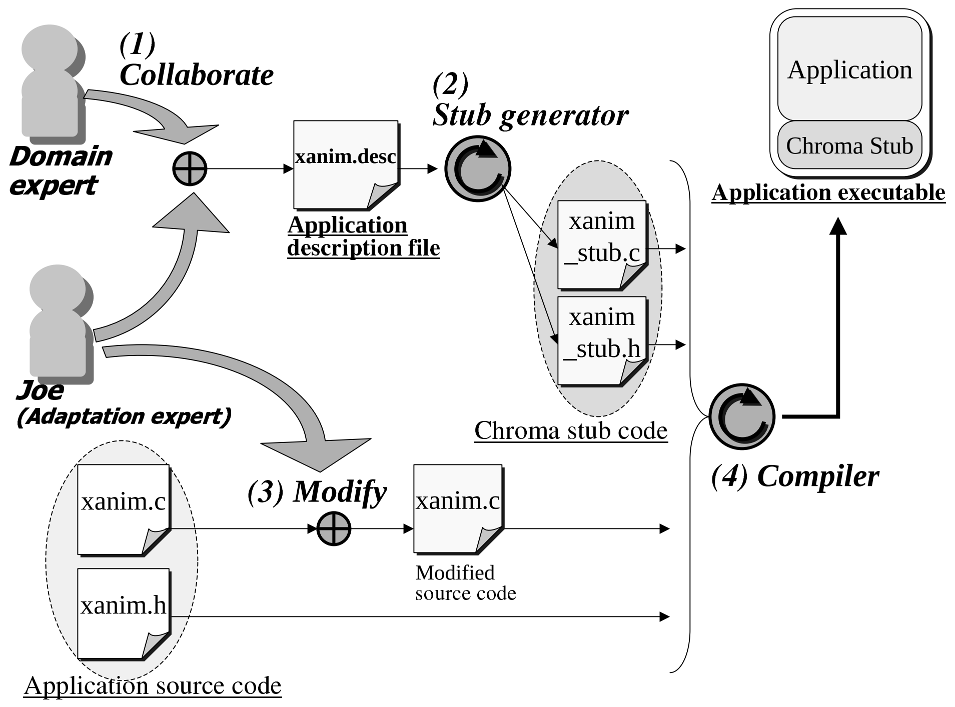

Figure 1: Process for adding adaptation to an application

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.

The case studies presented in this paper show how steps 1 and 3 can be accomplished easily (Steps 2 and 4 are automated).

4 Case study: XAnim

In this section, we will show how Joe can use the process described in Section 3 to make XAnim adaptive.

4.1 Creating the description file

The first thing that Joe does is to get in touch with somebody who understands video well. Together with this domain expert, Joe produces the description file for XAnim that captures its adaptive behavior.

The version of XAnim that Joe is 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

```
.
.
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

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. 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.

4.2 Modifying the application

Joe has his description file after talking with the domain expert. He now has to modify XAnim to work with the underlying runtime system (Chroma in this case). Usually, this would be a lot of work to try to figure out the underlying runtime system and things look bleak for Joe. But help is at hand!! A stub generator is available to process the description file for XAnim and automatically generate XAnim-specific code which will make Joe’s life a lot simpler. Joe eagerly uses the stub generator and then modifies XAnim in less than an hour. Life is good again!

Figure 3 shows the modifications that Joe will have to make to XAnim. The shaded lines are the lines that Joe added. Note that all Joe needed to do was place these calls in the correct places in XAnim!

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:

1. 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. This data structure is provided as an input to all the function calls that the stub generator generates.

2. In the video playing loop, a call to the `find_fidelity` function is made. 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.

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

4. 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.

5. 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. Using a stub generator to automatically create application specific code greatly reduces the amount of work involved in modifying an application to be adaptive.

5 Case Study: Pangloss-Lite

One important way that applications can adaptis 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.

```
APPLICATION panlite;

IN INTEGER nwords FROM 0 TO infinity DEFAULT 1;

RPC server_gbt   (IN STRING line, OUT STRING gbt_out);  // RPC spec. for the glossary engine
RPC server_ebmt  (IN STRING line, OUT STRING ebmt_out); // RPC spec. for the ebmt engine
RPC server_lm    (IN STRING gbt_out, IN STRING ebmt_out,
                  OUT STRING translation);              // RPC spec. for the language modeler

TACTICS           = gbt OR ebmt OR gbt_ebmt;

DEFINE gbt        = server_gbt & server_lm;  // glossary engine followed by language modeler
DEFINE ebmt       = server_ebmt & server_lm;   // ebmt engine followed by language modeler
DEFINE gbt_ebmt   = (server_gbt, server_ebmt) & server_lm; // both engines run in parallel
```

Figure 4: Description file for Pangloss-Lite

5.1 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. 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 and the permitted *execution tactics*. Each execution tactic specifies a way of executing the functions in some parallel or sequential order.

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 of the three functions that can be remotely executed are given on the next three lines. These functions handle the GBT engine, EBMT engine and the language modeler.

The possible remote execution possibilities for Pangloss-Lite are specified on the TACTICS line. As shown, Pangloss-Lite has three tactics: gbt, ebmt and gbt_ebmt. A tactic is a specific way of executing the functions in some parallel or sequential order. Pangloss-Lite has three tactics for remote execution. The *gbt* tactic executes just the GBT engine and sends the output to the language modeler. The *ebmt* tactic 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.

5.2 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.

1. An `initialize` call is made at the start of the application with a corresponding `cleanup` call at the end of the application.

2. The single IN variable for Pangloss-Lite is set via a `set` function call before calling `find_fidelity`.

3. 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.

4. 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

```
.
.
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

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.

6 Driving adaptation

Even the most efficient adaptation mechanisms are not very useful unless they are driven by an appropriate policy. The question then becomes what would be an *appropriate policy*. The answer to this question is not trivial since the user often perceives more than one quality attribute, and hence there is more than one degree of freedom for the adaptation policy. 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 they run off to board their plane?

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 drive the adaptation mechanisms.

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.

6.1 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.

```
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 6: Description of the utility function for XAnim

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 and 5) as values for the OUT

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

Figure 7: XML passed between the layer modeling the user and XAnim

parameters. Naturally, the maximization of the utility functions is constrained by the available resources.

The description for XAnim in Figure 6 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.

The big advantage of such a representation is that a particular utility function can be encoded in a totally parametric way. For the example in Figure 6, a utility function is encoded by six numeric parameters: the "good" and "bad" thresholds and the weight for each of the sigmoids.

6.2 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.

To give a concrete 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 6. Although higher quality is better, it is of secondary importance.

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. The XML that encodes this utility function is given in Figure 7.

7 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 of the key observations 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 enabled the reuse of user expectation models. This paves the way for 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.

8 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 was concerned with low-level system parameters, whereas 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.

9 Conclusions

In this paper we have shown a painless approach to extend applications for adaptation. Specifically, our approach is based on:

- 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* that guides experts in 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 two of these experiments as case studies in this paper (Xanim and Pangloss-Lite). Although more case studies are needed to further validate the applicability of the approach, we are confident that the mechanisms that we have created so far can be used to extend a broad class of applications for adaptability.

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: From computers everywhere to tasks anywhere. *Submitted for publication at the Working IEEE/IFIP Conference on Software Architecture (WICSA 2002)*, Montreal, Canada, Aug. 2002.

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