**15-712 Advanced Operating System**

# Dynamic execution placement for mobile application adaptation

Rajesh Balan, Tadashi Okoshi, SoYoung Park
{rajesh,slash,seraphin}@cs.cmu.edu

Group01_Balan_Okoshi_Park.doc

## Abstract

In a pervasive environment, the users are mobile; the devices that they carry are often small and limited in resources. Therefore, the applications that run on them benefit greatly by taking advantage of the computing infrastructure of the environment through adaptation and remote execution. Systems exist today that provide support for such adaptation. However, developing such an adaptive application is a challenging task due to the lack of a well-defined method to describe the adaptive behavior of the application as well as an API to interface with the operating system support. In this work, we present a language that concisely captures the application's behavior and an API that truly provides an abstract interface to the operating system.

## 1. Project Background

A mobile and pervasive computing environment presents a number of challenges to application and system designers. In this environment, users are assumed to be moving around with a variety of computing devices. These could range from powerful notebook to less powerful PDAs. The computational ability of the devices used are widely different as well as their connectivity to the surrounding environment. For example, a notebook could be connected to the environment via a fast Wavelan connection while a PDA may only be connected via a low bandwidth, high latency infrared connection.

However, regardless of this, users expect to get the same behavior from their applications when run on any platform. In other words, somehow all the applications that a user is depending on should still be able to be accessed regardless of which computing device is being used. The application may have to run in a degraded mode but it should still run. However, the degradation has to be dependent on the environment, as the performance of an application may be improved by remotely executing parts of it on fast servers located nearby. The overall goal of the system is thus to maximize the performance of applications as much as possible by utilizing all available resources in the environment. This may include remotely executing parts of the application on fast remote servers.

Currently, Odyssey [Noble97] supports application degradation and Spectra [Flinn01] supports remote execution. However, for applications to use Odyssey and Spectra requires the application writer to make extensive modifications to his application. The application writer also has to understand the internal workings of the Odyssey and Spectra runtimes. This is undesirable from a software engineering perspective as it presents a large barrier to application writers wanting to port their applications to use the benefits of Odyssey and Spectra.

Our project explores layer abstraction, application behavior distillation, and automatic code generation to reduce the amount of work required from the application developer. Providing the right abstraction is a core systems topic and a fine choice for a project for an operating systems course.

# 2. System design consideration

The main goal for this project is to show that a well-designed abstraction layer can provide the services of adaptation and remote execution to application developers while shielding them from the nitty-gritty details of using the underlying layer. More specifically, our project goals are as follows:

1) To develop a language which application writers can use to specify the application behavior, in quality degradation and remote execution possibilities.
2) To write a stub generator that will automatically parse the input from part (1) and generate application specific procedures which provide the interface between the application and Odyssey/Spectra. These procedures need to be added to the application at the appropriate places. This is very similar to RPC stub generation.
3) To develop one set of API for Odyssey and Spectra that isolates the developer from the details of these systems.

## 2.1 Capturing application behavior

The system can provide better support for adaptation if the application developer provides guidelines to the underlying system on how to make adaptive tradeoffs. These guidelines fall into several categories: attributes about the input that affects the resource consumption, how the application can degrade quality to met the resource availability, and how the application is partitioned for remote execution. These characteristics of the application are captured in a single application description file using a custom language that we developed.

## 2.2 A better API

There have been systems like Coign[Hunt99] that provided remote execution services without needing any application source code modifications. They managed to do this by exploiting externally visible closed interfaces. Not requiring applications to modify their source code is great from a software engineering perspective but we believe that we can provide a better system by requiring applications to make small changes to their source code.

We have chosen Odyssey and Spectra as our system playground, so our API is tailored for those systems. The development of the API posed the following questions:

- What is the minimum amount of information that needs to be exposed to the application?
- What interactions do we need between the system and the application to provide the relevant services?
- What is the minimum set of functions that provide those interactions?

We decided that only the application-specific parts that the developer mentioned in the description file should be exposed to the application. We provide the details to the answers to these question in the subsequent sections of the report.

# 3. Reducing application developer's effort

## 3.1. Problem statement

A major lesson learned over the lifetime of Odyssey is that it had a steep learning curve which deterred application developers from using the system. Concrete difficulty for application developers in using Odyssey system is necessity of handling several Odyssey's internal parameters, which are the unessential system-specific information for the applications, as well as application-specific information, such as fidelity values and parameters needed for fidelity decision-making. Figure 1 shows one of old Odyssey's API provided by the Odyssey library to applications and one of our new Chroma stub API provided by application-specific Chroma stub to applications.

| Odyssey API | Chroma API |
|---|---|
| **begin_fidelity_op(domain,**<br>**operation_type,**<br>**num_params,**<br>**parameters,**<br>**num_fidelities,**<br>**fidelities,**<br>**operation_id);** | **panlite_translate_begin_op(struct *parameters);** |

Figure 1: Odyssey API and Chroma API

In **begin_fidelity_op()** Odyssey API, application has to provide several system-specific information, such as domain, operation_type, and operation_id. Those many arguments values for API can lead several problems in porting applications to Odyssey in terms of software engineering. (1) Those unnecessarily visible parameters for application can be unnecessary factor of bugs. (2) Being able to set internal parameters of underlying system can be a security violation, allowing malicious application developer setting invalid values into them. And, (3) unnecessarily visible many parameters for application simply can be a barrier for application developers willing to port their application to Odyssey.

Thus, in order to solve those problems above, simplifying APIs for application is a significant issue in a way of achieving wide deployment of the system. More concretely,

1. Parameters irrelevant for applications, i.e. system-specific parameters, should be hidden from applications.
2. Out of application-specific parameters, such as fidelity value and parameters for fidelity decision-making, those which are static in application run-time should not be written in the application source code directly, but should be specified in some other "easier" way for application developers. Thus, only parameters which are set dynamically by application in run-time are visible into application source code, simplifying source code modification more.

Figure 2 shows the difference between Odyssey and Chroma. Chroma is the replacement of Odyssey, addressing the problems of Odyssey and changing the places those parameters specified.

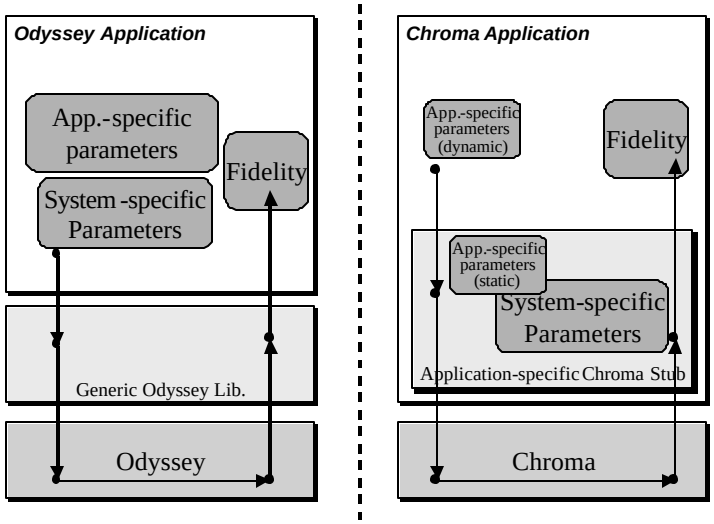


Figure 2: Location of parameters in Odyssey and Chroma

## 3.2. Our solution based-on automatic stub generation

We have rectified that problem in Chroma by employing automatic function stub generation. Figure 3 illustrates the procedure which application developers port their applications to Chroma.

In the following subsections, we describe detail of "application description file" for application-specific information, a stub generation tool called "Chroma stub generator", "Chroma API" embedded into application source code, and the application developer's effort with our solution.

As mentioned before, key design point of our solution is a separation between application-specific information and system-specific information in order to minimize the application developer's effort in porting their applications to Chroma. In our solution, management of all system-specific information is done inside Chroma API which is generated by Chroma stub generator. Thus, application developer does not have to care about this type of information at all. Regarding the application-specific information, information which is static during application run-time is written in the application description file and the stub generator generates codes for their management inside Chroma API functions. Application developers are not required to manage those information once they specified them inside the description file. Only for application-specific information which are set dynamically during the application run-time, application developer takes care by inserting Chroma API functions into the original application source code.

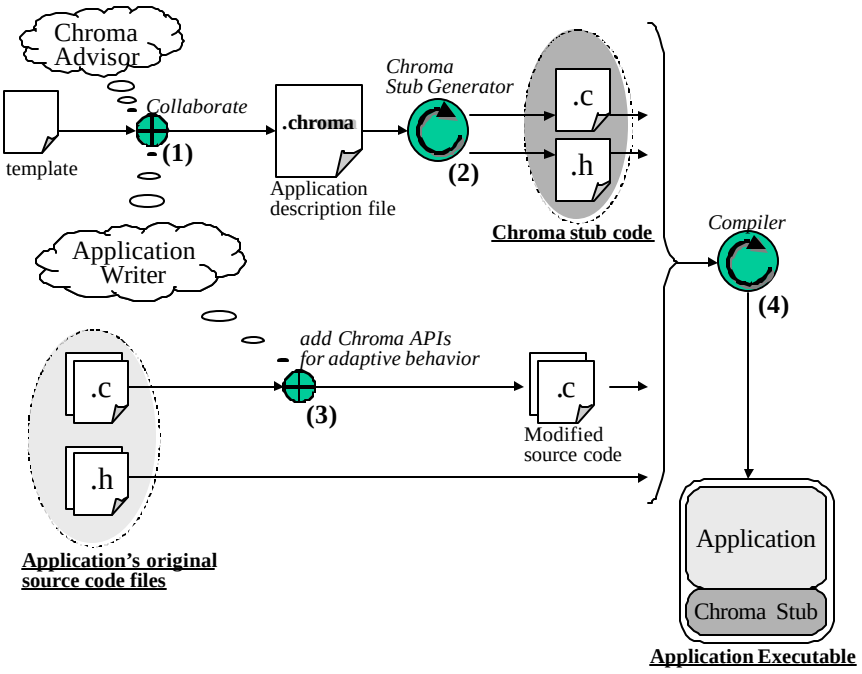


Figure 3: Stub generation and application compilation

## 3.3. Application description file

Application description file is a file which contains all application-specific information Chroma needs to know in order to support adaptation. The information includes (1) basic information about application and its operation to be executed remotely, (2) information of fidelity parameters which is needed by Chroma to provide fidelity value calculated from them to the application, and (3) information on detailed remote execution with possible execution plan specified by the application.

The application description file is generated from a template file with collaboration between the application developer and the Chroma advisor who has the expertise about porting applications to Chroma.

For detailed language of the application description file, please refer to Appendix A.

## 3.4. Chroma Stub Generator

Chroma stub generator is a tool which takes in on input file, application description file, and creates two output files, (1) a header file which contains the Chroma API function prototypes needed by the application, and (2) the corresponding source file which defines the functions. The header file also contains the declarations of all the data types needed by the application to use Chroma.

After the application description files is generated through the collaboration above, the application developer executes the stub generator to prepare those header file (.h) and the source code (.c) of Chroma API which will be next inserted into the developer's original application source code.

## 3.5. Chroma API

We preserve the Odyssey notion of an operation as the unit of adaptation. Examples of an operation include a single speech recognition, the translation of one sentence, and the playing of a video segment. Adaptation cannot occur in the middle of an operation. Exploring a space of possibilities to make a good decision is expensive, so for multiple-stage operations it makes sense to pay all of the costs up front at the beginning of the operation. We expect that the computing environment is not so unstable that the resource state would change often in the middle of an operation.

An application's adaptaptive use of the computational resources in its environment happens through the following six basic Chroma functions:

<application>_<operation>_initialize_params (<application>_<operation>_params_t * )

<application>_<operation>_register (<application>_<operation>_params_t * )

<application>_<operation>_find_fidelity (<application>_<operation>_params_t * )

<application>_<operation>_start_operation (<application>_<operation>_params_t * )

<application>_<operation>_stop_operation (<application>_<operation>_params_t * )

<application>_<operation>_cleanup_params (<application>_<operation>_params_t * )

The argument for the functions is a structure which encapsulates all of the variables found in the application description file as well as a few others that Chroma needs. The <application>_<operation> prefixes the function prototypes as well as the argument, thereby making them specific to a particular application and operation. For brevity, we drop the prefix when referring to a function or its argument.

These functions must be used in the order listed. The initialize_params function sets the elements in the structure to legal undefined values. This must be called before the structure is used in any other Chroma function. The cleanup_params function frees any dynamically allocated structure elements.

The register function registers the application with Chroma. The effect of this function is that the application has identified itself to Chroma as an adaptive application. Chroma can make use of this knowledge if it has any previous knowledge about this application, and also the number of adaptive processes in the system affects the adaptability of the entire system. This function is called once, after the Chroma function parameter has been initialized and before any operation.

The find_fidelity function is where Chroma makes adaptive decisions – e.g. whether to degrade the quality of the application, and whether to run parts of the computation locally or remotely. This is the single place where every variable of how an operation is executed (fidelity values as well as which specific remote server to use for remote execution) becomes concretized.

After the fidelity has been decided, the operation begins. The start_operation and stop_operation functions mark the beginning and the end of the operation. These must be inserted into the application so that Chroma can monitor and later predict the application resource demand per operation.

All retrieval and setting of the parameter structure elements should be done through macros that are also defined in the stub-generated files. The application developer should have no reason to dig into the structure or to be even aware of the layout of the structure.

This set of Chroma API does not have the capability for remote execution. For remote execution, the following function must be called.

<application>_<operation>_do_tactic (<application>_<operation>_params_t *, … )

This function handles all of the coordination and data marshalling necessary to carry out the particular execution tactic chosen in find_fidelity. It is the guts of this functionality, the decision-making to choose an execution tactic as well as the mechanism to carry it out, which is the focus of this project.

## 3.6. Application developer's effort

Required work for application developer in porting the application is (1), (2), (3), and (4) in Figure 3. Since (2) execution of Chroma stub generator and (4) final compilation are just command execution procedures, actual work for the developer is (1) writing the application description file and (3) insertion of Chroma API functions to the original source code.

Also separation of system-specific information and application-specific information will lead the following advantages.

- System-specific information

  Now application developer does not have to take care of this type information at all. Chroma API functions generated by the stub generator manages them internally. This can achieve followings.

  (1) Easiness for application developers without essentially unnecessary system-specific information

  (2) Higher overall system quality by avoiding configuration of those information by the developer's hand-written code

- Application-specific information

  Amount of application-specific information developer is required to handle is minimized by describing static information into the application description file. The information written inside the description file is managed inside the Chtoma API functions generated the stub generator which reads the description file.

  (3) Minimizing modification inside the original application source code will contribute to bug elimination during the porting phase.

  (4) The developer can concentrate their effort to insertion of Chroma API functions to the source code. This can be a good incentive of porting for application developers.

We want to state that the amount of effort spent is inevitably left up to the developer. Taking advantage of a few fidelity knobs returns some benefit for some amount of work. Adding remote execution returns more benefits for more work. We want to leave the choice to the developer.

# 4. Runtime system structure

Figure 2 shows the overview of runtime system structure including the chroma stub code inside applications. Application offers parameters and remote execution possibilities to the Chroma runtime. The

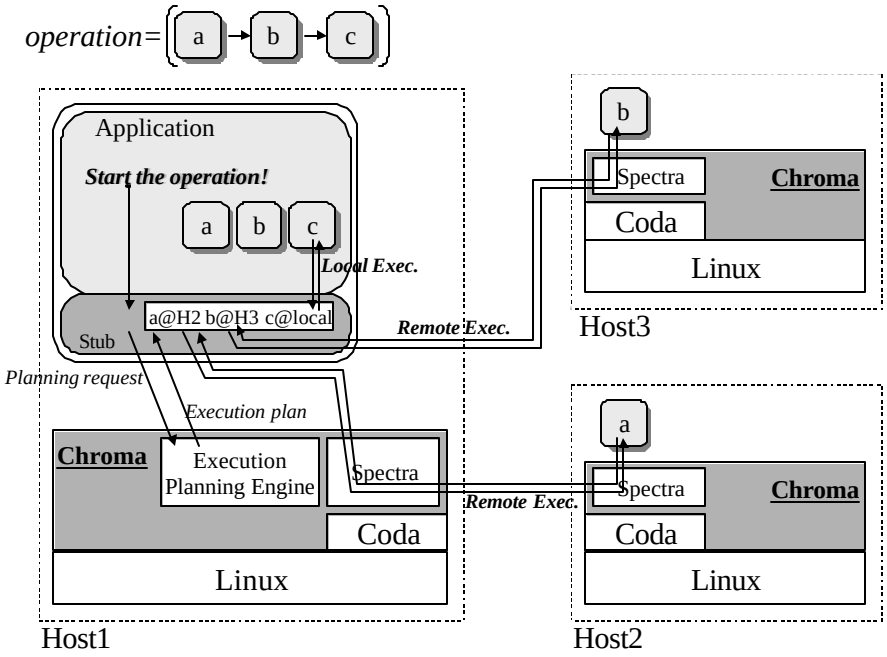


Figure 2: Stub generation and application compilation

Chroma runtime determines a fidelity value of application as well as the remote execution plan, according to the information on available resource. Receiving these values from the runtime, application invokes the stub function to start an operation. According to the execution plan provided by the runtime, the stub executes functions of the operation either locally or remotely.

# 5. Evaluation

Evaluating the value of our work poses its own challenges. A study of usability is beyond the scope of this project. Nevertheless, we provide some arguments from our experience of using the system to port a mobile application to run adaptively using Odyssey and Spectra. First, we capture its adaptive behavior, then insert the API calls into the application code.

## 5.1. Case study: Pangloss-Lite

Pangloss-Lite is a language translation system. It uses three translation engines which receive the same input sentence but produce varied output phrases according to their own algorithms and data files. The output is collected and fed into a language modeler which finds the best combination among the output phrases to produce the final translation.

We decided that Pangloss-Lite would degrade its quality by using fewer engines. There are several reasons for this decision. Each engine operates fairly independently from the other: each has its own data set, and the ability to switch a particular engine on or off was already a built-in feature. Each engine also had different resource consumption characteristics: one would be more CPU-bound, while another suffered noticeably from file cache misses.

The data file sizes of Pangloss-Lite are enormous. The data file for the language modeler, which is needed for every translation, is too large to sit in a hand-held mobile device. Therefore, remote execution

fit very well into this application. Furthermore, the three engines could potentially be run in parallel. Our language for parallel execution would greatly reduce the total latency of a translation. Since the choice of which engines to run and where to run them are tied together, we can express the both as possible remote execution tactics in the application description file. Once the hard work of making the monolithic application distributed, specifying the execution behavior was very simple.

The stub generator produced the source and header files for the API. The next step was to insert the custom API into the application.

The application was well-structured. This greatly eased the process of adding the API calls. Where to put them was obvious. The only change to the application code besides adding the API calls were minor: debugging code and memory allocation.

# 6. Related work

Our goal in this project was to develop a mechanism to make it easy for application developers to add new application to Odyssey [Noble97] [Flinn99] [Flinn01]. As such, we are not actually building an adaptive system, but rather, the tools needed to facilitate rapid application integration for an adaptive system. This related work will detail various runtime systems and mention the level of application developer tools provided for rapid application integration.

The adaptive system that we are using for our work is Odyssey. Odyssey was developed to support mobile applications and it currently supports fidelity degradation and remote execution of applications. However, it requires a lot of effort to add new applications into the Odyssey framework. This is because the application writer has to understand the internal workings of the Odyssey runtime and then write code to interface his application with Odyssey. We hope to make it easier to add applications into the Odyssey framework through the use of the specification language and the stub generator. At the same time, we hope to improve on Odyssey's method of deciding where and how to remotely execute applications by allowing applications writers to specify the remote execution possibilities of their applications.

Abacus [Amiri00] is a system that does remote function placement for data intensive applications. Abacus requires the application writer to create a few new functions for use by the Abacus runtime. Abacus also decides how to do the function placement using black box monitoring techniques. Abacus requires application writers to explicit modify their code to add the functions that the Abacus system requires.

River [Arpaci99] does parallel computation of large stream-based, compute-intensive applications. River requires the application writer to explicitly state the way in which the application should be run in parallel.

The Emerald [Jul88] system provides a programming framework for developing mobile code. It works at a much finer granularity than the system we are developing. Emerald is also a complete programming language that requires application writers to write their applications in this new language. Our system aims to be simpler to use for application writers who want to quickly integrate their applications into Odyssey.

Rover [Joseph96] is a toolkit developed at MIT that provides a way to construct mobile applications. Exploiting two ideas, Queued RPC and Relocatable Distributed Objects (RDOs), this toolkit provides applications with a uniform distributed object system. In terms of ease of use, Rover requires the application writer to rewrite quite a bit of his application in order to make it compatible with the Rover system. In our system, we hope to minimize the number and difficulty of the changes that need to be made to an application. Rover does not seem to have any mechanisms to automatically decide where to remotely execute applications. All remote execution decisions have to be explicitly stated by the application writer.

The Ninja [Gribble01] project at Berkeley is similar to Rover in that it also aims to provide a software infrastructure to support mobile applications. The Ninja architecture is basically implemented in Java and focuses on Java applications. Ninja requires application writers to write wrapper functions in order to add their new services into the Ninja infrastructure. These wrapper functions can be long and complicated to write. Ninja automatically places computational components on various servers according to the server's load. However, applications cannot specify specific remote execution plans.

There have also been several remote execution systems designed for fixed environments that analyzed application behavior to decide how to locate functionality. Coign [Hunt99] statically partitions objects in a distributed system by logging and predicting communication and execution costs. Our system is different in that it is allows application writers to specify more dynamic remote execution policies.

Condor [Basney99] monitors goodput to migrate processes in a computing cluster. Our system hopes to use hints from the application developer along with the current resource availability to determine how to remotely execute applications.

There have been a number of languages that allow application writers to specify remote execution possibilities for their applications. These include MPI [MPI94], PVM [Geist94], Obliq [Cardelli95], HORB [Horb], Telescript [White94], CORBA[Vinoski97] and Orca [Bal92]. Our system is different from these in that we ask the application writer to specify only the behavior of the application using our custom language. Based on this, our stub generator will create application specific code. The other languages require the application develop to code his application in those languages and this requires massive changes to the application if the application was initially coded in some other language.

# 7. Future work

There are still a number of areas in which this research can be improved. Firstly, it is necessary to derive a way for users to also provide input as to how they would like the applications to behave. Chroma currently only obtains information from the application developer. This information may be quite different from what the user requires and a way is needed to easily obtain that information.

Secondly, Chroma needs to be upgraded to make full use of the information provided by the application developer. Currently, Chroma makes very simplistic remote execution decisions even though the application writer is providing it with enough information to make more sophisticated decisions. Finally, more applications and users need to use Chroma to better evaluate our system's usefulness.

# 8. Conclusion

In this work, we have developed a language that allows application developers to specify the adaptive nature of their applications. This information is processed by a stub generator that generates application specify code to interface the application with the underlying runtime. We have shown, via case studies, that this method might be viable as a way of enabling rapid application integration into the Chroma environment.

# Bibliography


[Amiri00] Amiri, K., Petrou, D., Ganger, G., Gibson, G., "Dynamic Function Placement for Data-Intensive Cluster Computing", *Usenix Annual Technical Conference*, June 2000, pp. 307-322.

[Arpaci99] Arpaci-Dusseau, R., Anderson, E., et. al., "Cluster I/O with River: Making the Fast Case Common", *Workshop on Input/Output for Parallel and Distributed Systems (IOPADS)*, May 1999, pp. 10-22.

[Bal92] Bal, H.E., Kaashoek, M. F., Tanenmaum, A.S., "Orca: A Language for Parallel Programming of Distributed Systems.", *IEEE Transactions on Software Engineering*, 18(3), Marc 1992, pp. 190-205.

[Basney99] Basney, J. and Livny, M., "Improving Goodput by Co-scheduling CPU and Network Capacity", *International Journal of High Performance Computing Applications*, 13(3), Fall 1999

[Cardelli95] Cardelli, L., "A Language with Distributed Scope.", *Journal of Computing Systems*, 8(1), January 1995, pp. 27-59.

[Flinn99] Flinn J. and Satyanarayanan, M., "Energy-aware adaptation for mobile applications", *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December, 1999, Kiawah Island Resort, SC

[Flinn01] 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.

[Geist94] Geist, A. Beguelin, A., et. al., "PVM: Parallel Virtual Machine", *MIT press*, 1994.

[Gribble01] Gribble, S., Welsh, M., et al., "The Ninja Architecture for Robust Internet-Scale Systems and Services", *To appear in a Special Issue of Computer Networks on Pervasive Computing*.

[Horb] HORB home page, http:/www.horb.org

[Hunt99] Hunt, G. C. and Scott, M. L., "The Coign Automatic Distributed Partitioning System", *3rd Symposium on Operating System Design and Implemetation*, New Orleans, LA, Feb. 1999.

[Joseph96] Joseph, A., Tauber, J., and Kaashoek, M., "Building Reliable Mobile-Aware Applications using the Rover Toolkit", *Proceedings of the Second ACM International Conference on Mobile Computing and Networking (MobiCom'96)*. November 1996.

[Jul88] Jul, E., Levy, H., Hutchinson, N., Black, A., "Fine-Grained Mobility in the Emerald System", *ACM Transactions on Computer Systems, vol. 6, no. 1,* February 1988, pp. 109-133.

[MPI94] Message Passing Interface Forum, "MPI: A message-passing interface standard", *International Journal of Supercomputer Application*, 8(3/4), 1994, pp. 165-416.

[Noble97] Noble, B., Satyanarayanan, M., Narayanan, D., Tilton, J.E., Flinn, J., and Walker, K., "Agile Application-Aware Adaptation for Mobility", *Proceedings of the 16th ACM Symposium on Operating System Principles*, October 1997, St. Malo, France.

[White94] White, J. E.,"Telescript Technology: The foundation for the electronic marketplace", *White Paper*, General Magic Inc., 2465 Latham Street, Mountain View, CA 94040

[Wu98] Wu, D., Agrawal, D., and Abbadi, A. E., "Mobile Processing of Distributed Objects in Java", P*roceedings of the Fourth ACM International Conference on Mobile Computing and Networking (MobiCom'98)*. November 1998.

[Vinoski97] Vinsoki, S., "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments", *IEEE Communications Magazine*, 14(2), February 1997.

# APPENDIX A: Application description file language

# Language Grammar

The language grammar is divided into two main parts. They are the part relevant to Chroma and the part relevant to Prism. Some of the grammar is relevant to both Prism and Chroma.

## Chroma

Chroma requires three components. Namely

1) The basic information of the application (name, operation etc.)
2) The fidelity/tuning parameters of the application that are relevant to Chroma/Prism
3) The remote execution possibilities of the application. This is split into three subparts:
    a. Description of the procedures used for remote execution. This is similar to RPC definitions
    b. A list of servers that can be used by the application
    c. Description of the remote execution possibilities of the application. This involves the application writers specifying which procedures should run on which servers

## Prism

Prism requires two components. Namely

1) Mappings between Prism variables and application specified variables.
2) Mappings between Prism server names and application specified server names.

# Basic Information

This part of the grammar allows the application writer to specify the basic identifying information about the application. These definitions are also used by Prism.

## Grammar

Description = <APPLICATION> <OPERATION> <REST>
<APPLICATION> = **APPLICATION name** <TERMINATOR>
<OPERATION> = **OPERATION name** <TERMINATOR>
<TERMINATOR> = **;**

## Explanation

**APPLICATION** & **OPERATION** are keywords.
**name** is the name of the application and operation specified by the application writer.
The terminator **;** is used to demarcate the end of the line

## Example

APPLICATION janus;
OPERATION recognize;

# Fidelity parameters

This part of the grammar allows the application writer to specify the variables of the application that are required to be known by Chroma for the purposes of calculating fidelities etc. These definitions are also used by Prism.

## Grammar

<REST> = <FIDELITIES> | <REMOTE_EXECUTION> |
<FIDELITIES> = <TYPEDEF> | <VARIABLE> |

<VARIABLE> = <MODE> <TYPE>
<MODE> = **IN** | **OUT** | **INOUT**
<TYPE> = <INT> | <CHAR> | <BYTE> | <FLOAT> | <DOUBLE> | <STRING>
| <ENUM> | <TYPEDEF_NAME>

<INT> = **INT name** <FROM> <TO> <DEFAULT> <TERMINATOR>
<CHAR> = **CHAR name** <FROM> <TO> <DEFAULT> <TERMINATOR>
<BYTE> = **BYTE name** <FROM> <TO> <DEFAULT> <TERMINATOR>
<FLOAT> = **FLOAT name** <FROM> <TO> <DEFAULT> <TERMINATOR>
<DOUBLE> = **DOUBLE name** <FROM> <TO> <DEFAULT> <TERMINATOR>
<STRING> = **STRING name** <DEFAULT> <TERMINATOR>
<ENUM> = <ENUM_PARAMS> **name** <DEFAULT> <TERMINATOR>
<ENUM_PARAMS> = { <ENUM_VALUES> }
<ENUM_VALUES> = **name ,** <ENUM_VALUES> | **name**
<TYPEDEF_NAME> = **name name** <FROM> <TO> <DEFAULT> <TERMINATOR>

<FROM> = **FROM name** |
<TO> = **TO name** |
<DEFAULT> = **DEFAULT name** |
<TERMINATOR> = ;

<TYPEDEF> = **APPLICATION name** <TERMINATOR>
<OPERATION> = **OPERATION name** <TERMINATOR>
<TERMINATOR> = **;**

## Explanation

**APPLICATION** & **OPERATION** are keywords.
**name** is the name of the application and operation specified by the application writer.
The terminator **;** is used to demarcate the end of the line

### Example

APPLICATION janus;
OPERATION recognize;

# Remote Execution

## Defining the procedures used for remote execution and their interfaces

This part of the grammar allows application writers to specify the various Spectra operations in their application that would be of interest to Chroma. The interface for the Spectra operations is specified as such:

### Grammar

<REMOTE_EXECUTION> = <SPECTRA> | <SERVER> | <EXECUTION>

<SPECTRA> = **SPECTRA name** ( <parameters> ) <TERMINATOR>
<PARAMETERS> = <VAR> | <VAR> , <PARAMETERS>
<VAR> = <MODE> <TYPE> **name**
<MODE> = **IN | OUT | INOUT**
<TYPE> = **INT | CHAR | BYTE | FLOAT | DOUBLE | STRING |name**
<TERMINATOR> = **;**

### Example

*SPECTRA foo (IN STRING name, IN CHAR type, INOUT STRING Outfile, OUT BYTE flag);*
*SPECTRA bar (IN INT depth, INOUT STRING Outfile, IN BYTE flag);*

## Specifying the servers

This part of the grammar allows application writers to specify the servers that can be used by the application for remote execution. These definitions are also used by Prism.

### Grammar

<SERVER> = **SERVER** <SERVER_NAME> **= {** <SERVER LIST> }
TERMINATOR
<SERVER_NAME> = **REMOTE | name**
<SERVER LIST> = <SERVER ADDRESS> **,** <SERVER LIST> | <SERVER ADDRESS>
<SERVER ADDRESS> = **name** | **LOCAL** | **REMOTE**
<TERMINATOR> = **;**

### Example

SERVER fast_machines = {Delphi, Corinth, LOCAL};
SERVER REMOTE = {Mozart};
SERVER ALL = {REMOTE, LOCAL};

## Specifying the remote execution possibilities

This part of the grammar allows application writers to specify the ways in which the various SPECTRAs can be linked to one another. I.e., the various remote execution possibilities are defined here.

### Grammar

**<EXECUTION>** **= TACTICS operation_name = <POSSIBILITIES>**
TERMINATOR <DEFINES>

<POSSIBILITES> = <TACTIC_DEF> **OR** <POSSIBILITIES> |
<TACTIC_DEF> | **name OR** <POSSIBILITIES> |
**name**

<TACTIC_DEF> = <TACTIC_METHODS> <TACTIC_LEAFS> |
<TACTIC_LEAFS>

<TACTIC_METHODS> = <TACTIC_COMBO> | <TACTIC_PAR> |
<TACTIC_SEQ> |

<TACTIC_PAR> = **{** <TACTIC_LEAFS> **, (** <TACTIC_METHODS> **)**
<TACTIC_LEAFS> **}** | **{** <TACTIC_LEAFS> **}**

<TACTIC_COMBO> = **[** <TACTIC_LEAFS> **(** <TACTIC_METHODS> **)**
<TACTIC_LEAFS> **]** | **[** <TACTIC_LEAFS> **]**

<TACTIC_SEQ> = <TACTIC_LEAFS_N> **& (** <TACTIC_METHODS> **)**
| **(** <TACTIC_METHODS> **) &** <TACTIC_LEAFS_N>
| **(** <TACTIC_METHODS> **) & (** <TACTIC_METHODS> **) |**
<TACTIC_LEAFS_N> **&** <TACTIC_LEAFS_N>

<TACTIC_LEAFS> = <LEAF> <TACTIC_LEAFS> | <LEAF> |

<TACTIC_LEAFS_N> = <LEAF> <TACTIC_LEAFS> | <LEAF>

<LEAF> = **name** <SERVER_CONSTRAINT> **,** <POSSBILITIES> |
**name** <SERVER_CONSTRAINT>

<SERVER_CONSTRAINT> = **: server_name** |

<DEFINES> = <BEGIN> <DEFINES_VALUE> <END> |

<BEGIN> = **BEGIN**
<END> = **END**

<DEFINES_VALUE> = **#DEFINE name** <POSSIBILITIES> <TERMINATOR>
<DEFINES_VALUES> | **#DEFINE name**
<POSSIBILITIES> <TERMINATOR> |

<TERMINATOR> = **;**

## Example

```
SERVER fast  = {Delphi};

TACTICS   translate  = plan1 OR plan2 OR plan3;
BEGIN
#DEFINE plan1 {ebmt, ([dict gloss])} & lm;
#DEFINE plan2 {dict, ([ebmt gloss])} & lm;
# DEFINE plan3 {gloss:fast, ([ebmt dict])} & lm:fast;
END
```