This is to certify that the

dissertation entitled

ASAP FOR DEVELOPING ADAPTIVE SOFTWARE WITHIN DYNAMIC HETEROGENEOUS ENVIRONMENTS

presented by

Ren-Song Ko

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science

Matt W. Mutka
Major professor

Date April 29, 2003

ASAP FOR DEVELOPING ADAPTIVE SOFTWARE WITHIN DYNAMIC HETEROGENEOUS ENVIRONMENTS

By

Ren-Song Ko

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

2003

Abstract

ASAP for Developing Adaptive Software within Dynamic Heterogeneous Environments

By

Ren-Song Ko

As the improvement of hardware technology and increased deployment of wireless infrastructures, computer systems are increasingly heterogeneous and dynamic. Heterogeneity and dynamic raise several challenges for software developers and users. Such challenges are software portability, software or computing environments configuration, unexpected environments, and platforms with limited resources. A feasible approach for these challenges is to develop adaptive software that may be aware of environments and adapt itself.

We propose and implement the adaptive software architecture project (**ASAP**), which includes the software framework (**FRAME**), Component/Constraint Markup Language (**CSML**), and on-line adaptation, for the challenges under dynamic heterogeneous environments. Software may have a list of constraints for the environment. During execution, software may use the list to determine whether the environmental changes, and

then the software responds accordingly. **ASAP** may support applications in different domains, including ad hoc systems that are temporarily organized.

FRAME provides the APIs for adding adaptation to software. The adaptation and functionality of software are separately specified and implemented under **FRAME**, and hence, lessen development complexity. To further alleviate the complexity of developing adaptive software under **FRAME**, a high-level adaptive software specification language, **CSML**, may be used to specify the architecture of the software system and the constraints. The code for adaptation will be automatically generated by **CSML**. Then, developers only need to inherit the generated code to implement the functionality.

To avoid an unnecessary performance impact, **ASAP** uses two different on-line adaptation schemes, **Brew** and component reassembly, to let software respond to different scales of environmental changes. For small environmental changes, people may use **Brew** to repair the affected software without terminating the software. Otherwise, the execution of the software will be temporarily suspended to invoke the component reassembly, and then resumed with appropriate implementations of the components. Also, two approaches, partial reassembly and caching, are proposed to further reduce the performance impact of reassembly.

Three different applications are used to evaluate various features of **ASAP**. The GO game demonstrate the basic features of **ASAP**, including distributed component assembly. The Java MPEG Player has time constraints, which may not be satisfied under certain conditions and may be repaired by **Brew**. The last application demonstrates how **ASAP** may be used to realize behavior-based approach for the adaptive robotic control systems. Different performance results are also presented and discussed.

To *my family*

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Dr. Matt W. Mutka, my major advisor for his dedication and guidance throughout all the stages of the dissertation and for particularly standing by me throughout my doctoral program at Michigan State University. He allowed and even encouraged me to pursue my own interests, fostered my independent thinking, tolerated and helped me overcome my frustrations. He was a source of inspiration and a role model in many regards, especially discipline and high level of professionalism.

I am especially indebted to the faculty members of eLANS research lab, Dr. Lionel M. Ni and Dr. Abdol-Hossein Esfahanian, for their assistance in many aspects of my time at eLANS, helping me through the difficult times, and their invaluable time spent, with Dr. Chichia Chiu, on my committee.

I would also acknowledge all professors - too many to go through the list- who taught me the advanced and excited knowledge in engineering and mathematics. Special thanks also go to all my fellow students. They made the pursuit of my Ph.D. an enjoyable and invaluable professional experience.

I wish to thank Dr. Ronald S. Harichandran in the Department of Civil and Environmental Engineering for giving me the opportunity to work on the MFPDS pavement project. It not only provided the first three years of financial support for my study, but also a pleasant experience on cross-disciplinary research.

Words can not express my appreciation and gratitude for the love, sacrifice, support, and encouragement so generously given by wife, Su-Yin Li. Special thanks to my daughter, Meg, and son, Albert, for bringing joy to my life.

Finally, I would like to dedicate this dissertation to my parents. Without their love and moral support, it would not have been possible for me to complete this study.

TABLE OF CONTENTS

List of Tables

List of Figures

Chapter 1

Introduction and Motivation

1.1 Trend

1.1.1 Heterogeneity

The evolution of modern computing and computer technology may be traced back to the invention of the electronic digital computers, the Atanasoff-Berry Computer (ABC) and Electrical Numerical Integrator And Calculator (ENIAC) [1]. Since then, various scale of computers, from mainframe to handheld devices, have been developed with notable improvements in hardware technology.

Computer processing capability has basically followed Moore's law, which can be summarized as "The number of active devices we can place on a given area of silicon doubles every 18 months" [2] (This is actually a revision of Gordon Moore's 1965 estimate of doubling per year and the later 1995 estimate of doubling every two years.) This trend's obvious consequence is that we can continue to increase the capability of

devices fabricated in a given area of silicon. Instead of designing systems built from separate board-level components, we can integrate diverse functionality onto a single chip, resulting in remarkably compact consumer electronics. Similarly, the reduced capacitance resulting from smaller transistor dimensions means that we can operate these devices at higher speeds, increasing their effective performance. Additionally, reduced transistor sizes decrease power consumption, alleviating some of the perpetual problems surrounding energy storage technologies.

The combination of more transistors on a given area of silicon and a reduced power budget has brought us the capabilities of mid-1980's desktop computers in today's battery-operated, handheld PDAs. Two examples are the Motorola Dragonball [3] and Intel StrongARM[4] processors, the most common processors used by today's PDAs. Besides providing low power consumption and high performance, these processors integrate their DRAM and LCD controllers and a host of other interface I/O capability on the same die.

Also important trend over the last decade is the emergence of specialized and task-specific hardware, such as spell checkers, calculators, electronic translators, electronic books, and Web pads. These devices have a specialized interface and address the desired goal of ease of use. In contrast, a PC is a generalized machine, which makes it attractive to purchase, the one-time investment having the potential for many different uses. However, in the other ways it adds a level of complexity and formalism that hinders the casual user.

Thus, we can conclude that future computing environments are becoming heterogeneous, in which computing environments composed of a wide range of devices, with

diverse hardware architectures, operating systems, and purposes, interconnected networks.

1.1.2 Dynamic

Progress in wireless connectivity has increased. This area has witnessed two distinct development trends. The first is in short-range connectivity standards, such as Bluetooth (IEEE 802.15) [5] and the IrDA (Infrared Data Association) [6] standards, which are primarily for simple device-to-device communication. The second trend is in wireless LAN technology, such as the 11-Mbit-per-second IEEE 802.11b standard and the more recent 54-Mbps IEEE 802.11a standard [7], which allow for communication cells that span many hundreds of feet with sufficient bandwidth. Wireless technologies provide for two basic needs: the ability to detect location and the more basic ability to communicate, which nourishes a mobile computing environment. The document from Intel Corporation [8] reveals the trend of the future computing environments. The data it cited from Gartner, Inc. [9] shows that

> Market Trends Intel and third-party market research firms expect mobile PCs to comprise a strong segment of the overall PC market, with significantly higher demand for notebooks in the Mobility market segment. Gartner's forecast through 2006 shows approximately 50 percent growth in general PC sales, but 100 percent growth in the mobile PC segment alone. In other words, mobile PC sales will increase twice as fast as the overall PC market in the coming years.

Furthermore, the market research firm International Data Corporation (IDC) predicts [10] that

> Just as important is the rapidly growing deployment of wireless LANs and Bluetooth wireless technology, which are projected to reach adoption rates in new notebooks of 80 percent and 94 percent respectively through 2006. Higher speeds, greater interoperability and lower prices are driving the rapid, widespread adoption of wireless LANs. In turn, wireless connectivity enables new Mobility usage models, creating a user-driven demand for wireless LANs in new locations.

Also, in [11], IDC says that the number of mobile workers in the U.S. will increase by 12.7 million between 2001 and 2006, from 92 million to 105 million. In contrast, the number of workers who are not mobile will decline by 2 million through 2006, down to 53.8 million. What does this mean? By the end of 2006, almost two-thirds (66.0 percent) of U.S. workers will be classified as mobile workers. IDC's research findings include:

1. Mobile professionals will grow from 18.2 million in 2001 to 24.1 million in 2006, driving the highest technology investment of any mobile segment.

2. Mobile non-travelers, the workers who rarely leave town but who are often in meetings or away from their desks, will grow by 10 percent annually to more than 13 million in 2006.

3. "Occasionally mobile" workers, who travel less than 20 percent of the time, are actually declining in number. But rather than retiring their carry-on luggage, a growing number of these workers are graduating to the next level and joining the more frequently mobile professionals.

Thus, we can conclude that the future computing environments are dynamic; i.e., the usage of computers is not confined within a fixed location, and people carrying computers may join and leave the environment at their will. Computing environments may change during the execution of applications.

1.2 Identified Problems

Although hardware technology has significant improvements over years, unfortunately, progress in the area of software has not matched the great advances in hardware. Software has become the major cost of many systems because programming productivity has not increased very quickly. Although new programming techniques, such as object-oriented programming, have been developed to help relieve this problem, dynamic heterogeneous environments bring more challenges to the development, deployment, and usage of software.

Several software issues are discussed in the following subsections.

1.2.1 Software Portability

Software development usually consists of several stages, including design, implementation, test, and maintenance. Most of these stages are platform dependent. Thus,

with the number of platforms increasing over time, to make software cross platform portable may reduce the cost of software development on multiple platforms.

Based on amount of effort required for people to deploy and execute the software on different platforms, software portability may be classified into several levels [12].

Source Code Portability

Cross platform compilability provides source code level portability. With careful coding and help of some development tools, such as Autoconf [13] and Automake [14] on various Unix-like environments, the source code of software may be compiled into native code for various platforms. Nevertheless, it is probably difficult to deploy and use the software. For example, the source code has to be compiled for the target platform, either by vendors or users, and the computing environment needs to be correctly configured, such as required hardware and shared libraries. The software are usually distributed and executed in native code for a specific platform, and have performance advantage.

Nevertheless, software vendors may not have enough resources to support and accomplish the compilation on every platform. If done by users, the source code needs to be released to the users, which may not be practical for some vendors for the intelligence property issues, and users may not have the development tools, such as the compiler and linker, or knowledge to complete the compilation.

Bytecode Portability

Instead of native code, the software may be compiled into platform independent intermediate bytecode that will be executed by a program, usually referred to as a virtual machine, rather than by the "real" computer machine, the hardware processor. The software may be distributed in bytecode and is expected to work on any platform as long as there is a virtual machine. It mitigates the difficulty for software deployment and use, because recompilation for a target platform is not necessary.

The LISP language, used in artificial intelligence applications, is an earlier language that uses the bytecode and virtual machine approach. Other languages that use bytecode or a similar approach include Icon [15] and Prolog. More popular object-oriented language, the Java platform - i.e., language, class library, and virtual machine - promises "Write Once Run Anywhere," which, if even reasonably approximated, also provides platform independent bytecode level portability. The execution in bytecode has performance disadvantage, since extra work is needed for the virtual machine to interpret the bytecode. However, the benefits of platform independency often outweighs the performance as machines become more faster, because Java may substantially reduce software design cycle time, learning curve, development and maintenance costs.

Performance Portability

While the bytecode and virtual machine approach might greatly alleviate the difficulty of software portability, it does not apply in the performance- or resource-sensitive do-

main, such as real-time and multimedia, since it is impossible to predict the computation and resource capabilities of so many target platforms during the development stage.

For instance, the correctness of the real-time software depends not only on the logical result of the computation but also on the time at which the results are produced. Two platforms with different capability may produce the results at different times. The same application may run perfectly on a high-end machine but fail on a low-end machine because the results it produces could not meet the real-time constraints. Specifying the minimum requirements of the target platforms to ensure correctness may not solve completely solve this problem, since it limits portability. Furthermore, run-time environmental changes, such as a user invoke another application that may compete the same resource, still may cause failure of the execution.

In this situation, it might be desirable to lower the quality of output on a low-end platform to meet the time constraints. Moreover, if a platform provides some special functionality, such as QoS, the performance may improve if the software has knowledge of the functionality and may take advantage of it. Therefore, to achieve "same performance everywhere," software may probe dynamically computation or resource capabilities of the platform and reconfigure, or adapt, themselves to different computing and communication environments instead of assuming a priori knowledge of the capabilities of the target platform.

1.2.2 Software or Computing Environments Configuration

Obviously, software is getting complicated over time since more functionality is added as technology improves. Software and the computing environment sometimes have to be configured, such as existence of required libraries, matching of communication ports, exchange of security information, resolution of conflict between software, and so on. Without enough knowledge, it may be tedious for a user to get software to function correctly.

The heterogeneity and mobility will make configuration more difficult. For example, while a user watching a video on-line is moving around a building, the network traffic may change several times during the moving. It is impractical for user to change the resolution of video to maintain smooth playback. Also, the dynamic environment may make this problem unavoidable even if the user stays in a fixed place. Thus, to improve software usability under heterogeneous and dynamic environments, the software must be able to configure themselves automatically to the environments.

1.2.3 Unexpected Environments

Computing environments keep changing and it is impossible for software developers to model and predict what the computing environments will change over time. Unexpected situations may happen, and software have to be terminated or even systems need to be shut down for updating the system software for handling the unexpected environments. However, it may not be practical for some mission critical systems, since execution termination for updating system software is expensive and may lead

to disastrous consequences. Thus, for designing robust and fault-tolerant systems, there are needs for software dynamically to update the code without termination.

1.2.4 Platforms with Limited Resources

As mentioned earlier, today, computer devices are becoming smaller. Special purpose devices begin to emerge. For those devices with small form factors, such as mobile devices, the computation and memory are scarce resource, and the devices may not be able to execute some resource intensive software. However, if there are needs for such software, it will be nice for these small devices to seek extra resources automatically from environments.

For example, some computation intensive tasks may be automatically sent to the machine with plenty of computation capability, or several devices may form an *ad hoc systems* to perform the task together without too much human intervention. Furthermore, a device with limited memory may only load part of the software; that is, code may be dynamically loaded and unloaded as necessary to save memory resources.

1.3 Our Contribution

To overcome the above problems, software need to be aware of the computing environment. Rather than assuming what the environment should have, software will inquire what the environment has and makes an adjustment when the environment has something missing. Software may have a check list, or called *constraints* in later

chapters, to gather information about the environments. Criteria of the check list are specified during the development stage. During the execution, software may check the list to determine if there is something wrong or has changed in the environment, and then the software responds accordingly.

Nevertheless, specification of criteria raise challenges in complex software systems development, since the specified criteria may not be orthogonal. Because the criterion is checked individually and sequentially, mutual dependency of criteria may lead to false conclusions. For example, a video player may inquiry two criteria "Is it possible to play back at highest speed?" and "Is it possible to play back at highest resolution?" Both may be true if individually checked, but may not be true at the same time, since performance usually declines when achieving better quality. The mutual dependency of criteria needs to be considered. Moreover, today complex software systems are usually collaborative projects. The criteria that individual developers envision may sharply differ. While developers of higher level software are concerned about overall performance, developers of lower level software may be only focused on the performance of sub-tasks.

We propose and implement the adaptive software architecture project, or **ASAP**, for the problems and challenges under dynamic heterogeneous environments. The research problems we address are

- How does software gather resource information about a broad range of computing environments?
- How does software seek the best code for execution under specific environments?

- How may software be updated or repaired in response to changes in the computing environment?
- How may adaptation be easily integrated with the functionality of software?
- How do users specify performance and the quality of software for the purpose of enhancing performance portability?
- How do developers specify constraints and their dependencies for a variety of levels of software?
- How to minimize the performance impact of adaptation on the execution of software?

1.4 Possible Applications

There are many various possible applications of adaptive software in different domains, particular the applications that need

Environment-awareness. Some applications may need information about the environment to provide better services. For example, how to quickly and conveniently retrieve useful and desired information may be the top priority for users. Rather than downloading full multimedia content, web browser may be able to automatically extract text only information if it is aware of the slow network traffic, limited computation power, or battery capacity.

Portability. Some resource-sensitive applications, such as real-time, are not cross platform portable, particular at the performance portability level. It may be impractical to port the applications to every platform due to development cost. Adaptation may be a feasible solution. For example, a video player may reduce the resolution or drop the frames to boost the playback speed on mobile devices.

Robustness. For some mission critical systems, down time may be expensive. Some examples may be e-commerce servers and robots used to explore unknown territory. Software may sense system breakdown or unexpected events. Software may be able to update itself without down time.

On-the-fly task assignment. Some missions of systems may be determined on-the-fly. Thus, the code to handle the missions may be uploaded on-the-fly. For example, a robot may be assigned to different missions at different locations and different times. It may also be given a different mission for collaborative tasks if there are other robots nearby.

In addition to above applications, one particular interesting application is to realize ad hoc systems.

1.4.1 Ad Hoc Systems

As technology improves, small devices and task-specific hardware begin to emerge. These devices usually have limited resource or have a specialized interface to address the desired goal of ease of use. Each single device may not be able to handle resource-intensive software with reasonable performance, quality, or ease of use. A

straightforward approach is to combine several devices to performance the task together.

As illustrated in fig. 1.1, instead of running software on a cell phone, one may connect several computers or smart appliances together to form an ad hoc system. The **ASAP** will automatically distribute appropriate part of the code of the software to each participating machine based on the criteria list. Then these machines will execute the assigned code to accomplish the task collaboratively. Such a system is called ad hoc because it is an unplanned and temporarily organized system, only for short-term usage and execution for a specific task.

For example, a video player may run on a PDA as shown in fig. 1.2. Because of the limited resource form factors, the video and audio quality may not be as good as a home theater system, and the user interface may not be as intuitive and friendly as a remote control.

Fig. 1.3 illustrates that the video player runs on an ad hoc system consisting of smart LCD displayer, smart speaker, smart remote control, etc. Each participating machine is designed for handling special tasks, such as the remote control for user input. Appropriate software will be automatically distributed without human intervention. For example, **ASAP** may distribute the code for handling video rendering to the LCD displayer, the code for handling audio to the speaker, the code for user interface to the remote control, and the code for decoding video stream to the desktop computer. Therefore, instead of watching the video on a PDA, one may watch the video on a temporarily connected ad hoc system with better performance, quality, and ease of use.

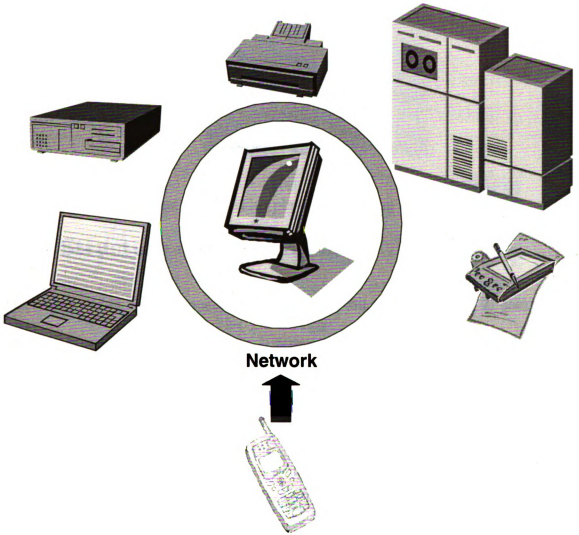

Figure 1.1: Architecture of an ad hoc system

Figure 1.2: A video player running on a PDA

1.5 Structure of the Content

The remainder of this document is structured as follows. In chapter 2, background information and survey of literature is presented. In chapter 3, the architecture of the adaptive software framework, **FRAME**, and development of adaptive software under **FRAME** are described. A solution to the problems for mutual dependency of criteria raised in complex and collaborative software systems development is also proposed. Chapter 4 describes **CSML**, Component/Constraint Markup Language, a high-level adaptive software specification language to alleviate the complexity development of adaptive software under **FRAME**. Chapter 5 addresses the performance issues for the adaptation during the execution, and proposes several approaches to improve the performance. In chapter 6, three applications from different domains are used to

Figure 1.3: A video player running on an ad hoc system

demonstrate and evaluate various features of **ASAP**. Finally, a summary and possible future work are outlined in chapter 7.

Chapter 2

ASAP Overview and Literature Review

2.1 ASAP Overview

The adaptive software architecture project, or **ASAP**, which includes **FRAME**, **CSML**, and **Brew** [12, 16], are designed to helps people develop, deploy, configure, and execute Java based software under heterogeneous and dynamic computing environments.

The Java platform - i.e., language, class library, and virtual machine - promises "Write Once Run Anywhere," which, if even reasonably approximated, provides bytecode level portability. Therefore, Java is used in the development of **ASAP**, since to achieve performance portability implies that the bytecode portability has to be achieved.

FRAME is the adaptive software framework that provides APIs for applications to adapt themselves to heterogeneous environments. The bytecode of applications are not linked during the development stage, but are "assembled" on-the-fly after con-

sulting the resources of the computing environments. If applications fail to work appropriately because of dynamic environmental change, applications may also use the **FRAME** APIs to replace part of the bytecodes to produce better performance or quality of execution without down-time.

CSML is a high level specification language that allows adaptive application developers to specify how the applications will be assembled, which part of the bytecodes may be replaced to respond to the dynamic environmental change, and the definition of resource requirements, desired performance or quality of execution. Once specified, **CSML** will automatically generate the code for adaptation. Thus, developers may focus on the design of software architecture and implementation of the functionality. The infrastructure for applications working under **FRAME** is automatically generated from the specifications.

Although **FRAME** provides APIs for applications to replace part of their bytecodes to respond to the dynamic environmental changes without the down-time, it may have significant performance impact since the execution of applications will be temporarily suspended for the bytecode replacement. `Brew` is a light weight on-line adaptation. The idea of `Brew` is computation steering, in which people may instrument computation data from the applications and then steer the execution behavior by changing some parameters of the applications. Thus, for minor environmental changes, the inappropriate execution of applications may be repaired by changing the applications' parameters with help of `Brew`.

The related work about the heterogeneous computing, adaptive systems, and high level software system specification are described in the following sections. Details about **FRAME**, **CSML**, and **Brew** are described in the next several chapters.

2.2 Pervasive Computing

For many tasks today, the use of computers is not entirely satisfactory. The interactions take effort and are often difficult. The traditional, and still prevalent, computing experience is sitting in front of a monitor, our attention completely absorbed in the dialog required to complete the details of a greater task. Although the real objective is the task's completion, not the interaction with the tools we use to perform it, it is not uncommon that sometimes the effort required to make the tools, i.e., computers, work is more complicated and difficult than completing the tasks themselves.

Mark Weiser wanted to explore whether we could design radically new kinds of computer systems. These systems would allow the orchestration of devices with non-traditional form factors that lend themselves to more natural and tacit interaction. Rather than just traditional interaction, such as keyboard and mouse, the working space, allowing position and manipulative [17] will be integrated into these systems. In addition to specialization and the use of embedded computers, support for mobile computing and wireless data networks is an important facet of this vision, in other words, invisible connectivity. With the invisible hardware and network, it will become possible to build computer systems that do not distract the user; ideally, the user might even forget the system is present [18]. In essence, Weiser was proposing

that well designed computer systems would become invisible to the user and that our conscious notion of computer systems would begin to disappear. Some years later, the book *The Invisible Computer* [19] by Don Norman popularized this concept. The survey of the progress toward Weiser's vision from various system viewpoints will be illustrated in the following subsections.

2.2.1 Hardware

Past and Current Projects

In the early 1990s, there are several projects at Xerox Palo Alto Research Center (PARC) embraced this research direction, including ParcTab [20], Mpad [21], and Liveboard [22]. Olivetti Research's Active Badge [23] and Berkeley's InfoPad [24] projects are also inspired by Weiser's vision about ubiquitous computing.

However, due to the limitation of technologies, many of the early systems fell short of designer expectations. For instance, in 1990, there were no Wireless Local Area Network standards. The processors suitable for mobile devices operated at only a few megahertz, while PCs were typically shipping with up to 50-MHz processors. The early electronic organizers were equipped with 128 Kbytes of memory, while PCs shipped with 30-Mbyte disks. The displays were not impressive: laptops used monochrome VGA, and the few handheld devices available mainly used character-based displays. There were no pen-based devices; most computers used keyboard and mouse as interface.

The combination of more transistors on a given area of silicon, increased storage density, improved display technology and a reduced power consumption brought the capabilities of mid-1980's desktop computers into today's battery-operated, handheld PDAs. Ubiquitous computing is ready to focus on getting computing "beyond the desktop."

Most current ubiquitous system research projects fall into two categories [25]. The first one is personal systems that give users access to computing independent of their physical location at the cost of them having to carry some equipment including mobile and wearable systems. For instance, personal servers [26] can enhance ubiquitous access to data. They form the computation and storage center of a person's digital experience. Devices such as a cell phone or PDA-like appliance communicate directly with this central server, providing a common representation of a user's data. These devices, therefore, merely represent an interface into this central repository. With this model, the personal server could be located out of easy reach, for example, in a user's shoe, handbag, or belt clip, without causing inconvenience.

The other category is infrastructure systems, which are associated with a particular physical locale. Many research challenges shift from issues of size, weight, and performance to those of deployment, management, and processing. Consider thousands of miniature temperature sensors around a room. How were they deployed? How is the data collected? How are faulty components identified and replaced [27]? Furthermore, with shrinking hardware devices, they may be out of sight and eventually out of mind. As a consequence, keeping track of these devices in human understandable way will become difficult.

Several hardware projects, such as the Berkeley motes [28], have started to explore this issue by creating a fairly small wireless sensor platform. This lets researchers actively explore the networking protocols necessary to organize large sets of nodes. Currently about the size of three stacked US quarter-dollar coins, these devices will keep shrinking until they reach the size of smart dust [29] and can no longer be seen or directly manipulated. The Berkeley Pico Radio project is pursing a single-chip system that incorporates both processing and radio frequency subsystems [30].

Future Directions

There have been three core challenges of ubiquitous computing hardware and will likely continue far into the future: form factors, energy, and the user interface. These problems are difficult because they are not independent and are somewhat contradictory; a solution in one space greatly confounds that in another.

- **Form factors**. Form factors, particularly size and weight, are significant obstacles to ubiquitous computing because they continually remind the user of the hardware's presence. This limitation has impact on both mobile systems when the user must carry the device and configuration of infrastructure-based systems involving many pieces of equipment. Unfortunately, the two main contributors to a device's size and weight come from the other two fundamental problems: batteries and the user interface.

For instance, the Itsy pocket computer [31] is just 70 percent larger than its 2.2 watt-hour battery and 320×200 pixel display. In addition, these two components represent 60 percent of the total weight without the case but 43 percent with the case.

- **Energy**. Another challenge for any mobile system is to reduce user involvement in managing its power consumption. Energy is a necessary resource for virtually all computing systems, but any reference to it detracts from a positive user experience. The degree to which an energy source distracts the user depends on the intended application, the hardware implementing the application, and the energy source's characteristics.

Solutions to this problem fall into two approaches. The first is to reduce power consumption. Part of this approach consists of designing energy-aware software that can identify the hardware states that provide a given service level and select those that are most energy efficient. For instance, in systems with a microprocessor whose energy consumption is greater at high speeds, the software can select the lowest speed possible that still achieves the required task's performance [32]. The control software can also modify the quality of service it seeks to deliver [33]. For instance, to save energy, the software could reduce the frame rate, or size, of an MPEG movie, incrementally resulting in a corresponding loss of fidelity. Or, the software could forward a voice utterance to a remote system for recognition rather than expending local energy on the task [33].

Another approach is to find alternative and improved energy sources. Although battery energy densities are projected to increase approximately 10 percent annually for the next three years [34], the energy consumption will likely increase at the same rate. This will lead to research for other technologies for storing and even generating energy. For example, MIT researchers have exploited the human body as an energy source by constructing sneakers that use flexible piezoelectric structures to generate

energy [35, 36]. Similarly, solar radiation, thermal gradients, mechanical vibration, and even gravitational fields all represent potential power sources for a mobile device [27].

Additionally, storage technologies under development promise much greater energy densities than those of conventional batteries. In the near term, one of the more promising technologies is fuel cells, particularly direct methanol fuel cells [37]. Pure methanol fuel offers an energy density roughly 40 times that of a Lithium-ion polymer battery, but 70 to 90 percent of this chemical energy is lost in conversion to electricity. In the longer term, technologies such as MIT's MEMS (micro-electromechanical systems)-based microturbine and associated micro electric generator might provide highly compact energy sources with significantly longer lifetimes.

An alternative approach is to transmit energy to a mobile device that, hence, may reduce its energy requirement. This technique is difficult to do safely over a long distance, but it is applicable to the field of passive electronic tagging. For example, *Radio Frequency Identification* [38] tags are inductively powered by the tag reader, typically up to a maximum of one meter, employing load modulation to transmit their data back to the interrogator. Such passive tags have unlimited lifetimes, are smaller, and cost less; however, unlike battery-powered (active) tags, they can communicate only over a short range and cannot autonomously signal their presence. The ability to transmit energy, when combined with robotics, gives us the capability for mobile computation that can recharge itself.

- **User interface**. Making user interfaces invisible is fundamentally difficult due to the tradeoff between form factors and usability. Reducing the size and weight will

make the device less visible but might decrease its usability. The degree to which these two components matter depends on the specific properties of the interface and of the application for which it is being used.

Today's most popular user interfaces include buttons, keyboards, mice, pointers, LCD panels, touch screens, microphones, and speakers. Each is designed and suitable for a specific application class. For example, a keyboard and display seem ideal for writing an article, but a microphone and speaker would probably be better for communicating with another person. For other applications, such as alerting a user that he or she has received mail, these user interfaces are unnecessarily complex.

Although these interfaces are distinctly separate from our bodies, the natural implication for disappearing interfaces is for the two to blend together. Several researchers are exploring the possibility of interpreting information from our neurons to let us control computers and other machines by just thinking about doing so [39]. Early work involving a monkey with neural implants has demonstrated the ability to gather sufficient information to let a robot mimic the monkey's arm movement.

2.2.2 Software

Ubicomp computing systems are prone to be local and unpredictable, two characteristics identified as *Boundary Principle* and *Volatility Principle* by Kindberg et al. [40].

Boundary Principle. A ubicomp system involves some integration between computing nodes and the physical world and ubicomp takes place in more or less

discrete environments based, for example, on homes, rooms, or airport lounges. Therefore, from physical integration, the Boundary Principle may be drawn:

> Ubicomp system designers should divide the ubicomp world into environments with boundaries that demarcate their content. A clear system boundary criterion often, but not necessarily, related to a boundary in the physical world should exist. A boundary should specify an environment's scope but doesn't necessarily constrain interoperation.

Volatility Principle. In an environment, there are components, units of software, that implement abstractions such as services, clients, resources, or applications. An environment can contain fixed infrastructure components, and spontaneous components based on devices that arrive and leave routinely. In an ubiquitous system, components must spontaneously interoperate in changing environments. That is, if a component interacts with a set of communicating components, it may change both identity and functionality over time as its circumstances change. From spontaneous interoperation, the Volatility Principle may be drawn:

> You should design ubicomp systems on the assumption that the set of participating users, hardware, and software is highly dynamic and unpredictable. Clear invariants that govern the entire system's execution should exist.

Physical integration and spontaneous interoperation have major implications for software infrastructure. These ubicomp challenges include a new level of component interoperability and extensibility, such as discovery and integration, and new dependability guarantees, including adaptation to changing environments, tolerance of routine failures or failure-like conditions, and security.

Discovery and Interaction

Spontaneous interoperation makes shorter-term demands on our components' interoperability. That is, devices may interact with partners of varying functionality over time. Thus, as a component enters an environment, it will face some issues such as bootstrapping, service discovery, and interaction. Additionally, if the arriving component implements a service, the components in the service's environment might need to discover and interact with it. Service discovery and interaction are generally separable (although the Intentional Naming System [41] combines them).

- **Bootstrapping.** The component requires a priori knowledge of addresses (for example, multicast or broadcast addresses) and any other parameters needed for network integration and service discovery. This is largely a solved problem. Protocols exist for joining underlying networks, such as IEEE 802.11 [7]. It is also possible to dynamically assign IP addresses either statefully, using DHCP [42], or statelessly in IPv6 [43].

- **Service discovery.** A service discovery system dynamically locates a service instance that matches a component's requirements, and thus solves the association problem. Address allocation and name resolution are two examples of local services

that an arriving component might need to access. Client components generally require an a priori specification of required services, and corresponding service components must provide similar specifications to be discoverable. Several systems perform service discovery [41, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53]. Each provides syntax and vocabulary for specifying services. Each service discovery protocol has adopted different service description models: Java service interface and assistant attribute objects of Jini, string-based attribute-value pairs of SLP, and XML descriptions in case of UPnP. Although there are pros and cons of their design decisions, e.g., their query efficiency and expressiveness, they typically consists of attribute-value pairs such as *serviceType=printer* and *type=laser*.

A challenge in ubicomp service description is avoiding over-specification, in which service requester and provider must obey the same vocabulary and syntax. Association can't take place if a component requests *serviceType=printing* when the provider has *service=print*. Agreement on such details is a practical obstacle, and it's debatable whether service specification can keep up with service development.

Other designs, such as Cooltown [53], tried to use more abstract and much less detailed service specifications to get around such problems. For example, a camera could send its images to any device that consumes images of the same encoding (such as JPEG) without needing to know the specific service. But abstraction may lead to ambiguity. If a camera were to choose a device in the house, it would potentially face many data consumers with no way of discriminating between them. Also, it's not clear how a camera should set the target device's parameters, such as image resolution, without a model of the device.

Another problem is that most of the service advertisements and queries are only able to capture only static aspects of context: for example, a server load attribute can indicate the load on a server machine at the moment of service announcement but not the one at the time of service query later, because it is dynamically changing. Service matching based on those declarative, static service descriptions provides minimal service discovery and filtering, leaving the rest of the work to users' manual selection; they have to try discovered service instances one-by-one, until they find an instance with satisfactory quality of service. But it would be a painful process if their mobile devices do not have enough computing and network resources needed for the manual selection. The problem gets even worse, when service population is large.

One possible approach [54] is to exploit useful context information relevant to service discovery. By adding extra dynamically determined attributes, called *context attribute*, it enables the right services to be delivered to the right users at the right time and place via refined service selection and recommendation, which is not attainable by declarative service description models. A good survey on how context can be beneficial in the ubiquitous environment is given by Chen et al. at [55].

To define the *boundary* is also an issue for meaningful discovery. A client in San Francisco may not interested in the services in Tokyo. One approach is based on communication; that is, the collection of services that a group, or multicast, communication can reach. However, it ignores some human issues such as territory or use conventions. It is unlikely, for example, that the devices connected to a particular subnet are in a meaningful place such as a room. Using physically constrained media such as infrared, ultrasound, or 60-GHz radio (which walls substantially attenuate)

helps solve territorial issues, but it does not solve other cultural issues. Another alternative is to rely on human supervision to determine the scope of each instance of the discovery service [56].

- Interaction.

After associating to a service, a component employs a programming interface to use it. Components must conform to a common interoperation model to interact. The problem is: how the common interoperation model may be obtained? For instance, Jini lets service-access code and data, in the form of a Java object, migrate to a device. However, how can the software on the device use the downloaded object without a priori knowledge of its methods?

Event systems [44, 57] or tuple spaces [44, 58, 59, 60] offer two different approaches. In each case, components don't need direct knowledge of one another. A component registers a service to a service server via self-describing data items, either events or tuples. The service server matches the data to specifications that other components requiring the service have registered, and then forwards it to all components with a matching specification. The interactions are data-oriented, and only requires a few fixed operations.

In each case, components interact by sharing a common service, a tuple space or an event service. They don't need direct knowledge of one another. However, these data-oriented systems have shifted the burden of making interaction consistent onto data items, events and tuples. If one component publishes events or adds tuples of which no other component has a priori knowledge, interaction may fail. Thus, data-oriented

interaction is a promising model, but requires ubiquitous data standardization for it to work across environment boundaries.

Adaptation

As mentioned earlier, an ubicomp system is prone to be unpredictable, and, hence, dynamic. Thus, a new challenge for ubiquitous systems arises to deal with limited and dynamically varying computational resources, and adaptation must often take place without human intervention to achieve what Weiser calls *calm computing* [61]. Several possible extensions of existing mobile computing techniques, transformation and adaptation for content and the human interface, will be examined in the following.

- **Content**. Mobile computing research has successfully addressed content adaptation for resource-poor devices at both the OS and application levels. Pioneering work such as the Coda file system explored network disconnection and content adaptation within the context of a specific application, i.e., the file system, handling all such adaptation in the operating system to enable application transparency. Coda's successor, Odyssey [62], and other, later work on adapting Web content for slow networks and small devices [63, 64] moved some responsibility to the application layer, since the OS is rarely familiar with the semantics of the application's data and it is not always in the best position to make adaptation decisions; this reflected the thinking of application-level framing [65].

Adaptation in ubicomp is quantitatively harder. Instead of 1-to-n adapting, i.e., a small fixed set of content types to a variety of device types, there must be potentially handle n-to-n adapting. Several general approaches mobile computing research

has explored have immediate applicability. For example, the Stanford Interactive Workspaces' smart clipboard can copy and paste data between incompatible applications on different platforms [66]. It builds on well-known mobile computing content adaptation, the main difference being that the smart clipboard must transparently invoke the machinery whenever the user performs a copy-and-paste operation. A more sophisticated but less general approach, *semantic snarfing*, as implemented in Carnegie Mellon's Pebbles project, captures content from a large display onto a small display and attempts to emulate the content's underlying behaviors [67]. Rather than relying on a generic transformation framework, it requires special-case code for each content type (such as menus and Web page content) as well as special-case client-server code for each pair of desired devices (Windows to Palm, X/Motif to Windows CE). Nonetheless, both approaches demonstrate the applicability of content transformation to the ubicomp environment.

- **The human interface.** For a user entering an environment and immediately accessing or controlling various aspects of the environment through her PDA, it may require separating user interfaces (UIs) from their applications. In ubicomp, however, the goal might be to move the human interface to a physically different device chosen on the fly.

One approach, such as VNC [68], lets a client display the bits and collects user input, without knowledge of UI widget semantics. On the other hand, X Windows and its low-bandwidth derivatives, such as LBX (Low-Bandwidth X), use declarative description of the interface elements and a client has the intelligence to render them itself.

A noteworthy variant involves delivering an HTML UI (based on forms and HTML widgets) via HTTP to any device that can host a Web browser, thus removing the need to reconfigure the device for different interfaces. However, HTML provides only limited expressiveness for a UI. Also, HTML and HTTP address delivering, rendering, and letting the user interact with the UI, but they do not address how to determine what belongs in it or how UI elements associate with the applications or devices they control.

Todd Hodes and his colleagues [69] proposed a Tcl/Tk framework in which controllable entities export Tcl-code descriptions, from which a client can generate its UI. The descriptions are declarative and high level, letting the geometry and the generated interface's visual details vary among platforms. The Stanford Interactive Workspaces project has successfully generalized Hodes's approach: The Interface Crafter framework [70] supports a level of indirection of specialized per-device or per-service interface generators and a generic interface transformation facility. In this system, the high-level service descriptions permit adaptation for non-traditional UI modalities such as voice control.

Finally, a fully generalized mobile code facility, such as Jini, lets the client download and execute arbitrary code or pseudocode that implements the interface and communication with the service. Because the protocol between a Jini service and its UI is opaque, i.e., embedded in the code or pseudocode, it is usually impossible to enlist a different user agent to interpret and render the UI. It is also impossible to realize the UI on a device not running a Java Virtual Machine or on one with non-traditional I/O characteristics.

Integration with the Physical World

To integrate computing environments with the physical world, there are needs for low-level application programming interfaces that let software deal with physical sensors and actuators, and a high-level software framework that lets applications sense and interact with their environment, including physical sensors and actuators. Traditional device drivers offer APIs that are too low-level, because application designers usually think of functionality at the level of widgets in standard toolkits such as Tcl/Tk, Visual Basic, and X/Motif. A potentially more useful low-level API is a Phidget [71], a GUI widget element whose state and behaviors correspond to those of a physical sensor or actuator. At a higher level, the Context Toolkit framework [72] provides applications with a context widget software abstraction, which lets an application access different types of context information while hiding how the information was sensed or collected. For example, one type of context widget provides information on the presence of a person in a particular location, without exposing how it collected the information. The Context Toolkit's middleware contribution is its ability to expose a uniform abstraction of information, such as location tracking, hiding the details of the sensing system or systems used to collect the information.

Location sensing and tracking figure prominently in several projects. The early Active Badge work [23] and the more recent Sentient Computing effort [73] use infrared, radio, and ultrasound to track physical objects that users carry or wear. Knowing the users' locations and identities enables location tracking and novel behaviors for existing devices, such as a camera that knows who it's photographing. On the other

hand, EasyLiving [74] and the MIT Intelligent Room [75] use location tracking as one of several elements that help infer or disambiguate a user action in a context-sensitive manner.

Several groups have investigated ways of linking physical entities directly to services using identification technologies. For example, Roy Want and his colleagues at Xerox Parc [76] augmented books and documents by attaching radio frequency identification tags to them and presenting the electronic version to users who scanned them with handheld devices. Jun Rekimoto and Yuji Ayatsuka [77] attached symbols specially designed for digital camera capture to physical objects, to link them to electronic services. Hewlett-Packard Labs' Cooltown project focuses on associating Web resources with physical entities. Although the Cooltown infrastructure has primarily targeted direct human interaction, work is in progress on interoperation between Web presences using XML over HTTP to export query and update operations [78].

Robustness and Routine Failures

Ubiquitous systems may see a radical increase of "failure" frequency compared to a wired distributed system. Some of these failures are not literal failures but unpredictable events from which it is similarly complicated to recover. For example, batteries have a relatively short mean time to failure due to battery exhaustion, and wireless networks have limited range and are prone to interference from nearby structures. Furthermore, service associations are sometimes gained and lost unpredictably, since a device may suddenly leave an environment.

Although the distributed systems literature contains techniques for fault-tolerant computing, they are often based on resource redundancy rather than scarcity. Furthermore, the common assumption in distributed systems that failures are relatively rare could lead to expensive or ungraceful recovery, contrary to the expectation of calm behavior for ubiquitous systems.

In ubicomp systems, various transient failures can actually characterize steady-state operation. The Internet protocol community and the distributed systems community have considered such scenarios. There exists several techniques from both specifically designed to function under the assumption that "failure is a common case."

Consider the PointRight system [79], in which a single mouse and keyboard can control a collection of otherwise-independent displays in a ubicomp environment. When a user enters or leaves a ubicomp environment that has PointRight, the user's device could register with a centralized directory that tracks the room's screen geometry. But this directory is a single point of failure. If it fails and has not saved its data to stable storage, it must notify all devices to reregister when the directory comes back up. Furthermore, if a device dies or fails to deregister before leaving the environment, it creates inconsistency between the directory contents and the environment's true state.

One solution requires each available device or service to send a periodic advertisement announcing its presence or availability to the directory service, which collects these advertisements and expires them when a new advertisement arrives or after a designated timeout period. Should the directory service fail, new advertisements will repopulate it when it restarts; hence, the directory itself constitutes soft state. Should

a device fail, it will stop sending advertisements, and once the device's previous advertisement has expired, the directory will no longer list the device. A detailed analysis of how to choose a timeout to balance the resulting "window of potential inconsistency" with the network and computation capacity consumed by advertisements is available elsewhere [80].

In addition to maintain consistency between the directory contents and the environment's state, For robustness, a process must always be prepared to rediscover and reacquire lost resources. In a dynamically changing environment, some operations can fail while others must necessarily succeed. The one.world middleware separates application code into operations that may fail, and logic that does not fail except under catastrophic circumstances [57]. Generally, an operation is anything that might require allocating or accessing a potentially nonlocal resource whose availability is dynamic, such as file or network I/O. When opening a file on a remote server, the application creates a binding between the local file handle and the remote file. If the application thread is migrated to another host, the file might still be available on the server, but it will be necessary to establish a new binding on the new host. The application must provide code that deals with such conditions when doing operations that can fail. One.world automatically provides for some common cases, such as retrying idempotent operations a finite number of times.

Security

Security has always been an important issue in network computing, and it may also apply to ubiquitous computing. Certain services in a smart room, for example, need

protection from devices belonging to visitors. Similarly, data and devices brought into an environment, such as users' PDAs, require protection from hostile devices nearby. Viewed from a higher level, users require security for their resources and, in many cases, privacy for themselves [81]. Mobile computing has led to an understanding of vulnerabilities such as the openness of wireless networks [82]. But physical integration and spontaneous interoperation raise new challenges, requiring new models of trust and authentication as well as new technologies.

In ubiquitous systems, trust is an issue because of spontaneous interoperation. What basis of trust can exist between components that can associate spontaneously? Components might belong to disparate individuals or organizations and have no relevant a priori knowledge of one another or a trusted third party. Fortunately, with help of physical integration, humans can make judgments about their environments' trustworthiness [83], and the physical world offers mechanisms for bootstrapping security based on that trust. For example, users might exchange cryptographic keys with their environment or each other over a physically constrained channel such as short-range infrared, once they have established trust.

Many extremely resource-poor devices do not have sufficient computing resources for asymmetric (public key) encryption, even when using elliptic curve cryptography [84], and protocols must minimize communication overheads to preserve battery life. For example, Spins (security protocols for sensor networks) provides security guarantees for the data that smart dust particles exchange in a potentially hostile environment [85]. They use only symmetric-key cryptography, which, unlike asymmetric-key cryptography, works on very low-power devices. But this does not address what has

been called the "sleep deprivation torture attack" on battery-powered nodes [86]: an attacker can always deny service by jamming the wireless network with a signal that rapidly causes the devices to run down their batteries.

Another problem in ubicomp systems is basing access control on authenticating users' identities, particularly knowing an identified user's whereabouts raises privacy issues. Besides, spontaneous interoperation makes it problematic for environments, which have to integrate a stream of users and devices that can spontaneously appear and disappear.

It can be advantageous to issue time-limited keys to devices that have spontaneously appeared in an environment, rather than setting up access control lists. Keys could be issued on the basis of dynamically established trust and used without explicitly disclosing an identity. Also, they reduce the need for system-wide configuration and avoid communication with a central server [51].

Physical integration has several implications for security through location. In location authentication, the system establishes a component's physical location, for example, as a criterion for providing services. That is, the system provides services to users, but do not matter care who the users are; the system cares only about where they are. Physically constrained channels, such as ultrasound, infrared, and short-range radio transmission, particularly at highly attenuated frequencies, have enabled work on protocols that prove whether clients are where they claim to be [87, 88].

People commonly use intranet architectures to protect resources sharing. Mobile ambients [89] is a mathematical model that aims to describe the movement of both mobile software agents and mobile computing devices and emphasizes the handling of

administrative domains and the authorization to enter or exit certain domains. Security is an inherent property, since any movement must obtain permission in advance. This property reflects nowadays Internet reality very well, particular the Internet partitioned by firewalls into administrative domains. However, it does not limit resource sharing within domains. Some security problems may occur when "home" devices do not want to share resources with visitors. Work on securing discovery services [51] and access to individual services [90] tends to assume the opposite of the firewall model: that locals and visitors go through the same security interface to access resources. Weiser envisioned devices you can use temporarily as personal devices and then return for others to use and a more convenient and fine-grained model of protected sharing is required. The "Resurrecting Duckling" paper [86] an approach that the user imprints the device by transmitting a symmetric key to it over a physically secure channel, such as by direct contact. Once imprinted, the device only obeys commands from other components that prove possession of the same key until it receives instructions to return to the imprintable state.

2.3 Reflective System

2.3.1 Concept of Reflection

Abstractly, reflection refers to the capability of a system to reason about and act upon itself. More specifically, a reflective system is one that provides a representation of its own behavior which is amenable to inspection and adaptation, and is causally

connected to the underlying behavior it describes. *Causally-connected* means that changes made to the self-representation are immediately mirrored in the underlying system's actual state and behavior, and vice-versa. It can therefore be said that a reflective system is one that supports an associated causally connected self representation (CCSR). Note that this definition was first used by Pattie Maes in her 1987 thesis from Vrije Universiteit Brussel, entitled: "Computational Reflection" [91].

For example, consider a run-time interpreter for a Java-like language that is reflective. Imagine further that the interpreter's CCSR is concerned with representing the following aspects of interpretation: method dispatch, garbage collection and class loading. Given such a system, it should be possible to alter the semantics of program execution by manipulating appropriate aspects of the CCSR. For example, one could dynamically add debugging facilities to the interpreter by inserting calls to debugger routines before and after each method call, one could alter the garbage collection policy (or disable it to help support real-time capabilities, for example), or one could insert additional security checks into the class loader.

It can be seen from the above example how reflection enables both inspection and adaptation of systems at run time. Inspection allows the current state of the system to be observed, while adaptation allows the system's behavior to be altered at run time to better match the system's current operating environment. Although both inspection and adaptation are essential aspects of a reflective system, not all systems badged as reflective necessarily support both of these properties. For instance, Java's reflective facilities [92] allow the programmer to extract, at run time, the names of methods in some target class together with the types of their arguments and return

values. This makes it possible to write code that may call methods in objects whose class was not known to the programmer when the program was written. However, while Java certainly provides an accessible self-representation of certain of its features, this self-representation cannot be said to be causally connected, for the simple reason that it is not possible to change it. In short, inspection is provided but not adaptation. Reflective computation does not directly contribute to solving problems in the external domain of the system. Instead, it contributes to the internal organization of the system or to its interface to the external world. Its purpose is to guarantee the effective and smooth functioning of the object-computation. Particularly, reflective systems provide a fundamentally new paradigm for thinking about computational systems. In a reflective system, a computational system is viewed as incorporating an object part and a reflective part. The task of the object computation is to solve problems and return information about an external domain, while the task of the reflective level is to solve problems and return information about the object computation.

At first sight the concept of reflection may seem a little far-fetched. However, there is a substantial practical value to reflection. There are many research about adding reflection into various systems, such languages, operation systems, and middleware.

2.3.2 Reflective Language

Traditional programming languages today do not fully recognize the importance of reflective computation. They do not provide adequate support for its modular implementation. For example, if the programmer wants to follow temporarily the computa-

tion, e.g., during debugging, he often changes his program by adding extra statements. When finished debugging, these statements have to be removed again from the source code, often resulting in new errors.

A programming language is said to have a *reflective architecture* [93] if it recognizes reflection as a fundamental programming concept and thus provides tools for handling reflective computation explicitly. Concretely, this means that:

1. The interpreter of such a language has to give any system that is running access to data representing the system itself. Systems implemented in such a language then have the possibility to perform reflective computation by including code that prescribes how these data may be manipulated.

2. The interpreter also has to guarantee that the causal connection between these data and the aspects of the system they represent is fulfilled. Consequently. the modifications these systems make to their self-representation are reflected in their own status and computation.

Thus, in a reflective architecture one can temporarily associate reflective computation with a program such that during the interpretation of this program some tracing is performed. The programmer may conclude the run-time information, such execution errors, from the tracing data, and then dynamically debugging.

Procedure-based, logic-based and rule-based languages incorporating a reflective architecture can be identified. 3-LISP [94] and BROWN [95] are two such procedural examples (variants of LISP). They introduce the concept of a reflective function, which is just like any other function, except that it specifies computation about the

currently ongoing computation. Reflective functions should be viewed as local (temporary) functions running at the level of the interpreter: they manipulate data representing the code, the environment and the continuation of the current object-level computation.

FOL [96] and META PROLOG [97] are two examples of logic-based languages with a reflective architecture. These languages adopt the concept of a meta-theory. A meta-theory again differs from other theories (or logic programs) in that it is about the deduction of another theory, instead of about the external problem domain. Examples of predicates used in a meta-theory are "provable(Theory, Goal)", "clause(Left-hand, Right-hand)", etc.

TEIRESIAS [98] and SOAR [99] are examples of rule-based languages with a reflective architecture. They incorporate the notion of meta-rules, which are just like normal rules, except that they specify computation about the ongoing computation. The data-memory these rules operate upon contains elements such as "there-is-an-impasse-in-the-inference-process", "there-exists-a-rule-about-the-current-goal", "all-rules-mentioning-the-current-goal-have-been-fired", etc.

Almost all of these languages, except FOL, are implemented by means of a procedural reflection, in which the data structure for self-representation is actually used to implement the system. Hence, the consistency between the self-representation and the system itself is automatically guaranteed. However, one problem with procedural reflection is that a self-representation has to serve two purposes, providing a good basis to reason about the system and used to implement the system, which means that it has to be effective and efficient. These are often contradicting requirements.

Consequently, people have been trying to develop a different type of reflective architecture in which the self-representation of the system would not be the implementation of the system. This type of architecture is said to support declarative reflection because it makes it possible to develop self-representations merely consisting of statements about the system. However, the causal connection requirement is more difficult to realize here because it has to be guaranteed that the explicit representation of the system and its implicitly obtained behavior are consistent with each other.

Such an architecture can be viewed as incorporating representations in two different formalisms of one and the same system. During computation the most appropriate representation is chosen. The implicit (procedural) representation serves the implementation of the system, while the explicit (declarative) representation serves the computation about the system. In architectures for declarative reflection more interesting self-representations can be developed, and GOLUX [100] and Partial Programs [101] are two attempts to realize such architectures.

Actually the distinction between declarative reflection and procedural reflection should more be viewed as a continuum. A language like FOL [96] is situated somewhere in the middle: FOL guarantees the accuracy of the self-representation by a technique called *semantic attachment.* The force of the self-representation is guaranteed by reflection principles. It is far less trivial to prove that the combination of these two techniques actually also succeeds in maintaining the consistency between the self-representation and the system.

Although the first OOLs, such us SIMULA [102] or SMALLTALK-72 [103], did not yet incorporate facilities for reflective computation, it must be said that the concept

of reflection tits most naturally in the spirit of object-oriented programming. An important issue in OOL is abstraction: an object is free to realize its role in the overall system in whatever way it wants to. Thus, it is natural to think that an object not only performs computation about its domain, but also about how it can realize this (object-) computation.

Several OOLs are implemented with reflective facilities being mixed in the object-level structures. In languages such as SMALLTALK-72 [103] and FLAVORS [104], an object not only contains information about the entity that is represented by the object, but also about the representation itself, i.e., about the object and its behavior. There are two problems with this way of providing reflective facilities. One is that these languages always support only a fixed set of reflective facilities. Adding a new facility means changing the interpreter itself. A second problem is that they mix object-level and reflective level, which may possibly lead to obscurities. For example, if we represent the concept of a book by means of an object, it may no longer be clear whether the slot with name "Author" represents the author of the book, i.e., domain data, or the author of the object, i.e., reflective data.

One step towards a cleaner handling of reflective facilities was set by the introduction of *meta-classes* by SMALLTALK-80 [105]. In SMALLTALK-72, classes are not yet objects. The internal structure and message-passing behavior of an object can be specified in its class, but the structure and behavior of a class cannot be specified. The idea behind this development in SMALLTALK-80, which was later also adopted in LOOPS [106], is that it should also be possible to specify the internal structure and computation of a class. Meta-classes serve this purpose.

Another step towards the origin of reflective architectures was taken by the development of OOLs such as PLASMA [107], ACTORS [108], RLL [109], and OBJVLISP [110]. These languages try to bring more uniformity in object-oriented programming by representing everything in terms of objects. They all contribute to the uniformity of the different notions existing in OOLs by representing everything in terms of objects: class, instance, meta-class, instance-variable, method, message, environment and continuation of a message. This increased uniformity makes it possible to treat more aspects of object-oriented systems as data for reflective computation.

Pattie Maes described an object-oriented reflective language called 3-KRS [93] based on KRS [111]. 3-KRS partitions every object into a domain specific referent and a reflective metaobject. The referent contains information describing the real-world entity that the object represents. The metaobject contains information describing its referent as a computational entity in itself. Enforcing this separation encourages the emergence of standard protocols for communicating between objects and metaobjects. The distribution of the system's self-representation among the system's metaobjects makes this self-representation easier to think about and manipulate.

The Common Lisp Object System (CLOS) [112, 113, 114] provides a very powerful generic function-based method combination mechanism. The CLOS Metaobject Protocol (MOP) provides a metacircular definition of the entire CLOS system, and is intended to permit open-ended experimentation with new object-oriented programming paradigms. The MOP has been used to successfully implement a number of distinctly un-CLOS-like object-oriented programming systems, and is designed to

permit the modification of basic language mechanisms such as method lookup from with CLOS itself.

In general, it can be said that the evolution of OOLs tends towards a broader use of reflective facilities. In the beginning reflective facilities were only used in minor ways. A class would for example only represent the reflective information telling what its instances were. However, as OOLs evolved, the self-representations became richer and applied in a broader way (from instances only, to classes, to meta-classes, to messages, etc).

Applying a reflective computing technology is not limited to programming languages. Silica [115] is an example applied to a CLOS-based window system. Apertos [116] is the first example which applies the notion of reflective computing to a distributed object-oriented operating system. Many other reflection systems could be found in [117].

2.3.3 Reflective Operating Systems

An operating system is inherently reflective if it has facilities to inspect system data structure maintained by a kernel and to inspect the internals of a process, such as **/dev/kmem** in UNIX [116]. However, these have a limitation in their use; that is, since they are mechanisms provided by an operating system kernel, it is difficult to alter their behavior.

Also, in virtual memory management, we can specify its management policy. UNIX provides **madvise()** for it. Mach [118] and Chorus [119] introduce user-level virtual

memory management as an external pager. Using an external pager approach allows people to write a code for an application specific policy for memory management. Further, in real-time operating system, people may describe a real-time scheduling policy which determines the process and thread next to be executed. ARTS [120] is an example of such a system providing policy and mechanism separation, where people may write a policy module which uses mechanisms given by its kernel to choose the thread to be run.

In contrast to these approaches, the framework proposed in Apertos [116], a reflective object-oriented operating system for an open and mobile computing environment, has significant advantages, in which the system is constructed based on object and metaobject separation, metahierarchy, and object migration. Object and metaobject separation and object migration help users and programmers to accommodate object heterogeneity, and, hence, easily provide the mechanisms such as changing a communication paradigm, inspecting the internals of an object for a debugger, and changing resource management policy. Metahierarchy provides discipline for programming in object and metaobject separation. Object migration is a basic mechanism of the operating system in order to accommodate object heterogeneity. These hide the underlying implementation from an object and increases mobility of object, so that these contribute to the realization of an open and mobile computing environment.

Another object-oriented reflective operating system is Merlin [121] based on the Self programming language [122]. It uses a reflective structure that allows the programmer to move cleanly between different abstraction levels to get around the problem existing in many pure object oriented programming languages (like Self) that present a very

abstract view of the hardware to the programmer, and so are unlikely candidates for system software development. However, this introduces other difficulties in the form of circular definitions. It also provides *Adaptive Compilation* [123], a system of optimizing code based on information gathered while running the code itself, to match the parallelism of the application to the characteristics of the underlying hardware. TUNES [124] is an ambitious project to a machine with some intelligence by using reflective architecture. Thus, many features of TUNES, such as, higher-order functions, self-extensible syntax, fault-tolerant computation, dynamic code regeneration, etc., can be dynamically migrated into the system. The goal is to allow users or programmers to specify what they care about and what they don't care about, and people simply trust the computer to handle it all by itself.

2.3.4 Reflective Middleware

Middleware refers to a distributed platform of interfaces and services that reside between the application and the operating system and aim to facilitate the development, deployment and management of distributed applications. OMG's CORBA [125], Microsoft's DCOM [126], or Java RMI [127] are the examples of object-based platforms. The main function of these platforms is to mask the inherent heterogeneity of distributed systems and provide a standard set of interfaces and services which distributed applications can assume present in any participating language, operating system or machine environment.

Why should middleware be reflective? What potential benefits are there in providing platforms such as CORBA with a CCSR? The most general answer to this question is that reflection does for middleware what it does for any system: it makes it more adaptable to its environment and better able to cope with change. In a middleware context, the need to cope with change particularly arises when middleware based applications are deployed in hostile or dynamic environments such as multimedia, group communication, real-time and embedded environments, handheld devices and mobile computing environments.

Existing middleware specifications such as CORBA and DCOM simplifies the construction of component-based distributed applications. Nonetheless, they do not define strategies for customizing themselves flexibly to dynamic environment. Recent research in reflective middleware tries to overcome this limitation. For example, a reflective architecture from the Distributed Multimedia Research Group at the Lancaster University [128] uses the Python interpreted language to inspect and change the implementation at run-time. COMERA [129] provides a framework based on Microsoft COM that allows users to modify several aspects of communication middleware at run-time. DynamicTAO [130] supports on-the-fly reconfiguration on CORBA ORB based on The ACE ORB(TAO) [131] that is the open source CORBA-compliant ORB.

Besides, it has been widely recognized that many QoS-constrained distributed applications need to be adaptive in heterogeneous environments. For example, Li et al. [132] propose a hierarchical adaptive QoS architecture for multimedia applications. A multimedia service is delivered by multiple service configurations, each of which

involves a different set of service components. Each service component is executed as a process. Components cooperate through protocols over network communication.

2.4 Other Related Work

2.4.1 Architectural Description Languages

Software architecture research is directed at reducing costs of developing applications and increasing the potential for commonality between different members of a closely related product family [133, 134]. Software development based on common architectural idioms has its focus shifted from lines-of-code to coarser-grained architectural elements (software components and connectors) and their overall interconnection structure. To support architecture-based development, formal modeling notations and analysis and development tools that operate on architectural specifications are needed. Architecture description languages (ADLs) and their accompanying toolsets have been proposed as the answer. Loosely defined, "an ADL for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module" [135]. ADLs have become an area of intense research in the software architecture community [136, 137, 138, 139]. A number of ADLs have been proposed for modeling architectures, both within a particular domain and as general-purpose architecture modeling languages. The languages most commonly referred to as ADLs:

Aesop [140] provides a generic toolkit and communication infrastructure that users can customize with architectural style descriptions and a set of tools that they would like to use for architectural analysis.

C2 [141, 142] is a general component and message based architectural style that is well-suited for large-scale, heterogeneous, and distributed applications.

Darwin [143, 144] is to specify system architectures in terms of components and their interconnections.

MetaH [145] specifies how software modules developed in a variety of styles are composed together with hardware objects to form a complete system architecture.

Rapide [146, 147] focuses on developing a new technology for building large-scale, distributed multi-language systems.

SADL [148, 149] formalizes architectures in terms of theories, shows how generic refinement operations can be proved correct, and describes a number of flexible refinement patterns.

UniCon [150, 151] is an ADL whose focus is on supporting the variety of architectural parts and styles found in the real world and on constructing systems from their architecture descriptions.

Weaves [152] are networks of concurrently executing tool fragments that communicate by passing objects.

Wright [153] allows architects to specify temporal communication protocols and check properties such as deadlock freedom.

ACME [154] is a simple, generic software ADL that can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools.

xADL [155, 156] is an XML-based ADL designed to support the description of architectures as explicit collections of components and connectors. xADL 2.0 supports aspects of structure, grouping, and configuration management through an set of XML Schemas which can be applied in whole or in part to architectures described by it.

Medvidovic and Taylor [157] propose a classification and comparison framework for software architecture description languages, and they also gave an excellent overview and comparison of ADLs prior to 2000.

There are also some meta languages that are used to specify the high level characteristics of system. One example is an XML-based Hierarchical QoS Markup Language, called HQML [158]. HQML is introduced to enhance distributed multimedia applications on World Wide Web with QoS capability. Another example is the megaprogramming language [159] that provide the glue for joining together computations spanning several modules.

2.4.2 Real-Time Java

Adding real-time capability to Java brings many challenge, including the analysis of real-time components, deadline-based scheduling, asynchronous event handling, stack allocation and garbage collection [160, 161, 162]. There have been several group involved in real-time extension of Java. The National Institute of Standards and Technology (NIST) sponsored a series of workshops beginning in June 1998. The goal was to develop requirements for supporting real-time programming on Java platform. The final workshop report [163] defines nine core requirements for a Real-Time Java specification, as well as a number of derived requirements.

The Real-Time for Java Experts Group (RTEG) under Sun's community process is working on the real-time specification for Java. The draft has been made available for public view [164]. Another group, the Real-Time Java Working Group (RTJWG) also has announced the public review period for its Real-Time Core Extensions Draft Specification document ("RT Core") [165]. Both specifications use the NIST requirements as guiding principles, but are different in some aspects. Sun's specification tends to preserve compatibility with existing Java run-time semantics. They intended to not support portability of real-time Java applications because they want difference between underlying real-time operating systems could be reflected at Java level [166]. Since it is tightly coupled with underlying operating system, the VM has to be re-implemented. On the other hand, RTJWG specification can be separated into a baseline Java VM which could be a generic off-the-shelf JVM, and a real-time

core execution engine which is portable and dynamically loadable. More details about comparison on these two specifications can be found at [166].

Java distributed system mechanisms and platforms - RMI, JavaSpaces, Jini - suffer from additional disabilities for most of the real-time domain. An expert group for distributed real-time specification was formed to address the need for many real-time applications (for example, in the industrial automation, defense, and telecommunication domains) which involve a multiplicity of computing nodes, must satisfy trans-node end-to-end timeliness requirements. More information can be found at [167].

2.4.3 Computation Steering

Interactive computational steering permits users to dynamically adjust parameters of an executing computation or system. Such steering permits users to achieve higher productivity in the execution of long-running simulation programs by providing more immediate feedback, or to direct performance optimizations in systems with temporally varying demands on and availability of resources. Steering techniques allow users to interact with and adjust the input parameters of algorithms for which the execution behavior varies randomly, varies dynamically with characteristics of the input data, or for which the relationship between the input data and execution behavior are not yet well understood.

Generally, visualizations are used to provide the user with insight into the state and behavior of the underlying system, and as a feedback mechanism, enabling users to gauge the effectiveness of these parameter adjustments. Tools and techniques for in-

teractive steering are being applied to simulation applications and environments, for network configuration and management, load balancing and scheduling, modeling and design, the interactive refinement of scientific visualizations, and performance optimization of WWW servers and parallel I/O systems. The goal of interactive steering is to enable researchers to monitor and guide their applications during runtime, to help users more effectively address the target problems of their computations, and to improve the performance of these applications. The ability to observe and adjust large scale simulation and modeling codes in an online fashion allows researcher to more easily detect the cause-and-effect relationships at work, and promotes the development of greater intuition regarding the effects of program parameters, the presence of bugs, the behavior of algorithms, and characteristics of the target problem.

In addition to designing application-specific steering solutions, researchers are designing general tools to facilitate the steering process. Pablo [168, 169] is an event based performance analysis environment that offers performance data capture, analysis, and presentation. Two newer interoperable toolkits, Autopilot and Virtue, are developed on top of the Pablo [170]. Autopilot emphasizes on performance steering, close-loop adaptive control and decoupling the steering infrastructure from the policy domain. Virtue focuses on performance data immersion, real-time display, and direct manipulation for grid control.

The Falcon system [171] demonstrates the potential performance improvements possible through online monitoring and steering of parallel programs. Later work by the same group has produced Progress(PROGram and REsource Steering System) [172, 173], which supports the addition of steering functionality to multi-threaded C pro-

grams executing on multiprocessors, through the use of a steering toolkit that provides sensors, probes, and actuators. The CUMULVS system [174] developed at Oak Ridge National Labs, supports users in integrating steering into existing PVM programs, and handles the details of collecting and sending distributed data fields to and receiving steering parameters from dynamically attached viewers.

Computational steering work at Utah, SciRun [175, 176], is a scientific problem-solving environment (PSE) which provides the ability to interactively guide or steer a running computation. It addresses the construction of steerable simulation programs via a data-flow programming model, and the addition of steering capabilities to large-scale simulations through the use of scripting languages. The entire process of computation modeling, simulation and visualization is built and executed within the PSE. SciRun was designed initially for multi-threaded shared-memory multiprocessors using C++ classes. A distributed-memory version is being produced and threading is now used to hide latency and perform other tasks. SciRun aims to address the problems of interaction and integration of scientific simulation and visualization in a distributed computing environment.

Other researchers are working to apply interactive steering in their particular application domains. Parallel performance instrumentation and evaluation concerns are being addressed by Waheed et al. [177] and Reed et al. [178]. An interactive simulation of material growth has been developed in the UK [179], a tool for the real-time visualization of computational fluid dynamics developed at NEC Research [180], interactive flow simulation tools developed in Germany [181] and at the Army High

Performance Computing Research Center [182], and interactive simulation tools for finite element analysis developed at Boeing [183].

Specialized interactive simulation environments have been developed for automobiles [184], for distributed modeling applications [185], for electronic manufacturing systems [186], and for large multimedia networks [187]. Finally, the publications regarding the presentation of information for interactive exploration include [188] and [189].

2.4.4 Robot Control Systems

Early robotic researchers attempted to create robot control systems that reasoned about and planned every action. The general approach was to sense the world, build a world model, plan actions with respect to goals, and then execute the plans via motor control systems. The first major development was Shakey from SRI [190]. These systems were often brittle control systems incapable of operating outside of controlled environments. Because they required an accurate world model to reason properly about what to do, environmental or sensory noise often made them unreliable. When they worked properly, they usually only operated under controlled and limited situations.

Brooks later created several robots that were designed to operating in cluttered, unfriendly environments using the "subsumption architecture" [191]. This architecture allows lower, more critical levels of operation to subsume control from higher levels. Though these special-purpose robots interacted with this relatively hostile environ-

ment brilliantly, their ability to reason about the world was intentionally neglected thus limiting the robots to reactive control schemes. Furthermore, since the controllers for many of these robots were designed in hardware, self-adaptation was not feasible.

Pham et al. have created two systems [192] that facilitate the creation of self-adaptive control software: PB3A and RAVE. PB3A, the Port-Based Adaptable Agent Architecture, is a Java-based programming framework that aims to facilitate the development and deployment of self-adaptive, distributed, multi-agent applications. RAVE, the Real And Virtual Environment, is a mixed-reality simulation environment for mobile robots. Together, these two systems allow for the creation, testing, and analysis of self-adaptive control software by on- and off-line simulation. PB3A decomposes a complex systems into a hierarchy of fundamental unit of execution called Port-Based Module, PBM. Two PBMs are linked by mapping their input and output ports. PBM may be dynamically loaded on demand across a network, and the control software can completely change its structure of the executing robot program by remapping ports or instantiating new PBMs. However, the down-time and disruption is unavoidable for PB3A to integrate newly developed modules into running applications.

For multi-robot systems, adaptive control systems become more important because the possibility of system failure is higher than single robot system. Furthermore, the coordination communication between robots may be expensive and unstable. Yamada et al. propose an adaptive action selection without explicit communication for multi-robot box-pushing, which changes an available behavior set depending on a situation [193], i.e., the existence of other robots and the task difficulty. Also using the

behavior-based approach, a set of behaviors or actions is designed for each situation. Each robot may determine its own situation by itself without explicit communication with other robots.

Software engineers have pursued many techniques, such as specification languages and object-oriented analysis and design, for achieving the software's original promise – applications that retain full plasticity throughout their life-cycle and that are as easy to modify in the field as they are on the drawing board. The article [194] examines the fundamental role of software architecture in self-adaptive systems and outlines several design issues, such as open or closed-adapted, type of autonomy, frequency, and cost effectiveness.

2.5 Summary

The adaptive software architecture project (**ASAP**), which includes **FRAME**, **CSML**, and **Brew**, is briefly described in this chapter. **FRAME** is the adaptive software framework that provides APIs for applications to be "assembled" on-the-fly after consulting the resources of the computing environments. If applications fail to work appropriately because of dynamic environmental change, applications may also use the **FRAME** APIs to replace part of the codes to produce better performance or quality of execution without down-time.

CSML is a high level specification language that allows developers to specify the definition of adaptation for applications. The infrastructure for applications working under **FRAME** is automatically generated from the specifications.

Brew is a light weight on-line adaptation, which allows the inappropriate execution of applications, due to minor environmental changes, to be repaired without significant performance impact from **FRAME**.

Some related work from the research areas, such as pervasive computing, reflective systems, architectural description languages, and others, are also presented. They are related to **ASAP** from the heterogeneous computing, adaptive systems, or high level software system specification perspectives. The common goal is to relieve the burden of development, configuration, execution, and management of systems, a particular research challenge in terms of quantitative differences for heterogeneous and varying physical circumstances.

Chapter 3

Adaptive Software Framework:

FRAME

3.1 Motivation

In many mass production industries, such as automobiles and electronics, the final products are assembled from parts. It is technically impossible for vendors to develop parts or products that may work perfectly under every environment. As a consequence, vendors usually specify how well the parts may perform under certain environments. The parts may be built by various vendors and have different specifications, but they are plug-in compatible if they have the required functionality and satisfy requirements. Therefore, for certain environments, users may be able to select appropriate parts based on the specifications. If under some situations, a part fails or does not perform as well as required, it is possible to replace the part with an appropriate one. Take the automobile tire as a real-world example. The regular tires

are usually designed for normal weather condition. These tires may be prone to skid under snowing days and may not meet the safety requirements. Consequently, special snow tires may be used to achieve better safety or performance.

The similar idea may be applied to the development of the adaptive software, in which the applications are assembled from parts, or called *components* in this context. Since it is impossible for software developers to develop a component that will work perfectly for every computing environment, they may implement the component that will only work flawlessly under certain environment and specify the performance information in the implementation specification or called *component constraint* in this context. To guarantee that the application will work correctly as required, a set of appropriate components for the application has to been identified and replaced, if necessary, for every computing environment. The whole adaptation process includes detecting the change of environment and diagnosing the performance. If the performance requirement is not satisfied, the appropriate set of components has to be identified and then perform the component replacement.

However, given the trend that current and future computing environment may change frequently due to the development of the mobile devices, the adaptation process to identify and replace an appropriate set may be tedious or even impossible for users. Therefore, it will benefit the users if software may automatically engage the adaptation process.

To realize the automatic adaptation process, the information about component constraints has to be embedded into implementation of components, so the software may be able to diagnose the feasibility of each component implementation without hu-

man intervention. That is, once the desired performance requirements are specified by users, the software itself will detect environmental change, search for the feasible implementation of each component by checking whether their specification meets the requirements or not, assemble those feasible components into software, and then the software is ready to execute. Such an automatic assembly process is called *component assembly* in this context.

Because of component assembly, applications do not need to be assembled, or linked, during the development stage. Particularly, until being executed, an application does not need to exist as a concrete entity but virtually as a set of components. With the assembly of the applications being delayed until execution time, applications may have an opportunity to know their computing environments and be able to select the appropriate implementations of the components to adapt the applications to the specific computing environments. Furthermore, the performance portability illustrated in the first chapter may be realized since the constraints of each component will satisfy the users' desired performance requirements.

3.2 Overview of FRAME

FRAME is a adaptive Java software framework that is implemented in Java. The main reason for choosing Java is that Java has provided bytecode level portability and to achieve performance level portability implies that bytecode level portability has been accomplished. The goal of **FRAME** is to help people develop, deploy, and use adaptive software in a heterogeneous environment by providing the following features:

- Developers may specify constraints for applications.
- Users may specify the intended quality of service before executing the application.
- Applications may be automatically distributed to single or multiple target platforms.
- Before execution, applications may probe the available resources from the computing environment and then adapt themselves to the computing environment.
- During execution, applications may detect the run-time environmental changes and invoke the assembly process if necessary without down-time.

Basically, **FRAME** accomplishes the above features by providing a generic component assembly framework. An application should be composed of components under **FRAME**. As shown in fig. 3.1, the kernel represents component functionality and is encapsulated by the adaptation shell that consists of **FRAME** APIs and constraints and provide the adaptation capability for the application. Furthermore, an application only needs **FRAME** APIs and constraints, but not its functionality, to be adaptive. Theoretically, it is possible to create a dummy adaptive software, in which the software is adaptive, but functionality of its components is not implemented. Thus, the functionality kernel may be implemented independently with the adaptation shell. Obviously, separation of the adaptation shell and the component functionality kernel will reduce the complexity of developing adaptive software.

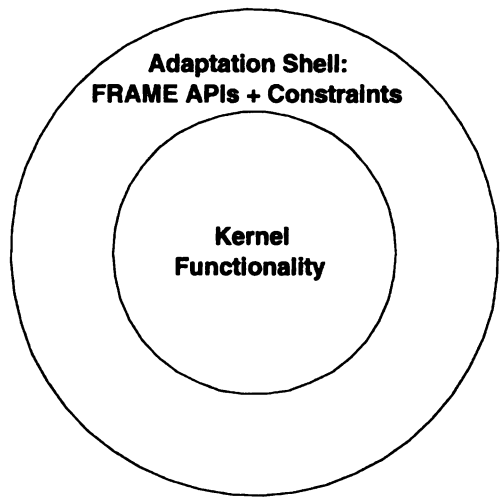

Figure 3.1: The separation of the adaptation shell and the functionality kernel for a FRAME component

As just mentioned, an application under FRAME should be in a hierarchical architecture that is built from components. Each component has information about quality of service or resource requirements specified as constraints, and cooperates with other components through an unambiguous interface. The application may be executed as a process on single machine, or distributedly on multiple machines, which require a special entity called the *component stub* to handle the communication between multiple machines.

Fig. 3.2 is a flowchart for developing an application under FRAME. First, developers need to decompose the application into a hierarchical architecture that contains components. The software hierarchy information, including the information about how components are dependent from each other and where these components may be located, should be stored in a database server called the *component registry.* For each

component, developers need to design its interface consisting of the functionality or services it will provide and the constraints it has. Once the interface is specified, various implementations with different resource requirements and quality of service may be developed and used for specific computing environments. For each component, each of its implementation will have a unique version number to be easily distinguished from other implementations. Besides, each implementation of a component needs to be stored in an server called the *component repository.* Finally, information about each implementation of a component, including version number and location of the component repository, should also be registered to the component registry.

Fig. 3.3 is a flowchart for executing an application under **FRAME**, and fig. 3.4 shows how an application execution under **FRAME** interacts with the component registry and the repository. The software is not directly executed by users but via an entity called the *launcher*, which will trigger the component assembly and the execution of the software. Users specify intended performance or quality of service as constraints in the application launcher and execute it on the target platform. The launcher will query the component registry about the software components. Based on the results from the registry, the launcher will load each implementation of components from the component repository. During the component assembly, the launcher will diagnose each implementations based on whether the constraints are satisfied under the computing environment. After the appropriate implementation of each component being identified, they will be assembled with appropriate parameters and then the application will begin to execute.

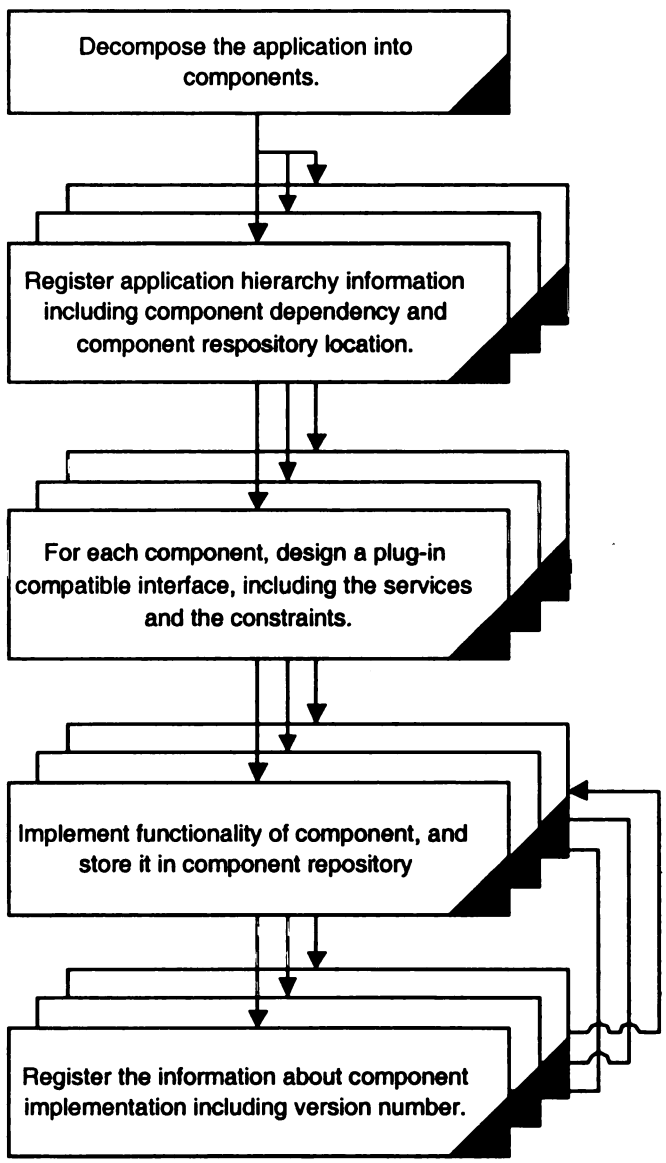

Figure 3.2: Flowchart of developing an application

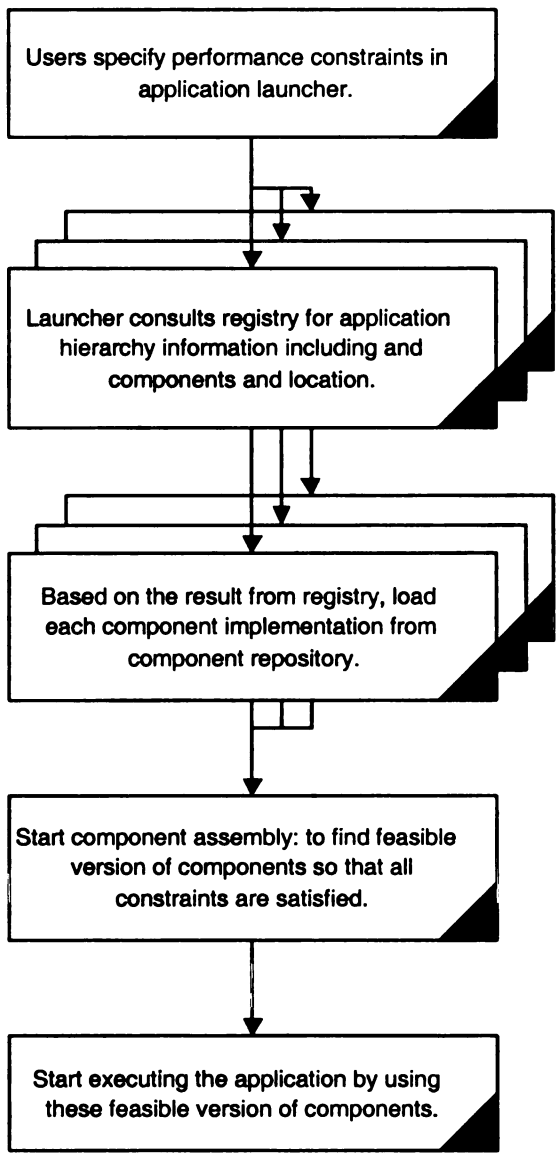

Figure 3.3: Flowchart of executing an application

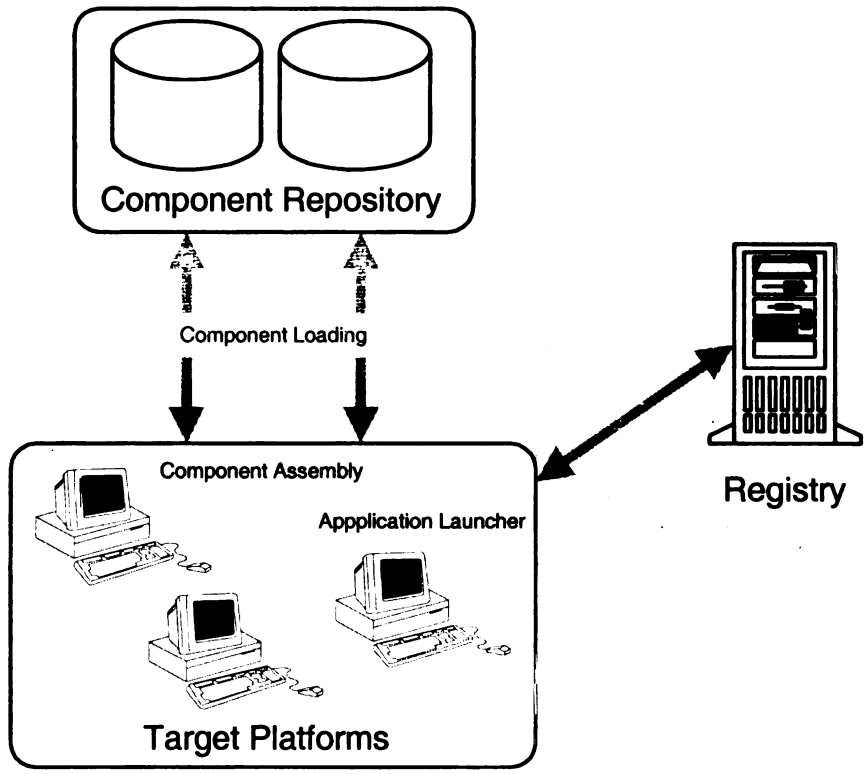

Figure 3.4: The interaction between a target platform, component registry and repository during the execution of an application

FRAME provides the APIs to accomplish the above flowcharts, including the communication between target platforms and registry, component loading from repository, component assembly, and inducing the application execution. It is deserved to be noticed that the usage of **FRAME** APIs depends on the results from the component registry and the results of constraint diagnosis. Since the component registry contains the information about the software hierarchy, it is obvious that the usage of **FRAME** APIs, and hence the adaptive capability, depends on the software hierarchy and constraints, but not the functionality of components.

Details about **FRAME** will be explained with an example, the GO game, in the following sections.

3.3 Components

As shown in fig. 3.5, the structure of a general software hierarchy may be viewed as a graph with components being represented by nodes. The dependencies of components are represented by arcs and defined via services; that is, a parent component requires services from its child components, and vice versa. Each component, except for the root component, might have more than one implementation or version. Some components may need services from their child components. Only one version of each component is needed to execute a program. The component dependency information needs to be registered to the component registry and, of course, the whole software hierarchy has to be resolved during run-time by querying the component registry.

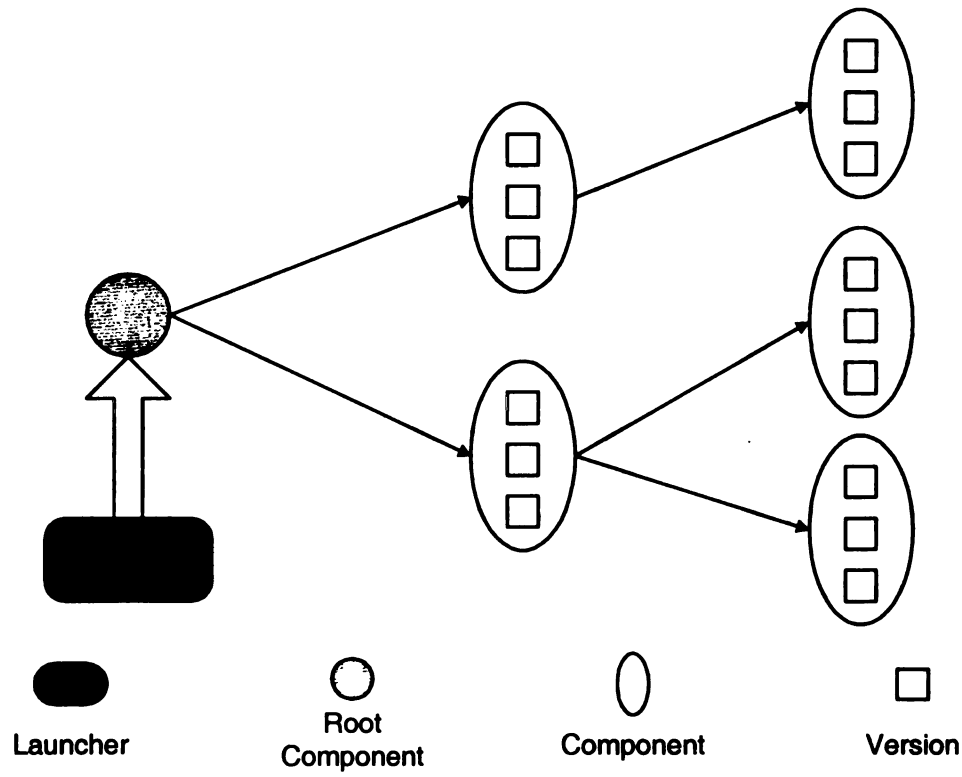

Figure 3.5: Software hierarchy under **FRAME**

Components are key entities under **FRAME**. Fig. 3.6 shows a general component architecture. A component may provide functionality or services that may be used by other components, i.e., the parent components, and it in turn may need some services from its child components. As an example, fig. 3.7 is software hierarchy of a GO game that consists of three components. Component **board** is the user interface and the root component. It needs service from its two child components: **AI** and **audio**. The component **AI** is the opponent that a user plays against and has two different implementations, **GnuGO** and **Random**. The former implementation is more competitive. Its AI engine is inherited from GnuGO source code developed in C [195]. This implementation has many recursive function invocations that require much stack memory and CPU computation. Therefore, this version is not suitable for slow, battery powered machines with limited memory. The second implementation has very limited intelligence. It is less competitive to play, but requires less computation and memory resources. The **audio** component may play background music and also has two different implementations: **MIDI** and **dummy**. **MIDI** implementation is able to play a MIDI audio file and requires that the target platform has a sound device. If the target platform cannot use **MIDI**, such as having no sound device, the **dummy** implementation will be used, which does not have any performance effect on components **board** and **AI**. To simplify the example, each implementation of each component is labelled according to table 3.1.

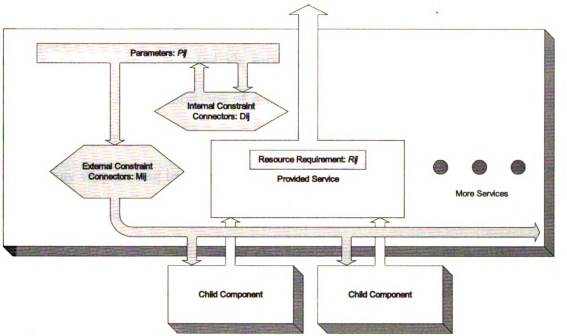

Figure 3.6: Component Architecture

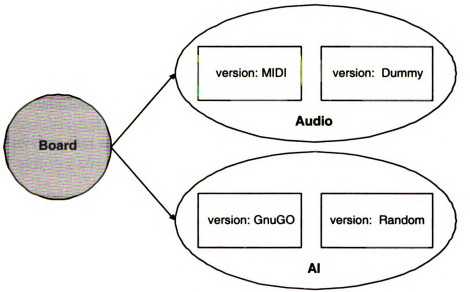

Figure 3.7: Software hierarchy of the GO game

3.3.1 Services

The functionality of a component is defined by the services it may provide, and different components are connected via services. To ensure the plug-in compatibility between different implementations of a component, each implementation of a component would provides the same services, although the performance and quality of services may vary among implementations.

In the GO game example, the **AI** component provides the service, **play**, to its parent component, **board**. Both **GnuGO** and **Random** implementations will implement the service, **play**, but with different resource requirements and, of course, different intelligence.

The services provided by each component of the GO game example is listed in table 3.2.

3.3.2 Constraints

In addition to services, a component may also have constraints as shown in fig. 3.6. Since implementations of a component may have different resource requirements and quality of services that are specified as a set of constraints, unlike services, implemen-

	component 1	component 2	component 3
version 1	**board**	**AI** with **GnuGO** implementation	**audio** with **MIDI** implementation
version 2	N/A	**AI** with **Random** implementation	**audio** with **dummy** implementation

Table 3.1: Label of each component

tations of the same component may have different set of constraints. More precisely, component assembly identify the feasible implementation of a component solely based on constraints, so each implementation should have a unique set of constraints because, otherwise, component assembly will not be able to distinguish the implementations.

The constraints may be classified into four different sets, resource requirements (R_{ij}), parameter constraints (P_{ij}), internal connectors (D_{ij}), and external connectors (M_{ij}). Each constraint is a predicate and defined as a boolean function in Java, in which the truth value represents whether a constraint is satisfied or not. If all constraints of a set, e.g. R_{ij}, P_{ij}, D_{ij}, and M_{ij}, are true, the set is defined as true; otherwise, the set is false. By checking the truth value of constraints, component assembly will be able to identify the feasibility of a component implementation.

Resource Requirements: R_{ij}

To guarantee the promised quality of services, some minimum resources may be required by the component implementation. For example, in the GO game, the component implementation, `MIDI`, requires at least a sound device to play a MIDI audio file. Such minimum required resources are specified as *resource requirement constraints.*

component	`board`	`AI`		`audio`	
implementation		`GnuGO`	`Random`	`MIDI`	`dummy`
service	`main`	`play`		`playAudio`	

Table 3.2: List of the services of the GO game component

The collection of the resources required for a component i with version number j are denoted as R_{ij}. For example, the required resource for the **MIDI** implementation of the component **audio** is a sound device. Also, it is not appropriate to use the **GnuGO** version for a battery powered platform because of power consumption. Therefore, by following the labels used in table 3.1, the overall resource requirement for the **MIDI** implementation would be R_{31} = {"A sound device exists.", "Platform is AC powered."}. Since the **dummy** should be used whenever R_{31} is false, R_{32} is actually the complement of R_{31}, denoted as R_{31}^c. Instead of specifying R_{32}, the **dummy** implementation is specified as the "default" version of the **audio**; that is, if the required resources of other versions are not satisfied, the default version will be used.

Parameter Constraints: P_{ij}

Each component may have *parameters* that represent some quantifiable or enumerable metrics, such as performance or output quality of implementation. Parameter constraints are used to specify the finite domains of these parameters, in which the performance is reasonable or the output quality is desired. For example, consider the parameter specified for performance. If there exists such a value within the specified finite domain that equals to the performance, the performance is said to be reasonable.

If a parameter is the kth parameter of component i, it is denoted as p_i^k. Table 3.3, 3.4, and 3.5 are the complete list for the parameters of each component of the GO game. These parameters should be in some default finite domain that may be specified by parent components or the application launcher. The collection of

default parameter within a domain for a component i with version j is called the *parameter constraints* and denoted as P_{ij}. For example, **board** component developers may be interested in the response time of the first move, i.e., the elapsed time from player move to computer move, power consumption of first move, and the competition level. Thus, these three metrics could be represented by three parameters p_1^1, p_1^2, p_1^3 respectively. The maximum response time might be set at 10 seconds. The power consumption is measured in percentage of a fully charged battery, and the maximum power consumption of the first move might be 0.5% since a typical GO game may involves more than 100 moves. The competition level is divided into two different level. These metrics determine the default reasonable domain of p_1^1, p_1^2, and p_1^3. Therefore, the collection of parameter default domains for the component **board** is $P_{11} = \{\text{“}0 \leq p_1^1 \leq 10\text{”}, \text{“}0 \leq p_1^2 \leq 0.5\text{”}, \text{“}1 \leq p_1^3 \leq 2\text{”}\}$.

parameter	comment
p_1^1	the response time of the first move
p_1^2	power consumption of first move
p_1^3	intelligence

Table 3.3: Complete parameter list for component **board**

parameter	comment
p_2^1	the response time of the first move
p_2^2	power consumption of first move
p_2^3	intelligence

Table 3.4: Complete parameter list for component **AI**

Internal Connectors: D_{ij}

As the software is decomposed into a hierarchy, the above constraints are also under the hierarchy and raise the problem that the specified constraints may not be orthogonal. For instance, performance usually declines when achieving better output quality. Their relationship must also be specified as a special constraint, say $C(p, q)$ with p and q being denoted as performance and quality. Without these constraint, we may specify the highest reasonable performance and intended quality within their domains. The parameter constraints are satisfied individually, but, in reality, it is impossible for the implementation to produce the highest performance and output quality at the same time and the assembly process may select a false implementation. The truth of these special constraints are examined with parameters specified in their domains. For example, if there exist such values p and q within their respective specified domains that $C(p, q)$ is true, $C(p, q)$ is satisfied in conjunction with parameter constraints. Thus, similar to that the components are connected via services, the dependency of constraints are connected by special constraints called internal constraint connectors and external constraint connectors.

For the non-orthogonal parameters within a component, the relations between these parameters, by analyzing or modeling, are specified as *internal constraint connectors*, denoted as D_{ij}. Take the component `board` as an example, it is obvious that a longer

parameter	comment
p_3^1	power consumption of first move

Table 3.5: Complete parameter list for component `audio`

response time, p_1^1, will consume more power, p_1^2. Their proportional relation may be modeled as a linear relation, $p_1^1 \geq 20p_1^2$. Therefore, the collection of the internal constraint connectors for the component **board** is $D_{11} = \{\text{“}p_1^1 \geq 20p_1^2\text{”}\}$.

External Connectors: M_{ij}

Moreover, today's complex software systems are usually collaborative projects. Each component may be designed by different developers, and the constraints that individual developers envision may sharply differ. For example, the developers of the different components may be interested in different performance metrics. While developers at higher level software are concerned about overall performance, developers at lower level software may be only focused on the performance of sub-tasks, and these parameters may be dependent or independent. Also, a parent component may need to redefine the parameter domain of its child components, and the application launcher may redefine the parameter domain of the root component. These relations among the parameters cross the components are specified by *external constraint connectors.* For example, although the **board** component and **AI** component developers may be interested in the response time, power consumption of the first move, and competition level, all these parameters may have different meaning from different perspective. Since component **audio** may also consume resources, i.e., CPU and power, the response time of component **board**, p_1^1, should take component **audio** into consideration and may be longer than the response time of component **AI**, p_2^1. Therefore, there must be some connections between the parameters of the component i with version j and its child components, and the collection of all these connections are specified as ex-

ternal constraint connectors, M_{ij}. For component **board** and its child component, the relation between the response time parameters may be modeled as a linear relation, $p_1^1 \leq 1.1p_2^1$. Similarly, the total power consumption p_1^2 should include the power consumption from **AI**, p_2^2, and **audio**, p_3^1. The relation between these power consumption parameters may be modeled as a linear relation, $p_1^2 \leq 1.1(p_2^2 + p_3^1)$. Finally, the same metrics may be used for competition level of the component **board** and **AI**, and the relation would simply be $p_1^3 = p_2^3$. Thus, external constraint connectors between the parameters of the component **board** and its child component, the **AI**, is $M_{11} = \{\text{“}p_1^1 \leq 1.1p_2^1\text{”}, \text{“}p_1^2 \leq 1.1(p_2^2 + p_3^1)\text{”}, \text{“}p_1^3 = p_2^3\text{”}\}$.

Each element in R_{ij}, P_{ij}, D_{ij}, and M_{ij} is called a *constraint.* The component constraint $\mathbb{C}_{ij}$ for component i with version j is defined as the set of all constraints, or

$$\mathbb{C}_{ij} = R_{ij} \cup P_{ij} \cup D_{ij} \cup M_{ij}$$

and the feasible component i is defined as the component i with version j such that all constraints of $\mathbb{C}_{ij}$ are satisfied. Finally, it is deserved to notice that each implementation of a component should have a unique set of constraints, i.e., unique component constraint $\mathbb{C}_{ij}$. Complete constraints for each component implementation of the GO game are listed in table 3.6, 3.7, and 3.8.

3.3.3 Root Component

The root component of the software hierarchy is a special component and the starting point of component assembly. While it shares the some common properties as a

constraint	board $(i = 1, j = 1)$
R_{ij}	$\emptyset$
P_{ij}	$\{\text{“}0 \leq p_1^1 \leq 10\text{”}, \text{“}0 \leq p_1^2 \leq 0.5\text{”}, \text{“}1 \leq p_1^3 \leq 2\text{”}\}$
D_{ij}	$\{\text{“}p_1^1 \geq 20p_1^2\text{”}\}$
M_{ij}	$\{\text{“}p_1^1 \leq 1.1p_2^1\text{”}, \text{“}p_1^2 \leq 1.1(p_2^2 + p_3^1)\text{”}, \text{“}p_1^3 = p_2^3\text{”}\}$

Table 3.6: Complete constraint list for component **board**

constraint	AI	
	GnuGO $(i = 2, j = 1)$	**Random** $(i = 2, j = 2)$
R_{ij}	$\emptyset$	$\emptyset$
P_{ij}	$\{\text{“}0 \leq p_2^1 \leq 10\text{”}, \text{“}0 \leq p_2^2 \leq 0.5\text{”}, \text{“}p_2^3 = 2\text{”}\}$	$\{\text{“}0 \leq p_2^1 \leq 10\text{”}, \text{“}0 \leq p_2^2 \leq 0.5\text{”}, \text{“}p_2^3 = 1\text{”}\}$
D_{ij}	$\emptyset$	$\emptyset$
M_{ij}	$\emptyset$	$\emptyset$

Table 3.7: Complete constraint list for component **AI**. Here the fact that competition level of **GnuGO** is higher than the competition level of **Random** means that **GnuGO** is more intelligent than **Random**.

constraint	audio	
	MIDI $(i = 3, j = 1)$	**dummy** $(i = 3, j = 2)$
R_{ij}	{“A sound device exists.”, “Platform is AC powered.”}	default, i.e., R_{31}^c
P_{ij}	$\{\text{“}0 \leq p_3^1 \leq 0.5\text{”}\}$	$\{\text{“}0 \leq p_3^1 \leq 0.5\text{”}\}$
D_{ij}	$\emptyset$	$\emptyset$
M_{ij}	$\emptyset$	$\emptyset$

Table 3.8: Complete constraint list for component **audio**

regular component, such as the specification of R_{ij}, P_{ij}, D_{ij}, and M_{ij}, it has only one implementation and one provided service, `main` as shown in table 3.2, which is the starting point of the application execution and will be called by the launcher after the component assembly.

The root component also has parameters in a default finite domain as normal components. Nonetheless, the range of its parameters may be redefined by users via an application launcher, unlike the normal components being redefined by their parent components. For example, users may specify the response time and power consumption of the root component **board** in an application launcher.

3.3.4 Component Stub

Instead of running on single machine, **FRAME** is designed to be able to automatically distribute components of applications to multiple platforms in order to support ad hoc systems. This is done via the component stub that is a special implementation of the component.

Component stubs provide the communication mechanism between components on different platforms, and need to cooperate with the real implementations of the components to provide functionality. Conceptually, what a component stub does is to transfer the services of an implementation of a component from one platform to another.

Fig. 3.8 is the communication mechanism between two non-distributed component objects, A and B, executing on a single platform. Component object A and B actu-

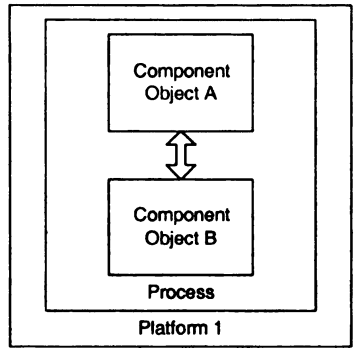

Figure 3.8: Non-Distributed Component Invocation

ally execute within a process and communication with each other through a regular method invocation mechanism within the process. Fig. 3.9 shows the underlying communication mechanism between two distributed component objects executing on two different platforms. Instead of using the real implementation of component B, component A actually interacts with the stub of component B, which implements the communication infrastructure with a remote daemon process called the **FRAME agent**. During the remote method invocation, the component stub will transfer necessary information about the method and its the associated object, which is B, to the **FRAME agent**; it will also pass the method arguments. Once the information is received, the **FRAME agent** will locate the object of the specified method, invoke the object method with the arguments from the stub, and then pass back the returned value to the stub.

3.4 Component Assembly

Since a component may have multiple implementations that work appropriately under a specific environment, an application may be assembled from several possible

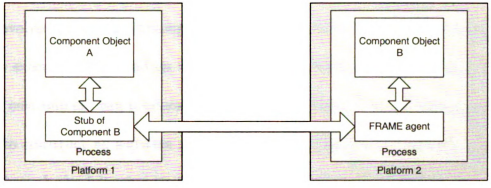

Figure 3.9: Distributed Component Invocation

combinations of components. The main purpose of component assembly is to identify the **feasible combination** consisting of all feasible component implementations. Suppose that a program requires components $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_i$. The collection of $\mathbb{C}_{iv_i}$ for component $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_i$ with version $v_1, v_2, \ldots, v_i$, respectively, is called a *software constraint* and denoted as $\mathbb{S}_{v_1 v_2 \ldots v_i}$. For instance, all four possible software constraints of the GO game, corresponding to all four possible combination of components for the GO game, will be $\mathbb{S}_{111} = \mathbb{C}_{11} \cup \mathbb{C}_{21} \cup \mathbb{C}_{31}$, $\mathbb{S}_{112} = \mathbb{C}_{11} \cup \mathbb{C}_{21} \cup \mathbb{C}_{32}$, $\mathbb{S}_{121} = \mathbb{C}_{11} \cup \mathbb{C}_{22} \cup \mathbb{C}_{31}$, and $\mathbb{S}_{122} = \mathbb{C}_{11} \cup \mathbb{C}_{22} \cup \mathbb{C}_{32}$. If all constraints within $\mathbb{S}_{v_1 v_2 \ldots v_i}$ are satisfied, $\mathbb{S}_{v_1 v_2 \ldots v_i}$ is called a *feasible software constraint.*

As mentioned earlier, each implementation of a component should have a unique component constraint $\mathbb{C}_{ij}$. Thus, the software constraint would be different from each other, and the mapping between combinations and software constraints are one-to-one. Such a one-to-one property may be used to identify the feasible combination. That is, if $\mathbb{S}_{v_1 v_2 \ldots v_i}$ is a feasible software constraint, the combination of components $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_i$ with version $v_1, v_2, \ldots, v_i$ respectively will be a feasible combination.

Before an application is executed, it needs to search for the appropriate version of each involved component through component assembly. Whether an application is able to execute will depend on whether the user specified parameters and the computing environment can produce a feasible combination. First, component assembly needs to construct all software constraints from all possible combinations, which is done through the following steps, starting from the root component as the current component:

1. Query the implementation information about the current component.
2. Query the location information, i.e., the location of component repository, for each implementation of the current component.
3. Load each implementation of the current component from the component repository.
4. Build component constraint $\mathbb{C}_{ij}$ from loaded component i with version number j.
5. Query all child component information of the current component.
6. Repeat the first step for each child component.
7. Build all possible combinations from all implementation of each components.
8. For combinations, the corresponding software constraints $\mathbb{S}_{v_1 v_2 \ldots v_i}$ may be built from $\mathbb{C}$'s.

After constructing all possible software constraints, then the remaining assembly process is basically a constraint satisfaction problem (CSP); that is, determining the truth value of all possible software constraints to find a feasible software constraint, which will in turn give the corresponding feasible combination. Once the feasible combination is found, the feasible implementation of each involved component will be initialized with appropriate values of parameters that will satisfy the software constraint. For the GO game, one difference for a given `AI` component between these two software constraints ($\mathbf{S}_{111}$ and $\mathbf{S}_{112}$, or $\mathbf{S}_{121}$ and $\mathbf{S}_{122}$) is the existence of a sound device. If no sound device exists, the feasible program will be the one using the **dummy** version but not the `MIDI` version because the constraint, "A sound device exists.", is not satisfied.

A constraint satisfaction problem may be solved using the generate-and-test paradigm (GT). In this paradigm, each possible combination of the variables is systematically generated and then tested to see if it satisfies all the constraints. The first combination that satisfies all the constraints is the solution. The number of combinations considered by this method is the size of the Cartesian product of all the variable domains.

A more efficient method uses the backtracking (BT) paradigm. In this method, variables are instantiated sequentially. As soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial instantiation violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has alternatives available. Clearly, whenever a partial instantiation violates a constraint, backtracking is able to eliminate a subspace

from the Cartesian product of all variable domains. The backtrack method essentially performs a depth-first search [196] of the space of potential solutions of the CSP.

This backtracking algorithm is currently implemented in **FRAME**, i.e., using multiple nested loops for parameters, which take each values in the specified parameter domains and examine if the each constraint of the software constraint is satisfied. If, for a parameter, such a value exists that make a software constraint to be true, the value is called the *feasible value* of the parameter. Then the application will be assembled from the corresponding combination with each parameter being initialized to its feasible value. This may not be an efficient approach. For a static component environment, the component assembly process only needs to be invoked once, right before the execution of the application, since the computing environment does not change and the feasible software constraints will remain invalid during the course of execution. In this case, the component assembly will not have any performance impact on the application execution. However, this case may not be true for a dynamic computing environment, since the computing environment may change much such that the feasible software constraint becomes false and causes the invocation of component assembly. More details about the performance impact of the component assembly will be discussed in chapter 5.

For distributed applications, the component assembly process will distribute components to specified platforms before constructing software constraints. All components except the root component may be distributed to any possible specified platform as long as their component constraints $\mathbb{C}_{ij}$'s are true on the platform. The root compo-

nent is not distributable, since it has to interact with the application launcher and is restricted to execute on the same platform as the application launcher.

As a consequence, there might be more than one possibility to distribute components depending on the number of components and specified platforms, i.e., $n_p^{(n_c-1)}$ possibilities with n_c being the number of components, n_p being the total number of specified platforms, and the root component always being executed on the first specified platform. We call each possibility a *distribution*, and a distribution as an n-distribution, $1 \leq n \leq n_p$, if all components are distributed to n of n_p specified platforms. For each distribution, we may construct all possible software constraints and determine if a feasible software constraint exists by solving the constraints. A distribution is called feasible if a feasible software constraint can be found within the distribution. Therefore, the component assembly process for distributed applications is to find a feasible distribution from all possible distributions.

The current component distributing strategy is a top-down strategy with load distribution. Indeed, the component assembly process first tries to find a feasible distribution from all the n_p-distributions. If fails, it repeats the search from the $(n_p - 1)$-distributions, and then $(n_p - 2)$-distributions, until it find a feasible distribution or fails at 1-distribution.

3.4.1 Application Launcher

As mentioned earlier, an application is not directly invoked by a user but via an application launcher. The main task of the launcher is to interact with the root

component. That is, the launcher may take user specified constraints of applications, i.e., parameters domain range of the root component. It then will retrieve the root component by consulting the location information of the root component from the component registry, redefine parameter constraints of the root component, and then start the component assembly process.

Users may specify the target platform that the application will execute on. To execute a distributed application, users only needs to specify a set of, instead of one, target platforms. The first platform of the set is the platform that the launcher will execute. Because we want users to interact with the application on a certain platform via launcher that will be the first platform of the set, the root component will execute on the first specified platform, and the rest of the components may be distributed to all specified platforms.

3.5 Discussion

3.5.1 Alternative Adaptation Approach

A more simple and straightforward approach for implementing the adaptation is to use the condition statements such as if-else statements. The general structure of the code would be using nested if-else statements and each is used to decide the appropriate implementation of a component. Once an implementation is selected, execution flow may go into the inner if-else statements to select the appropriate implementation of other components. Compared to the steps that component assembly needs to go

through, it is not difficult to argue that such an approach has a performance advantage over component assembly. The comparison results will be revealed in chapter 6.

However, the condition statements approach is primitive from the software engineering perspective. The condition statements contain constraint information to select the implementation and also the software hierarchy information that is required to direct the execution flow to the condition statements for selecting the implementation of the different components. While the number of components and their implementations increase, the code tends toward so called "spaghetti code" that has a complex and tangled control structure and the software will become more difficult to maintain or modify.

The most important limitation of the condition statements approach, is that condition statements are hard-coded, so the availability of all implementations need to be known during the development stage, which implies the assumption that all the computing environments may be anticipated and correctly modeled and the corresponding component implementations are implemented. It is not flexible enough to integrate newly developed implementations for the environments unknown during the development stage without rewriting and recompiling the code, and, of course, the down-time. For some systems, in which the robustness is important and the down-time is very expense, the condition statements approach is not appropriate particularly under dynamic environment,

On the contrary, **FRAME** might greatly alleviate the spaghetti code problem since the software hierarchy and the component constraints are separately specified at the software system level. The information may be stored in component registry and then

resolved by component assembly. Besides, **FRAME** may remove the barrier of inflexible integration of newly developed component implementations by simply adding the new implementation into component repository and update the component registry.

3.5.2 Components

Component-based software engineering has existed in one form or another for a number of years. So far, we did not clearly clarify the meaning of **component**. Actually, there is no universal definition of **software component**, and the term **component** has come to mean many different things to different people. A component may have a very shallow meaning such as a collection of subroutines and libraries[197]. It may also have a very deep semantic meaning and is abstract across various domains [198]. Cheesman [199] identifies the component concept from a number of different perspectives, examines the characteristics and benefits of each, and shows how they relate to each other. The components under the **FRAME** fall into the *Service Perspective*, which considers a component to be a software entity that offers services (operations or functions) to its consumers. It also introduces the notion of a contract between the provider and the consumer of those services. By focusing on the notion of a contract, the service perspective introduces the important distinction between the specification of a component (what it does) and its implementation and executable forms (how it does it) . This distinction is fundamental to the management of dependencies between components and begins to address the important requirement to be able to replace a component with minimal impact on the consumer, often referred to

as "plug-and-play". Furthermore, consumers of a component should only be dependent on the specification of that component. Any dependency on its implementation (whether through direct knowledge or due to unspecified assumptions which happen to be supported) will mean that the application is likely to fail when the component is upgraded or replaced.

3.6 Summary

In this chapter, the adaptive Java software framework (**FRAME**) is presented. An application should be composed of components under **FRAME**. Each component has information about quality of service or resource requirements specified as constraints, and cooperates with other components through services. Furthermore, an application only needs **FRAME** APIs and constraints, but not its functionality, to be adaptive. Thus, the adaptation and functionality of an application may be separately specified and implemented, and, hence, alleviate the complexity of development.

The services define the dependency of the components, i.e., software hierarchy. Each component, except for the root component, might have more than one implementation of the provided services. Each implementation of a component will produce reasonable performance or appropriate response for some specific situations. The software hierarchy information needs to be registered to the component registry, and needs to be resolved during run-time by the component assembly via the component registry in order to know which components are required for an application.

Each implementation may contain a set of constraints that specify how well the implementation will perform or response under certain environment. There are several different types of constraints, such as parameters and connectors. Parameters specify the quantifiable or enumerable metrics and the feasible domains of the metrics. Besides, connectors specify the mutual dependency of parameters.

An application may be assembled from several possible combinations of components. Each combination has a set of constraints, or a software constraint, which consists of all the constraints from involved component implementations. Since no two implementations should have same set of constraints, the mapping between combinations and software constraints are one-to-one. By solving which software constraint is feasible, i.e., all constraints in the software constraint are satisfied, the corresponding feasible combination will be found.

Chapter 4

Component/Constraint Markup Language (CSML)

4.1 Motivation

For complicated software projects, the implementations may become difficult to maintain. Software architecture research is directed at reducing the cost of application development by focusing attention on high level design issues and increasing reuse. Over the past decade, software architecture research has emerged as the principled study of the overall structure of software systems, especially the relations among subsystems and components. From its roots in qualitative descriptions of useful system organizations, software architecture has matured to encompass broad explorations of notations, tools, and analysis techniques [200]. Whereas initially the research area interpreted software practice, it now offer concrete guidance for complex software design and development.

Similar reasoning may be applied to the development of adaptive software under **FRAME**, where the concept of a software architecture has been used in **FRAME**. Moreover, although **FRAME** provides the framework to allow component-based software to be assembled during the run-time, the usage of the **FRAME** APIs may not be straightforward and the plug-in compatibility of components will become more difficult to maintain for collaborative projects. For instance, in addition to the code implementing the component services, additional code needs to be added in order to work under **FRAME**, such as the Java implementation of constraints for each component assembly, communication for registering and querying the necessary information in the registry, and interpretation for the result from the registry. Particularly, the constraints may be different among implementations. Accordingly, a high level software architecture specification language will be helpful for developers to maintain the consistency of component interfaces and implementations.

From the previous chapter, the architecture of **FRAME** is described with the relations between components, not the detailed functionality of components. Actually, the implementation of adaptation and the component functionality is mutually independent. Hence, the usage of the **FRAME** APIs only depends on the relations between components. It will be possible to develop an application to use **FRAME** APIs with the only information about the software architecture.

4.2 Overview of CSML

Component/Constraint Specification Markup Language (**CSML**) is a specification language to abstract the information about the relations between components. It uses Extensible Markup Language (XML) [201], a very flexible text format derived from SGML (ISO 8879) [202], to help software developers specify the high level structure of adaptive software systems under **FRAME**. The software hierarchy information is captured by specifying child component dependency. The consistency of component interfaces will be maintained by specifying the provided services and the required services from child components. Individual implementation information such as constraints and other information about locations of the component registry and repository may also be specified. Once specified, **CSML** will compile the specifications and generate the code for adaptation, i.e., the adaptation shell including **FRAME** APIs and constraints as shown in fig. 3.1. Next, the developers need to implement the functionality of the components.

Fig. 4.1 illustrate the flow of using **CSML** to develop a component. First, **CSML** compiles the component specification, and generates a component interface and a base class of a component implementation in Java. The generated code includes adaptation shell code such as methods for each constraint and **FRAME** APIs for communicating with the component registry, accessing the parameters, constructing the all software constraints, and constraint solving. The component stub is the communication mechanism for distributed computation that has been described in the previous chapter. The component register is used to register component information into the registry.

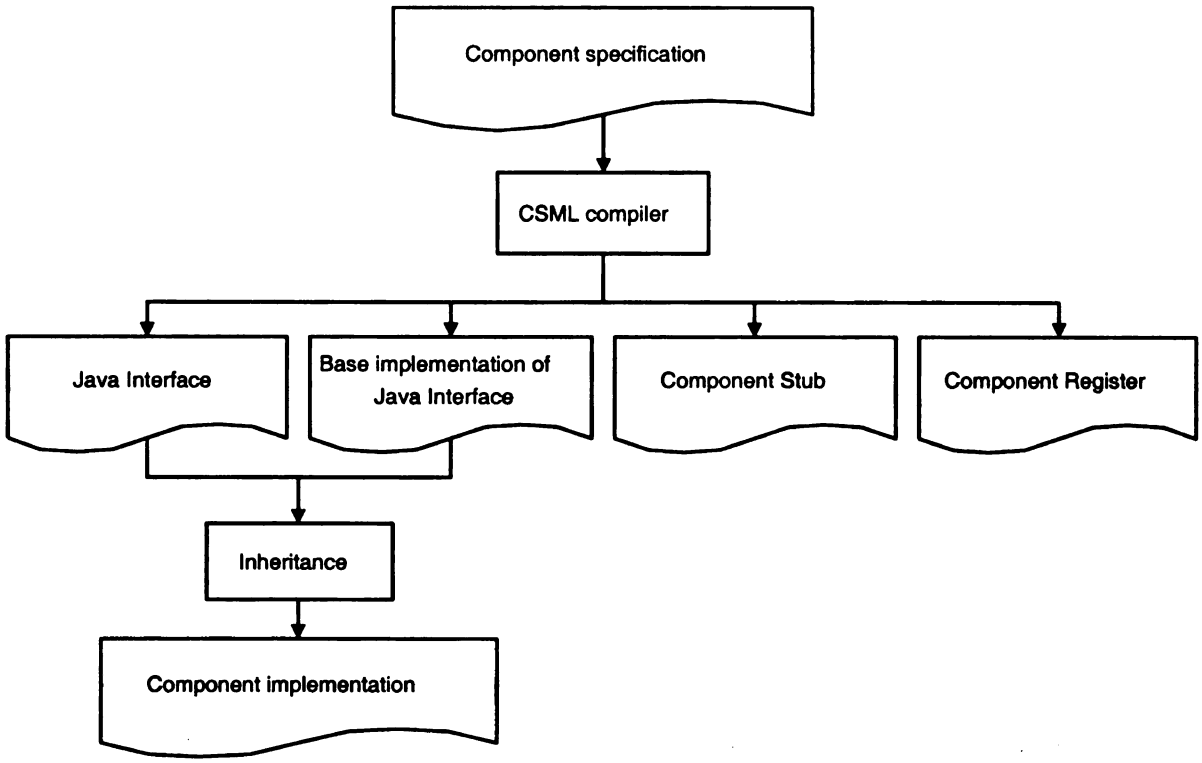

Figure 4.1: Flow chart of using CSML to develop a component

Both may also be generated from the specification. Then, component developers only need to inherit the base class to implement the functionality of the component; basically, the infrastructure needed to work under FRAME is generated from the specification by CSML.

Similarly, application users may use CSML to specify the intended performance or quality of the application execution. An application launcher that carries the information about the desired performance and pass it to the root component of the application will be automatically generated from the user specification as shown in fig. 4.2.

The details of the format of CSML specification will be illustrated with the GO game example in the following sections.

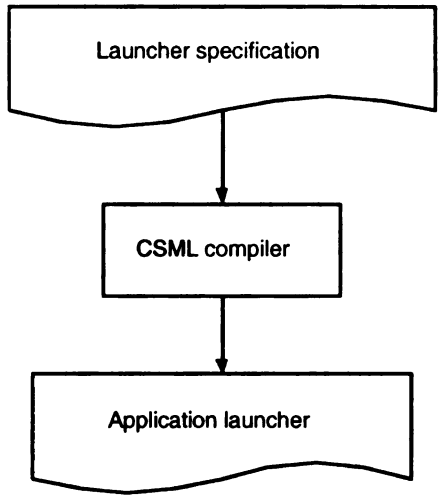

Figure 4.2: Flow chart of using CSML to specify a launcher

4.3 Root component specification example

Table. 4.1 is the specification of the root component **board** in CSML. The element *root-component* specifies identity of a root component, including the name of root component (in attribute *name*), host-name of component registry (in attribute *registry-host*), and location of the root component itself (in attribute *uri*), as shown in lines 1-2.

Parameters are specified in the element *parameter* with name and their range (in attribute *upper* and *lower*), lines 4-6. For example, from the parameter list of the component **board**, as shown in table 3.3, the parameter p_1^1 may be specified with name being **response_time** and id being **p11**. Also, from the constraint list of the component **board**, as shown in table 3.6, the parameter constraint "$0 \leq p_1^1 \leq 10$" will specify the upper and lower bound of p_1^1 to be 10 and 0 respectively.

Child components are specified in the element *child-component*, which will be used to resolved the software hierarchy by component assembly. Similar to the element *root-*

```
 1 <root-component name="board" application="GO"
 2   uri="http://www.cse.msu.edu/~korenson/class/JavaGO.jar" registry-host="192.168.1.111">
 3   <description> GO board. </description>
 4   <parameter name="response_time" id="p11" value-type="int" upper="10" lower="0" />
 5   <parameter name="power_consumption" id="p12" value-type="float" upper="0.5" lower="0" />
 6   <parameter name="intelligence" id="p13" value-type="int" upper="2" lower="1" />
 7   <internal-connector id="d1">
 8     <from-current parameter-id="p11" alias="var1" />
 9     <from-current parameter-id="p12" alias="var2" />
10     <definition>
11       return ~var1$ >= 20 * ~var2$;
12     </definition>
13   </internal-connector>
14   <child-component name="AI" id="c1" registry-host="192.168.1.111"
15     uri="http://www.cse.msu.edu/~korenson/class/GOPlayer.jar" />
16   <child-component name="audio" id="c2" registry-host="192.168.1.111"
17     uri="http://www.cse.msu.edu/~korenson/class/audio.jar" />
18   <external-connector id="m1">
19     <from-current parameter-id="p11" alias="var1" />
20     <from-child child-id="c1" parameter="response_time" alias="var2" />
21     <definition>
22       return ~var1$ <= 1.1 * ~var2$;
23     </definition>
24   </external-connector>
25   <external-connector id="m2">
26     <from-current parameter-id="p12" alias="var1" />
27     <from-child child-id="c1" parameter="power_consumption" alias="var2" />
28     <from-child child-id="c2" parameter="power_consumption" alias="var3" />
29     <definition>
30       return ~var1$ <= 1.1 * (~var2$ + ~var3$);
31     </definition>
32   </external-connector>
33   <external-connector id="m3">
34     <from-current parameter-id="p11" alias="var1" />
35     <from-child child-id="c1" parameter="intelligence" alias="var2" />
36     <definition>
37       return ~var1$ == ~var2$;
38     </definition>
39   </external-connector>
40   <main>
41     <instance name="Ai" child-id="c1" />
42     <instance name="Audio" child-id="c2" />
43     <required-service service="play" child-id="c1" />
44     <required-service service="playAudio" child-id="c2" />
45     <definition>
46       try {
47         Audio.playAudio()
48         Ai.play();
49       } catch(Throwable e) {
50          e.printStackTrace();
51       }
52     </definition>
53   </main>
54 </root-component>
```

Table 4.1: CSML for the component board

component, the identity related information of child components, including name and locations, need to be specified here. As shown in fig. 3.7, two child component, `AI` and `audio`, are needed by `board`, and are specified in lines 14-15 and 16-17 respectively. The internal connector is specified in the element *internal-connector* with an id in attribute *id*. The parameters that are related in the internal connector needs to be specified in the element *from-current* with the parameter id being specified in the attribute *parameter-id*. An alias (in attribute *alias*) also needs to be specified for the purpose of being used in the definition. Finally, the definition of the internal connector will be specified in the element *definition*. The format of *definition* is very close to Java boolean function definition except the aliases of parameters, instead of parameters themselves, are used. For example, the internal connector, "$p_1^1 \geq 20p_1^2$", in table 3.6 is specified in lines 7-13. The elements *from-current* specify that the internal connector specify the relation between p_1^1 and p_1^2". The definition is specified in the form similar to Java code "`return ^var1# >= 20 * ^var2#;`" with var1 and var2 being aliases of parameter p_1^1 and p_1^2 respectively.

Similar to the internal connector, the external connector is specified in the element *external-connector*. Both elements are very close, except *external-connector* contains an addition element *from-child* that specify the involved parameter from child component. The involved parameter from child component is identified by the attributes *child-id* that must match the attribute *id* from one of *child-component* and the parameter name that is specified in the CSML specification of the child component. For example, the external connector, "$p_1^1 \leq 1.1p_2^1$", in table 3.6 is specified in lines 18-24. The elements *from-current* and *from-child* specify that the external connector

specify the relation between p_1^1 and p_1^2", where p_1^2" is identified through the attributes *parameter* and *child-id* that links to the attribute *id* of child component **AI**. The definition is also specified in the form similar to Java code "**return ^var1# <= 1.1 * ^var2#;**" with var1 and var2 being aliases of parameter p_1^1 and p_2^1 respectively.

As mentioned in the previous chapter, the root component has only one provided service, **main**. The definition of **main** may be specified in the element *main*. For example, in lines 40-53, *main* specifies that the **main** function will instantiate the two child components in the element *instance*. The instance of the child component is identified through the attributes *child-id* that must match the attribute *id* from one of *child-component* and will be give a name in the attribute **name**. The required service from the child components is specified in the element *required-service* with the attributes *child-id* to identify the child component and *service* to the name of service. In this example, **main** will call the service **playAudio()** provided by **Audio**, an instance of the component **audio**, and **play()** provided by **Ai**, an instance of the component **AI**.

4.4 Non-Root Component specification example

The CSML format of the non-root component is similar to the root component. However, unlike the root component, the non-root component may have multiple implementations. Hence, the CSML format of the non-root component is divided into two parts and is distinguished by the elements *general* and *customized* as shown in table. 4.2 for the CSML specification of the component **audio**.

```
 1 <component name="audio" registry-host="192.168.1.111"
 2   uri="http://www.cse.msu.edu/~korenson/class/audio.jar">
 3   <general>
 4     <provided-service id="s1">
 5       <declaration method-name="playAudio" return-type="void" />
 6     </provided-service>
 7     <parameter name="power_consumption" id="p31" value-type="float" upper="0.5" lower="0" />
 8   </general>
 9   <customized version="MIDI" uri="http://www.cse.msu.edu/~korenson/class/midi.jar">
10     <description> MIDI version </description>
11     <constraint-definition constraint-id="s1">
12       <resource-requirement>
13         <resource name="sound-device" id="r1">
14           <built-in><sound /></built-in>
15         </resource>
16         <resource name="AC-powered" id="r2">
17           <built-in><AC /></built-in>
18         </resource>
19       </resource-requirement>
20     </constraint-definition>
21   </customized>
22   <customized version="dummy" uri="http://www.cse.msu.edu/~korenson/class/dummy.jar"
23     default="yes">
24     <description> Dummy version </description>
25   </customized>
26 </component>
```

Table 4.2: CSML for the component audio

The element *general*, lines 3-8, specifies the information that is implementation independent. In the example, it specifies that the **audio** component have one provided service in the element *provided-service* and one parameter in the element *parameter*. The element *provided-service* specifies the name and type of the return object of the service in the attributes *method-name* and *return-type* of the element *declaration*. Lines 4-6 specify that the **audio** component provides a service called **playAudio** without returning any object or value.

The element *customized* specifies the information that varies among implementations. Lines 9-21 specify the **MIDI** implementation and lines 22-25 specify the **dummy** implementation. The identity information of the implementation is specified in the attributes *version* and *uri* of the element *customized*. The constraints of each implementation are specified in the element *constraint-definition*. The constraints may be either resource requirements, parameter constraints, internal connectors, or external connectors. In the example, lines 12-19 specify the resource requirements of the **MIDI** implementation, including sound device, lines 13-15, and AC power, lines 16-18, in the elements *resource*. The element *built-in* means that there is a built-in Java method that would be automatically generated by **CSML** to check the availability of the specified resource, such as sound devices or AC power. In line 23, there is an attribute *default* in the element *customized*. With the value being set to **yes**, the **dummy** implementation is specified as the "default" version of the **audio**; that is, if the required resources of other versions are not satisfied, the default version will be used.

Table. 4.3 is the specification of the non-root component **AI** in **CSML**. It specifies the **GnuGO** implementation in lines 11-16 and the **Random** implementation in lines 17-22. Each implementation specifies different parameter constraints for the parameter p_2^3 in the elements *parameter-constraint*. The parameter is identified through the attributes *constraint-id* that must match the attribute *id* from one of *parameter*. For example, line 14 specifies the parameter constraint "$2 \leq p_2^3 \leq 2$", i.e., "$p_2^3 = 2$" as shown in table 3.7.

4.5 Launcher specification example

CSML also allows users to specify target platforms and the intended performance of the application, and generates the application launcher. Table. 4.4 is the specification of the launcher. The element *launcher* specifies the name of the launcher (in the attribute *name*), and the name and registry location of the application (in the attributes *application* and *registry-host*).

The element *target* specifies the information about the target platform, such as the location (in the attribute *host*) and the architecture (in the attribute *arch*). Lines 3 and 4 specifies that the application may execute on two target platforms with different architectures. Notice that the component **board** is the root component and will be executed on the first platform and the other two components may be distributed to the second platform.

The element *constraint* specifies the intended performance and output quality of the application by specifying the range of parameters of the root component. The pa-

```
 1 <component name="AI" registry-host="192.168.1.111"
 2   uri="http://www.cse.msu.edu/~korenson/class/AI.jar">
 3   <general>
 4     <provided-service id="s1">
 5       <declaration method-name="play" return-type="void" />
 6     </provided-service>
 7     <parameter name="response_time" id="p21" value-type="int" upper="10" lower="0" />
 8     <parameter name="power_consumption" id="p22" value-type="float" upper="0.5" lower="0" />
 9     <parameter name="intelligence" id="p23" value-type="int" upper="2" lower="1" />
10   </general>
11   <customized version="GnuGO" uri="http://www.cse.msu.edu/~korenson/class/GnuGo.jar">
12     <description> GnuGo version </description>
13     <constraint-definition constraint-id="p23">
14       <parameter-constraint upper="2" lower="2" />
15     </constraint-definition>
16   </customized>
17   <customized version="Random" uri="http://www.cse.msu.edu/~korenson/class/Random.jar">
18     <description> Random version </description>
19     <constraint-definition constraint-id="p23">
20       <parameter-constraint upper="1" lower="1" />
21     </constraint-definition>
22   </customized>
23 </component>
```

Table 4.3: CSML for the component AI

```
 1 <launcher name="GO launcher" application="GO"
 2  registry-host="192.168.1.111">
 3  <target host="192.168.1.34" arch="arm" />
 4  <target host="192.168.1.111" arch="i686" />
 5  <constraint name="response_time" value-type="int"
 6   lower="0" upper="15" />
 7  <constraint name="power_consumption"
 8   value-type="float" lower="0" upper="1" />
 9  <constraint name="intelligence"
10   value-type="int" lower="1" upper="2" />
11 </launcher>
```

Table 4.4: CSML for the GO game launcher

rameter is identified with the attribute *name* that must match the attribute *name* from one of *parameter* in the root component specification. For example, lines 5 and 6 specify **response_time**, i.e., the parameter p_1^1 of the root component **board**, should be between the range 0 and 15.

Once the constraints are specified, the range of the root component parameters, p_1^1, p_1^2, and p_1^3, will be replaced by the user specified range and the new range will be used during the component assembly.

4.6 Summary

Since the architecture of **FRAME** only depends on the relations between components but not the detailed functionality of components, it will be possible to develop the adaptation of an application with the only information about the software architecture.

Component/Constraint Specification Markup Language (**CSML**) is a specification language to abstract the information about the relations between components. The software hierarchy information is captured by specifying child component dependency. The consistency of component interfaces will be maintained by specifying the provided services and the required services from child components. Individual implementation information such as constraints and other information about locations of the component registry and repository may also be specified. Once specified, **CSML** will compile the specifications and generate the code for adaptation. Then, component developers only need to inherit the generated code to implement the functionality of the compo-

nent; basically, the infrastructure needed to work under **FRAME** is generated from the specification by **CSML**.

Similarly, application users may use **CSML** to specify the intended performance or quality of the application execution. An application launcher that carries the information about the desired performance and pass it to the root component of the application will be automatically generated from the user specification.

Chapter 5

On-line Adaptation: Brew and Component Reassembly

5.1 Motivation

As mentioned in chapter 1, computing environments are becoming more dynamic due to the population of mobile devices. For the environment-sensitive applications, such as real-time applications, they have to be adaptive to the environmental change to ensure the correctness of the execution. The general approach for this on-line adaptation may be achieved in the following steps:

1. Periodically sample or log the run-time information of the applications, particularly the data that are sensitive to the environment.

2. Measure the environmental change by interpreting the sample data.

3. Respond according to the level of the environmental changes.

The extra work above for the on-line adaptation definitely consumes computing resources and it will have a performance impact on the execution of applications. Actually, the purpose of the on-line adaptation is to sacrifice the performance in order to improve the quality of the execution, such as stability or predictability of real-time applications. Thus, a design challenge for the on-line adaptation is to make such a trade-off more promising; that is, increase the quality of the execution without losing too much performance or even interrupting the application execution.

There are two design factors may have a performance impact. The first one is the frequency of run-time data sampling. Obviously, with a higher data sampling rate, applications will have more up-to-date knowledge about the computing environment, but will consume more computing resources and more likely will interrupt the execution. On the contrary, a lower data sampling rate may be appropriate for resource limited computing environment, but applications may not be able to respond to the environmental change in time and the execution will likely fail.

How does software respond to the different scales of the environmental changes, i.e., how radically does the environmental change, may also have significant performance impact from resource management perspective. Particularly, inappropriate over-reacting, such as major responses to minor environmental changes, may simply waste the computing resources. As a real-world analogous example, a worn tire should be replaced with a new tire. However, when it is possible to simply inflate a low-pressured tire, it will be over-reacting and waste the money to replace the tire with a new tire.

ASAP uses two different on-line adaptation schemes, **Brew** and *component reassembly*, to let applications be able to respond to different scales of environmental changes. For a small scale of environmental changes, people may use **Brew** to steer and repair the affected applications that do not work properly under the new environment without terminating the applications. For a large scale of environmental change, the execution of the applications will be temporarily suspended, component reassembly will be invoked, and then resume the execution with appropriate implementations of the components. Furthermore, the run-time data sampling rate may be adjusted dynamically with help of **Brew**.

5.2 Java Steering System: Brew

5.2.1 Overview

Brew is an interactive computation steering environment for sampling and interpreting run-time information of the Java remote applications, such as performance data, in real-time. People then may visualize the instrumented data and steer the remote applications, as shown in fig. 5.1.

Fig. 5.2 shows how the execution quality of the application may be changed on the fly. The local machine collects the run-time environment-sensitive data from the remote application, a Java MPEG player, which will be described in more details in the next chapter. The run-time environment-sensitive may reveal the change of computing environment, and people may conclude the execution quality of the remote

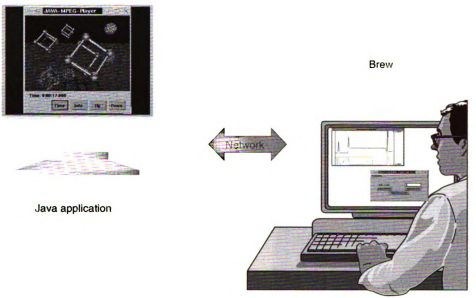

Figure 5.1: **Brew** is an interactive Java environment that people may visualize the instrumented data and steer the remote applications.

application from the data and send a command to respond the environment, such as to reduce the image quality in order to improve the frame displaying rating.

As shown in fig. 5.3, the overall architecture of **Brew** includes the instrumentation system, control system, and a Java application with the **Brew** subsystem embedded. More details about how people may interact with the application via these subsystems are described in the following two subsections.

5.2.2 Run-Time Data Collection and Visualization

For collecting and visualizing run-time data, **Brew** uses PG^{RT} [203], an environment for integration of tools and systems for instrumentation, performance visualization, and analysis of complex real-time systems. Two parts of PG^{RT} used are **BRISK** and the

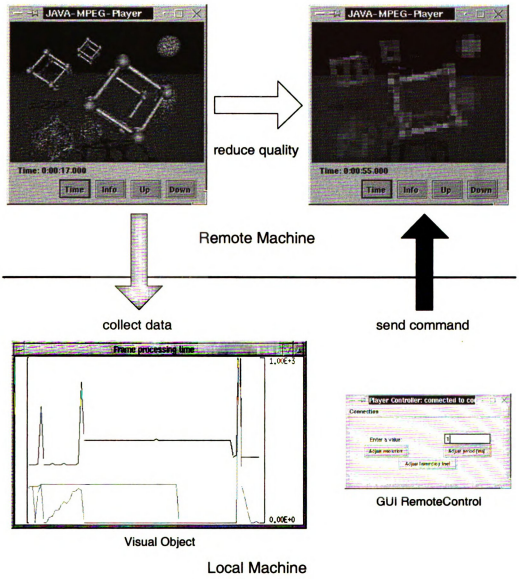

Figure 5.2: On-line change of execution quality via **Brew**.

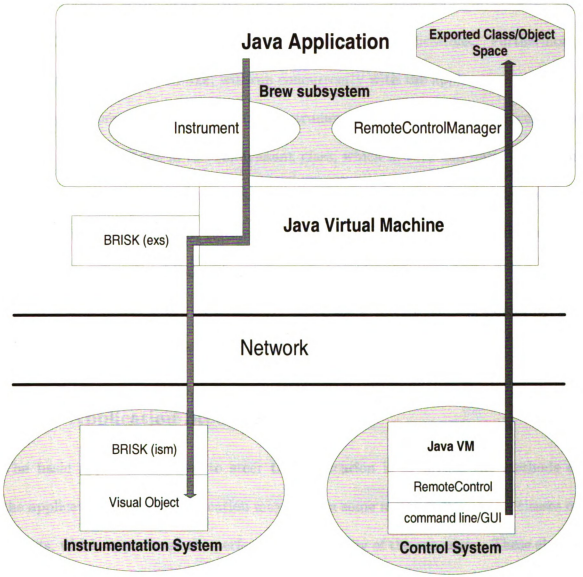

Figure 5.3: Architecture of `Brew`

Visual Object Framework (VO). **BRISK** is used for communication between applications and VOs. VO is used to develop application specific performance visualizations, which includes processing and rendering of instrumentation data.

To incorporate with **Brew**, the **Brew** subsystem, including **Instrument** and **RemoteControlManager**, need to be embedded in the applications. Furthermore, a daemon of **BRISK**, called **exs**, will run concurrently with the application.

Java applications may put the run-time instrumentation data in a shared memory by using wrapper methods in the **Instrument** class, which in turn call native functions to access **exs**. The instrumentation data is collected by **exs**, and then delivered to the peer **BRISK** daemon, **ism**, through the network. VO then retrieves, processes and renders these data for visualization.

A screen shot of VO is shown in fig. 5.2. Also, the arrow on the left side of fig. 5.3 shows the data path for the run-time data collection and visualization.

5.2.3 Application Steering

The basic approach of **Brew** to steer the application is to invoke the methods of the application. That is, application may contain some methods from some classes or objects, which may change the performance or quality of the execution. These classes or objects need to be exported as shown in fig. 5.4, so the methods will be able to be accessed from different processes through an agent called **RemoteControlManager**.

Thus, after concluding the environmental change from VO, people may able to steer the application by sending commands through a remote control in order to respond

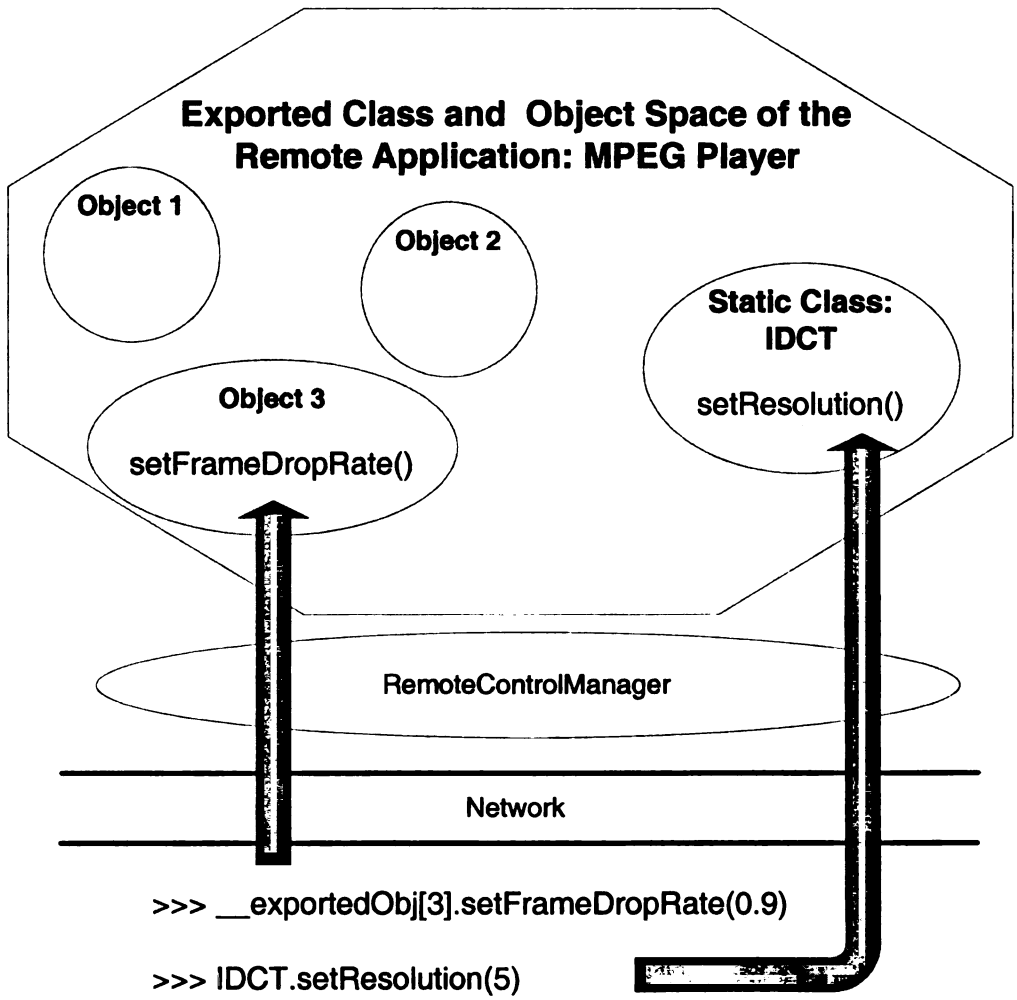

Figure 5.4: Architecture of `Brew`

to the environmental change. The commands may be sent via command line interface or graphical user interface (GUI). GUI is actual the wrapper of the command line interface, which simply send the text format of command to **RemoteControl**.

The semantic of the commands is to invoke a method of a class or object of the remote applications. The syntax is similar to Java as shown in fig. 5.4. For example, the command "**IDCT.setResolution(5)**" will invoke the method, **setResolution**, of the exported static class, **IDCT**, of the remote application with argument 5. After the invocation, the execution quality of the application has been changed, e.g., the image quality of MPEG the player is changed and the frame rate may be improved.

Similar to **exs** and **ism** of **BRISK**, **RemoteControl** and **RemoteControlManager** are the peer communication infrastructure as shown in fig. 5.3 and completely implemented in Java. They encapsulate the basic communication details based on the steering protocol that the steered functions of the remote application will be easily identified and invoked.

RemoteControl is the infrastructure of remote controls. Its main responsibility is to parse the text format of command from the user interface based on the steering protocol with and send the parsed information to **RemoteControlManager**.

RemoteControlManager of the **Brew** subsystem uses Java reflection APIs [92] to identity, locate, and invoke the exported classes, objects, and the methods. As an example shown in fig. 5.4, one static class **IDCT** and three objects with id being 1, 2, 3 have been exported. The method of the exported static class may be invoked by directly using the class name with the method name, such as "**IDCT.setResolution(5)**." The method of the exported object has to be invoked through an id number,

since there may be more than one object instances of the same class. A special array, `__exportedObj`, is used to locate the exported object. For example, the method, `setFrameDropRate`, of the object 3 may be invoked by the command "`__exportedObj[3].setFrameDropRate(0.9)`." The arrow on the right side of fig. 5.3 shows the data path for sending the commands to invoke the remote methods. Furthermore, since the remote method invocation commands are text formatted, the application may be steered through a script. The scripting steering enables a complicated method to be constructed from several primitive method invocations, and time-driven remote method invocations, i.e., a remote method may be invoked at a specified time. These two features may be very useful for testing and debugging remote real-time software. For example, it is possible to write a test suite as a script. People may check how the real-time software responds after some events by simulating the events as method invocations at a specified time. For example, the following script segment will invoke the method, `ping`, of the exported object 1 for 100 times. Here the keyword, `least`, specifies that each loop will wait for at least 500 ms.

```
I = 0;
while (!(I == 100)) {
  least(500) {
    __exportedObj[1].ping();
  }
  ++I;
}
```

5.3 Component Reassembly

5.3.1 Overview

Computation steering may only change the execution performance and quality by sending commands with appropriate parameters. There are some limitations such as some components may not be parameterized or the environmental change so much that simply changing the parameters is not enough. In these cases, a more radical adaptation must be invoked. That is, the execution of the applications will be temporarily suspended. The component assembly will be invoked to search for more appropriate implementation of components, reassemble the application from the new components, and then resume execution. The whole process is called *component reassembly.* The main difference between component assembly and component reassembly is the synchronization between old and new implementations of components. Particularly, component reassembly needs to ensure that the applications will produce the same results with the newly assembled implementations of the components after component reassembly.

A straightforward approach for implementing component reassembly is to extend assembly to reassembly by using a separate constraint monitoring thread to determine the run-time computing environmental changes by examining if any one of the constraints fails, and re-invoke the assembly process whenever necessary. Unlike the component assembly that only execute once before the execution and has no performance impact on the run-time performance of the applications, one of the challenges for reassembly is performance, since to re-invoke assembly process involves I/O activ-

ities, such as communication between component registry, and intense computation, such as constraints solving to find the feasible combination. Reassembly may even fail on a small temporal scale of environmental change, since the environment may have changed before reassembly finishes.

In our experiments illustrated in the next chapter, the assembly process of the robot application is about 650 times slower than the similar application hard-coded by if-else condition statements, an simple alternative adaptation approach discussed in chapter 3. Therefore, it will be not feasible to simply re-invoke the assembly process for the reassembly.

In the next two subsections, we will discuss the performance of assembly and propose several approaches to improve the performance. The experiment results on the performance improvements will be illustrated in the next chapter.

5.3.2 Performance Analysis of Component Assembly

As discussed in chapter 3, component assembly may be divided into two stages:

1. **Software constraint sets building**: In this step, the assembly process needs to resolve the software hierarchy, i.e., to know the components that the application has and the implementations of each component. As a consequence, all possible combinations of the component implementations will be known, and then all software constraints will be built by adding the embedded constraints, parameters and connectors, from each involved component implementations.

2. **Constraints solving**: In this step, the assembly process will find which software constraints are feasible by checking the truth of each included constraints within the specified parameter domains. As mentioned in chapter 3, **FRAME** uses a backtracking algorithm for solving constraint satisfaction problems, which use multiple nested loops for parameters, which take each values in the specified parameter domains and examine if the software constraint is satisfied.

The performance for software constraint sets building, denoted as P_b, depends on the number of software constraint sets and the number of constraints needed to be added to each set, i.e., the summation of constraints embedded in each involved component implementation. With the backtracking constraints solving algorithm, the performance for constraints solving, denoted as P_s, depends on, similar to P_b, the number of constraints sets and the number of constraints in each set, and it also is proportional to the range of each parameter domain. That is,

$$P_b \propto \sum_{i}^{\text{all possible sets}} |S_i| \tag{5.1}$$

and

$$P_s \propto \left(\sum_{i}^{\text{all possible sets}} |S_i| \right) \times \left(\prod_{i}^{\text{all possible parameters}} |D_i| \right) \tag{5.2}$$

where S_i is constraints set i, D_i is the domain of parameter i. Mackworth, et al. has a paper in the area of the complexity of constraint satisfaction problems [204].

5.3.3 Performance Improvement

The basic idea to improve the performance of reassembly is to avoid unnecessary communication and computation. There are two approaches, partial reassembly and caching.

First, we observe that not all component implementations will become inappropriate after the computing environmental change and it is unnecessary to examine the constraints of these components. Take an automobile as a real-world analogous example, although the tires of the automobile may have different safety performance during weather changes, many other parts of the automobile, such as body or seats, are not under any influence of the weather. Thus, it is not necessary to pay attention to these parts.

As shown in fig. 5.5, one or more subtrees of the software architecture may be specified for partial reassembly. A monitor thread is associated with each reassembly subtree to check if all the constraints of the subtree are satisfied. Developers may also specify only the subset of all the constraints contained in the subtree to be examined to reduce the run-time performance impact by the monitors. The reassembly process will start when some of monitored constraints fail.

The other performance improvement is to use cache, which may be done in two different levels. The first level is to cache the software constraints set building, i.e., the first stage of component assembly. Since the purpose of the software constraints set building is to build all possible software constraints from all component implementations. If no component implementation is added or removed, the possible software

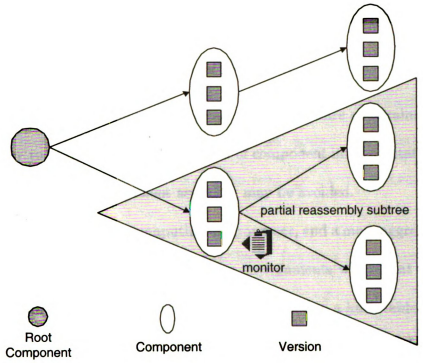

Figure 5.5: Software architecture

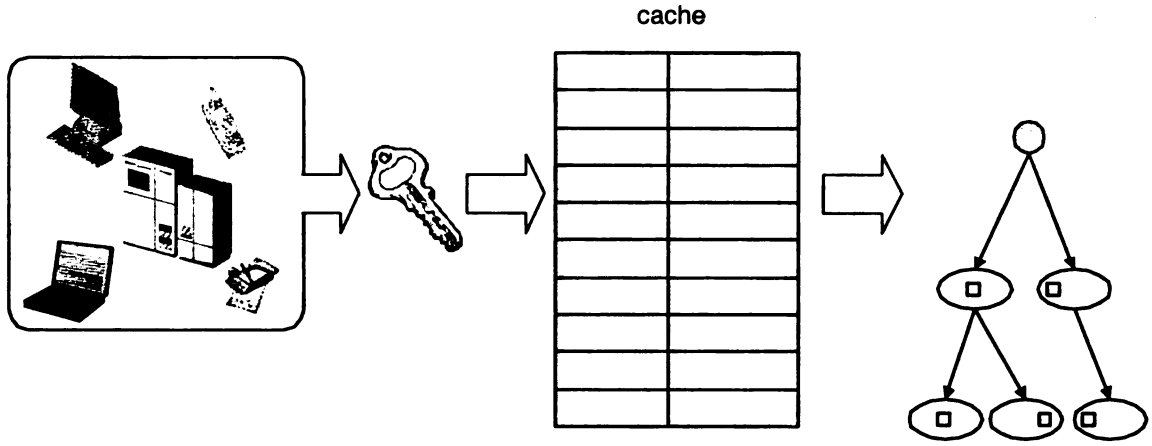

Figure 5.6: Flow of reassembly cache

constraints set will remain the same. By caching the software constraints set, it will be unnecessary to re-build the set if no status of component implementations change, and the first stage of the component assembly may be avoided.

The second level is to cache the computing environment, and a more aggressive scheme based on the assumption that the computing environments will repeat again. Currently, this caching scheme is key-value and implemented in a hash table as shown in fig. 5.6. The key will be unique for the computing environment; that is, developers have to provide a mapping that will convert a computing environment into a unique key. The value associated with the key will be the feasible components under the computing environment. To improve the probability of cache hit, the cache table may be stored in persistence storage, and then used or shared for subsequence execution of the same application.

5.4 Summary

A design challenge for the on-line adaptation is to make the trade-off between quality of execution and performance more promising. **ASAP** uses two different on-line adaptation schemes, **Brew** and component reassembly, to avoid possible over-reacting problems, such as major responses to minor environmental changes, which may simply waste the computing resources.

For a small scale of environmental changes, people may use **Brew** to steer and repair the affected applications that do not work properly under the new environment without terminating the applications. For a large scale of environmental change, the execution of the applications will be temporarily suspended, component reassembly will be invoked, and then resume the execution with appropriate implementations of the components.

Moreover, under some conditions, it is possible to improve the performance of reassembly by avoiding unnecessary communication and computation. Two different approaches, partial reassembly and caching, are also presented in this chapter.

Chapter 6

Application Demonstration and Performance Evaluation

6.1 Overview

There are three application demonstrations from different domains in this chapter, and each will demonstrate different features of **ASAP**. The GO game that has been used in the previous chapters as an example to describe **FRAME** and **CSML** will demonstrate the basic features of **ASAP**, including distributed component assembly. The second application is a soft real-time application, the Java MPEG Player. The Java MPEG Player has certain time constraints, which may not be satisfied under different environments. We will demonstrate that the unsatisfied constraints may be repaired by **Brew**. The last application is a self-adaptive robotic control system that control modules may be replaced by component reassembly. Different performance results will also be presented and discussed.

6.2 The GO Game

6.2.1 Application Description

In this section, we describe a demonstration of **FRAME** using the GO game described. The component **board** is inherited from JavaGO, a Java Applet that is originally developed by Alain Papazoglou [205]. **Board** provides the graphical user interface and game rules checking. Users may play against computer implemented as component **AI** with two different level of competition.

The first implementation of the component **AI**, **GnuGO**, is inherited from GnuGO source code developed in C [195] and is compiled as a native shared library. A Java wrapper class is developed to access the native library. The second implementation, **Random**, is written in Java with very limited intelligence.

6.2.2 The First Experiment: the Different Implementation of AI

The first experiment demonstrates that different memory resources lead to different imlementations of **AI** by component assembly. The experimental platform is a Linux PCs with 500 MHz Celeron Processor and 128 MB RAM. In order for the application to use each version of the **AI** component, we would execute VM with two different pre-allocated amounts of stack.

The **GnuGO** implementation is competitive and actually played well in several computer GO tournaments [195]. It has many recursive function invocations. Hence, the

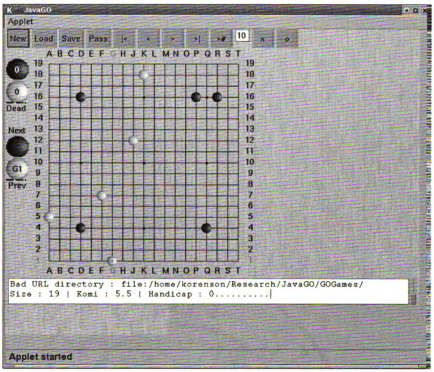

Figure 6.1: GO game with **Random** implementation

GnuGO implementation requires much stack allocation. Memory becomes a constraint for **GnuGO**. In fact, the default pre-allocated amount of stack of VM was not large enough to meet the memory size constraint. The **Random** implementation of component **AI** was used. Fig. 6.1 is the screen shot of the GO game played with the **Random** implementation; here, human played black and computer played white. A typical strategy to develop a GO game begins playing around the corners, then the edges, and finally the center. Although less memory resource is required, fig. 6.1 shows that **Random** implementation is not competitive since computer made several bad moves by playing at the edge, even the center during the opening of game.

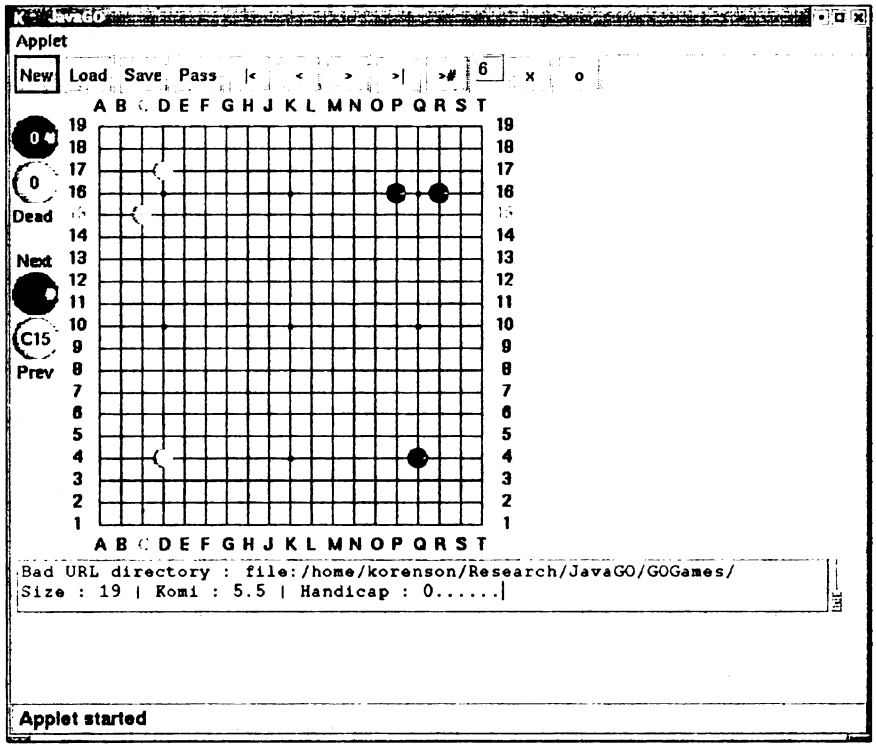

Figure 6.2: GO game with **GnuGO** implementation

We tested the same application on a VM with increased pre-allocated amount of stack. With the enough pre-allocated amount of stack, the memory size constraint was met and the **GnuGO** implementation was used.

Fig. 6.2 shows the first several moves of the GO game with **GnuGO** implementation. The screen shot shows that computer developed the game around the corners and demonstrates that **GnuGO** implementation has enough intelligence to realize the typical GO game strategy.

6.2.3 The Second Experiment: Hetergeneous Multiple Platforms

The second experiment evaluates the GO game under **FRAME** on a hetergeneous computing environment with different computational power. The experimental platforms include a Compaq iPAQ H3670, which is a small powerful handheld computer based on the Intel StrongARM 32-bit RISC processor running at 206 MHz with 16 MB ROM and 64 MB RAM, and a desktop with a 900 MHz Pentium III processor and 256 MB RAM.

Fig. 6.3 compares the response time of the first several moves on the iPAQ and the desktop. The response time is no more than 30 seconds on the desktop, but may exceed 400 seconds on the iPAQ. Fig. 6.4 compares the CPU load of the first several moves with both the **board** component and the **GnuGO AI** component on the iPAQ, and the **board** component on the iPAQ and the **GnuGO AI** component on the desktop. Most of time the CPU is busy for the first case, the **GnuGO AI** component on the iPAQ, but idle for the second case. Power is a scarce resource on a handheld computer and high CPU utilization consumes much battery power [206]. Together with a longer response time, it is probably not possible to finish a game on the iPAQ with the **GnuGO AI** component on the same iPAQ. Based on our experiments, the remaining power after ten moves is only 76% for the first case and 98% for the second case.

Because of limited computation power of the iPAQ, it may be frustrating and impractical to execute the entire GO game with the **GnuGO AI** component on the iPAQ. Under **FRAME**, if users specify one target platform, the less competitive **AI** will be

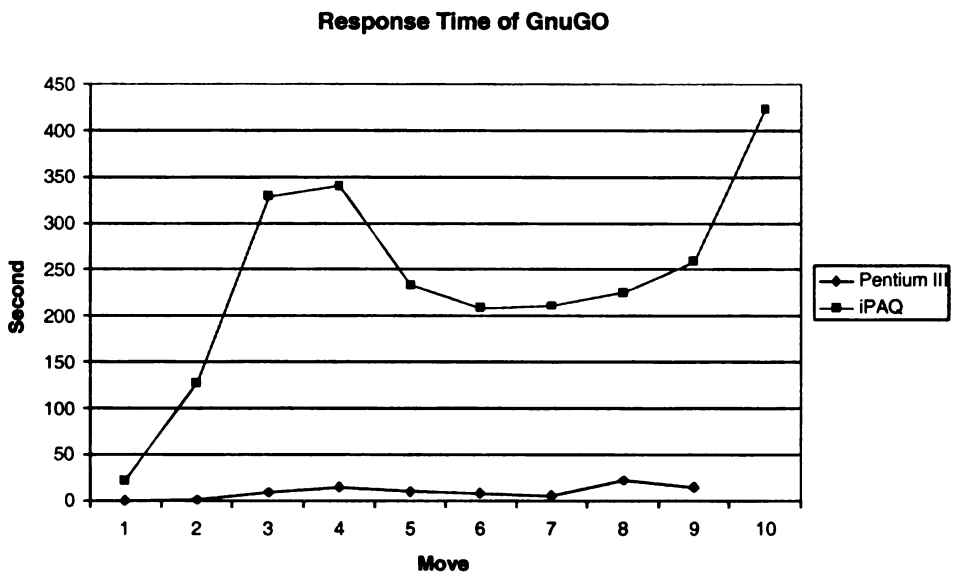

Figure 6.3: Response time of the GnuGO AI engine

used since the response time constraint of `GnuGO` version is not satisfied on the iPAQ. Alternatively, users may specify multiple target platforms to distribute the `GnuGO` version of `AI` component to a remote machine.

6.3 MPEG Video Player

6.3.1 Introduction

With the advances in hardware technology, the demands increase for soft real-time capable software on various platforms. Such demands include communication software for cellular phone and video playback on PDAs. Traditionally, real-time software developers make design decisions based on the best technical solution with full knowl-

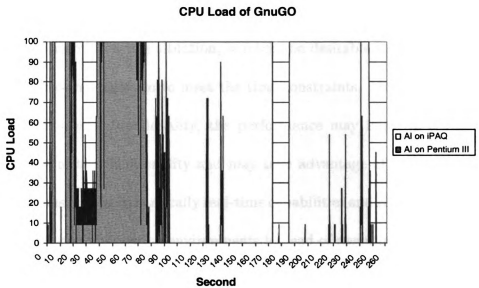

Figure 6.4: CPU load of the GnuGO AI engine

edge of resource capability of target platforms. Through human effort and experimentation, developers try to minimize cost while maximize performance and compliance with real-time constraints. With the tendency of computing environments moving toward diversity, such a development methodology for porting software to each platform is time and cost consuming.

While Java's promise of "Write Once Run Anywhere" might greatly alleviate the difficulty of software portability, it does not apply in the real-time domain because it is impossible to predict the resource capability of so many target platforms during the development stage. Since the correctness of the real-time software depends not only on the logical result of the computation but also on the time at which the results are produced, two platforms with different capability may produce the results at different

times. For instance, the same application may run perfectly on a high-end machine but fail on a low-end machine because the results it produces could not meet the real-time constraints. In this situation, it might be desirable to lower the quality of output on a low-end platform to meet the time constraints. Moreover, if a platform provides some special functionality, the performance may improve if the software has knowledge of the functionality and may take advantage of it. Therefore, many applications must probe dynamically real-time capabilities and reconfigure to different computing and communication environments instead of assuming a priori knowledge of the capabilities of the target platform. Criteria for probing real-time capabilities are specified as constraints during the development stage and the applications are built "on the fly."

Changes in computing environments make application behavior even more unpredictable. For instance, if two or more simultaneous executing applications compete for resources, it is impossible to guarantee that the time constraints of real-time applications will be satisfied without support from the OS. For a soft real-time application, unsatisfied time constraints might not be critical, so it is preferred to repair instead of terminating the application.

In the following subsections, we use a soft real-time application, the Java MPEG video player, as an example to demonstrate how **FRAME** and **Brew** may be used to realize the performance portability.

6.3.2 Application Description

The Java MPEG player, originally developed by Joerg Anders [207], is a MPEG1 [208] video player. These kinds of multimedia applications usually have time constraints that applications need to accomplish some tasks before the deadline for reasonable performance. For example, a video player may need to decode and display a image frame every four seconds to be able to play the video smoothly at the rate 15 frames per second.

To apply **ASAP** to the Java MPEG player, we decomposed the MPEG player into the component hierarchy as fig. 6.5 that consists of four components. The component **player** is the root component that needs service from its child component, **displayer**, which in turn has two child components, **decoder** and **RTOS**. The **displayer** has two different implementations, **on-the-fly** and **buffer**. The former implementation will display a frame when there is a frame decoded by the **decoder**. The latter implementation will buffer the decoded frames first, and then display them later. This implementation is suitable for a slower machine with a large amount of memory. The component **RTOS** has two different implementations, **timesys** and **dummy**. The **timesys** implementation can only be used on the TimeSys real-time Linux platform[209][1]. It provides a CPU reservation service to guarantee quality of service provided by the **displayer** and the **decoder**. If the underlying OS is not a TimeSys real-time Linux, the **dummy** implementation will be used, which does not have any performance effect

[1]Several real-time Linux operating systems are available. We chose the TimeSys real-time Linux operating system for this example.

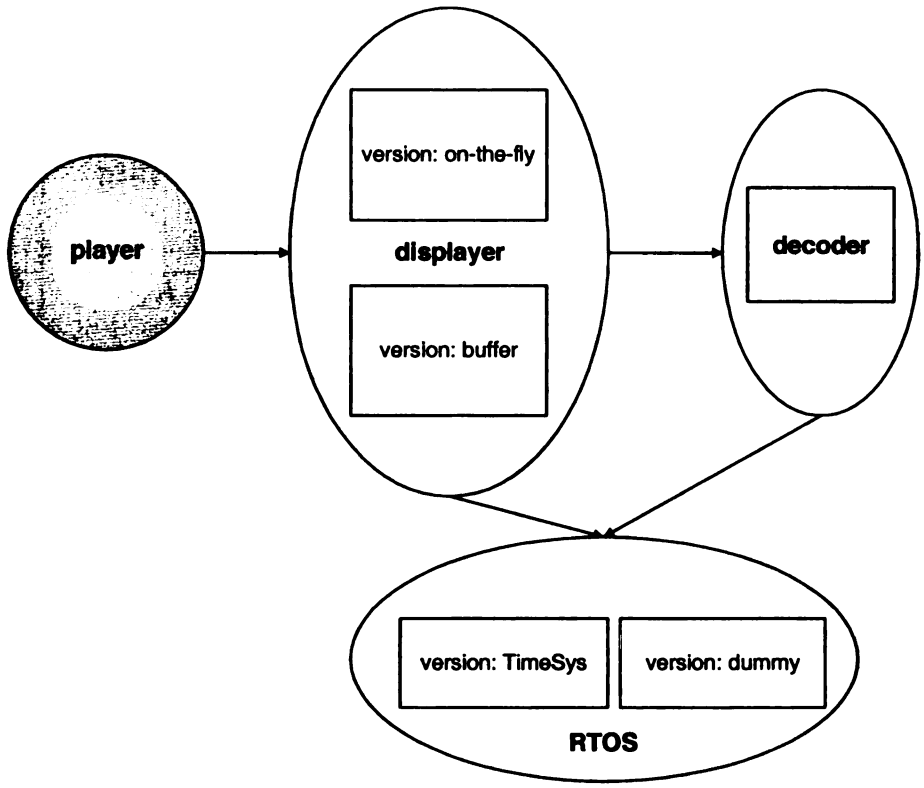

Figure 6.5: Software hierarchy of an MPEG video player

on the `displayer` and the `decoder`. To simplify the example, each implementation of each component is labelled according to table 6.1.

We also made some modifications such that it may display the frame periodically. That is, if a frame is decoded within a period, it will be displayed at the end of the period; if not, the displaying will be delayed to the end of the next period. We also embedded some code in the application so that it could be instrumented and steered remotely by `Brew`.

	component 1	component 2	component 3	component 4
version 1	**player**	**displayer** with **on-the-fly** implementation	**decoder**	**RTOS** with **timesys** implementation
version 2	N/A	**displayer** with **buffer** implementation	N/A	**RTOS** with **dummy** implementation

Table 6.1: Label of each component

parameter	comment
p_1^1	number of frames played per second
p_1^2	image quality

Table 6.2: Complete parameter list for component **player**

constraint	**player** $(i = 1, j = 1)$
R_{ij}	$\emptyset$
P_{ij}	$\{\text{“}5 \leq p_1^1 \leq 30\text{”}, \text{“}1 \leq p_1^2 \leq 8\text{”}\}$
D_{ij}	$\{\text{“}7p_1^1 + 25p_1^2 \leq 235\text{”}\}$
M_{ij}	$\{\text{“}p_1^1 = \frac{1000}{p_2^1}\text{”}, \text{“}p_1^2 = p_2^2\text{”}\}$

Table 6.3: Complete constraint list for component **player**

6.3.3 Constraints

Table 6.2 and 6.3 list the complete parameters and constraints of the component **player**. For users or developers of a MPEG video player, how well a video player is executing may justified by the number of frames played per second and image quality. Hence, these two numbers may be used as the parameters of the root component **player** for users to specify, and are denoted as p_1^1 and p_1^2 within the domains $[5, 30]$ and $[1, 8]$ as P_{ij} respectively. Here the image quality is divided into eight different levels. The internal connector D_{ij} specifies the linear relation between p_1^1 and $p1^2$.

parameter	comment
p_2^1	time interval, in milliseconds, between frames displayed
p_2^2	image quality

Table 6.4: Complete parameter list for component **displayer**

Table 6.4 list the complete parameters of the component **displayer**. The parameters in which developers may be interested are the time interval, in milliseconds, between frames displayed, p_2^1, and the image quality, p_2^2. The number of frames played per second and the time interval between frames played are actually inverse proportional to each other. The external connector, D_{ij}, in table 6.3 simply specifies that the inverse relation between p_1^1 and p_2^1 and also both **player** and **displayer** use same metric for image quality.

Table 6.5 list the constraints of the component **displayer**. The parameter constraints, P_{ij}, specifies that the time interval between frames displayed might be within 40 milliseconds and 200 milliseconds, and the image quality would be between 1 and

8. For internal connector, D_{ij}, it is obvious that a better image quality, p_2^2, will require a longer time interval, p_2^1, to display a frame. Their proportional relation may be modeled as a linear relation, $7p_2^1 - 160p_2^2 \geq 120$ and specfied in D_{ij}.

Table 6.6 list the complete parameters of the component **decoder**. The parameters in which developers may be interested are the time interval, in milliseconds, between frames decoded, p_3^1, and the image quality, p_3^2. Similarly, both the **displayer** and the **decoder** may use the same metric for the image quality, i.e., $p_2^2 = p_3^2$ specified in the external connector, M_{ij} in table 6.5.

However, the time to decode a frame is different than the time to display a frame. For the **on-the-fly** implementation of the **displayer**, a frame is displayed when a frame is decoded by the **decoder**. Hence the time to decode a frame is actually only a fraction of the time to display a frame, and the relation between these two time intervals may be modeled as $p_2^1 \leq p_3^1 + 70$; that is, once a frame is decoded, the **displayer** may need extra time, no more than 70 milliseconds, to display the frame. Thus, external constraint connectors between the parameters of the **displayer** with

constraint	**displayer**	
	on-the-fly $(i = 2, j = 1)$	**bufer** $(i = 2, j = 2)$
R_{ij}	$\emptyset$	$\emptyset$
P_{ij}	{"$40 \leq p_2^1 \leq 200$", "$1 \leq p_2^2 \leq 8$"}	{"$40 \leq p_2^1 \leq 200$", "$1 \leq p_2^2 \leq 8$"}
D_{ij}	{"$7p_2^1 - 160p_2^2 \geq 120$"}	{"$7p_2^1 - 160p_2^2 \geq 120$"}
M_{ij}	{"$p_2^1 \leq p_3^1 + 70$", "$p_2^2 = p_3^2$"}	{"$190 \leq p_3^1 \leq 1000$", "$p_2^2 = p_3^2$"}

Table 6.5: Complete constraint list for component **displayer**

the **on-the-fly** implementation and the **decoder** is {"$p_2^1 \leq p_3^1 + 70$", "$p_2^2 = p_3^2$"} in table 6.5.

On the other hand, for the **buffer** implementation of the **displayer**, the decoded frames are buffered first and then displayed later together. These two time intervals, p_2^1 and p_3^1, are irrelevant. However, **displayer** developers may specify a constraint whether the **decoder** may decode frames in some specific time interval, such as $190 \leq p_3^1 \leq 1000$. The constraint reflects the fact the decoding time is not related to the displaying time, and we would like the **decoder** that can decode a frame within 1000 milliseconds. If decoding speed is too fast, i.e., $p_{31} < 190$, then we would not prefer to use this version. Thus, external constraint connectors for the **buffer** implementation of the **displayer** is {"$190 \leq p_3^1 \leq 1000$", "$p_2^2 = p_3^2$"} in table 6.5.

Table 6.7 and 6.8 list the complete constraints of the component **decoder** and **RTOS**. The constraints of **decoder** is simliar to the ones of displayers, except **decoder** has no external connectors since it does not have any child components.

RTOS only specifies the required resource for different implementations. For example, the required resource for the **timesys** implementation is TimeSys real-time Linux OS. Therefore, its resource requirement would be {"OS is a TimeSys real-time Linux."}. Instead of specifying the resource requirement, the **dummy** implementation is specified

parameter	comment
p_3^1	time interval, in milliseconds, between frames decoded
p_3^2	image quality

Table 6.6: Complete parameter list for component **decoder**

as the "default" implementation; that is, if the required resources of other implementations are not satisfied, the default implementation will be used.

All the software hierarchy, parameters, and constraints are specified in **CSML** listed in appendix A.

The implementation of the **displayer** component that will be used depends on which time constraint, $p_2^1 \leq p_3^1 + 70$ or $190 \leq p_3^1 \leq 1000$, is satisfied. If decoding speed is fast enough, the feasible program will use the **on-the-fly** version because the first constraint would be satisfied. The first two experiments will demonstrate that the different implementations of displayer are chosen based on the time constraint and, if unsatisfied during the run-time, may be repaired by **Brew**. The last experiment demonstrates that **FRAME** will use the special functionality of the TimeSys real-time Linux.

6.3.4 MPEG player on JIT engine

The first experiment was conducted on a Linux PC with 900 MHz Pentium III Processor, 256 MB RAM, and the Java Virtual Machine (JVM) with just-in-time (JIT)

constraint	**decoder** $(i = 3, j = 1)$
R_{ij}	$\emptyset$
P_{ij}	$\{\text{“}40 \leq p_3^1 \leq 200\text{”}, \text{“}1 \leq p_3^2 \leq 8\text{”}\}$
D_{ij}	$\{\text{“}7p_3^1 - 160p_3^2 \geq 120\text{”}\}$
M_{ij}	$\emptyset$

Table 6.7: Complete constraint list for component **decoder**

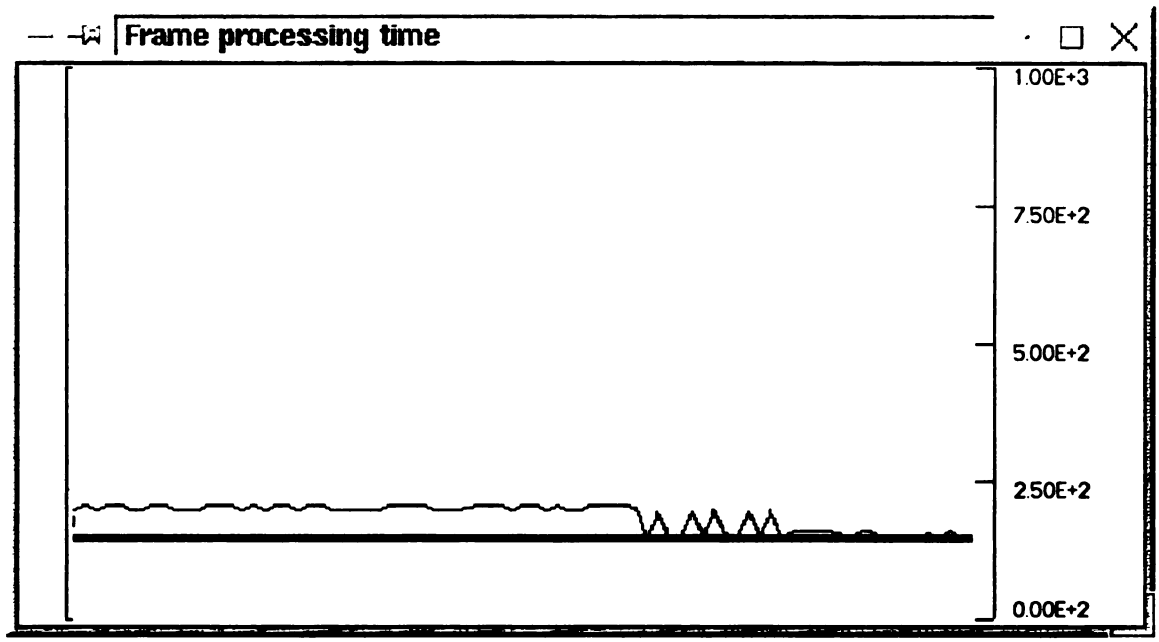

Figure 6.6: Frames processing time of the MPEG player on Linux PC

engine [210, 211]. The instrumented metric is the time to display a frame. The steered parameters are the number of frames played per second and the quality of the image. The JIT will take the bytecodes and compile them into native code for the machine that Java applications are running on. It can actually be faster to grab the bytecodes, compile them, and run the resulting executable than it is to just interpret them. As a consequence, with JIT optimization, Java VM has the ability to decode

constraint	RTOS	
	TimeSys $(i = 4, j = 1)$	**dummy** $(i = 4, j = 2)$
R_{ij}	{"OS is a TimeSys real-time Linux."}	default, i.e., R^c_{41}
P_{ij}	$\emptyset$	$\emptyset$
D_{ij}	$\emptyset$	$\emptyset$
M_{ij}	$\emptyset$	$\emptyset$

Table 6.8: Complete constraint list for component RTOS

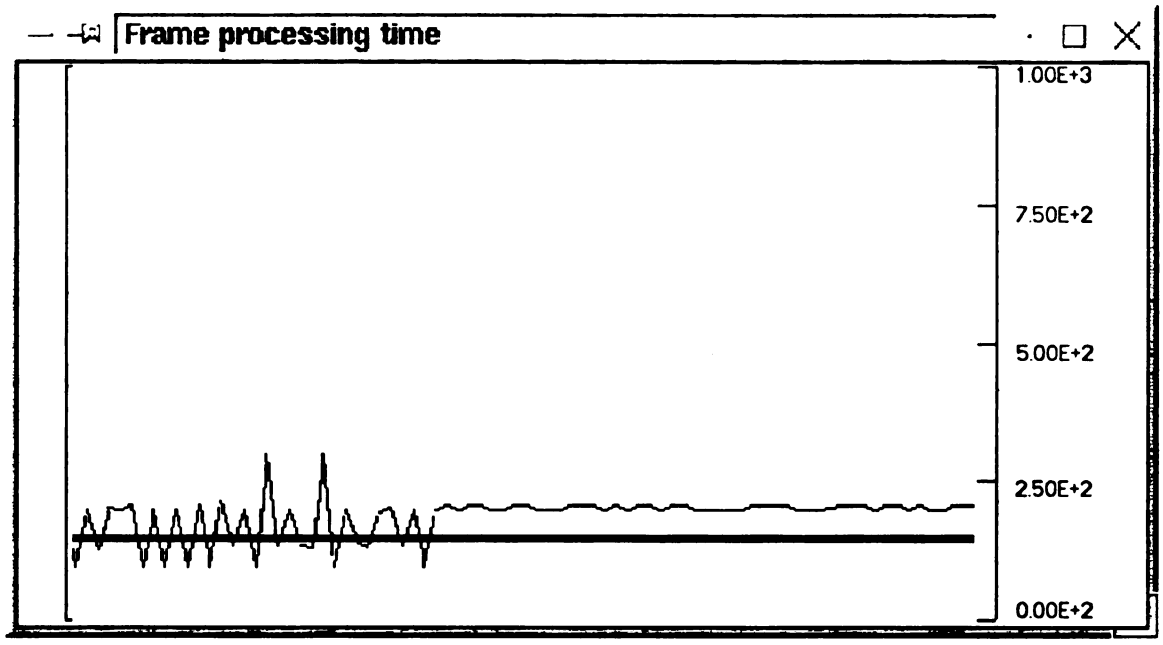

Figure 6.7: The MPEG player is steered to meet the time constraint

the frame fast enough so the time constraint $p_2^1 \leq p_3^1 + 70$ can be satisfied and the **on-the-fly** implementation of displayer component will be used. After the components are assembled, the MPEG player begins to execute with parameters that satisfy the software constraints. The value of parameter p_1^1, the number of frames played per second, is 5, which in turn gives the displaying period 200 ms. The value of parameter p_1^2, image quality, is 8. Fig. 6.6 is the screen shot of the VO for the Java MPEG player executing on a JVM with JIT engine.

The thinner line represents the time (in milliseconds) needed to display a frame. Initially, the displaying period is 200 ms. We used **Brew** to reduce image quality, which reduces the time to decode a frame, and therefore we can decrease the period to 150 ms. The variation of the thinner line reflects such a scenario.

In fig. 6.7, in which the MPEG player is interfered by other applications, the spikes of the thinner line shows that some frames may not be decoded within one period and displaying them needs to be delayed to the next period. Because of delay, it may take up to one period, 150 ms, to display these frames after they are decoded and the time constraint, "$p_2^1 \leq p_3^1 + 70$", is not satisfied, which requires that the extra time should be less than 70 ms. Thus, we increase the period to 200 ms to repair the time constraint.

6.3.5 MPEG player on interpreter engine

The second experiment was conducted on a Linux PC with 900 MHz Pentium III Processor, 256 MB RAM, and the Java Virtual Machine (JVM) with the interpreter engine that JVM simply interpret the bytecodes without compiling them into native code. The performance of the interpreter is usually slower than JVM with JIT engine, and actually is not able to decode the frame fast enough so the time constraint $p_2^1 \leq p_3^1 + 70$ can be satisfied. As shown in fig. 6.8, we execute the **on-the-fly** version. It requires about 800 ms (the thicker line) to decode and more than 1 second to display a frame. It does not satisfy the constraint $p_2^1 \leq p_3^1 + 70$ but $190 \leq p_3^1 \leq 1000$; thus it will load and execute the **buffer** implementation of the displayer component. Fig. 6.9 shows the time between each frame is reduced to 50 ms, i.e., 20 frames per second. Thus, the constraints, $5 \leq p_1^1 \leq 30$, can be satisfied.

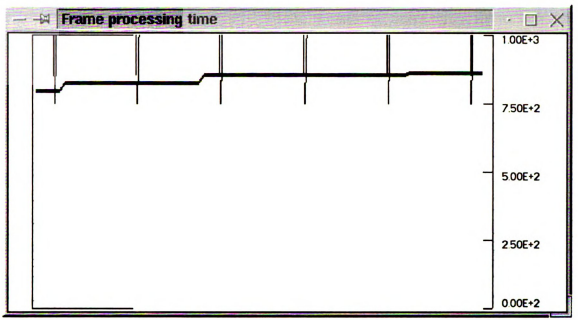

Figure 6.8: Frames processing time of the MPEG player on Java VM with JIT engine

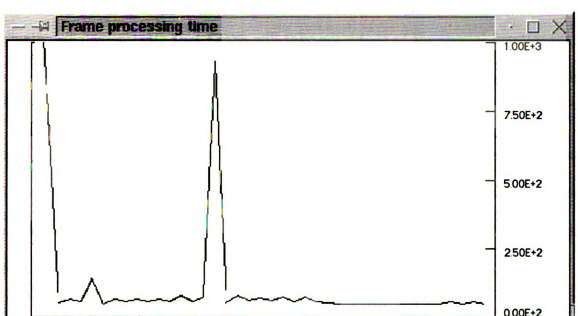

Figure 6.9: Frames processing time of the MPEG player on Java VM with interpreter engine

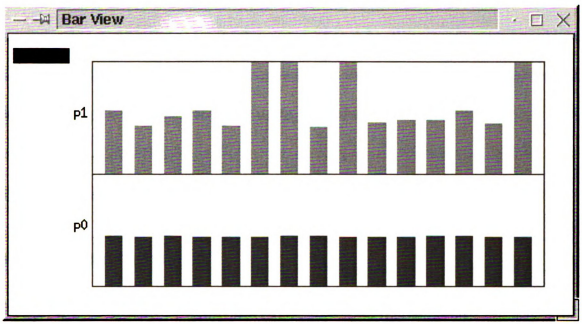

Figure 6.10: Time interval to display a frame for the MPEG player with and without CPU reservation

MPEG player on TimeSys real-time Linux

Finally, we demonstrate that **FRAME** will take advantage of the special functionality of computing environments and, thus, applications may provide better or guaranteed quality of service. For comparison, we had two MPEG players, $P0$ and $P1$, running on a 500MHz Celeron processor and 128 MB RAM PC with the real-time Linux from TimeSys simultaneously. $P0$ is normally executed under **FRAME** and the **timesys** version is used because the constraint, "OS is a TimeSys real-time Linux.", is satisfied. $P1$ is forced to use **dummy** version, so all the real-time features are disabled.

Fig. 6.10 shows the performance difference for the MPEG players with and without CPU reservation. The height of bars is the time between two consecutive frames. $P0$ has 40% CPU reserved (40 ms for every 100 ms). $P1$ does not have CPU reserved and

has to compete for the remaining 60% CPU with other applications. After launching several other applications, the performance impact by those applications could be significant. The time between two frames is still kept as the same (100 ms) for $P0$, but varies significantly and could be up to more than 200 ms for $P1$.

6.4 Self-Adaptive Robot

6.4.1 Introduction

Nowadays, robots may be used to handle hazard materials or explore distant unfriendly locations. Since the failure of these missions may cause disastrous consequences, the design of robust, fault-tolerant robots will become important. One approach for robustness is to design self-adaptive robots that control systems may diagnose, overcome, and adapt to adverse and rapidly changing conditions. For example, the Air Force Research Laboratory Information Directorate and IS Robotics of Somerville, Mass., used the firm's "Ariel 2" autonomous legged underwater vehicle to demonstrate the feasibility of software that automatically adjusts to failures and changes in a system [212], where the adaptive features of the robot include tolerance for sensor inadequacy, environment adaptation, tolerance to actuator failure, and goal-driven choice of behaviors.

Self-adaptive robotic systems are usually developed by using the behavior-based approach that does not require an accurate model of robots and environments [191]. Rather than attempting to model the world, the systems have multiple lower and

more critical level control modules that react directly to sensory information and then complicated behaviors may be achieved from these low level control modules. However, these control modules must be pre-defined during the development stage and probably pre-loaded before execution. Therefore, the control software developers need to predict all the challenges and conditions that the robotic systems may meet. However, sometimes it is impossible to make such a prediction and the robotic systems will not be able to handle the unexpected situations without terminating the robotic systems for adding the control modules for the unexpected situations. Although the control modules may be loaded during the run-time, the availability of the control modules still need to be known for module selection algorithms to select. Since module selection algorithms are usually hard-coded during the development stage, it is impossible to avoid the down-time for adding newly developed modules and updating the module selection algorithms. Furthermore, some situations are rare and the memory, which may be a scarce resource on robotic systems, for the control modules to handle those rare situations may not be well utilized. Thus, there needs to be a flexible approach for the adaptive robotic systems that will dynamically integrate new and select the appropriate control module to handle the situation they have ahead.

One of novel features of **ASAP** is that **ASAP** allows newly developed modules to be added and selected without the necessity of updating the module selection algorithms and therefore the down-time. Therefore, we illustrate how to use **ASAP** to implement the behavior-based approach to the adaptive robotic systems in this section, which will be realized by the reassembly process if each control module is envisioned as an

implementation of an abstract component and the situations it needs to handle are implemented as constraints.

Besides, the reassembly process may have a significant performance impact on the execution of the applications, since it needs to locate and load new implementations of components, and examine whether the constraints of the implementation are satisfied. Without any improvement, the assembly process does not apply well to the reassembly on small temporal scale of environmental change, i.e., relatively highly frequent computing environmental changes. Various performance evaluation, analysis, and comparison will be also illustrated in this section.

6.4.2 Application Description

Fig. 6.11 is the software hierarchy of a simple adaptive robotic control system that will retrieve obstacle information from sensors and then respond according to the type of obstacle ahead. Each low level control module is envisioned as an implementation of a component. The system consists of five components, and the component **robot** is the root component that needs service from its child components, **sensor** and **response**. The **sensor** component will collect the information about the obstacle ahead, determine the type of the obstacles, and send the information to **robot**. The **response** component will retrieve the obstacle information from **robot**. The response of robot are actually performed by the child components of **response**, **movement** and **other**. The **other** component handles the robot actions, except how the robot will move, which is handled by the **movement** component.

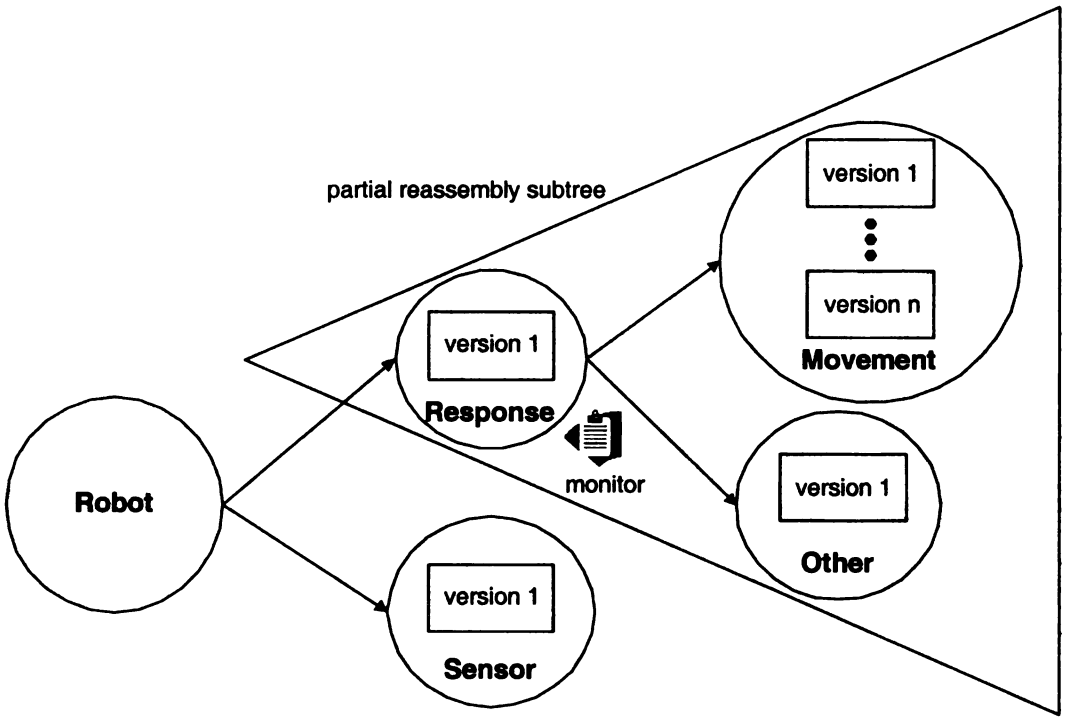

Figure 6.11: Software hierarchy of the simple robot application

In this section, we focus on the performance of the assembly, and measure how performance depends on the number of the implementations. Thus, we simplify the software hierarchy by limiting each component to one implementation except **response**, which may have up to twenty versions of implementations. Each implementation of the **movement** component performs some certain pre-defined movement for the obstacle, such as left circle move or right square move to pass the obstacle ahead.

6.4.3 Constraints

As illustrated in chapter 3, each implementation may contain a set of constraints that specify how well the implementation will perform or response under certain environ-

ment. For this simple robotic control system, one implementation of the **movement** component may move the robot in a left semi-circle pattern, so one of its constraints may be {"the obstacle ahead has free space at left side."}, which specifies such a movement implementation will perform an appropriate response for the obstacles that have free space at the left side. Therefore, when the robot encounters a circle obstacle, the assembly process may select such a movement to pass the obstacle.

The parameters specified in the first two examples are numerical metrics, such as the performance or output quality, so they may be confined in finite domains as parameter constraints and be used for solving constraint satisfaction problems by component assembly. However, the parameters are not necessarily limited to numerical metrics, but may be any metrics that can be enumerated or quantized. To apply the constraints to the applications that component implementations are not distinguished by numerical metrics but other concepts such as the situations need to handle, the concepts of parameters need to be extended by enumerating or quantizing the concepts. For the robot application, there may be twenty different kinds of obstacles ahead. Therefore, the constraint, {"the obstacle ahead has free space at the left side."}, of the left semi-circle implementation of the **movement** component may be specified as a parameter within a finite domain as {"$14 \leq p \leq 14$"}, where p is the parameter representing the obstacle ahead and 14 represents the obstacle ahead has free space at the left side.

6.4.4 Performance Evaluation

We implemented the simple adaptive robotic control system on a robot, XR4000 [213], to evaluate the performance of component reassembly. As shown in fig. 6.11, we only specify partial reassembly for the subtree rooted at the **response** component. Furthermore, there is a separate constraint monitoring thread automatically generated by CSML, which will determine the run-time environmental changes by examining if any one of the constraints fails and invoke reassembly process if necessary. Since the constraints are embedded into the implementation of components, the assembly process is able to bring in the newly developed implementation and examine its feasibility without down-time.

The only constraint to be monitored by the monitoring thread is the type of the obstacle ahead. Thus, whenever the robot encounter a obstacle, a reassembly process may be invoked depending on the type of obstacle ahead. Since only the **movement** component has multiple versions of implementations, what the reassembly does is actually select an appropriate implementation of the **movement** component to pass the obstacle. We compare the performance of component reassembly with and without caching, and also evaluate the performance of the similar application using hard-coded if-else condition statements, an alternative adaptation approach described in subsection 3.5.1. It needs to be noticed that, from formulae 5.1 and 5.2, performance is application dependent, and, therefore, the performance comparison or improvement may not be same for different applications.

Fig. 6.12 compares the time required to search for selecting an implementation of the **movement** component by the different scheme, i.e., non-cached reassembly, cached reassembly, and hard-coded if-else statement. The if-else scheme requires about $0.003 \sim 0.018$ ms that depends on the number of implementations. The non-cached reassembly requires about $2.1 \sim 12.1$ ms that also depends on the number of implementations, it is about 650 times slower than the if-else scheme as shown in fig. 6.13. Fig. 6.13 also reveals that the performance factor of the non-cached reassembly over the if-else scheme fluctuate between $600 \sim 700$ and center around 650. It shows that the non-cached reassembly algorithm is not harder than if-else algorithm in terms of computational complexity, which is $O(n)$ with n being the number of implementations, as shown in fig. 6.14 and 6.15.

The non-cached reassembly time is of the order of ms and actually is good compared to the robot movement time that is usually in the order of seconds. However, for highly frequent environmental changes, i.e., in the order of ms, it may have significant performance impact and even may get stuck in the reassembly process since another environmental change occurs before the reassembly finishes.

The result in fig. 6.15 shows that cached reassembly requires about 0.29 ms independent on the number of implementations and improves the reassembly speed by a factor of $7 \sim 40$, as shown in fig. 6.16, and may be only about 15 times slower than if-else scheme. Unlike if-else and non-cached scheme, the cache access time is constant and independent on the number of implementations. Thus, the performance improvement becomes more significant while the number of implementations increases. Also,

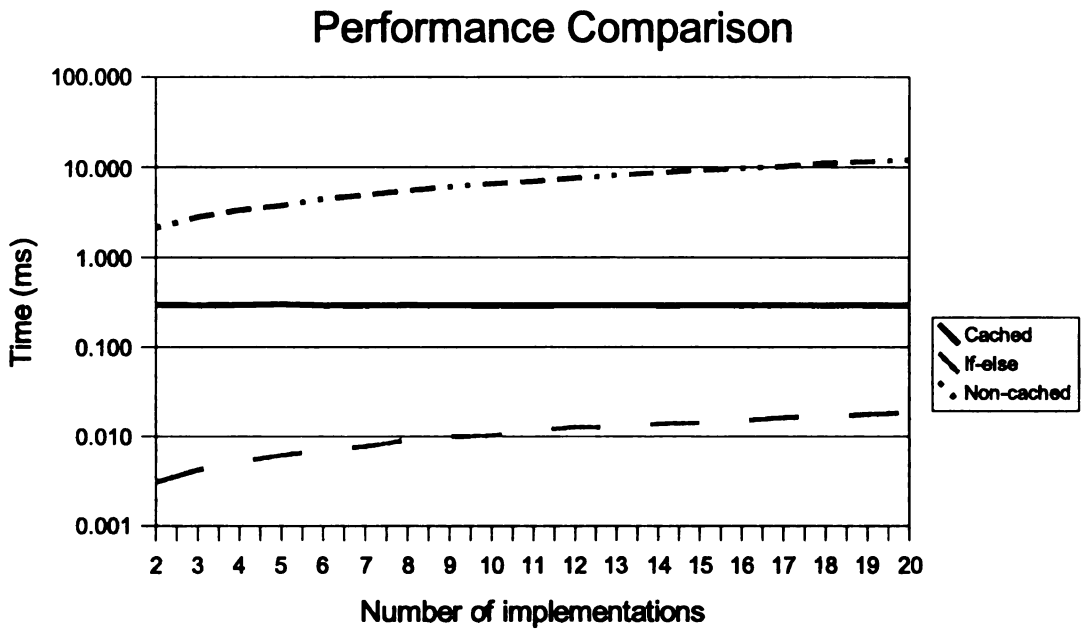

Figure 6.12: Performance comparison for different component selection scheme

the constant assembly time of the cache makes the execution time of the application more predictable, which is an important issue for real-time applications.

Subsection 5.3.2 illustrates that two steps are involved in assembly, or non-cached reassembly. Fig. 6.17 shows that the time required for the constraint solving, which is about 50% $\sim$ 60%. If the application structure does not change and no new implementation is added, the subsequence software constraints sets building may be avoided, and the non-cached reassembly performance may be approximately reduced to constraint solving step, which is a 40% $\sim$ 50% time saving in this case.

From formulae 5.2, the constraints solving performance, P_s, depends on the number of constraints sets and the number of constraints in each set, $\sum |S_i|$, and it also is proportional to the range of each parameter domain, $\prod |D_i|$. To improve the backtracking constraints solving algorithm, if more information may be extracted

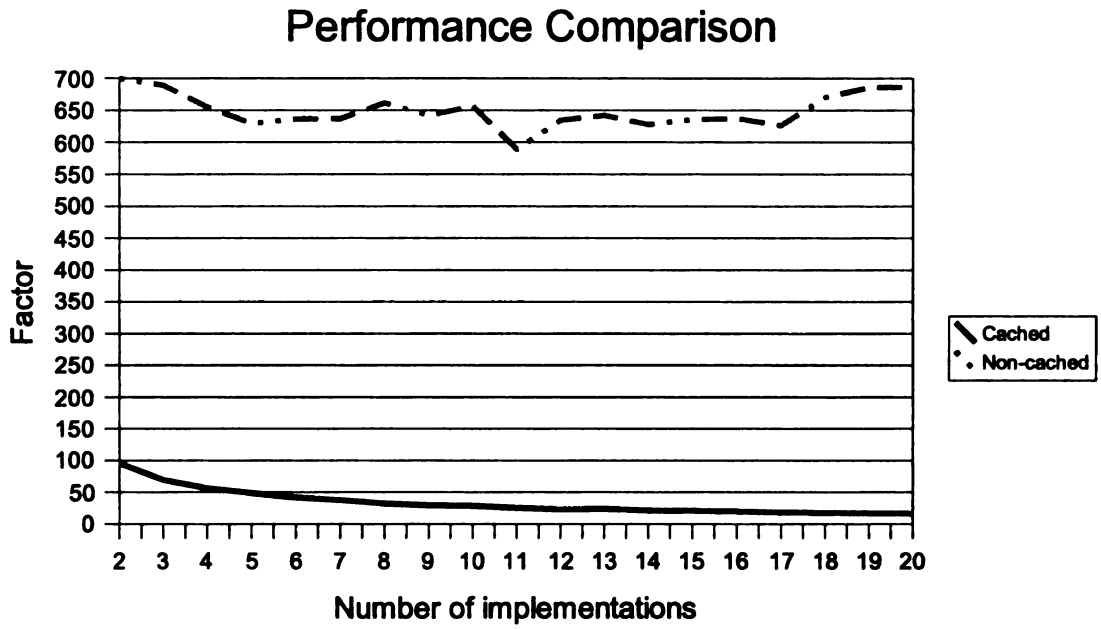

Figure 6.13: Performance factor over hard-coded if-else scheme for other different component selection scheme

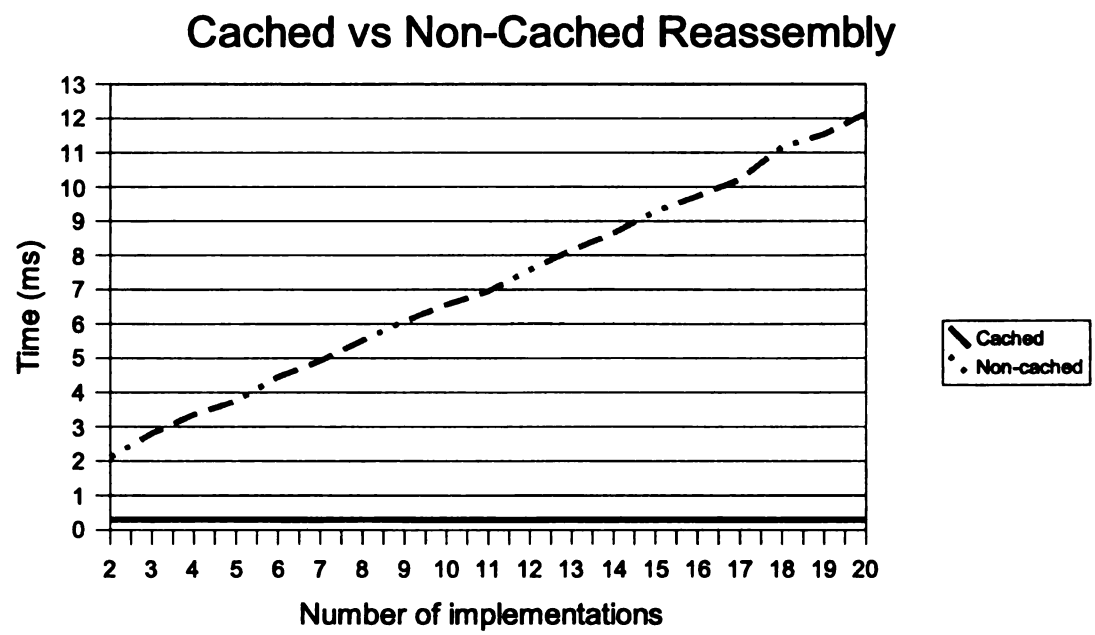

Figure 6.14: Performance comparison of cached reassembly and non-cached reassembly

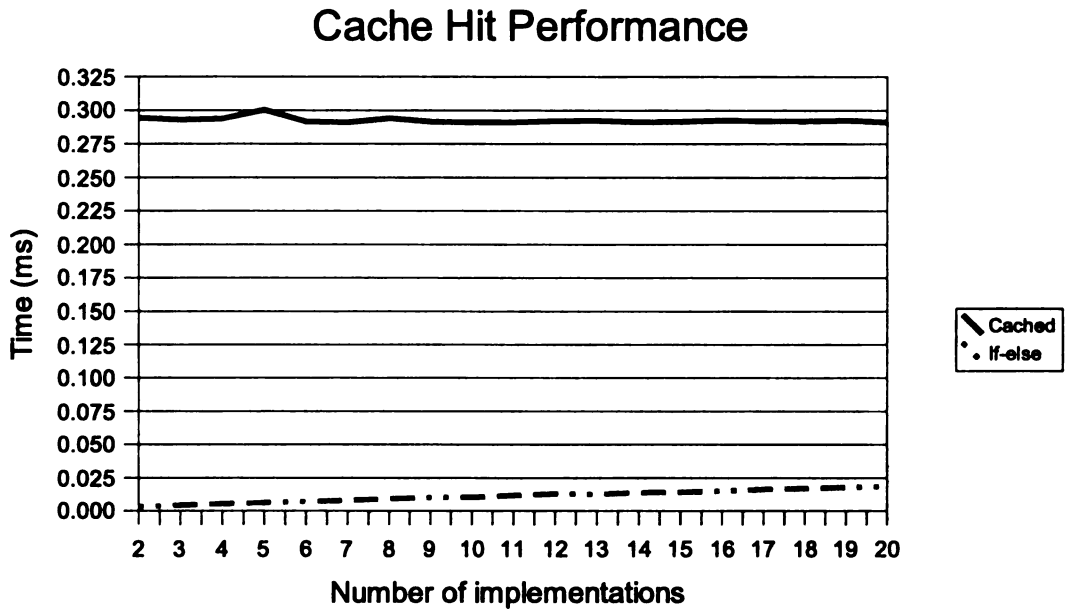

Figure 6.15: Performance comparison of cached reassembly and if-else scheme

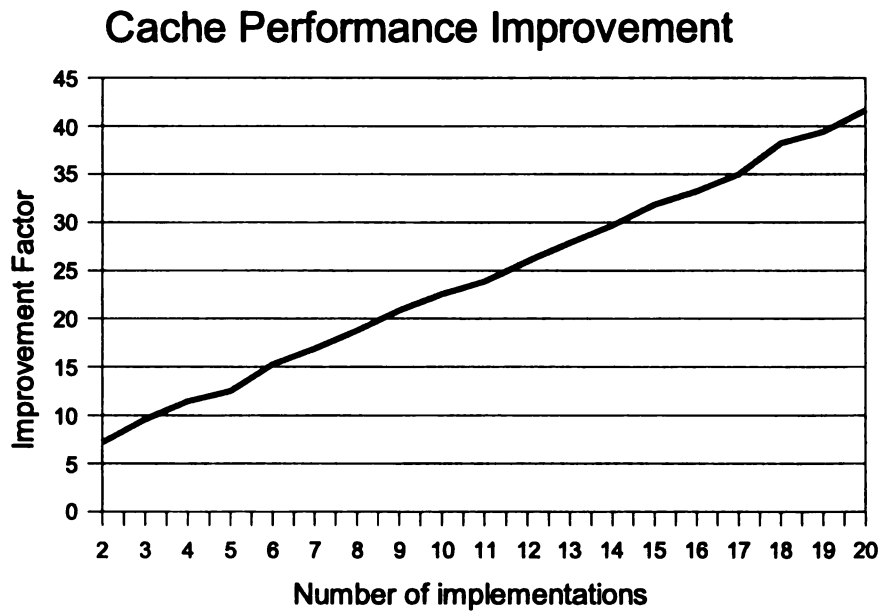

Figure 6.16: Performance improvement of cached reassembly over non-cached reassembly

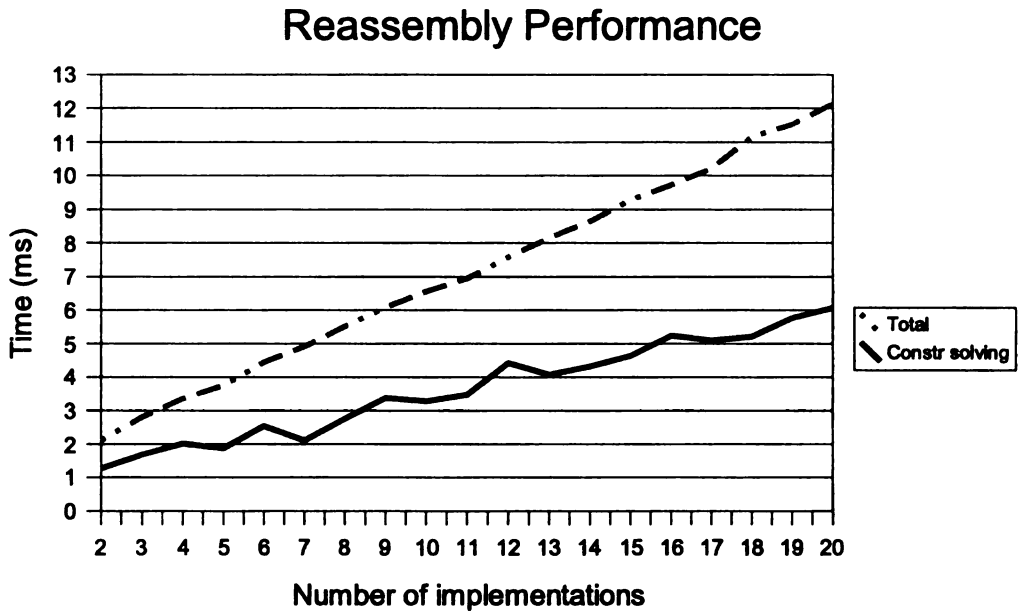

Figure 6.17: Constraints solving performance of reassembly

from the relationship between parameters, i.e., connectors, some redundancy may be found in the constraints sets. Thus, truth checking for some values in domains or some constraints may be avoided, which will reduce the terms $\prod |D_i|$ and $\sum |S_i|$ respectively.

Memory usage, a scarce resource in embedded systems, may be a challenge issue for developing software on embedded systems. Reassembly will load and unload the implementations of components whenever necessary, which will free memory. Depending on how the application is developed, the reassembly may use less memory than the hard-coded if-else scheme. For example, the robot application using hard-coded if-else statement has all implementations of the **movement** component preloaded to improve performance. However, this is a trade-off with memory usage. Fig. 6.18 shows that preloaded components require about 50% more memory than **ASAP**.

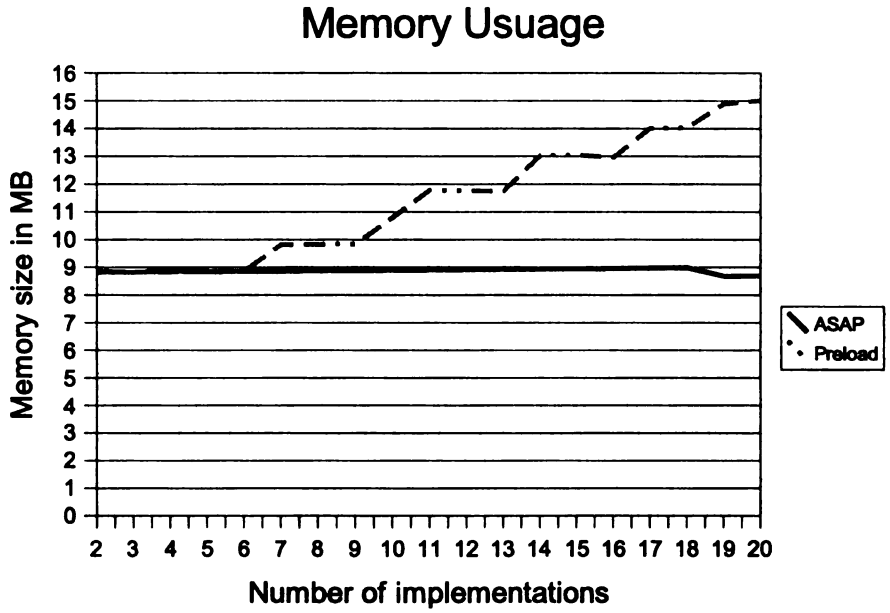

Figure 6.18: Memory usage comparison for **ASAP** and component-preloaded

6.5 Summary

Three different applications from different domains are used to demonstrate and evaluate various features of **ASAP**. The GO game demonstrates the basic features of **ASAP**, including distributed component assembly. The Java MPEG Player has certain time constraints, which may not be satisfied under different environments and may be repaired by `Brew`. The last application is a self-adaptive robotic control system that demonstrates how **ASAP** may be used to realize behavior-based approach for the adaptive robotic control systems, in which control modules may be replaced by component reassembly.

Different performance comparisons between non-cached reassembly, cached reassembly, and hard-coded if-else statement are also presented. The if-else scheme requires

about $0.003 \sim 0.018$ ms that depends on the number of implementations. The non-cached reassembly requires about $2.1 \sim 12.1$ ms that also depends on the number of implementations, it is about 650 times slower than the if-else scheme. On the other hand, cached reassembly requires about 0.29 ms independent on the number of implementations and improves the reassembly speed by a factor of $7 \sim 40$, and may be only about 15 times slower than if-else scheme. Unlike if-else and non-cached scheme, the cache access time is constant and independent on the number of implementations. Thus, the performance improvement is getting more significant while the number of implementations increases. Also, the constant assembly time of the cache makes execution time of the application more predictable, which is an important design requirement for some software such as real-time applications.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

As hardware technology improves and the deployment of wireless infrastructure increases, the computer systems are being more heterogeneous and dynamic. Heterogeneity and dynamic may raise several challenges for software developers and users. Such challenges are software portability, software or computing environments configuration, unexpected environments, and platforms with limited resources. A feasible approach for these challenges is to develop adaptive software that may be aware of environments and adapt themselves.

The adaptive software architecture project, or **ASAP**, is proposed and implemented for the problems and challenges under dynamic heterogeneous environments. Software may have a list of constraints to gather the information about the environments. Constraints are specified during the development stage. During the execution, software may check the list to find out if there are something wrong or change in the

environments, and then software will respond accordingly. However, specification of constraints raise challenges in complex and collaborative software systems development, since mutual dependency of constraints may leads to false conclusion. Several possible applications of adaptive software are presented, including applications that need environment-awareness, portability, robustness, and on-the-fly task assignment. One particular interesting application is also presented, i.e., ad hoc systems that are temporarily organized for accomplishing tasks.

The adaptive software framework, **FRAME**, of **ASAP** provides the APIs for software to check the constraint list and then adapt itself to the computing environments. The adaptation and functionality of software are separately specified and implemented under **FRAME**, and, hence, alleviate the complexity of development. Furthermore, the problems for mutual dependency of constraints are solved by using connectors.

To further alleviate the complexity of developing adaptive software under **FRAME**, a high-level adaptive software specification language, Component/Constraint Markup Language or **CSML**, may be used to specify the architecture of a software system and the constraints. The code for adaptation will be automatically generated by **CSML**. Then, developers only need to inherit the generated code to implement the functionality of software; basically, the infrastructure needed to work under **FRAME** is generated from the specification by **CSML**.

Adaptive software will definitely consume computing resources to detect and respond to the environmental changes during the run-time, and it will have a performance impact on the execution of software. To avoid unnecessary waste on computing resources, **ASAP** uses two different on-line adaptation schemes, **Brew** and component

reassembly, to let software respond to different scales of environmental changes. For a small scale of environmental change, people may use **Brew** to steer and repair the affected software that do not work properly under the new environment without terminating the software. For a large scale of environmental change, the execution of the software will be temporarily suspended, component reassembly will be invoked, and then the execution will resume with appropriate implementations of the components. To further reduce the performance impact of reassembly, two approaches, partial reassembly and caching, are proposed.

Three different applications from different domains are used to demonstrate and evaluate various features of **ASAP**. The GO game demonstrates the basic features of **ASAP**, including distributed component assembly. The Java MPEG Player has certain time constraints, which may not be satisfied under different environments and may be repaired by **Brew**. The last application is a self-adaptive robotic control system that demonstrates how **ASAP** may be used to realize behavior-based approach for the adaptive robotic control systems, in which control modules may be replaced by component reassembly. Different performance results are also presented and discussed. The results show that the cache may improve the reassembly speed by a factor of $7 \sim 40$ and the time for reassembly is constant and hence predictable, an important design requirement for real-time applications.

7.2 Future Work

7.2.1 FRAME

One possible extension of **ASAP** is to use multiple registries, instead of one global centralized registry, which may have location-dependent implementations that are specifically used within physical boundaries, such as the neighborhood of the registry. It requires that an application is able to dynamically locate and contact an appropriate registry, a research problem similar to service discovery, where clients generally have a priori specification of required services. Not only in the spatial domain, it is also possible to extend the registry in the time domain, where the implementation information stored in a registry may be changed over time. Thus, people may design a task flow plan that will assign certain tasks to robots at given time, and therefore may be used to coordinate the missions of complex systems, such as multi-robot systems, in which multiple robots collaborate for complex tasks.

One important aspect of ubiquitous computing is the existence of disappearing hardware [214] that are mobile, have small form factors and usually limited computation resource. Since the constraints solving may require a lot of computation, these disappearing hardware may not have enough resources. One solution is to use a dedicated server for the off-site assembly process. Therefore, the participating platforms may send the environment information to the server for assembly, and retrieve assembly result and the appropriate implementations of the components.

To further realize ubiquitous computing, it may be necessary to implement **ASAP** on small devices, ranging from pagers and mobile phones to set-top boxes and car naviga-

tion system. Sun Microsystems introduced the Java 2 Platform, Micro Edition [215], a set of specifications that pertain to Java on small devices. Some of features eliminated from Java Standard Edition include reflection, the Java Native Interface, and user-defined class loader [216]. These features are heavily used in current **ASAP** implementation. To port ASAP with J2ME, the entire architecture may need to be reorganized.

There are some other improvements for performance. For instance, constraints solving is a performance bottleneck during component assembly. The current implementation uses a backtracking algorithm to solve the constraints, which basically will iterate through each possible valid value of parameters to check if all the constraints are satisfied. Although, backtracking is strictly better than generate-and-test, its runtime complexity for most nontrivial problems is still exponential. One of the reasons for this poor performance is that the backtracking paradigm suffers from thrashing [217], i.e., search in different parts of the space keeps failing for the same reasons. The situation is worse for distributed applications because, in addition to the slower performance of remote method invocation than direct invocation, constraints solving probably needs to examine all possible distributions in which the complexity is $n_p^{(n_c-1)}$ times that of non-distributed application. A number of different approaches have been developed for solving these problems [218], such as using constraint propagation to simplify the original problem. It will be nice to define a clear interface for constraint solving algorithms, so that different constraint solving algorithm could be implemented and be selected by the application launcher. Furthermore, the component assembly results may be cached. For subsequence execution of the same software, it would be possible

to accelerate the assembly process by "perturbating" the results, if the computing environment does not fluctuate significantly.

The current assembly process will use the first found feasible combination, although there may be more than one feasible combination. However, the first found feasible combination may not be what the user really wants. For example, both combinations, **GnuGO** with no sound and **Random** with **MIDI**, may be feasible. The user may prefer to play **GnuGO** with no sound, although the first found feasible combination is **Random** with **MIDI**. This could be solved by allowing users specify the priority of versions of components. The prioritization of the implementations of components will reorder the examination of implementations, so the first found feasible program will use the high priority version. However, this may raise some research challenges in terms of where the priority should be specified. It is not practical to specify the priority within component specification, because when developers specify components, they actually specify their interface without knowing details about implementations. To prioritize implementations requires knowledge about available implementations and entangles the interfaces and implementations, which, as discussed in subsection 3.5.2, may cause the interfaces to become implementation dependent and implementations may not be interchangeable. Alternatively and maybe more properly, the priority may be specified by users before execution. However, this approach may place the burden on users for priority specification, since they need to retrieve information about all implementations of each components and make judgments. More concepts and research should be clarified and investigated.

The current communication implementation of each remote method invocation requires opening a socket, data transfer, and closing the socket. Obviously, when there are many remote invocations involved, such a implementation is not efficient for software execution and burdens the **FRAME** agents. We are investigating the possibility of aggregating several remote invocations within one socket session, in which the biggest challenge is the possible data corruption for simultaneous remote method invocations within one session.

Other minor improvements includes implementation dependent software hierarchy and prioritizing the constraints. First, current software hierarchy are implementation independent; that is, each implementation of a component will have the same child component dependency. Such a hierarchy may be not valid for a later developed version of a component with more features added. We would like to extend it to an implementation dependent software hierarchy, so each version of a component may have its own child component dependency, which will give component developers more flexibility to develop later versions of components that can use different child components. Again, this may raise problems that similarly occur in the prioritization of the implementations.

Meanwhile, some constraints are critical, which have to be always satisfied, and best-effort would be enough for others. Thus, the critical constraints will be assigned a higher priority, and the feasible software constraints will be the ones in which the critical constraints are satisfied. Consequently, the goal of the component assembly is to search for the best software constraint, in which there are the most lower priority constraints satisfied, from all the feasible software constraints.

7.2.2 CSML

Another performance improvement may be accomplished in the component specification stage; use **CSML** to optimize the component constraints by checking the redundant constraints for avoiding unnecessary constraints solving. From formulae 5.2, the constraints solving performance, P_s, depends on the number of constraints sets and the number of constraints in each set, $\sum |S_i|$, and it also is proportional to the range of each parameter domain, $\prod |D_i|$. To improve brute force constraints solving algorithm, if more information may be extracted from the relationship between parameters, i.e., connectors, some redundancy may be found in the constraints sets. Thus, truth checking for some values in domains or some constraints may be avoided, which will reduce the terms $\prod |D_i|$ and $\sum |S_i|$ respectively.

It also will be helpful to add more built-in macros for checking common used resource and reservation, such as the remaining power, CPU speed, I/O bandwidth, etc. For example of the GO game, by simply specifying macro for power constraint, **CSML** could generate the Java code which can determine if a machine is powered by a battery.

Also, in the current implementation of **CSML**, the components are specified individually without checking the validity of dependency of child components during **CSML** compilation time. Some errors will not show up until run-time. Therefore, it will be nice to add some features so **CSML** will load the specification of child components and do the dependency validity checking during the compilation time. Other minor improvement may be a graphical user interface for **CSML**, so developer may have a better overview about software architectures.

7.2.3 On-line Adaptation

Brew and reassembly are not fully integrated yet in the sense that software does not know threshold of environmental change scale to use **Brew** or reassembly; that is, how significant in the change of the environment should reassembly be used rather than **Brew**. It is difficult and ambiguous to define the threshold because it is necessary to quantize the environmental change. Currently, it still depends on human beings to determine when one approach is better than the other. An automatic approach may be accomplished by using the second level on-line adaptation with indirect information; that is, to monitor the adaptation and efficiency of **Brew** and reassembly. For instance, if an application repaired by **Brew** needs to be repaired frequently and the repairing time is significant, it may indicate that **Brew** may not be ideal approach for on-line adaptation and reassembly may be used. If the repairing time for using reassembly is short, it may indicate that **Brew** may be better approach. The second level on-line adaptation is applied to an adaptation, not an application, which may be referred as first level on-line adaptation, and may bring some interesting research challenges.

There is another extension of the functionality of **Brew**. The applications and **Brew** form a closed loop, where the gap between the VO and the remote controller is bridged by a human. It is possible to add an adaptive algorithm that will automatically send a control command based on the instrumentation results from VO. It requires to build a interface that allows an adaptive algorithm to easily fill the gap to form a closed loop that does not require any human intervention. The development of the adaptive

algorithm is separated from the applications and may be easily customized. Different adaptive algorithms may also be swapped dynamically without terminating and restarting the applications. Furthermore, they have less impact on the performance of the applications because they are executed on different platforms.

Security also needs to be considered, particularly under wireless environments. Security falls into two issues. The first one is to authenticate trustworthy persons to steer the software. The second one is to avoid unwanted methods be invoked remotely. Since **brew** use Java reflection APIs, basically all public methods of exported objects and all static public methods of all classes could be invoked remotely, including the methods which may cause unwanted consequences. One possible solution is to enforce the developers to carefully design their software so that the methods not appropriate for remote invocation would not be public, but this may not be practical because it brings more constraints that are not directly related to software specifications and it could take time to verify. It also does not allow different levels of authorities; that is, different users may have different authorities to steer the software. A more practical solution is to use a security profile file, so the **RemoteControlManager** may filter out the unwanted invocations based on predefined profile and grant users different levels of authorities.

APPENDICES

Appendix A

CSML of the Java MPEG Video Player

This appendix lists the component and constraint specifications of the Java MPEG video player in CSML. The specifications may be derived from table 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, and 6.8.

A.1 CSML for the root component player

```
<application name="player"
  uri="http://www.cse.msu.edu/~korenson/class/player.jar"
  steerable="on" registry-host="sushi">
  <description>
    the root component of the Java MPEG video player.
  </description>
  <parameter name="frames" id="p11" value-type="int" upper="30"
    lower="5" />
  <parameter name="quality" id="p12" value-type="int" upper="8"
    lower="1" />
```

```
11   <internal-connector id="f3">
12     <from-current parameter-id="f1" alias="var1" />
13     <from-current parameter-id="f2" alias="var2" />
14     <definition>
15       return 7 * ^var1# + 25 * ^var2# == 235;
16     </definition>
17   </internal-connector>
18   <child-component name="displayer" id="c1" registry-host="sushi"
19     uri="http://www.cse.msu.edu/~korenson/class/displayer.jar" />
20   <external-connector id="f4">
21     <from-current parameter-id="f1" alias="var1" />
22     <from-child child-id="c1" parameter="time" alias="var2" />
23     <definition>
24       return ^var1# * ^var2# == 1000;
25     </definition>
26   </external-connector>
27   <external-connector id="f5">
28     <from-current parameter-id="f2" alias="var1" />
29     <from-child child-id="c1" parameter="quality" alias="var2" />
30     <definition>
31       return ^var1# == ^var2#;
32     </definition>
33   </external-connector>
34   <main>
35     <instance name="player" child-id="c1" />
36     <required-service service="play" child-id="c1" />
37     <definition>
38       try {
39         player.play(args);
40       } catch(Throwable e) {
41          e.printStackTrace();
42       }
43     </definition>
44   </main>
45 </application>
```

A.2 CSML for the component displayer

```
1 <component name="displayer" steerable="on"
2   registry-host="192.168.1.111"
```

```
3   uri="http://www.cse.msu.edu/~korenson/class/displayer.jar">
4   <general>
5     <provided-service>
6       <declaration method-name="play" return-type="void">
7         <argument name="videoFileName" value-type="String[]"/>
8       </declaration>
9     </provided-service>
10    <parameter name="time" id="p21" value-type="int" upper="200"
11      lower="40" />
12    <parameter name="quality" id="p22" value-type="int" upper="8"
13      lower="1" />
14    <internal-connector id="f1">
15      <from-current parameter-id="p21" alias="var1" />
16      <from-current parameter-id="p22" alias="var2" />
17      <definition>
18        return 7 * ^var1# - 160 * ^var2# &gt;= 120;
19      </definition>
20    </internal-connector>
21    <child-component name="decoder" id="c1"
22      registry-host="192.168.1.111" />
23    <child-component name="RTOS" id="c2"
24      registry-host="192.168.1.111" />
25  </general>
26  <customized version="on_the_fly"
27    uri="http://www.cse.msu.edu/~korenson/class/on_the_fly.jar">
28    <description> on-the-fly version </description>
29    <constraint-definition constraint-id="f2">
30      <external-connector id="f2">
31        <from-current parameter-id="p21" alias="var1" />
32        <from-child child-id="c1" parameter="time" alias="var2" />
33        <definition>
34          return ^var1# &lt;= ^var2# + 70;
35        </definition>
36      </external-connector>
37    </constraint-definition>
38  </customized>
39  <customized version="buffer"
40    uri="http://www.cse.msu.edu/~korenson/class/buffer.jar">
41    <description> buffer version </description>
42    <constraint-definition constraint-id="f2">
43      <external-connector id="f2">
44        <from-current parameter-id="p21" alias="var1" />
45        <from-child child-id="c1" parameter="time" alias="var2" />
46        <definition>
47          return 190 &lt;= ^var2# &amp;&amp;
```

```
48            ^var2# &lt;= 1000;
49          </definition>
50        </external-connector>
51      </constraint-definition>
52    </customized>
53 </component>
```

A.3 CSML for the component decoder

```
 1 <component name="decoder" steerable="on"
 2   registry-host="192.168.1.111"
 3   uri="http://www.cse.msu.edu/~korenson/class/decoder.jar">
 4   <general>
 5     <provided-service>
 6       <declaration method-name="decode" return-type="void">
 7         <argument name="videoFileName" value-type="String[]"/>
 8       </declaration>
 9     </provided-service>
10     <parameter name="time" id="p31" value-type="int" upper="200"
11       lower="40" />
12     <parameter name="quality" id="p32" value-type="int" upper="8"
13       lower="1" />
14     <internal-connector id="f1">
15       <from-current parameter-id="p31" alias="var1" />
16       <from-current parameter-id="p32" alias="var2" />
17       <definition>
18         return 7 * ^var1# - 160 * ^var2# &gt;= 120;
19       </definition>
20     </internal-connector>
21     <child-component name="RTOS" id="c1"
22       registry-host="192.168.1.111" />
23   </general>
24   <customized version="decoder_impl"
25     uri="http://www.cse.msu.edu/~korenson/class/decoder_impl.jar">
26     <description> the only one implementation </description>
27   </customized>
28 </component>
```

A.4 CSML for the component RTOS

```
<component name="RTOS" registry-host="192.168.1.111"
  uri="http://www.cse.msu.edu/~korenson/class/RTOS.jar">
  <general>
    <provided-service id="s1">
      <declaration method-name="setStartTime" return-type="void">
        <description>
          Set the start time when the CPU reservation is taking
          effect. More precisely, after (millis + nanos) later,
          the reservation will take effect.
        </description>
        <argument name="millis" value-type="long"/>
        <argument name="nanos" value-type="long"/>
      </declaration>
      <declaration method-name="setComputeTime" return-type="void">
        <description>
          Reserve the CPU time for each period.
        </description>
        <argument name="millis" value-type="long"/>
        <argument name="nanos" value-type="long"/>
      </declaration>
      <declaration method-name="setPeriod" return-type="void">
        <description>
          Set period.
        </description>
        <argument name="millis" value-type="long"/>
        <argument name="nanos" value-type="long"/>
      </declaration>
  </general>
  <customized version="TimeSys"
    uri="http://www.cse.msu.edu/~korenson/class/timesys.jar">
    <description> TimeSys version </description>
    <constraint-definition constraint-id="s1">
      <resource-requirement>
        <resource name="timesys-rt-linux" id="r1">
          <built-in><timesys-linux-rt /></built-in>
        </resource>
      </resource-requirement>
    </constraint-definition>
  </customized>
  <customized version="dummy"
    uri="http://www.cse.msu.edu/~korenson/class/dummy.jar"
```

```
42      default="yes">
43      <description> Dummy version </description>
44    </customized>
45 </component>
```

Bibliography

Bibliography

[1] Timeline of Computing History. Information available at *http://www.computer.org/computer/timeline/timeline.pdf.*

[2] Gordon Moore. Electronics. *Cramming More Components onto Integrated Circuits*, 38(8):114–117, April 1965.

[3] Motorola Semiconductor Products Sector. Information available at *http://e-www.motorola.com/.*

[4] Intel(R) PCA Applications Processors. Information available at *http://www.intel.com/design/pca/applicationsprocessors/index.htm.*

[5] The Official Bluetooth Wireless Info Site. Information available at *http://www.bluetooth.com/.*

[6] Infrared Data Association. Information available at *http://www.irda.org/.*

[7] IEEE P802.11, The Working Group for Wireless LANs. Information available at *http://grouper.ieee.org/groups/802/11/.*

[8] Intel Corporation. *Mobile Platform Vision Guide for 2003*, September 2002. Information available at *http://www.intel.com/design/mobile/platform/downloads/IMPVG03.pdf.*

[9] Gartner, Inc. Information available at *http://gartner.com.*

[10] International Data Corporation. *Worldwide PC Forecast Update 2002-2006*, July 2002. Information available at *http://www.idc.com.*

[11] International Data Corporation. *U.S. Mobile Worker Population Forecast and Analysis, 2002-2006*, June 2002. Information available at *http://www.idc.com.*

[12] Ren-Song Ko and Matt W. Mutka. **FRAME** for Achieving Performance Portability within Heterogeneous Environments. In *Proceedings of the 9th IEEE Conference on Engineering Computer Based Systems (ECBS)*, Lund University, Lund, SWEDEN, April 2002.

[13] Autoconf - GNU Project - Free Software Foundation (FSF). Information available at *http://www.gnu.org/software/autoconf/autoconf.html.*

[14] Automake - GNU Project - Free Software Foundation (FSF). Information available at *http://www.gnu.org/software/automake/.*

[15] The Icon Programming Language. Information available at *http://www.cs.arizona.edu/icon/.*

[16] Ren-Song Ko and Matt W. Mutka. Adaptive Soft Real-Time Java within Heterogeneous Environments. In *Proceedings of Tenth International Workshop on Parallel and Distributed Real-Time Systems*, Fort Lauderdale, Florida, April 2002.

[17] Kenneth P. Fishkin, Anuj Gujar, Beverly L. Harrison, Thomas P. Moran, and Roy Want. Embodied User Interfaces for Really Direct Manipulation. *Communications of the ACM*, 43(9):75–80, September 2000.

[18] Mark Weiser and John Seeley Brown. The Coming Age of Calm Technology. In Peter J. Denning and Robert M. Metcalfe, editors, *In Beyond Calculation: The Next Fifty Years of Computing*, pages 75–85. Springer Verlag, 1997.

[19] Donald A. Norman. *The Invisible Computer.* MIT Press, Cambridge, Mass., 1998.

[20] R. Want, B. N. Schilit, N. I. Adams, R. Gold, K. Petersen, D. Goldberg, J. R. Ellis, and M. Weiser. An Overview of the PARCTAB Ubiquitous Computing Experiment. *IEEE Personal Communications*, 2(6):28–33, December 1995.

[21] C. K. Kantarjiev, A. Demers, R. Frederick, R. T. Krivacic, and M. Weiser. Experiences with X in a Wireless Environment. In *Proceeding of the USENIX Mobile and Location-Independent Computing Symposium*, pages 117–128, Cambridge, MA, 1993.

[22] Scott Elrod, Richard Bruce, Rich Gold, David Goldberg, Frank Halasz, William Janssen, David Lee, Kim McCall, Elin Pedersen, Ken Pier, John Tang, and Brent Welch. Liveboard: A Large Interactive Display Supporting Group Meetings, Presentations, and Remote Collaboration. In *Conference proceedings on Human factors in computing systems*, pages 599–607. ACM Press, 1992.

[23] Roy Want, Andy Hopper, Veronica Falco, and Jonathan Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems (TOIS)*, 10(1):91–102, 1992.

[24] Thomas E. Truman, Trevor Pering, Roger Doering, and Robert W. Brodersen. The Infopad Multimedia Terminal: A Portable Device for Wireless Information Access. *IEEE Transactions on Computers*, 47(10):1073–1087, 1998.

[25] Roy Want, Trevor Pering, Gaetano Borriello, and Keith I. Farkas. Disappearing Hardware. *IEEE Pervasive Computing*, 1(1):36–47, January/March 2002.

[26] Roy Want and Bill N. Schilit. Guest editors' introduction: Expanding the horizons of location-aware computing. *IEEE Computer*, 34(8):31–34, August 2001.

[27] Anantha Chandrakasan, Rajeevan Amirtharajah, SeongHwan Cho, James Goodman, Gangadhar Konduri, Joanna Kulik, Wendi Rabiner, and Alice Wang. Design Considerations for Distributed Microsensor Systems. In *Proceedings of 1999 IEEE Custom Integrated Circuits Conference*, pages 279–286, Piscataway, N.J., 1999. IEEE Press.

[28] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System Architecture Directions for Networked Sensors. In *Proceedings of 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, New York, 2000. ACM Press.

[29] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next Century Challenges: Mobile Networking for "Smart Dust". In *Proceedings of 5th ACM/IEEE International Conference on Mobile Computing and Networks*, pages 271–278, New York, 1999. ACM Press.

[30] Jan M. Rabaey. Wireless beyond the Third Generation: Facing the Energy Challenge. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 1–3, New York, 2001. ACM Press.

[31] William R. Hamburgen, Deborah A. Wallach, Marc A. Viredaz, Lawrence S. Brakmo, Carl A. Waldspurger, Joel F. Bartlett, Timothy Mann, and Keith I. Farkas. Itsy: Stretching the Bounds of Mobile Computing. *Computer*, 34(4):28–37, April 2001.

[32] Mark Weiser, Brent Welch, Alan J. Demers, and Scott Shenker. Scheduling for Reduced CPU Energy. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 13–23, USENIX, Berkeley, Calif., 1994.

[33] Jason Flinn. *Extending Mobile Computer Battery Life through Energy-Aware Adaptation.* PhD thesis, Carnegie Mellon University, December 2001.

[34] Y. Kim. Capacity Enhancement Technology of Samsung Soft Case Li-ion batteries. In *Proceedings of Power 2001: 9th Annual International Conference on Power Requirements for Mobile Computing, Wireless Electronic Devices & Business/Consumer Applications*, 2001.

[35] Thad Starner. Human Powered Wearable Computing. *IBM Systems Journal*, 35(3-4):618–629, 1996.

[36] Nathan S. Shenck and Joseph A. Paradiso. Energy Scavenging with Shoe-Mounted Piezoelectrics. *IEEE Micro*, 21(3):30–42, May/June 2001.

[37] Sharon Thomas and Marcia Zalbowitz. Fuel cells, green power. Los Alamos National Laboratory in Los Alamos, New Mexico, 1999. Information available at *http://education.lanl.gov/resources/fuelcells/fuelcells.pdf.*

[38] Klaus Finkenzeller. *RFID Handbook: Radio-Frequency Identification Fundamentals and Applications.* John Wiley & Sons, New York, 2000.

[39] Antonio Regalado. Brain-Machine Interface. *Technology Review*, pages 98–100, January/February 2001.

[40] Tim Kindberg and Armando Fox. System Software for Ubiquitous Computing. *IEEE Pervasive Computing*, 1(1):70–81, January/March 2002.

[41] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The Design and Implementation of an Intentional Naming System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 186–201. ACM Press, 1999.

[42] R. Droms. Dynamic Host Configuration Protocol. RFC 2131, Internet Engineering Task Force. Information available at *http://www.ietf.org/.*

[43] S. Thomson and T. Narten. IPv6 Stateless Address Autoconfiguration. RFC 1971, Internet Engineering Task Force. Information available at *http://www.ietf.org/.*

[44] Ken Arnold, Robert Scheifler, Jim Waldo, Bryan O'Sullivan, and Ann Wollrath. *The Jini Specifications.* Addison-Wesley, 1999.

[45] Jini Technology Architectural Overview, January 1999. Information available at *http://wwws.sun.com/software/jini/whitepapers/architecture.html.*

[46] Jini Specifications v1.2, December 2001. Information available at *http://wwws.sun.com/software/jini/specs/.*

[47] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. IETF RFC 2608, June 1999. Information available at *http://www.ietf.org/rfc/rfc2608.txt.*

[48] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Templates and Service: Schemes. IETF RFC 2609, June 1999. Information available at *http://www.ietf.org/rfc/rfc2609.txt.*

[49] Universal Plug and Play Device Architecture, June 2000. Information available at *http://www.upnp.org/download/UPnPDA10_20000613.htm.*

[50] Salutation Consortium. *Salutation Architecture Specification Version 2.0c*, June 1999. Information available at *http://www.salutation.org/specordr.htm.*

[51] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An Architecture for a Secure Service Discovery Service. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networks*, pages 24–35. ACM Press, 1999.

[52] Erik Guttman. Service Location Protocol: Automatic Discovery of IP Network Services. *IEEE Internet Computing*, 3(4):71–80, July/August 1999.

[53] Tim Kindberg and John Barton. A Web-Based Nomadic Computing System. *Computer Networks*, 35(4):443–456, March 2001.

[54] Choonhwa Lee and Sumi Helal. Context Attributes: An Approach to Enable Context-awareness for Service Discovery. In *Proceedings of the Third IEEE/IPSJ Symposium on Applications and the Internet*, Orlando, Florida, January 2003.

[55] Guanling Chen and David Kotz. A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, November 2000.

[56] John Barton, Tim Kindberg, and Shreyas Sadalgi. Physical Registration: Configuring Electronic Directories using Handheld Devices. Technical Report HPL-2001-119, HP Labs, Palo Alto, CA 94304, USA, May 2001. Available at *http://www.hpl.hp.com/techreports/2001/HPL-2001-119.pdf*.

[57] Robert Grimm, Janet Davis, Ben Hendrickson, Eric Lemar, Adam MacBeth, Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. Systems Directions for Pervasive Computing. In *Proceedings of the 8th Workshop Hot Topics in Operating Systems*, pages 128–132, 2001.

[58] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

[59] N. Davies, S. P. Wade, A. Friday, and G. S. Blair. Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications. In *Proceedings of the IFIP/IEEE International Conference on Open Distributed Processing and Distributed Platforms*, pages 291–302. Chapman & Hall, Ltd., 1997.

[60] Brad Johanson and Armando Fox. Tuplespaces as Coordination Infrastructure for Interactive Workspaces. In *Proceeding of Workshop Application Models and Programming Tools for Ubiquitous Computing*, 2001. Information available at *http://choices.cs.uiuc.edu/UbiTools01/pub/08-fox.pdf*.

[61] Mark Weiser and John Seely Brown. Designing Calm Technology. *PowerGrid Journal*, 1, 1996. Information available at *http://www.ubiq.com/weiser/calmtech/calmtech.htm*.

[62] Brian D. Noble, Morgan Price, and Mahadev Satyanarayanan. A Programming Interface for Application-Aware Adaptation in Mobile Computing. In *Proceeding of USENIX Symposium on Mobile and Location-Independent Computing*, USENIX, Berkeley, Calif., 1995. Information available at *http://choices.cs.uiuc.edu/UbiTools01/pub/08-fox.pdf.*

[63] Bruce Zenel and Dan Duchamp. A General Purpose Proxy Filtering Mechanism Applied to the Mobile Environment. In *Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networks*, pages 248–259. ACM Press, 1997.

[64] Armando Fox, Steven D. Gribble, Yatin Chawathe, and Eric A. Brewer. Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives. *A special issue of IEEE Personal Communications on Adapation*, pages 10–19, August 1998.

[65] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. *ACM SIGCOMM Computer Communication Review*, 20(4):200–208, 1990.

[66] Emre Kiciman and Armando Fox. Using Dynamic Mediation to Integrate COTS Entities in a Ubiquitous Computing Environment. In *Proceeding of the 2nd International Symposium on Handheld and Ubiquitous Computing*, volume 1927 of *Lecture Notes in Computer Science*, pages 211–226, Berlin, 2000. Springer-Verlag.

[67] Brad A. Myers, Rishi Bhatnagar, Jeffrey Nichols, Choon Hong Peck, Dave Kong, Robert Miller, and A. Chris Long. Interacting at a Distance Using Semantic Snarfing. In *Ubicomp 2001: Ubiquitous Computing*, volume 2201 of *Lecture Notes in Computer Science*, pages 305–314, Berlin, 2001. Springer-Verlag.

[68] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, January/February 1998.

[69] Todd D. Hodes, Randy H. Katz, Edouard Servan-Schreiber, and Lawrence A. Rowe. Composable Ad Hoc Mobile Services for Universal Interaction. In *Proceeding of the 3rd International Symposium on Mobile Computing and Networking*, pages 1–12, New York, 1997. ACM Press.

[70] Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. In *Ubicomp 2001: Ubiquitous Computing*, volume 2201 of *Lecture Notes in Computer Science*, pages 56–75, Berlin, 2001. Springer-Verlag.

[71] Chester Fitchett and Saul Greenberg. The Phidget Architecture: Rapid Development of Physical User Interfaces. In *Proceeding of Workshop Application*

Models and Programming Tools for Ubiquitous Computing, 2001. Information available at *http://choices.cs.uiuc.edu/UbiTools01/pub/17-fitchett.pdf*.

[72] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The Context Toolkit: Aiding the Development of Context-Enabled Applications. In *Proceeding of ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 434–441, New York, 1999. ACM Press.

[73] Mike Addlesee, Rupert Curwen, Steve Hodges, Joe Newman, Pete Steggles, Andy Ward, and Andy Hopper. Implementing a Sentient Computing System. *Computer*, 34(8):50–56, August 2001.

[74] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven A. Shafer. EasyLiving: Technologies for Intelligent Environments. In *Proceeding of the 2nd International Symposium on Handheld and Ubiquitous Computing*, volume 1927 of *Lecture Notes in Computer Science*, pages 12–29, Berlin, 2000. Springer-Verlag.

[75] Michael H. Coen, Brenton Phillips, Nimrod Warshawsky, Luke Weisman, Stephen Peters, and Peter Finin. Meeting the Computational Needs of Intelligent Environments: The Metaglue System. In *Proceeding of the 1st International Workshop Managing Interactions in Smart Environments*, Berlin, 1999. Springer-Verlag.

[76] Roy Want, Kenneth P. Fishkin, Anuj Gujar, and Beverly L. Harrison. Bridging Physical and Virtual Worlds with Electronic Tags. In *Proceeding of 1999 Conference on Human Factors in Computing Systems*, pages 370–377, New York, 1999. ACM Press.

[77] Jun Rekimoto and Yuji Ayatsuka. CyberCode: Designing Augmented Reality Environments with Visual Tags. In *Proceeding of Designing Augmented Reality Environments*, pages 1–10, New York, 2000. ACM Press.

[78] Deborah Caswell and Philippe Debaty. Creating Web Representations for Places. In *Proceeding of the 2nd International Symposium on Handheld and Ubiquitous Computing*, volume 1927 of *Lecture Notes in Computer Science*, pages 114–126, Berlin, 2000. Springer-Verlag.

[79] Brad Johanson, Greg Hutchins, and Terry Winograd. PointRight: A System for Pointer/Keyboard Redirection among Multiple Displays and Machines. Technical Report CS-2000-03, Computer Science Department, Stanford University, Stanford, Calif., 2000.

[80] Suchitra Raman and Steven McCanne. A Model, Analysis, and Protocol Framework for Soft State-Based Communication. In *Proceeding of Special Interest Group on Data Communication*, pages 15–25, New York, 1999. ACM Press.

[81] Marc Langheinrich. Privacy by Design: Principles of Privacy-Aware Ubiquitous Systems. In *Ubicomp 2001: Ubiquitous Computing*, volume 2201 of *Lecture Notes in Computer Science*, pages 273–291, Berlin, 2001. Springer-Verlag.

[82] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting Mobile Communications: The Insecurity of 802.11. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networks*, pages 180–188. ACM Press, 2001.

[83] Laura Marie Feeney, Bengt Ahlgren, and Assar Westerlund. Spontaneous Networking: An Application-Oriented Approach to Ad Hoc Networking. *IEEE Communications Magazine*, 39(6):176–181, June 2001.

[84] Ian F. Blake, Gadiel Seroussi, and Nigel P. Smart. *Elliptic Curves in Cryptography.* Cambridge University Press, Cambridge, UK, 1st edition, 2000.

[85] Adrian Perrig, Robert Szewczyk, J.D. Tygar, Victor Wen, and David E. Culler. SPINS: Security Protocols for Sensor Networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networks*, pages 189–199. ACM Press, 2001.

[86] Frank Stajano and Ross Anderson. The Resurrecting Duckling: Security Issues for Ad Hoc Wireless Networks. In *Security Protocols*, volume 1796 of *Lecture Notes in Computer Science*, pages 172–194, Berlin, 1999. Springer-Verlag.

[87] Eran Gabber and Avishai Wool. How to Prove Where You Are: Tracking the Location of Customer Equipment. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 142–149. ACM Press, 1998.

[88] Tim Kindberg and Kan Zhang. Context Authentication Using Constrained Channels. Technical Report HPL-2001-84, Hewlett-Packard Laboratories, Palo Alto, Calif., 2001.

[89] Luca Cardelli and Andrew D. Gordon. Mobile Ambients. In *Foundations of Software Science and Computation Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155, Berlin, 1998. Springer-Verlag.

[90] Pasi Eronen and Pekka Nikander. Decentralized Jini Security. In *Proceeding of Network and Distributed System Security 2001*, pages 161–172, The Internet Soc., Reston, Va., 2001.

[91] Pattie Maes. *Computational Reflection.* PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels, Belgium, January 1987.

[92] Java 2 Platform, Standard Edition (J2SE) API Specification. Information available at *http://java.sun.com/apis.html.*

[93] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 147–155. ACM Press, 1987.

[94] Brian C. Smith. Reflection and Semantics in a Procedural Language. Technical Report 272, Massachusetts institute of Technology. Laboratory for Computer Science, Cambridge, Massachusetts, 1982.

[95] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without Metaphysics. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 348–355, 1984.

[96] R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. In B. L. Webber and N. J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 173–191. Kaufmann, Los Altos, CA, 1981.

[97] Kenneth A. Bowen. Meta-Level Programming and Knowledge Representation. In *Proceedings of the International Conference on Artilicial Intelligence and its Applications*, 1986.

[98] R. Davis and D.B. Lenat. *Knowledge-Based Expert Systems in Artificial Intelligence.* McGraw-Hill, New York, 1982.

[99] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in SOAR: The Anatomy of a General Learning Mechanism. *Machine Learning*, 1:11–46, 1986.

[100] P. Hayes. *The Language GOLUX.* PhD thesis, University of Essex, United Kingdom, 1974.

[101] M. Genesereth. Prescriptive Introspection. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection.* June 1987.

[102] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[103] Adele Goldberg and Alan Kay. Smalltalk-72 Instruction Manual. Technical Report SSL 76-6, Learning Research Group, Xerox Palo, Alto Research Center, 1976.

[104] D. Moon and D. Weinreb. *The Lisp Machine Manual.* MIT AI Lab, 1981.

[105] Adele Goldberg and David Robson. *Smalltalk-80: the Language and its Implementation.* Addison-Wesley Longman Publishing Co., Inc., 1983.

[106] Mark Stefik and Daniel G. Bobrow. Object-Oriented Programming: Themes and Variations. *AI Magazine*, 6(4), 1986.

[107] Brian C. Smith and Carl Hewitt. A PLASMA Primer (draft). MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, October 1975.

[108] H. Lieberman. A Preview of ACT1. MIT Artificial Intelligence Laboratory, MIT AI-MEMO 625, Cambridge, Massachusetts, 1981.

[109] Russel Greiner. RLL-1: A Representation Langauge Language. Working Paper 80-9, Stanford Heuristic Programming Project, October 1980.

[110] Jean-Pierre Briot and Pierre Cointe. The OBJVLISP Model: Definition of a Uniform, Reflexive and Extensible Object Oriented Language. In *Proceeding of the 7th European Conference on Artificial Intelligence*, pages 270–277, Brighton, UK, 1986.

[111] L. Steels. The KRS Concept System. Technical Report 86-1, Artificial Intelligence Laboratory, Brussels Free University, Brussels, Belgium, 1986.

[112] Daniel G. Bobrow and Gregor Kiczales. The Common Lisp Object System Metaobject Kernel - A Status Report. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 309–315, Snowbird, Utah, United States, 1988. ACM Press.

[113] *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS.* Addison-Wesley, 1989.

[114] Gregor Kiczales, Jim des Rivires, and Daniel G. Bobrow. *The Art of Metaobject Protocol.* MIT Press, 1991.

[115] Ramana Rao. Implementational Reflection in Silica. In *Proceedings of European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 251–266, Geneva, Switzerland, July 1991. Springer-Verlag.

[116] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27, pages 414–434, New York, NY, 1992. ACM Press.

[117] Mamdouh H. Ibrahim, editor. *Proceedings of the OOPSLA '91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, 1991.

[118] Jr Avadis Tevanian. Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach. Technical Report CMU-CS-88-106, Carnegie Mellon University, December 1987.

[119] E. Abrossimov, M. Rozier, and M. Shapiro. Generic Virtual Memory Management for Operating System Kernels. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 123–136. ACM Press, 1989.

[120] H. Tokuda and C. W. Mercer. Arts: a distributed real-time kernel. *ACM SIGOPS Operating Systems Review*, 23(3):29–53, 1989.

[121] Jecel M. Assumpcao Jr and Sergio T. Kofuji. Bootstrapping the Object Oriented Operating System Merlin: Just Add Reflection. In Chris Zimmerman, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*, chapter 5. CRC Press, 1996.

[122] Urs Hölzle, Bay-Wei Chang, Craig Chambers, Ole Agesen, and David Ungar. The SELF Manual. Version 1.1. Unpublished manual, February 1991.

[123] Jecel M. Assumpcao Jr. Adaptive Compilation in the Merlin System for Parallel Machines. In *Proceeding of IEEE/USP International Workshop on High Performance Computing*, pages 155–166, March 1994.

[124] The TUNES Project for a Reflective Computing System. Information available at *http://tunes.org/.*

[125] Object Management Group, Framingham, Mass. *CORBA v2.2 Specification*, 1998.

[126] N. Brown and C. Kindel. *Distributed Component Object Model Protocol-DCOM/1.0.* Information available at *http://www.microsoft.com/com.*

[127] Java Remote Method Invocation (RMI). Information available at *http://java.sun.com/products/jdk/rmi/.*

[128] Gordon S. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, 1998. Springer-Verlag.

[129] Yi-Min Wang and Woei-Jyh Lee. COMERA: COM Extensible Remoting Architecture. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 79–88. USENIX, April 1998.

[130] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.

[131] D. Schmidt and C. Cleeland. Applying Patterns to Develop Extensible ORB Middleware. In *IEEE Communications Magazine*, volume 37, pages 54–63. IEEE CS Press, Los Alamitos, Calif., 1999.

[132] Baochun Li, William Kalter, and Klara Nahrstedt. A Hierarchical Quality of Service Control Architecture for Configurable Multimedia Applications. In *Journal of High Speed Networks, Special Issue on Management of Multimedia Networking.* IOS Press, 2001.

[133] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[134] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, April 1996.

[135] Steve Vestal. A Cursory Overview and Comparison of Four Architecture Description Languages. Technical report, Honeywell Technology Center, February 1993.

[136] D. Garlan, editor. *Proceeding of the 1st International Workshop Architectures for Software Systems*, April 1995.

[137] David Garlan, Walter Tichy, and Frances Paulisch. Summary of the dagstuhl workshop on software architecture. *ACM SIGSOFT Software Engineering Notes*, 20(3):63–83, 1995.

[138] A.L. Wolf, editor. *Proceeding of the 2nd International Software Architecture Workshop*, October 1996.

[139] J. Magee and D.E. Perry, editors. *Proceeding of the 3rd International Software Architecture Workshop*, November 1997.

[140] David Garlan, Robert Allen, and John Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering.* ACM Press, December 1994.

[141] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 24–32. ACM Press, 1996.

[142] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 44–53, 1999.

[143] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proceeding of the 5th European Software Engineering Conference*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[144] Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 3–14. ACM Press, 1996.

[145] Pam Binns, Matt Englehart, Mike Jackson, and Steve Vestal. Domain-Specific Software Architectures for Guidance, Navigation and Control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227, June 1996.

[146] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[147] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[148] Mark Moriconi and R. A. Riemenschneider. Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. Technical Report SRI-CSL-97-01, SRI Int'l, March 1997.

[149] Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, 1995.

[150] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *Software Engineering*, 21(4):314–335, 1995.

[151] Mary Shaw, Robert DeLine, and Gregory Zelesnik. Abstractions and Implementations for Architectural Connections. In *Proceeding of the 3rd International Conference on Configurable Distributed Systems*, Annapolis, Maryland, May 1996.

[152] Michael M. Gorlick and Rami R. Razouk. Using Weaves for Software Construction and Analysis. In *Proceedings of the 13th International Conference on Software Engineering*, pages 23–34. IEEE Computer Society Press, 1991.

[153] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[154] David Garlan, Robert T. Monroe, and David Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[155] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A Highly-Extensible, XML-Based Architecture Description Language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architectures*, Amsterdam, Netherlands, August 2001.

[156] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages.

In *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, May 2002.

[157] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, January 2000.

[158] Xiaohui Gu, Klara Nahrstedt, Wanghong Yuan, Duangdao Wichadakul, and Dongyan Xu. An XML-based QoS Enabling Language for the Web. In *Journal of Visual Language and Computing (Special Issue on Multimedia Languages for the Web)*. Academic Press, December 2001.

[159] G. Wiederhold, P. Wegner, and S. Ceri. Towards Megaprogramming: A Paradigm for Component-Based Programming. *Communications of the ACM*, pages 89–99, November 1992.

[160] K. Nilsen. Adding Real-Time Capabilities to Java. *Communications ACM*, pages 49–56, June 1998.

[161] William Foote. Theory versus Practice in Real-Time Computing with the Java™ Platform. Available at *http://java.sun.com/people/billf/real-time/isorc99.ps.*

[162] William Foote. Real-Time Extensions to the Java™ Platform: a Progress Report. Available at *http://java.sun.com/people/billf/real-time/rtss98WIP.ps.*

[163] Lisa Carnahan and Marcus Ruark, editors. *Requirements for Real-Time Extensions for the Java Platform: Report from the Requirement Group for Real-Time Extensions for the Java Platform.* NIST Special Publication 500-243, September 1999. Available at *http://www.nist.gov/itl/div897/ctg/real-time/rt-doc/rtj-final-draft.pdf.*

[164] The Real-Time for Java™ Experts Group. *The Real-Time Specification for Java.* Available at *http://www.javaseries.com/rtj.pdf.*

[165] Real-Time Java™ Working Group. *Real-Time Core Extensions.* Available at *http://www.j-consortium.org/rtjwg/rtce.1.0.14.pdf.*

[166] K. Nilsen. Real-Time Core Extensions for the Java™ Platform. Available at *http://www.j-consortium.org/rtjwg/rtss.12-1-99.ppt.*

[167] JSR: Java Specification Request. Distributed Real-Time Specification. Information available at *http://jcp.org/jsr/detail/050.jsp.*

[168] Daniel A. Reed, Robert D. Olson, Ruth A. Aydt, Tara Madhyastha, Thomas Birkett, David W. Jensen, Bobby A. A. Nazief, and Brian K. Totty. Scalable Performance Environments for Parallel Systems. Technical Report 1673, Urbana, Illinois, 1991.

[169] Daniel A. Reed, Keith A. Shields, Will H. Scullin, Luis F. Tavera, and Christopher L. Elford. Virtual Reality and Parallel Systems Performance Analysis. *Computer*, 28(11):57–67, 1995.

[170] J. S. Vetter and D. A. Reed. Real-Time Performance Monitoring, Adaptive Control, and Interactive Steering of Computational Grids. *The International Journal of High Performance Computing Applications.*

[171] Weiming Gu, Greg Eisenhauer, Karsten Schwan, and Jeffrey Vetter. Falcon: On-line monitoring and steering of parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, 1998.

[172] Jeffrey Vetter and Karsten Schwan. Progress: A Toolkit for Interactive Program Steering. In *Proceedings of the 24th International Conference on Parallel Processing*, pages 139–142, Urbana, IL, 1995.

[173] Falcon and Progress: Platforms for Interactive High Performance Programs. Information available at *http://www.cc.gatech.edu/systems/projects/Steering/.*

[174] G. Geist, J. Kohl, and P. Papadopoulos. CUMULVS: Providing Fault-Tolerance, Visualization, and Steering of Parallel Applications. In *The International Journal of Supercomputer Applications and High Performance Computing*, volume 11, pages 224–235, 1997.

[175] S.G. Parker and C.R. Johnson. SCIRun: Applying Interactive Computer Graphics to Scientific Problems. In *SIGGRAPH 96*, 1996.

[176] S. G. Parker, M. Miller, C. D. Hansen, and C. R. Johnson. An Integrated Problem Solving Environment: the SCIRun Computational Steering System. In *31st Hawaii International Conference on System Sciences (HICSS-31)*, volume vii, pages 147–156, January 1998.

[177] Abdul Waheed, Diane T. Rover, Matt Mutka, Aleksander Bakić, and David Pierce. Vista: A Framework for Instrumentation System Design for Multidisciplinary Applications. In *Proceedings of the 4th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 192–195, San Jose, CA, February 1996.

[178] Daniel A. Reed, Christopher L. Elford, Tara M. Madhyastha, Evgenia Smirni, and Stephen E. Lamm. The Next Frontier: Interactive and Closed Loop Performance Steering. In *Proceedings of the 1996 International Conference on Parallel Processing Workshop*, pages 20–31, Bloomingdale, IL, August 1996.

[179] S.A. Khaddaj. Animation and Integration of Material Growth Simulations in a Visual Programming Environment. *International Journal of Imaging Systems and Technology*, 7(3):246–255, 1996.

[180] S. Doi, T. Takei, Y. Akiba, K. Muramatsu, H. Matsumoto, L.C.J. van Rijn, and B.C. Schultheiss. A Real-Time Visualization System for Computational Fluid Dynamics. *NEC Research and Development*, 37(1):114–123, January 1996.

[181] R. Grosso, K. Wechsler, T. Ertl, and M. Schafer. Computational Steering and Visualization for Multiblock Multigrid Flow Simulations. In *High-Performance Computing and Networking. International Conference and Exhibition*, pages 927–928, Brussels, Belgium, April 1996.

[182] Paul R. Woodward. Interactive Scientific Visualization of Fluid Flow. *Computer*, 26(10):13–25, October 1993.

[183] G.D. Kerlick and E. Kirby. Towards interactive Steering, Visualization and Animation of Unsteady Finite Element Simulations. *IEEE Visualization 1993*, pages 374–377, 1993.

[184] Matthias Deegener, Gerd Grosse, Werner John, Bettina Khnapfel, Michael L'ohr, and Hanno Wirth. Rapid Prototyping with MUSE. In *Proceeding for the Dedicated Conferences on Mechatronics and Supercomputing Applications in the Transportation Industries*, number 625-632, 1994.

[185] G. Hanyzewski and M. Mull. HERMES: A Tool Kit for Developing Distributed Modeling Applications. *Nanotechnology*, 7(3):193–196, 1996.

[186] R. de Souza and Yuanhong Zhang. SimEnvir++: An Object-Based Simulation Environment. *Computers in Industry*, 30(3):211–217, October 1996.

[187] M.C. Chan, G. Pacifici, and R. Stadler. A Platform for Real-Time Visualization and Interactive Simulation of Large Multimedia Networks. In *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems*, pages 47–52, Honululu, HI, April 1996.

[188] Mei C. Chuah and Steven F. Roth. On the Semantics of Interactive Visualizations. In *Proceedings of the Symposium on Information Visualization '96*, pages 29–36, San Francisco, CA, October 1996. IEEE Press.

[189] Mikael Jern and Rae A. Earnshaw. Interactive Real-Time Visualization System Using a Virtual Reality Paradigm. In M. Gobel, H. Muller, and B. Urban, editors, *Proceedings of the 5th Eurographics Workshop on Visualization in Scientific Computing*, pages 174–189, Rostock, Germany, May 1994. Springer-Verlag.

[190] Nils J. Nilsson. Shakey the robot. Technical Report 323, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, April 1984.

[191] Rodney A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[192] Theodore Q. Pham, Kevin R. Dixon, Jonathan R. Jackson, and Pradeep K. Khosla. Software Systems Facilitating Self-Adaptive Control Software. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2000.

[193] Seiji Yamada and Jun'ya Saito. Adaptive Action Selection without Explicit Communication for Multi-Robot Box-Pushing. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1999.

[194] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[195] GNU Go - GNU Project - Free Software Foundation (FSF). Information available at *http://www.gnu.org/software/gnugo/*.

[196] Vipin Kumar. Depth-First Search. In S. C. Shapiro, editor, *Encyclopaedia of Artificial Intelligence*, volume 2, pages 1004–1005. John Wiley and Sons, Inc., 1987.

[197] Allen Parrish, Brandon Dixon, and David Cordes. A Conceptual Foundation for Component-Based Software Deployment. *Journal of Systems and Software*, 2001.

[198] Don Batory and Sean O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[199] John Cheesman. What is a Component, February 1998. Information available at *http://www.cbdedge.com/cbdweb/technology/whatComp.html*.

[200] Mary Shaw. The Coming-of-Age of Software Architecture Research. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 656–664a, Toronto, Canada, 2001. IEEE Computer Society.

[201] Extensible Markup Language (XML). Information available at *http://www.w3.org/XML/*.

[202] ISO 8879:1986 Information processing - Text and office systems - Standard Generalized Markup Language (SGML). Information available at *http://www.iso.org/*.

[203] Aleksandar Bakić, Matt W. Mutka, and Diane T. Rover. Real-Time Performance Visualization and Analysis Using Distributed Visual Objects. In *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, pages 154–161, December 1997.

[204] Alan K. Mackworth and Eugene C. Freuder. The Complexity of Constraint Satisfaction Revisited. *Artificial Intelligence*, 59(1-2):57–62, 1993.

[205] Alain Papazoglou. JavaGO. Information available at *http://www.papazoglou.net/go/javago.html.*

[206] K. I. Farkas, J. Flinn, G. Back, D. Grunwald, and J. M. Anderson. Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine. *Communications of the ACM*, pages 49–56, June 1998.

[207] Joerg Anders. MPEG-1-Player. Information available at *http://rnvs.informatik.tu-chemnitz.de/~jan/MPEG/MPEG_Play.html.*

[208] Moving Picture Experts Group (MPEG). Information available at *http://mpeg.telecomitalialab.com.*

[209] TimeSys. Real-Time Embedded Linux. Information available at *http://www.timesys.com/.*

[210] Java JIT Compiler. Information available at *http://wwws.sun.com/software/solaris/jit/.*

[211] The JIT Compiler Interface Specification. Information available at *http://java.sun.com/docs/jit_interface.html.*

[212] Fran Crumb. Robot demonstrates self-adaptive software techniques. Air Force Research Laboratory Public Affairs, January 2000. Information available at *http://www.af.mil/news/Jan2000/n20000107_000021.html.*

[213] Nomadic Technologies, Inc., Mountain View, CA. *Nomad XRDEV Software Manual*, March 1999. Information available at *http://nomadic.sourceforge.net/production/manuals/xrdev-1.0.pdf.gz.*

[214] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):66–75, September 1991. Reprinted in IEEE Pervasive Computing, Jan-Mar 2002, pp. 19-25.

[215] Sun Microsystems, Inc., Palo Alto, CA. *Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices*, May 2000. Information available at *http://java.sun.com/products/cldc/wp/KVMwp.pdf.*

[216] Sumi Helal. Pervasive Java. *IEEE Pervasive Computing*, 1(1):82–85, January/March 2002.

[217] John Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1979. Available as Technical Report CMU-CS-79-124.

[218] Vipin Kumar. Algorithms for Constraints Satisfaction problems: A Survey. *The AI Magazine, by the AAAI*, 13(1):32–44, 1992.