FRAME for Achieving Performance Portability within Heterogeneous Environments*

Ren-Song Ko and *Matt W. Mutka*
Department of Computer Science and Engineering
3115 Engineering Building, Michigan State University
East Lansing, MI 48824-1226
E-mail: {korenson,mutka}@cse.msu.edu

Abstract

Resource heterogeneity offers a new challenge to portability of resource critical applications such as multimedia or interactive applications. Under heterogeneous environments, a priori knowledge of available resources is not always feasible during the development stage. As a consequence, resource critical applications must probe dynamically the resources and reconfigure to adapt themselves to different computing environments. We propose a component-based framework, FRAME*, and* CSML *to help people develop and deploy resource critical Java applications in a heterogeneous environment. Under* FRAME*, the assembly of an application is postponed to execution time so that the application may be customized by gathering resource information from the environments.* CSML *may help people develop components and specify the intended performance. Finally, an application, the GO game, is used to evaluate* FRAME *under various environments.*

1 Introduction

It is very likely that future computing environments are composed of a wide range of devices, with diverse architectures and purposes, interconnected via networks. Software portability under such a heterogeneous environment brings challenges to software developers. Cross platform compilability provides source code level portability. With careful coding, the software may be compiled into native code for various platforms. Nevertheless, it is probably difficult to deploy and use the software. That is, 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 Java platform promises "Write Once Run Anywhere," which provides platform independent code level portability. Instead of native code, the software is compiled into platform independent intermediate bytecode that will be executed by a virtual machine. It mitigates the difficulty for software deployment and use, because recompilation for a target platform is not necessary.

Resource heterogeneity offers a higher level challenge to portability to many applications including multimedia or interactive applications. For instance, while an interactive application may run perfectly on a desktop, it may not work on a handheld computer because of inadequate resources. It might be desirable to lower the quality of output on a handheld computer to achieve a better performance. Moreover, if a platform provides some special functionality, the performance may improve if the software has the knowledge and may take advantage of the special functionality.

In order to achieve such a higher level performance portability, the information of available resources on target platforms are necessary but, unfortunately, not available until run-time. Therefore, the application must probe dynamically performance capabilities and reconfigure to different computing environments instead of assuming the capability of target platform. Criteria for probing, such as resource requirements, performance, and quality are specified as constraints during the software development stage.

Nevertheless, specification of constraints raise challenges in complex software systems development. The specified constraints may not be orthogonal. For example, performance usually declines when achieving better output quality. Moreover, today complex software systems are usually collaborative projects. The constraints that individual developers envision may sharply differ.

*This work is supported in part by the National Science Foundation under grants no. 009017, 9911074, and 9700732.

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. The constraints in complex software systems will be hierarchical.

We propose a Java framework, FRAME, and CSML (Component/constraint Specification Markup Language) to help people develop, deploy, and use resource critical software on various platforms. We evaluate FRAME and CSML on an application that executes on a range of platforms, from a desktop computer to a mobile device. The research problems we address are

- How do users specify performance and quality of applications for the purpose of enhancing performance portability?
- How do applications customize themselves to a broad range of computing environments?
- How do developers specify hierarchical constraints?

The remainder of this paper is organized as follows. The architecture of FRAME is described in the next section, followed by the description of the specification markup language, CSML. Section 4 describes the application, the GO game, running under different environments consisting of a handheld computer and a desktop. Finally, the last two sections will give a summary, survey of related work, and then discuss potential future investigations.

2 FRAME

2.1 Overview

In many mass production industries, such as automobiles and electronics, the final products are assembled from parts. The parts may be built by various vendors, but they are plug-in compatible if they have the required functionality and satisfy specifications. This is the idea behind FRAME, in which the applications are assembled from parts (or components). Furthermore, the information about component specifications (or constraints) are actually embedded into implementation of components and an automatic assembling process is possible. Because of the automatic assembling process, the applications may not only be assembled during the development stage, but also be assembled "on the fly" so that the applications may be customized to specific computing environments.

An application should be composed of components under FRAME. Each component has constraints and co-

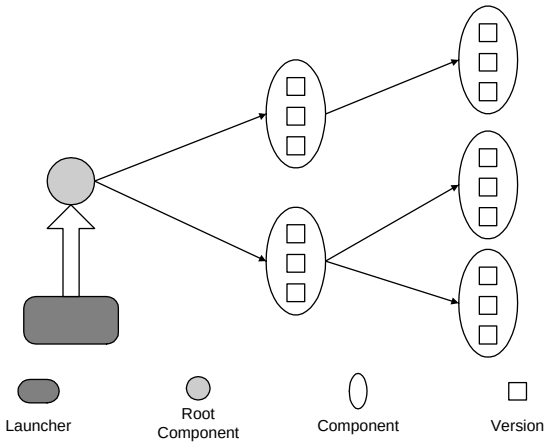

Figure 1. Software hierarchy under FRAME

operates 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. To develop an application, developers first need to decompose the application into components. The information about each component, including the component location and child component dependency, should be stored in a server called the *component registry*. For each component, developers need to design its interface consisting of the services it can provide and the constraints it has. Finally, each component needs to be implemented and information about the implementation, including version number, should also be registered to the component registry. Furthermore, various implementations with different resource requirement may be developed and used for specific computing environments.

Fig. 1 is a general software hierarchy. The dependency of components is 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. 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.

The software is not directly executed by users but via the *launcher*, which will trigger the *component assembling*, and the execution of the software. Users specify performance constraints in the launcher and execute it on the target platform.

FRAME provides the Java APIs for communication between target platforms and registry, component loading, component assembling, and component initialization. Details about FRAME will be explained with an example, the GO game in the following subsections.

2.2 Component registry

The component registry is a server that stores information about components. The information, such as the available versions of components and their locations, are necessary during the component assembling. Before a component may be used, it must register the information to the registry. Since new versions of components with different resource requirements might be developed after the software is developed, the software should not have any knowledge about the version-dependent information in advance. It must retrieve the version specific information from the registry.

2.3 Component

Components are key entities under FRAME. A component may provide services that may be used by other components, i.e., the parent components. As an example, fig. 2 is software hierarchy of a GO game that consists of three components. Component `board` is the user interface and the root component that 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 [1] and is compiled as a native shared library. 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 is less competitive to play, but requires less computation and memory resources. The `audio` component may play background music and 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 does not have a 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. 1.

As mentioned earlier, a component may have multiple versions of implementation with different requirements and qualities of service. Resource requirements

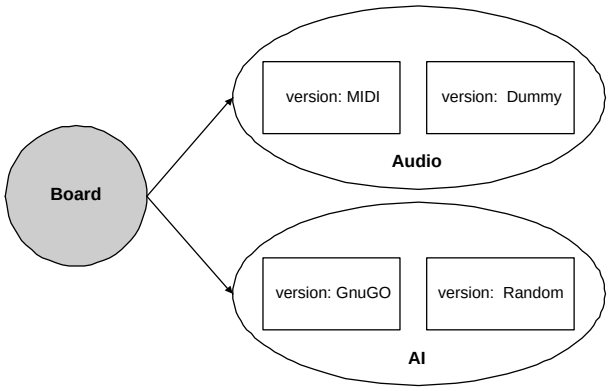

Figure 2. Software hierarchy of the GO game

of each service may vary among versions. The collection of these resources required for a component i with version j are denoted as R_{ij}. For example, the required resource for the `MIDI` version 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, its overall resource requirement would be $R_{31} = \{$"A sound device exists.", "Platform is not battery powered."$\}$. Instead of specifying the resource requirement, the `dummy` version 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.

Each component has parameters, p_i^k where p_i^k is the kth parameter of component i, to characterize its performance or quality for implementations. These parameters should be in some default finite domain that may be specified by parent components or the application launcher. The collection of default parameters within a domain for a component i with version j is 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, and power consumption of first move. Thus, these two metrics could be represented by two parameters p_1^1 and p_1^2 respectively. The maximum response time might be

	version 1	version 2
component 1	`board`	N/A
component 2	`GnuGO AI`	`Random AI`
component 3	`MIDI audio`	`dummy audio`

Table 1. Label of each component

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 involve more than 100 moves. These metrics determine the default domain of p_1^1 and p_1^2. Therefore, the collection of parameter default domains for both component `board` is $P_{11} = \{\text{“}0 \leq p_1^1 \leq 10\text{”}, \text{“}0 \leq p_1^2 \leq 0.5\text{”}\}$.

Because the components are hierarchical, the parameters are also hierarchical. While the components are connected via services, the parameters are connected by *internal connectors* and *external connectors*. For a component, its parameters may be not independent. The relations between the parameters, by analyzing or modeling, are specified as internal connectors, denoted as D_{ij}. For the component `board`, a longer response time, p_1^1, will usually 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 connectors for the component `board` is $D_{11} = \{\text{“}p_1^1 \geq 20p_1^2\text{”}\}$.

The developers of the different components may be interested in different performance metrics. These parameters may be dependent or independent. Also, a parent component may need to specify the parameter domain of its child components. For example, although the `board` component and `AI` component developers may be interested in the response time and power consumption of the first move, both 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 external 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 relation between the two power consumption parameters may be modeled as a linear relation, $p_1^2 \leq 1.1p_2^2$. Thus, external connectors between the parameters of the component `board` and its child component, the `AI`, is $M_{11} = \{\text{“}p_1^1 \leq 1.1p_2^1\text{”}, \text{“}p_1^2 \leq 1.1p_2^2\text{”}\}$.

Each element in R_{ij}, P_{ij}, D_{ij}, and M_{ij} is called a *constraint* that should be specified in the form of a predicate and implemented as a boolean function in Java. 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.

2.4 Root component

The root component of the software hierarchy is a special component and the starting point of component assembling. While it shares the some common properties as a regular component, such as the specification of R_{ij}, P_{ij}, D_{ij}, and M_{ij}, it has only one provided service, `main`, which is the starting point of the application execution and will be called by the launcher after the component assembling.

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

2.5 Component stub

Instead of running on a single machine, the components of applications may be distributed automatically to multiple platforms based on available resource and user specified constraints. For two non-distributed component objects, A and B, executing on a single platform, they actually execute within a process and communicate with each other through the regular method invocation mechanism within the process. For two distributed component objects executing on two different platforms, instead of using the real component B, component A actually interacts with the stub of the 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 associated object, which is B, to `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.

2.6 Component assembling

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* $\mathbb{S}_{v_1 v_2 \ldots v_i}$. 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* and components $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_i$ with version $v_1, v_2, \ldots, v_i$ respectively will form a feasible program. Before a program is executed, it needs to search for the appropriate version of each involved component through a process called *component assembling*. Whether a program is able to execute will depend on whether the user specified parameters and the computing environment can produce a feasible program; if not, the program will not execute.

The first step of the assembling process is to construct all possible software constraints $\mathbb{S}_{v_1 v_2 \ldots v_i}$, which is done by resolving the software hierarchy, loading each version of the involved component and building component constraint $\mathbb{C}_{ij}$ from loaded component. After constructing all possible software constraints, then the remaining assembling process is basically a constraints solving problem; that is, finding a feasible software constraint from all possible software constraints, which will in turn give the corresponding feasible program. Once the feasible program is found, the appropriate version 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 ($\mathbb{S}_{111}$ and $\mathbb{S}_{112}$, or $\mathbb{S}_{121}$ and $\mathbb{S}_{122}$) is 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.

For distributed applications, the component assembling process will distribute components to specified platforms before constructing software constraints. 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 assembling process for distributed applications is to find a feasible distribution from all possible distributions.

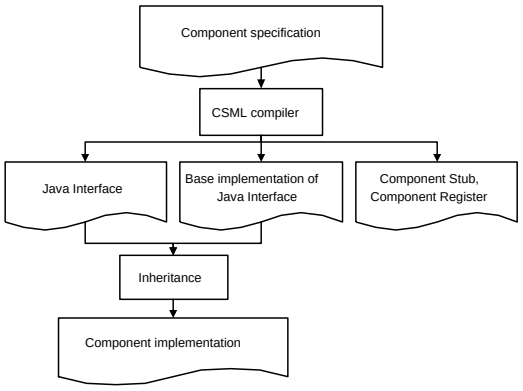

Figure 3. Flow chart of using `CSML` to develop a component

2.7 Application launcher

As mentioned earlier, an application is not directly invoked by a user but via a special program called the *launcher*. The launcher will take user specified constraints of software, i.e., parameters 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, and then start the assembling process.

To execute a distributed application, users only needs to specify a set of, instead of one, target platforms. Because we want users to interact with the application on a certain platform 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 `CSML`

While `FRAME` provides the framework to allow component-based software to be assembled before execution, the usage of the `FRAME` APIs may not be straightforward. Instead of implementing in Java code directly, developers specify services and constraints in `CSML` (Component/constraint Specification Markup Language) that is an XML-based markup language. As shown in fig. 3, `CSML` will generate a component interface and a base class of component implementation in Java from the specification and component developers only need to inherit the base class to implement the component. `CSML` will also generate the component stub and the component register that is used to register component information in the registry.

3.1 Root component specification example

```
 1 <root-component name="board" application="GO"
 2  uri="http://192.168.1.111/JavaGO.jar"
 3  registry-host="192.168.1.111">
 4  <parameter name="response_time" id="p11"
 5   value-type="int" upper="10" lower="0"> ...
 6  </parameter>
 7  <parameter name="power_consumption" id="p12"
 8   value-type="float" upper="0.5" lower="0"> ...
 9  </parameter>
10  <internal-connector id="f1">
11   <from-current parameter-id="p11" alias="var1" />
12   <from-current parameter-id="p12" alias="var2" />
13   <definition> return ^var1# >= 20 * ^var2#;
14   </definition>
15  </internal-connector>
16  <child-component name="AI" id="c1"
17   registry-host="192.168.1.111" />
18  <child-component name="audio" id="c2"
19   registry-host="192.168.1.111" />
20  <external-connector id="f2">
21   <from-current parameter-id="p11" alias="var1" />
22   <from-child child-id="c1"
23    parameter="response_time" alias="var2" />
24   <definition> return ^var1# <= 1.1 * ^var2#;
25   </definition>
26  </external-connector>
27  ...
28  <main>
29   <instance name="Ai" child-id="c1" />
30   <instance name="Audio" child-id="c2" />
31   <required-service service="play" child-id="c1" />
32   <required-service service="playAudio"
33    child-id="c2" />
34   <definition> ... </definition>
35  </main>
36 </root-component>
```

Table 2. `CSML` **for the component** `board`

Table. 2 is the specification of the root component `board` in `CSML`. The element *root-component* specifies the name of root component (in attribute *name*), hostname of component registry (in attribute *registry-host*), and location (in attribute *uri*) in lines 1-3. Parameters are specified in element *parameter* with name and their range (in attribute *upper* and *lower*), lines 4-9. Child components are specified in element *child-component*, lines 16-19. Other constraints are specified in element *internal-connector*, lines 10-15, and *external-connector*, lines 20-26. Their definition are specified in the form of Java code, which should return a boolean value, i.e., true if the constraint is satisfied. For example, the definition of the constraint $p_1^1 \geq 20p_1^2$ is specified as "`return ^var1# >= 20 * ^var2#;`" with var1 and var2 being aliases of parameter response time p_1^1 and power consumption p_1^2 respectively. There is also a specification for the `main` function in lines 28-35. In the 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`.

3.2 Component specification example

```
 1 <component name="audio" ...">
 2  <general>
 3   ...
 4   <provided-service id="s1">
 5    <declaration method-name="playAudio"
 6     return-type="void" /> ...
 7   </provided-service> ...
 8  </general>
 9  <customized version="MIDI"
10   uri="http://192,168.1.111/midi.jar">
11   <constraint-definition constraint-id="s1">
12    <resource-requirement>
13     <resource name="sound-device" ...>
14      <built-in><sound /></built-in>
15     </resource>
16    </resource-requirement> ...
17   </constraint-definition>
18  </customized>
19  <customized version="dummy"
20   uri="http://192.168.1.111/dummy.jar"> ...
21  </customized>
22 </component>
```

Table 3. `CSML` **for the component** `audio`

Table. 3 is the specification of the component `audio` in `CSML`. The format is very similar to the root component specification. In addition to constraints, the provided service of a component is specified in the element *provided-service*. The element *general*, lines 2-8, specifies the information that is version independent and element *customized*, lines 9-21, specifies the version dependent information where, except version number, there are resource requirements of `s1` for each version with `s1` corresponding to provided service `playAudio`. In the example, the `MIDI` version specifies a sound device as a resource requirement in lines 11-17.

3.3 Launcher specification example

```
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 </launcher>
```

Table 4. `CSML` **for the GO game launcher**

`CSML` also allows users to specify target platforms and the intended performance of the application, and generate the application launcher. Table. 4 is the specification of the launcher. It specifies the application to execute on two target platforms. The component `board` will be executed on the first platform and the other two

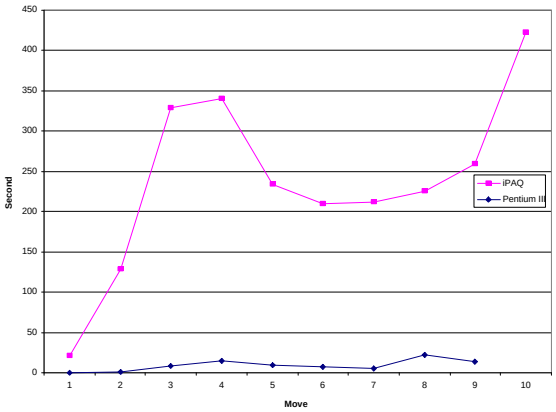

Figure 4. Response time of the GnuGO AI engine

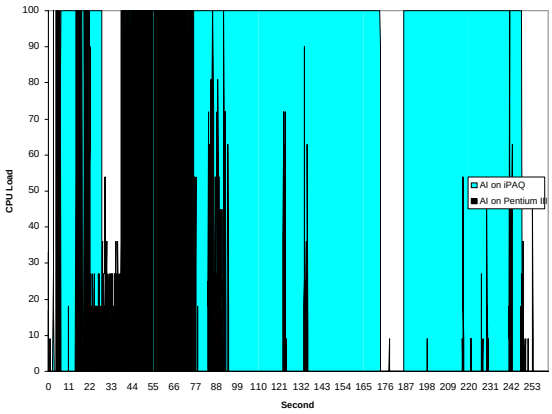

Figure 5. CPU load of the GnuGO AI engine

may be distributed to the second platform. It also specifies intended performance and output quality in the element *constraint*, lines 5-8, which corresponds to the parameters of application. Once the constraints are specified, the range of the root component parameters, p_1^1 and p_1^2, will be replaced by the user specified range and the new range will be used during the component assembling.

4 Application demonstration

In this section, we describe a demonstration of `FRAME` using the GO game described in section 3. The component `board` is inherited from JavaGO, a Java Applet that is originally developed by Alain Papazoglou [11]. 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. 4 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. 5 compares the CPU load of the first several moves with both `board` component and the `GnuGO AI` component on the iPAQ, and `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 [5]. 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 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.

5 Related work

Existing middleware specifications such as CORBA [9] and DCOM [4] simplify the construction of component-based distributed applications. Nonetheless, they do not define strategies for customizing themselves flexibly to dynamic environments. Recent research in reflective middleware tries to overcome this limitation. For example, COMERA [13] provides a framework based on Microsoft COM that allows users to modify several aspects of communication middleware at run-time. DynamicTAO [8] supports on-the-fly reconfiguration on CORBA ORB based on The ACE ORB(TAO) [12] that is the open source CORBA-compliant ORB.

Numerous architectural description languages (ADLs) have been developed to capture the key design properties of a system and provide mechanisms for specifying component requirements. For example,

Wright [2] supports the specification and analysis of interactions between components. ACME [6] supports the interchange of architectural descriptions between a variety of architectural design tools, and representation and satisfaction of constraints.

6 Conclusion and future work

Under heterogeneous environments, resource critical software should be aware of the heterogeneity and produce the needed performance on the available resources. FRAME is an adaptive software framework developed in Java, which allows Java applications to be assembled "on the fly" and, therefore, be customized to specific computing environments. CSML allows people to specify the high-level characteristics of component and will generate Java code to work under FRAME.

Currently, FRAME can only customize applications before execution. This implies the assumption that the computing environment does not change much such that feasible software constraints become invalid. Nonetheless, future computing systems have been envisioned as omnipresent [14], pervasive [3], and nomadic [7]. FRAME's assumption will not be appropriate under these environments. We plan to explore the possibility that applications may adapt themselves to new environment transparently at run-time. Such a run-time self-adaptive application brings several design issues [10], such as open or closed-adapted, type of autonomy, frequency, and cost effectiveness. Furthermore, to specify these design decisions and reflect them into an application is a challenge. We also plan to apply the extended framework to a more complicated embedded systems such as robots.

There are some other improvements for performance. For instance, constraints solving is a performance bottleneck during component assembling. The current implementation uses a brute force algorithm to solve the constraints, which basically will iterate through each possible valid value of parameters to check if all the constraints are satisfied. We will define a clear interface for constraint solving algorithm, so that different constraint solving algorithm could be implemented and be selected by the application launcher. Furthermore, the component assembling results may be cached. For subsequence execution of the same software, it would be possible to accelerate the assembling process by "perturbating" the results, if the computing environment does not fluctuate significantly.

References

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

[2] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, Jan. 1997. Issued as CMU Technical Report CMU-CS-97-144.

[3] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski. Challenges: An Application Model for Pervasive Computing. In *Proceeding of the 6 th Annual ACM/IEEE Intl Conf. Mobile Computing and Networking (MobiCom2000)*, pages 266–274, Boston, MA, Aug. 2000.

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

[5] 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. *Commun. ACM*, pages 49–56, June 1998.

[6] D. Garlan, R. T. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, Nov. 1997.

[7] T. Kindberg and J. Barton. A Web-Based Nomadic Computing System. Technical Report HPL-2000-110, HP Labs, Palo Alto, CA 94304, USA, Aug. 2000. Available at *http://www.hpl.hp.com/techreports/2000/HPL-2000-110.pdf*.

[8] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell. Monitoring, Security, and Dynamic Confi guration 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, Apr. 2000. Springer-Verlag.

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

[10] 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.

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

[12] 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.

[13] Y.-M. Wang and W.-J. Lee. COMERA: COM Extensible Remoting Architecture. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 79–88. USENIX, Apr. 1998.

[14] M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM*, 36(7):74–84, July 1993.