Adaptive Soft Real-Time Java 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

Traditional real-time software development methodologies require full knowledge of the resource capability of target platforms during the development stage. However, such knowledge is not always feasible within heterogeneous environments. Resource heterogeneity offers a new level challenge to real-time software portability, and changes in computing environments make application behavior even more unpredictable. As a consequence, real-time software has to probe dynamically real-time capabilities to adapt itself to different computing environments, and to respond to environment changes. We propose an approach such that the assembly of an application is postponed to execution time so that the application may be customized by gathering real-time capability information from the environments. After the application is launched, it may be finely tuned remotely by an interactive steering environment in response to changes in the computing environment. Finally, an application, the MPEG video player, is used to evaluate this approach under various computing environments.

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 knowledge 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.

Therefore, the following two problems must be addressed in order to achieve such a higher level perfor-

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

mance portability so that soft real-time software may be executed under a heterogeneous environment with as little human intervention as possible:

- How do applications customize themselves to a broad range of computing environments?
- How may applications be repaired in response to changes in computing environment?

In this paper, we address the first problem by applying the adaptive software framework, FRAME and CSML (Component/constraint Specification Markup Language) to soft real-time Java applications under different computing environments. After the application is executed, it may be finely tuned remotely by an interactive steering environment, brew, in response to changes in the computing environments. We use the application, the MPEG video player, to evaluate FRAME and brew on desktop computers with different resource capabilities. Finally, the last two sections will give a summary, survey of related work, and then discuss potential future investigations.

2 Adaptive software framework

2.1 FRAME

FRAME [5] is a framework, based on Java techniques, to help people develop and deploy Java applications that can customize themselves, based on specified constraints, to multiple computing environments. Under FRAME, an application should be composed of components that are implemented Java packages and will not be assembled until execution. Each component provides services to cooperate with other components and constraints that are criteria used to assemble an application. The application may be executed as a process on a single machine, or distributedly on multiple machines.

As an example of FRAME, fig. 1 is the software hierarchy of the MPEG video player that consists of four components. 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 a database server called the *component registry* and, of course, the whole software hierarchy has to be resolved during run-time by querying the component registry.

In the example, component player is the root component that needs service from its child component,

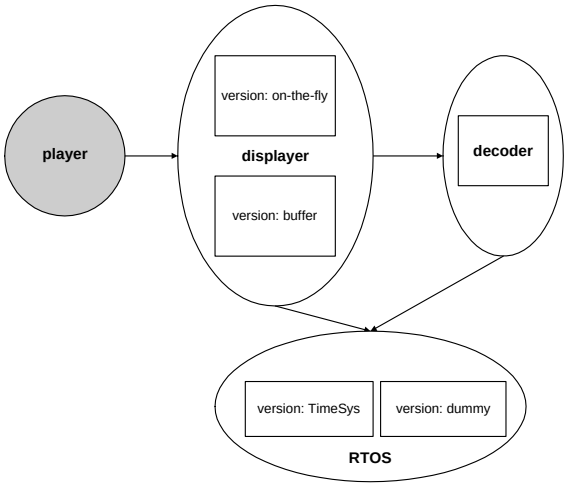

Figure 1. Software hierarchy of an MPEG video player

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[13][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 on the displayer and the decoder. To simplify the example, each implementation of each component is labelled according to table. 1.

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

Table 1. Label of each component

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

Since a component may have multiple versions of implementation, there may be more than one possible combinations of components for an application. The collection of constraints of each possible combination is called a *software constraint*. The software constraint of each combination is unique; that is, the mapping of combinations and software constraints is one-to-one. For example, there are total four possible combinations of components for the MPEG video player, corresponding to four possible software constraints of the MPEG video player.

Before a program is executed, it needs to be assembled from the feasible version of each component, in which all the constraints of the corresponding software constraint are satisfied. With the one-to-one property of the combinations and software constraints mapping, the process to find feasible components can be reduced to a constraints solving problem in the following steps.

- Resolve the software hierarchy by querying the component registry.
- All possible combinations will be constructed from the software hierarchy.
- All the corresponding software constraints will be built from the combinations.

By solving the constraints, a feasible software constraint, if exists, might be found, which will in turn give the corresponding feasible combination.

2.2 Component constraints

For each implementation of a component, developers not only need to specify the services but also the constraints. There are four different categories of constraints that may be specified; that is, resource requirements, parameters that characterize performance or quality of implementation, internal constraint connectors, and external constraint connectors.

For example, the required resource for the `timesys` version of the `RTOS` 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` version is specified as the "default" version of the `RTOS`; that is, if the required resources of other versions are not satisfied, the default version will be used.

For parameters, `displayer` component developers may be interested in the time interval, in milliseconds, between frames displayed and the image quality. These two metrics could be represented by two parameters p_2^1 and p_2^2 with p_i^k being the kth parameter of component i. The time interval between frames displayed might be set between 40 milliseconds and 200 milliseconds, and the image quality is divided into eight different levels. These metrics determine the default domain of p_2^1 and p_2^2. Therefore, the collection of parameter default domains for the both versions of the `displayer` are $\{\text{“}40 \leq p_2^1 \leq 200\text{”}, \text{“}1 \leq p_2^2 \leq 8\text{”}\}$.

As the component dependencies are defined via services, the parameter constraints are connected by special constraints called *internal constraint connectors* and *external constraint connectors*. For a component, its parameters may be not independent. The relations between the parameter constraints within a component, by analyzing or modeling, are specified as internal constraint connectors. For the component `displayer`, 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, $7p_2^1 - 160p_2^2 \geq 120$. Therefore, the collection of internal constraint connectors for the both versions of the `displayer` are $\{\text{“}7p_2^1 - 160p_2^2 \geq 120\text{”}\}$.

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, the `decoder` developers may be interested in the time interval, in milliseconds, between frames decoded, p_3^1, and the image quality, p_3^2. Both the `displayer` and the `decoder` may use the same metric for the image quality, i.e., $p_2^2 = p_3^2$, but the time to decode a frame is different than the time to display a frame. The relations between parameter constraints of parent and child components are specified as external constraint connectors. For the `on-the-fly` version 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 the `on-the-fly` version and the `decoder` is $\{\text{“}p_2^1 \leq p_3^1 + 70\text{”}, \text{“}p_2^2 = p_3^2\text{”}\}$. On the other hand, for the `buffer` version 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 can 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 version` of the `displayer` is $\{$"$190 \leq p_3^1 \leq 1000$", "$p_2^2 = p_3^2$"$\}$.

The version of the `displayer` 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.

2.3 Constraint specification in CSML

```
 1 <component name="displayer" steerable="on"
 2  registry-host="192.168.1.111"
 3  uri="http://192.168.1.111/displayer.jar">
 4  <general>
 5   ...
 6   <parameter name="time" id="p21" value-type="int"
 7    upper="200" lower="40"> ...
 8   </parameter>
 9   <parameter name="quality" id="p22"
10    value-type="int" upper="8" lower="1"> ...
11   </parameter>
12   <internal-connector id="f1">
13    <from-current parameter-id="p21" alias="var1" />
14    <from-current parameter-id="p22" alias="var2" />
15    <definition>
16     return 7 * ^var1# - 160 * ^var2# &gt;= 120;
17    </definition>
18   </internal-connector>
19   <child-component name="decoder" id="c1" ... />
20   <external-connector id="f2">
21    <from-current parameter-id="p21" alias="var1" />
22    <from-child child-id="c1" parameter="time"
23     alias="var2" /> ...
24   </external-connector>
25   ...
26   <provided-service>
27    <declaration method-name="play">
28     <argument name="args" value-type="String[]"/>
29    </declaration>
30   </provided-service>
31   ...
32  </general>
33  <customized version="on_the_fly"
34   uri="http://192.168.1.111/on_the_fly.jar">
35   <constraint-definition constraint-id="f2">
36    <definition>
37     return ^var1# &lt;= ^var2# + 60;
38    </definition>
39   </constraint-definition>
40  </customized>
41  <customized version="buffer"
42   uri="http://192.168.1.111/buffer.jar">
43   <constraint-definition constraint-id="f2">
44    <definition>
45     return 190 &lt;= ^var2# &amp;&amp;
46            ^var2# &lt;= 1000;
47    </definition>
48   </constraint-definition>
49  </customized>
50 </component>
```

Table 2. CSML for component displayer

Instead of implementing them in Java code directly, developers specify services and constraints in CSML (Component/constraint Specification Markup Language) [5] that is an XML-based markup language. 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; the infrastructure needed to work under FRAME is generated by CSML.

Table. 2 is the specification of the component `displayer` in CSML. The element *component* specifies the component name (in attribute *name*), host-name of component registry (in attribute *registry-host*), and location (in attribute *uri*) in lines 1-3. Parameters are specified in element *parameter*, lines 6-11, with name and their range (in attribute *upper* and *lower*). Component dependencies are specified in element *child-component*. Internal constraint connectors and external constraint connectors are specified in element *internal-connector*, lines 12-18, and *external-connector*, lines 20-24, respectively. 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 in line 16, the definition of the constraint $7p_2^1 - 160p_2^2 \geq 120$ is specified as "`return 7 * ^var1# - 160 * ^var2# >= 120;`" with var1 and var2 being aliases of parameter time p_2^1 and quality p_2^2 respectively. The provided service is specified in the element *provided-service*. Finally, the element *general*, lines 4-32, specifies the information that is version independent and element *customized*, lines 33-40 and lines 41-48, specifies the version dependent information such as version number and definitions of version dependent constraints.

3 Brew

Brew is an interactive environment for collecting performance data and repairing constraints missed applications because of changes in environments. The overall architecture of the interactive environment is shown in Fig. 2. The instrumentation component consists of visual objects (VO) and a daemon, called `ism`. The control component consists of a user interface and `RemoteController`. To incorporate with the interactive environment, `RemoteControlManager` and `Instrument` need to be embedded in the applications. Furthermore, a daemon, call `exs`, will run concurrently with the application.

3.1 BRISK and the visual object framework

The instrumentation component uses PG$^{\mathrm{RT}}$ [2], an environment for integration of tools and systems for instrumentation, performance visualization, and analysis of complex real-time systems. Two parts of PG$^{\mathrm{RT}}$ used are BRISK and the Visual Object Framework (VO). BRISK uses `ism` and `exs` for communication between

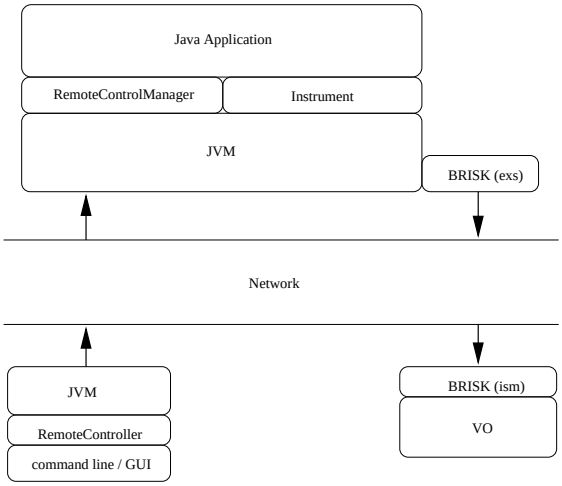

Figure 2. Architecture of `brew`

applications and VOs. VO is used to develop application specific performance visualizations, which includes processing and rendering of instrumentation data.

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

3.2 Remote controller

The control component is implemented in Java, so the application may be "steered" anywhere from any device regardless of its hardware architecture and operating system. It consists of two subcomponents, `RemoteController` and `RemoteControlManager`.

`RemoteController` is the infrastructure of remote controllers. Its main responsibility is to parse the command and send the parsed information to `RemoteControlManager`. Different user interfaces, such as a command line, GUI, or Java applet under web browser, may be used on top of `RemoteController`.

`RemoteControlManager` handles the steering commands from `RemoteController`, i.e., finds and invokes the requested methods with appropriate arguments. The application need to export the objects that would be remotely controlled to `RemoteControlManager`, and then all the public methods of the exported objects may be invoked remotely by `RemoteController`. Thus, the behavior of the application may be changed via these public methods.

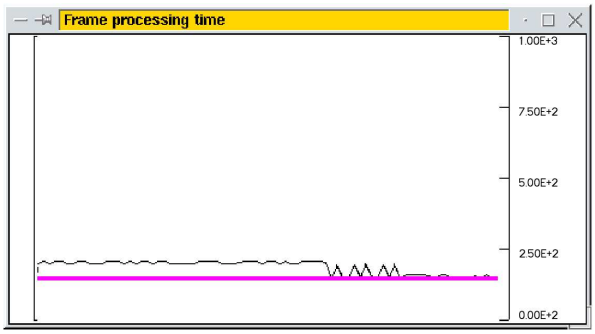

Figure 3. Frames processing time of the MPEG player on Linux PC

4 Demonstration

4.1 Application modifications

As described in section 3, we decomposed the MPEG player, originally developed by Joerg Anders [1], into the component hierarchy as fig. 1. The root component `player` has two parameters, the number of frames played per second (p_1^1) and image quality (p_1^2), and they are specified within the domains $[5, 30]$ and $[1, 8]$ respectively. We 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`. The instrumental metric is the time to display a frame. The steered parameters are the number of frames played per second and the quality of image. The first two experiments, which demonstrate the different versions of displayer are chosen based on the time constraint, are conducted on a Linux PC with 900 MHz Pentium III Processor and 256 MB RAM with two different VM engines, just-in-time (JIT) and interpreter respectively. The last experiment demonstrates that `FRAME` will use the special functionality of the TimeSys real-time Linux.

4.2 MPEG player on JIT engine

With JIT optimization, Java VM has the ability to decode the frame fast enough so the time constraint $p_2^1 \leq p_3^1 + 70$ can be satisfied and the `on-the-fly` version of displayer component will be used. After the components being assembled, the MPEG player begins to execute with parameters that satisfy the software constraints. The value of parameter p_1^1, number of frames

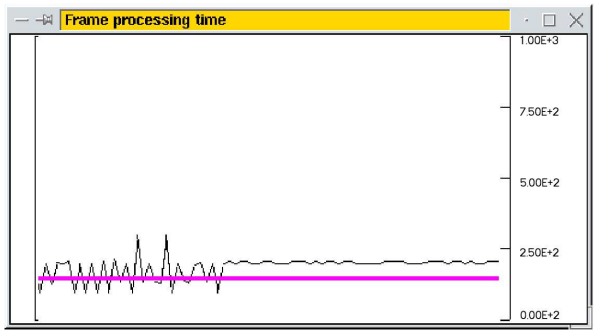

Figure 4. The MPEG player is steered to meet the time constraint

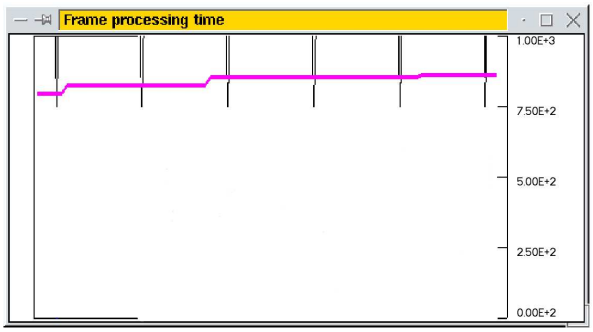

Figure 5. Frames processing time of the MPEG player on Java VM with JIT engine

played per second, is 5 which in turn gives displaying period 200 ms. The value of parameter p_1^2, image quality, is 8. Fig. 3 is the screen shot of the VO for the Java MPEG player executing on a VM 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. 4, 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 so that the time constraint can be satisfied.

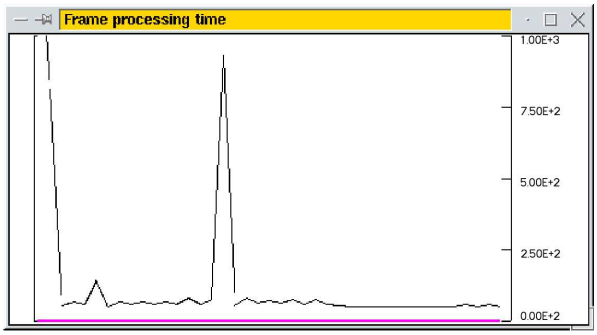

Figure 6. Frames processing time of the MPEG player on Java VM with interpreter engine

4.3 MPEG player on interpreter engine

We tested the same application on a Java VM with the interpreter engine. As shown in fig. 5, 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` version of the displayer component. Fig. 6 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.

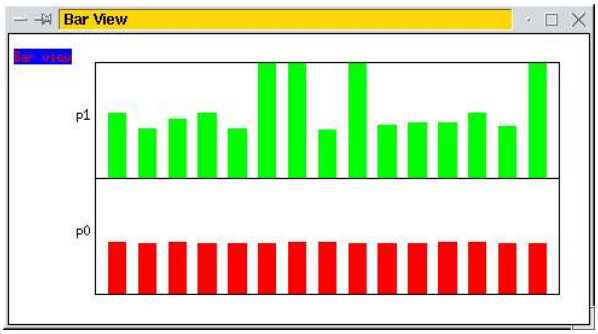

Figure 7. Time interval to display a frame for the MPEG player with and without CPU reservation

4.4 MPEG player on TimeSys real-time Linux

Finally, we demonstrate that FRAME will take advantage of special functionality of computing environments. 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. 7 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 same (100 ms) for $P0$, but varies significantly and could be up to more than 200 ms for $P1$.

5 Related work

There are two groups for defining real-time Java specification, The Real-Time for Java Experts Group (RTEG) [11] and the Real-Time Java Working Group (RTJWG) [12]. RTEG's specification tends to preserve compatibility with existing Java run-time semantics. They intended not to support portability of real-time Java applications because they want the difference between underlying real-time operating systems to be reflected at the Java level [7]. Since it is tightly coupled with the underlying operating system, the VM has to be re-implemented. On the other hand, RTJWG's specification separates VM into a baseline VM, which could be a generic off-the-shelf VM, 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 [7].

Bernat et al. [3] illustrates the challenges of using Java byte code to undertake worst-case execution time analysis. A prototype tool is being developed that analyses Java Class files and that identifies and extracts the annotations in the Java byte code.

Reflection has been widely adopted in language design, as witnessed by the Java Core Reflection API and its extension, such as Kava [17] and Dalang [16]. Reflection is also increasingly being applied to a variety of other areas including distributed system [15] and middleware, such as COMERA [14] and DynamicTAO [6]. `FRAME` is a framework that provide mechanisms to execute reflective applications. `CSML` may help people develop reflective applications, which will automatically add reflection to applications by only specifying the high level characteristics of components.

Program steering has been defined as the capacity to control the execution of long-running, resource-intensive programs. This may include modifying program state, managing data output, starting and stalling program execution, altering resource allocations etc. For example, SciRun [9] is a scientific problem-solving environment that provides the ability to interactively guide or steer a running computation. SciRun was designed initially for multi-threaded shared-memory multiprocessors. A distributed-memory version is being produced and threading is now used to hide latency and perform other tasks. The distributed laboratories project [10] addresses interactivity in a computationally diverse environment consisting of complex scientific applications, information brokers, and client sources through lightweight online steering and monitoring mechanisms, as well as decision mechanisms for controlling and optimizing data flow. The goal is aimed to be a distributed computational tool and focused on low monitoring latency and perturbation.

The important distinction between `brew` and the above steering systems is that `brew` adds reflection to the steering mechanism based on the Java Core Reflection API. It provides an interpreted language that allows users to write scripts for more complicated application steering.

6 Conclusion and future work

We demonstrate how the adaptive software framework, `FRAME` and `CSML`, may be applied to soft real-time Java applications under heterogeneous environment. `FRAME` provides the necessary APIs to allow applications to be built from constraints on the fly. `FRAME` does not have run-time performance impact on applications because the assembly has finished before execution. `CSML` allow people to use XML to specify component interfaces, constraints, and will generate Java code to preserve the plug-in compatibility. The interactive steering environment, `brew`, allows Java applications to export their raw performance information to be rendered and visualized. Users then may conclude the application performance and repair the constraints missed applications.

Currently, `FRAME` can only assemble applications before execution. This implies the assumption that the computing environment does not change much such that the feasible software constraints become invalid and cannot be repaired by `brew`. Nonetheless, fu-

ture computing systems have been envisioned as nomadic [4]. FRAME's assumption will not be appropriate under these environments. We plan to extend the reflectivity to dynamic environments that applications may be re-assembled for new environment transparently at run-time. Such a run-time reflective application brings several design issues [8], 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 plan an improvement for brew. The applications and brew form a closed loop, where the gap between the VO and the remote controller is bridged by a human. We plan to build a interface that allows an adaptive algorithm to easily fill the gap to form a closed loop that will automatically send a control command based on the instrumentation results from VO. The development of the adaptive algorithm is separated from the applications and can be easily customized.

References

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

[2] A. Bakić, M. W. Mutka, and D. 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, Dec. 1997.

[3] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. In *Proceedings of the 12th EuroMicro Conference on Real-Time Systems*, Stockholm, June 2000.

[4] 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*.

[5] R.-S. Ko and M. 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, Apr. 2002.

[6] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. 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, Apr. 2000. Springer-Verlag.

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

[8] 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.

[9] 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, Jan. 1998.

[10] B. Plale, G. Eisenhauer, K. Schwan, J. Heiner, V. Martin, and J. Vetter. From Interactive Applications to Distributed Laboratories. *IEEE Concurrency*, 6(2):78–89, /1998.

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

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

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

[14] 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.

[15] T. Watanabe, A. Noriki, and K. Shinbori. A Reflective Framework for Reliable Mobile Agent Systems. In W. Cazzola, S. Chiba, and T. Ledoux, editors, *On-Line Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures*, June 2000. Available at *http://www.disi.unige.it/person/CazzolaW/ewrma2000-proceedings.html*.

[16] I. Welch and R. Stroud. From Dalang to Kava - the Evolution of a Reflective Java Extension. In *Proceedings of Second International Conference on Metalevel Architectures and Reflection*, June 1999.

[17] I. Welch and R. Stroud. Kava - Using Bytecode Rewriting to add Behavioural Reflection to Java. In *Proceedings of USENIX Conference on Object-Oriented Technology*, 2001.