A COMPONENT-BASED APPROACH FOR ADAPTIVE SOFT REAL-TIME JAVA WITHIN HETEROGENEOUS ENVIRONMENTS*

REN-SONG KO † AND MATT W. MUTKA‡

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. Although Java provides platform independent code level portability, resource heterogeneity offers a higher level challenge to real-time software portability. Furthermore, 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 a component-based 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.

Key words. adaptive software, components, Java, real-time, steering, XML

AMS subject classifications.

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 maximizing performance and compliance with real-time constraints. It is very likely that future computing environments are composed of a wide range of devices, with diverse hardware architectures, operating systems, and purposes, interconnected via wired or wireless networks. Such a development methodology for porting software to each platform is time and cost consuming.

Software portability may be classified into several levels based on amount of effort required for people to deploy and execute an application on different platforms. 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. Besides, it may be sometimes impractical for vendors to release the source code.

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. It mitigates the difficulty for software deployment and use, because recompilation for a target platform is not necessary. The Java platform promises “Write Once Run Anywhere,” which provides platform independent code level portability. Although native software offers performance benefits over Java, the benefits of platform independency often outweighs the performance because Java may substantially reduce software design cycle time, learning curve, development and maintenance costs.

While Java 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

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

† Department of Computer Science and Engineering, 3115 Engineering Building, Michigan State University East Lansing, MI 48824-1226 (korenson@cse.msu.edu).

‡ Department of Computer Science and Engineering, 3115 Engineering Building, Michigan State University East Lansing, MI 48824-1226 (mutka@cse.msu.edu).

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 performance 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 the computing environment?

In this paper, we address the first problem by applying the adaptive software framework, FRAME, to soft real-time Java applications under different computing environments. We also introduce CSML (Component/constraint Specification Markup Language) that may help developers develop applications under FRAME. 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, and compare the performance of the MPEG video player with and without using FRAME. Finally, the last two sections will give a summary, survey of related work, and then discuss potential future investigations.

2. Adaptive software framework: FRAME. 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 assembly process is possible. Because of the automatic assembly 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.

FRAME [8] is a framework, based on Java techniques, to help people develop and deploy Java applications that may customize themselves, based on specified constraints, to multiple computing environments. Together with CSML, FRAME was developed with the following goals:

- Developers may specify constraints for software, and template code may be generated from the specification.
- Users may specify the intended quality of service before executing the application.
- Applications may be automatically distributed to single or multiple target platforms.
- Applications may probe the available resources before execution to adapt themselves to the computing environment.

An application should be composed of components under FRAME. Each component has constraints or resource requirements and cooperates with other components through an unambiguous interface. Furthermore, various implementations with different resource requirement may be developed and used for specific computing environments and are usually stored in a server called the *component repository*.

Before a component may be used, it must register some information to a database server called the *component registry*. The information, such as the available implementations of components, child component dependency and the component repository locations, are necessary during the component assembly. Since new implementations of components with different resource requirements might be developed after the software is developed, the software should not have any knowledge about the implementation-dependent information in advance. It must retrieve the implementation specific information from the registry.

The application may be executed as a process on single machine, or distributedly on multiple machines, which require a special component implementation called the *component stub* to handle the communication between multiple machines. Besides, the application is not directly executed by users but via an entity called the *launcher*, which will trigger the initialization, or called *component assembly*, and the execution of the application. Users specify performance constraints in the application launcher and execute it on the specified target platforms. The launcher will query the component registry about the components, and will load each implementation of components from the component repository. During the component assembly process, the launcher will test them and assemble the appropriate implementation of components with appropriate parameters. After the component assembly process, the

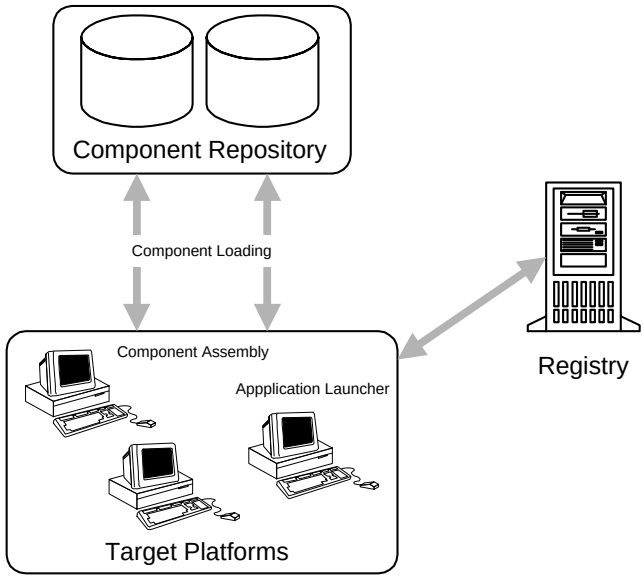

FIG. 2.1. *The interaction between target platforms, component registry and repository during the execution of an application. The application may be executed on one or more target platforms via the launcher. The launcher will query the component registry about the components, load each version of components from the component repository, and start component assembly before application execution.*

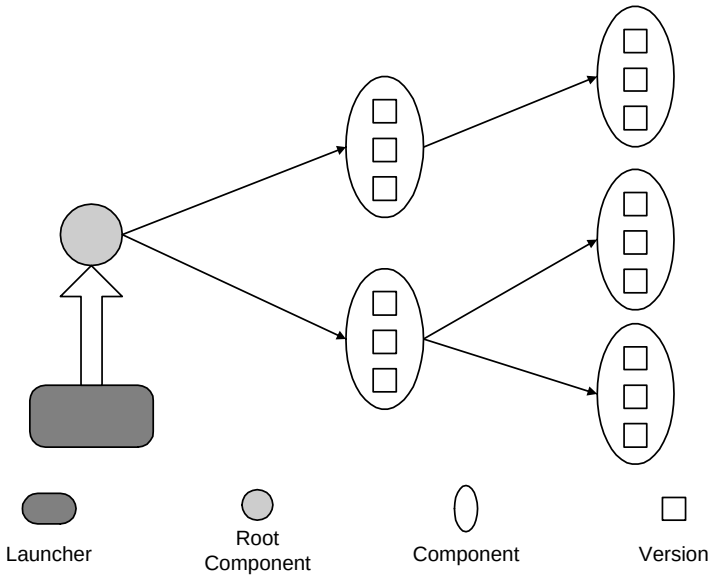

FIG. 2.2. *Software hierarchy under* FRAME

application will begin to execute. To execute a distributed application, users only needs to specify a set of, instead of one, target platforms, and FRAME will automatically distribute the appropriate components to appropriate platforms.

Fig. 2.1 shows how an application under FRAME interacts with the component registry and the repository. FRAME provides the APIs for communication between target platforms and registry, component loading, component assembly, and component initialization. Details about FRAME will be explained with an example, the MPEG video player in the following subsections.

2.1. Component. Components are key entities under FRAME. Fig. 2.2 is a general software hierarchy based on components. The dependency of components is defined via services; that is, a parent component requires services from its child components, and vice versa. The performance characteristics of child components may be specified by parent

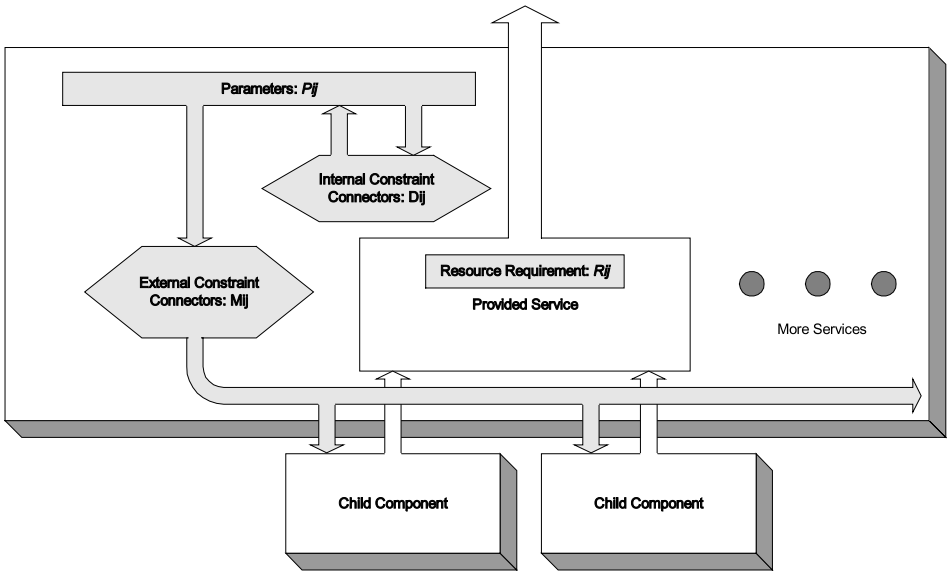

FIG. 2.3. *Component Architecture*

components, and the root component may be specified by users via an application launcher. 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 an application.

One special version of a component is the 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.

Fig. 2.3 shows a general component architecture. A component may provide 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 of `FRAME`, fig. 2.4 is the software hierarchy of the MPEG video player 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 may only be used on the TimeSys real-time Linux platform[16][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. 2.1.

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

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

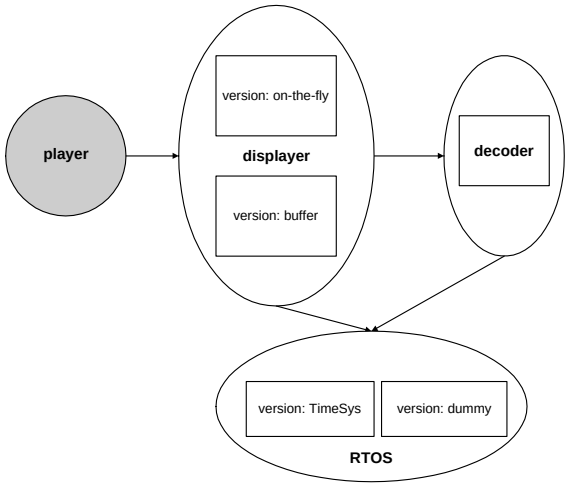

FIG. 2.4. *Software hierarchy of an MPEG video player*

is, resource requirements R_{ij}, parameters P_{ij} that characterize performance or quality of implementation, internal constraint connectors D_{ij}, and external constraint connectors M_{ij} for a component i with version j.

For example, the required resource for the `timesys` version of the `RTOS` is TimeSys real-time Linux OS. Therefore, its resource requirement would be $R_{41} = \{$"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 $P_{21} = P_{22} = \{$"$40 \leq p_2^1 \leq 200$", "$1 \leq p_2^2 \leq 8$"$\}$.

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 $D_{21} = D_{22} = \{$"$7p_2^1 - 160p_2^2 \geq 120$"$\}$.

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.

	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 2.1
Label of each component

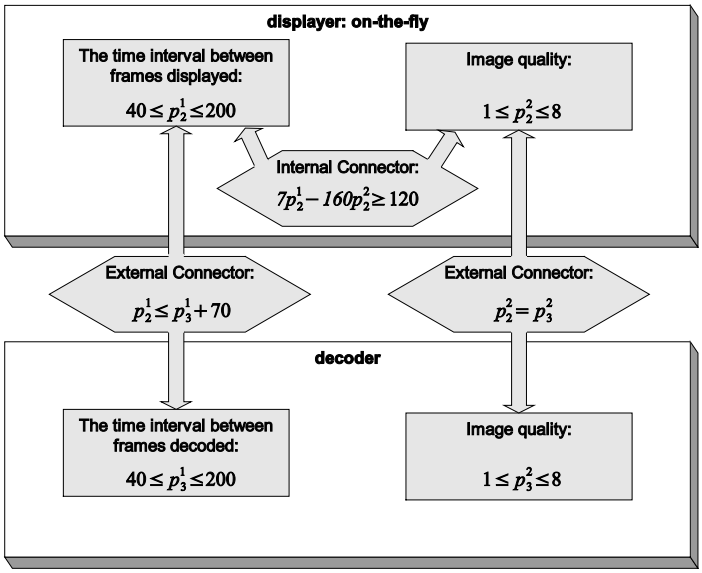

FIG. 2.5. *The parameters of the* `displayer` *with the* `on-the-fly` *version and the* `decoder` *are connected by connectors.*

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 $p_2^1 \le p_3^1 + 70$; 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 $M_{21} = \{\text{“}p_2^1 \le p_3^1 + 70\text{”}, \text{“}p_2^2 = p_3^2\text{”}\}$. The relations between parameters and connectors for the `on-the-fly` version of `displayer` are illustrated in fig. 2.5. 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 may specify a constraint whether the `decoder` may decode frames in some specific time interval, such as $190 \le p_3^1 \le 1000$. 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 the 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 $M_{22} = \{\text{“}190 \le p_3^1 \le 1000\text{”}, \text{“}p_2^2 = p_3^2\text{”}\}$.

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 $\mathfrak{C}_{ij}$ for component i with version j is defined as the set of all constraints, or

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

and the feasible component i is defined as the component i with version j such that all constraints of $\mathfrak{C}_{ij}$ are satisfied.

2.3. Component Assembly. Suppose that an application requires components $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_i$. The component $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_i$ with version $v_1, v_2, \ldots, v_i$ form a possible combination, denoted as $\mathbb{C}_{v_1 v_2 \ldots v_i}$, for the application. Regardless of constraints, each $\mathbb{C}_{v_1 v_2 \ldots v_i}$ is executable since components cooperate via services. The collection of $\mathfrak{C}_{iv_i}$ 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* $\mathbb{S}_{v_1 v_2 \ldots v_i}$. The software constraint of each combination should be unique; that is, the mapping of combinations, $\mathbb{C}_{v_1 v_2 \ldots v_i}$, and software constraints, $\mathbb{S}_{v_1 v_2 \ldots v_i}$, is one-to-one. For instance, all four possible software constraints of the MPEG video player, corresponding to all four possible combination of components for the MPEG video player, will be $\mathbb{S}_{1111} = \mathfrak{C}_{11} \cup \mathfrak{C}_{21} \cup \mathfrak{C}_{31} \cup \mathfrak{C}_{41}$, $\mathbb{S}_{1112} = \mathfrak{C}_{11} \cup \mathfrak{C}_{21} \cup \mathfrak{C}_{31} \cup \mathfrak{C}_{42}$, $\mathbb{S}_{1211} = \mathfrak{C}_{11} \cup \mathfrak{C}_{22} \cup \mathfrak{C}_{31} \cup \mathfrak{C}_{41}$, and $\mathbb{S}_{1212} = \mathfrak{C}_{11} \cup \mathfrak{C}_{22} \cup \mathfrak{C}_{31} \cup \mathfrak{C}_{42}$.

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 be feasible components by the definition and form a feasible combination $\mathbb{C}_{v_1 v_2 \ldots v_i}$. Before an application is executed, it needs to search for the feasible version of each involved component through a process called *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.

With the one-to-one property of $\mathbb{C}_{v_1\,v_2\ldots v_i}$ and $\mathbb{S}_{v_1v_2\ldots v_i}$ mapping, the process to find a feasible combination may be reduced to a constraints solving problem for $\mathbb{S}_{v_1v_2\ldots v_i}$ in the following steps.

- Resolve the software hierarchy by querying the component registry.
- Each possible combination, $\mathbb{C}_{v_1\,v_2\ldots v_i}$, will be constructed from the software hierarchy.
- Each software constraints, $\mathbb{S}_{v_1v_2\ldots v_i}$, will be built from the corresponding $\mathbb{C}_{v_1\,v_2\ldots v_i}$.

After constructing all possible software constraints, then the remaining assembly 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 combination. Once the feasible combination is found, the feasible version of each involved component will be initialized with appropriate values of parameters that will satisfy the software constraint. For the MPEG video player, the version of the `displayer` that will be feasible depends on which time constraint, $p_2^1 \leq p_3^1 + 70$ of $\mathbb{S}_{1111}$ or $190 \leq p_3^1 \leq 1000$ of $\mathbb{S}_{1211}$, is satisfied. If the decoding speed is fast enough, the feasible combination will include the `on-the-fly` version because the first constraint would be satisfied.

For distributed applications, the component assembly 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. Because we want users to interact with the application on a certain platform, the root component will be always executed on one of the specified platforms and non-distributed. Thus, there are $n_p^{(n_c-1)}$ possibilities with n_c being the number of components and n_p being the total number of specified platforms. 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.

3. Component/constraint Specification Markup Language: `CSML`. While `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 may not be easy to maintain. In addition to the code implementing the component services, additional functionality needs to be added in order to work under `FRAME`, such as the Java implementation of constraints for component assembly and communication for registering and querying the necessary information in the registry.

`CSML` [8] uses XML to help software developers develop components under `FRAME` and maintain consistent plug-in compatibility of components. Instead of implementing them in Java code directly, developers specify high level of characteristics of components, such as services, component dependencies, and constraints, in `CSML`. As shown in fig. 3.1, `CSML` will generate a component interface and a base class of component implementation in Java from a component specification. The generated code includes methods for each constraint, accessing the parameters, constructing the all software constraints. Component register is used to register component information in the registry. Component developers only need to inherit the base class to implement the component; the infrastructure needed to work under `FRAME` is generated by `CSML`.

Similarly, `CSML` allows software users to specify the intended performance and output quality of software and generate the application launcher as shown in fig. 3.2.

3.1. Component specification example. Table. 3.1 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 1-2. Parameters are specified in element *parameter*, lines 5-10, with name and their range (in attribute *upper* and *lower*). Component dependencies are specified in element *child-component*, lines 16 and 17. Internal constraint connectors and external constraint connectors are specified in element *internal-connector*, lines 11-15, and *external-connector*, lines 18-22, 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 14, 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 3-30, specifies the information that is version independent and element *customized*, lines 31-36 and lines 37-43, specifies the version dependent information such as version number and definitions of version dependent constraints. In the example, the `on-the-fly` version specifies the external connector $p_2^1 \leq p_3^1 + 70$ in lines 32-34.

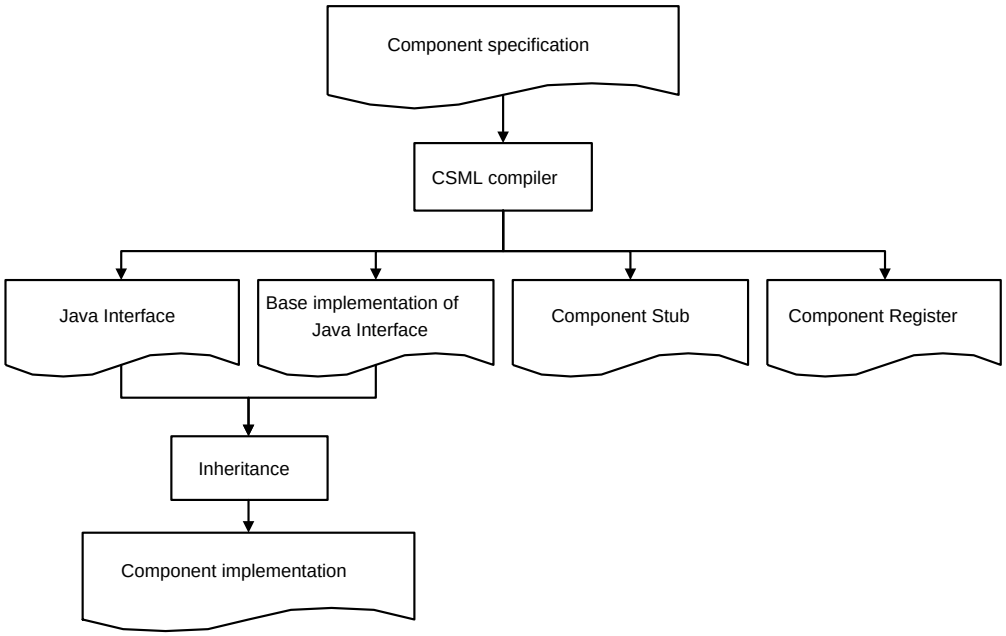

FIG. 3.1. *Flow chart of using* CSML *to develop a component*

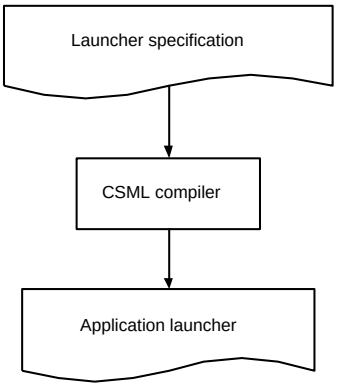

FIG. 3.2. *Flow chart of using* CSML *to execute a component*

3.2. Launcher specification example. CSML also allows users to specify target platforms and the intended performance of the application, and generate the application launcher. Table. 3.2 is the specification of the launcher. It specifies the application to execute on one target platform in line 3. It also specifies intended performance and output quality in the element *constraint*, lines 4 and 5, which correspond to the parameters of root component, i.e., number of frames played per second (p_1^1) and image quality (p_1^2). 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 assembly process.

4. Brew**.** Brew is an interactive computation steering environment for collecting performance data and repairing constraints missed applications because of changes in environments. As shown in Fig. 4.1, the overall architecture of brew consists of two subsystems, instrumentation and steering. More details are described in the following subsections.

4.1. Instrumentation subsystem. The instrumentation subsystem uses $\mathrm{PG}^{\mathrm{RT}}$ [3], an environment for integration of tools and systems for instrumentation, performance visualization, and analysis of complex real-time systems. Two parts of $\mathrm{PG}^{\mathrm{RT}}$ used are BRISK and the Visual Object Framework (VO). BRISK uses two daemons, ism and exs, 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 the instrumentation subsystem, the `Instrument` class needs to be embedded in the applications and `exs` will run concurrently with the application. 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.

4.2. Steering subsystem. The steering subsystem 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,

```
<component name="displayer" steerable="on" registry-host="192.168.1.111"
  uri="http://www.cse.msu.edu/~korenson/class/displayer.jar">
  <general>
    ...
    <parameter name="time" id="p21" value-type="int" upper="200" lower="40">
      ...
    </parameter>
    <parameter name="quality" id="p22" value-type="int" upper="8" lower="1">
      ...
    </parameter>
    <internal-connector id="f1">
      <from-current parameter-id="p21" alias="var1" />
      <from-current parameter-id="p22" alias="var2" />
      <definition> return 7 * ^var1# - 160 * ^var2# &gt;= 120; </definition>
    </internal-connector>
    <child-component name="decoder" id="c1" registry-host="192.168.1.111" />
    <child-component name="RTOS" id="c2" registry-host="192.168.1.111" />
    <external-connector id="f2">
      <from-current parameter-id="p21" alias="var1" />
      <from-child child-id="c1" parameter="time" alias="var2" />
      ...
    </external-connector>
      ...
    <provided-service>
      <declaration method-name="play" return-type="void">
        <argument name="args" value-type="String[]"/>
      </declaration>
    </provided-service>
    ...
  </general>
  <customized version="v1" uri="http://www.cse.msu.edu/~korenson/class/displayer_v1.jar">
    <description> on-the-fly version </description>
    <constraint-definition constraint-id="f2">
      <definition> return ^var1# &lt;= ^var2# + 70; </definition>
    </constraint-definition>
  </customized>
  <customized version="v2" uri="http://www.cse.msu.edu/~korenson/class/displayer_v2.jar">
    <description> buffer version </description>
    <constraint-definition constraint-id="f2">
      <definition> return 190 &lt;= ^var2# &amp;&amp; ^var2# &lt;= 1000; </definition>
    </constraint-definition>
  </customized>
</component>
```

TABLE 3.1
`CSML` *for component* `displayer`

```
<launcher name="MPEG_Player launcher" application="MPEG_Player" registry-host="192.168.1.111">
  <description> MPEG video player laucher </description>
  <target host="192.168.1.111" arch="i686" os="linux" />
  <constraint name="frames" value-type="int" lower="5" upper="30" />
  <constraint name="quality" value-type="int" lower="1" upper="8" />
</launcher>
```

TABLE 3.2
`CSML` *for the MPEG video player launcher*

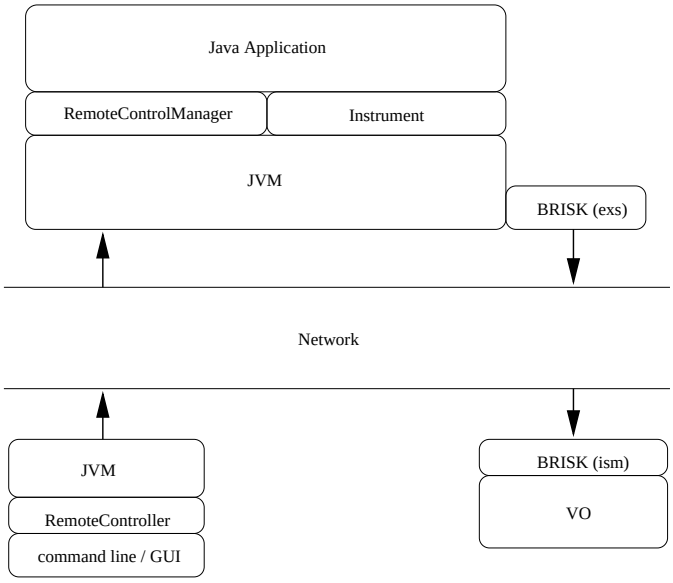

FIG. 4.1. *Architecture of* `brew`

GUI, or Java applet under web browser, may be used on top of `RemoteController`.

To process the steering commands from `RemoteController`, `RemoteControlManager` need to be embedded in the applications. Upon receiving the parsed command, `RemoteControlManager` will find 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.

5. Demonstration.

5.1. Application modifications. As described in section 2, we decomposed the MPEG player, originally developed by Joerg Anders [2], into the component hierarchy as fig. 2.4. The two parameters of the root component `player`, the number of frames played per second (p_1^1) and image quality (p_1^2), are specified within the domains $[5, 30]$ and $[1, 8]$ in table. 3.2 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.

5.2. MPEG player on JIT engine. With JIT optimization, Java VM has the ability to decode the frame fast enough so the time constraint $p_2^1 \leq p_3^1 + 70$ may 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 played per second, is 5 which in turn gives a displaying period of 200 ms. The value of parameter p_1^2, image quality, is 8. Fig. 5.1 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. 5.2, 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 may be satisfied.

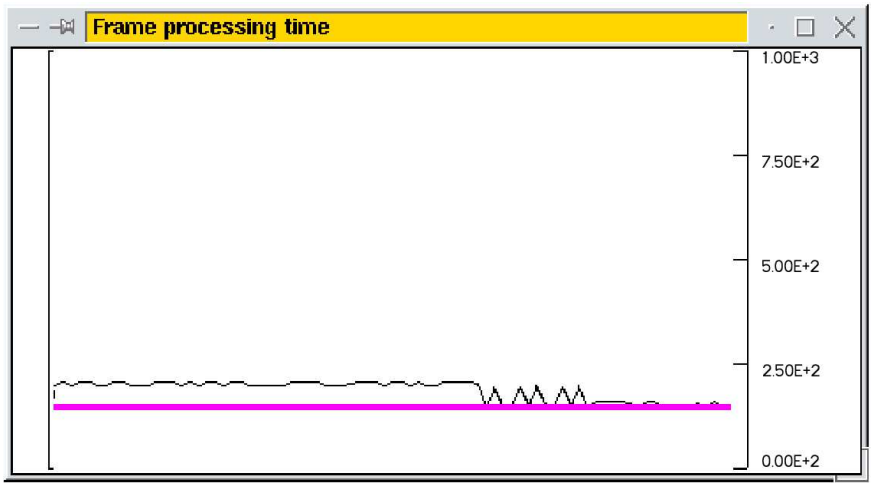

FIG. 5.1. *This figure is the screen shot from a VO on-line performance visualization, which illustrates how the frames processing time varies during the execution of the MPEG player on Linux PC. The Y-axis is the time in milliseconds. The thinner line represents the time needed to display a frame and the thicker line represents the time needed to decode a frame. The VO shows that the displaying time may be further reduced by reducing the image quality.*

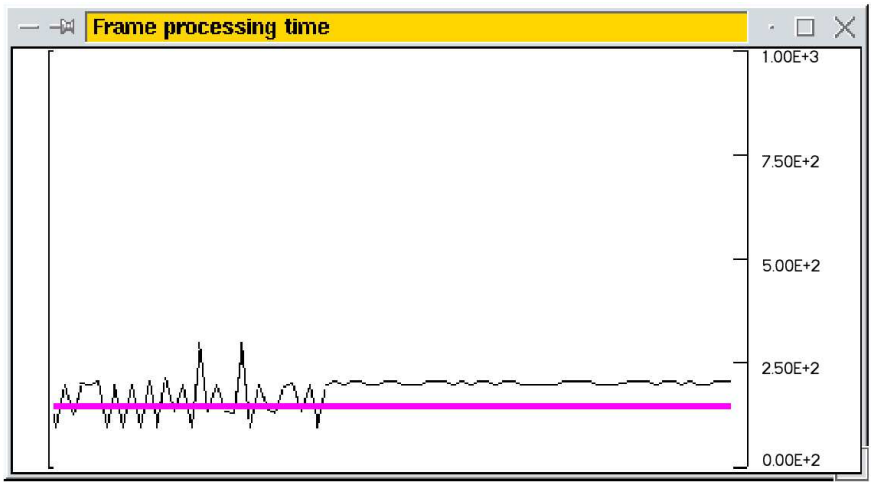

FIG. 5.2. *This figure is the screen shot from a VO on-line performance visualization, which illustrates how the frames processing time varies during the execution of the MPEG player on Linux PC. The Y-axis is the time in milliseconds. The thinner line represents the time needed to display a frame and the thicker line represents the time needed to decode a frame. The VO shows that the displaying time may not meet the time constraint and then is steered to meet the constraint by increasing the displaying period.*

5.3. MPEG player on interpreter engine. We tested the same application on a Java VM with the interpreter engine. As shown in fig. 5.3, we execute the `on-the-fly` version without using `FRAME`. 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 $190 \leq p_3^1 \leq 1000$. Therefore, if the MPEG player is executed under `FRAME`, the assembly process will load and execute the `buffer` version of the displayer component. Fig. 5.4 shows the time between each frame is reduced to 50 ms, i.e., 20 frames per second. Thus, the constraints, $5 \leq p_1^1 \leq 30$, may be satisfied.

5.4. 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 selected by the assembly process because the constraint, "OS is a TimeSys real-time Linux.", is satisfied. $P1$ is forced to use the `dummy` version without the help from `FRAME`, so all the real-time features are disabled. Fig. 5.5 shows the performance difference for the MPEG players with and

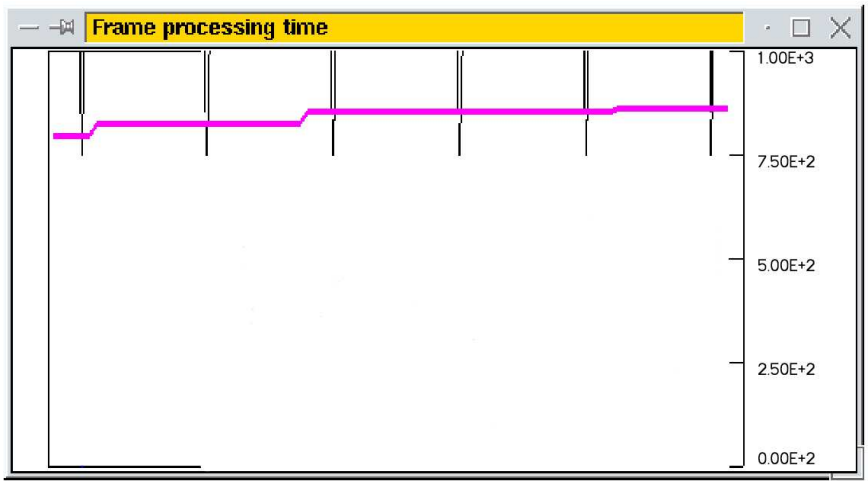

FIG. 5.3. *This figure is the screen shot from a VO on-line performance visualization, which illustrates how the frames processing time varies during the execution of the MPEG player on a Java VM with interpreter engine. The Y-axis is the time in milliseconds. The thinner line represents the time needed to display a frame and the thicker line represents the time needed to decode a frame. The* `on-the-fly` *version of the component* `displayer` *is forced to use. The VO shows that the displaying time constraint ($\leq$ 200 ms) is not satisfied.*

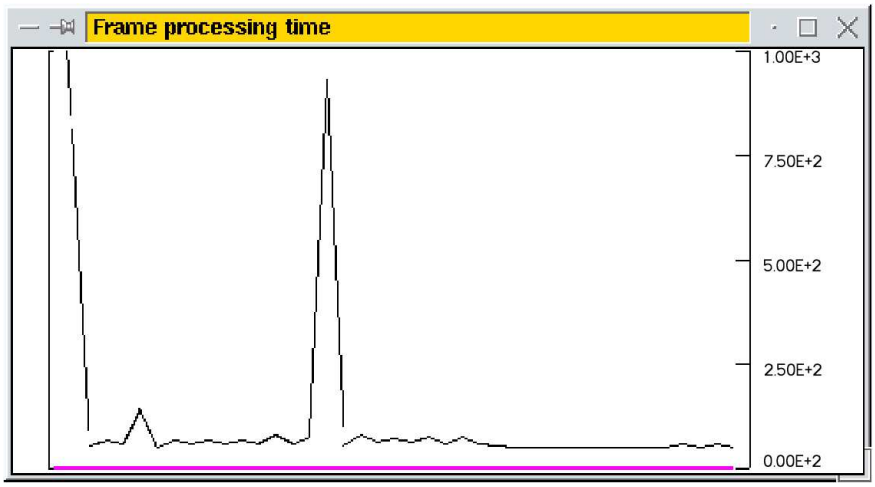

FIG. 5.4. *This figure is the screen shot from a VO on-line performance visualization, which illustrates how the frames processing time varies during the execution of the MPEG player on a Java VM with interpreter engine. The Y-axis is the time in milliseconds. The thinner line represents the time needed to display a frame and the thicker line represents the time needed to decode a frame. Under* `FRAME`*, the* `buffer` *version of the component* `displayer` *will be used. The VO shows that the displaying time constraint ($\leq$ 200 ms) is satisfied.*

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 to be the same (100 ms) for $P0$, but varies significantly and could be up to more than 200 ms for $P1$.

6. Related work. There are two groups for defining real-time Java specification, The Real-Time for Java Experts Group (RTEG) [14] and the Real-Time Java Working Group (RTJWG) [15]. 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 [10]. 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

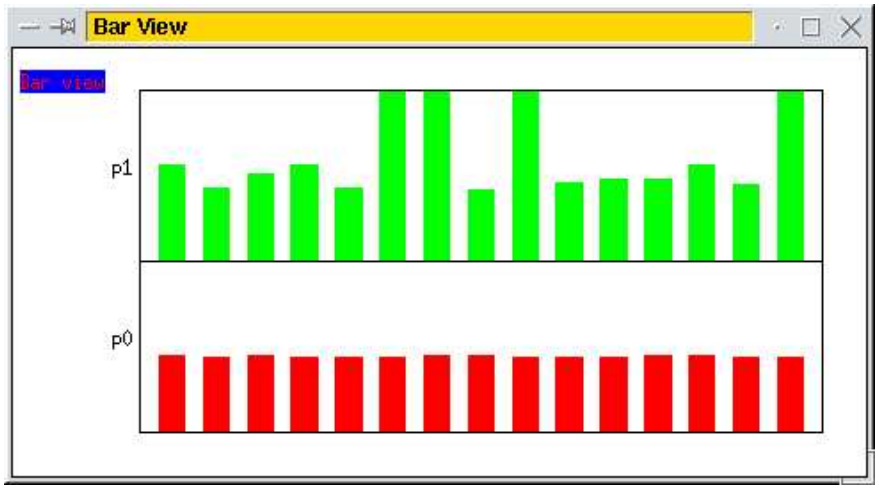

FIG. 5.5. *This figure is the screen shot from a VO on-line performance visualization, which compares the impact of different* RTOS *versions on the displaying time during the execution of the MPEG player on a TimeSys real-time Linux. The height of bars is the time between two consecutive frames. The* $P0$ *use the* timesys *version with the help of* FRAME *and* $P0$ *is forced to us the* dummy *version. The VO shows that* $P0$ *may take advantage of the services from real-time Linux and playback the video smoother than* $P1$.

these two specifications can be found at [10].

Bernat et al. [5] 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 [21] and Dalang [20]. Reflection is also increasingly being applied to a variety of other areas including distributed system [18] and middleware, such as COMERA [17] and DynamicTAO [9]. 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.

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 [1] 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.

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 [12] 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 [13] addresses interactivity in a computationally diverse environment consisting of complex scientific applications, information brokers, and client sources through light-weight 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.

7. Conclusion and future work. We classify the software portability into several levels and discuss that the bytecode portability is not enough in the real-time domain. However, a higher level performance portability may be achieved if the applications may adapt themselves to the computing environment and respond to the run-time environment changes. We also demonstrate how the adaptive software framework, FRAME, and brew, may be applied to soft real-time Java applications under a heterogeneous environment to achieve such a higher level performance

portability. FRAME provides the necessary APIs to allow applications to be built from constraints on the fly. FRAME does not have a run-time performance impact on applications because the assembly finishes 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, future computing systems have been envisioned as omnipresent [19], pervasive [4], and nomadic [7]. FRAME's assumption will not be appropriate under these environments. We plan to extend the reflection 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 [11], such as open or closed-adapted, type of autonomy, frequency, and cost effectiveness. Furthermore, performance may be another important issue. In our experiments, the assembly may be about 650 times slower than the similar application hard coded by if-else statements. Unlike assembly, re-assembly occurs during the execution of applications and may have greater performance impact on applications. Therefore, it will be not feasible to simply re-invoke the assembly process for the re-assembly. How to improve the performance is a challenge for the re-assembly.

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] 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.

[2] J. ANDERS, *MPEG-1-Player*. Information available at *http://rnvs.informatik.tu-chemnitz.de/˜jan/MPEG/MPEG_Play.html*.

[3] 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, Dec. 1997, pp. 154–161.

[4] 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), Boston, MA, Aug. 2000, pp. 266–274.

[5] 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.

[6] D. GARLAN, R. T. MONROE, AND D. WILE, *ACME: An Architecture Description Interchange Language*, in Proceedings of CASCON'97, Toronto, Ontario, Nov. 1997, pp. 169–183.

[7] T. KINDBERG AND J. BARTON, *A Web-Based Nomadic Computing System*, Tech. 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] 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.

[9] 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), no. 1795 in LNCS, New York, Apr. 2000, Springer-Verlag, pp. 121–143.

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

[11] 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 (1999), pp. 54–62.

[12] 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), vol. vii, Jan. 1998, pp. 147–156.

[13] B. PLALE, G. EISENHAUER, K. SCHWAN, J. HEINER, V. MARTIN, AND J. VETTER, *From Interactive Applications to Distributed Laboratories*, IEEE Concurrency, 6 (1998), pp. 78–89.

[14] THE REAL-TIME FOR JAVA™ EXPERTS GROUP, *The Real-Time Specification for Java*. Available at *http://www.javaseries.com/rtj.pdf*.

[15] REAL-TIME JAVA™ WORKING GROUP, *Real-Time Core Extensions*. Available at *http://www.j-consortium.org/rtjwg/rtce.1.0.14.pdf*.

[16] TIMESYS, *Real-Time Embedded Linux*. Information available at *http://www.timesys.com/*.

[17] 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), USENIX, Apr. 1998, pp. 79–88.

[18] T. WATANABE, A. NORIKI, AND K. SHINBORI, *A Reflective Framework for Reliable Mobile Agent Systems*, in On-Line Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures, W. Cazzola, S. Chiba, and T. Ledoux, eds., June 2000. Available at *http://www.disi.unige.it/person/CazzolaW/ewrma2000-proceedings.html*.

[19] M. WEISER, *Some Computer Science Issues in Ubiquitous Computing*, Communications of the ACM, 36 (1993), pp. 74–84.

[20] 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.

[21] ———, *Kava - Using Bytecode Rewriting to add Behavioural Reflection to Java*, in Proceedings of USENIX Conference on Object-Oriented Technology, 2001.