An On-Line Performance Visualization Technology*

Aleksandar M. Bakić, Matt W. Mutka
Michigan State University
Department of Computer
Science and Engineering
3115 Engineering Building
East Lansing, Michigan 48824
{bakicale,mutka}@cse.msu.edu

Diane T. Rover
Michigan State University
Department of Electrical
and Computer Engineering
2120 Engineering Building
East Lansing, Michigan 48824
rover@egr.msu.edu

Abstract

We present a new software technology for on-line performance analysis and visualization of complex parallel and distributed systems. Often heterogeneous, these systems need capabilities for flexible integration and configuration of performance analysis and visualization. Our technology is based on an object-oriented framework for rapid prototyping and development of distributable visual objects. *The visual objects consist of two levels, a platform/device-specific low level, and an analysis- and visualization-specific high level. We have developed a very high-level, markup language, called VOML, and a compiler for component-based development of high-level visual objects. The VOML is based on a software architecture for on-line event processing and performance visualization called EPIRA. The technology lends itself to constructing high-level visual objects from globally distributed component definitions. We present details of the technology and tools used, and show how an example visual object can be rapidly prototyped from several reusable components.*

1 Introduction

Performance analysis and visualization (PAV) tools are crucial components of an effective development cycle, as well as deployment, of parallel and distributed applications. On-line PAV is even becoming necessary for the latter. Since the amount of performance data to be analyzed and visualized increases with the size of a target parallel/distributed application, on-line PAV itself should be distributed. Heterogeneous systems, in addition, need PAV tools that provide flexible integration and configuration support for heterogeneous performance data. Extant generic and library-specific PAV tools for parallel/distributed systems can cover only low-level performance aspects, provided that the target systems fit into their generic schemes and/or use specific libraries, such as PVM [8] and MPI [7]. A wider range of performance aspects, at multiple levels, global and local, are needed to capture and visually explain the behavior of a heterogeneous system.

We have developed a framework for on-line PAV, called PG$^{\mathrm{RT}}$ visual objects[1], to address these issues. The framework is object-oriented and easily distributable via middleware software such as CORBA [16] and DCOM [3]. Within it, a visual-object developer can integrate low- and high-level, application-specific PAV. Furthermore, it is based on two visual-object levels for portability and code reuse: a device-dependent low level, and a device-independent high-level. Our goal was also to be able to integrate various sources of off- and on-line performance data. To achieve this flexibility, the visual objects consume performance data in the form of *event records* from an environment. To formalize the design of high-level visual objects, i.e., enforce a structured approach that is less error-prone, we have defined certain rules and a very high-level, component-based specification language, called Visual Object Markup Language (VOML). The language uses Standard Generalized Markup Language (SGML) markup for structuring visual objects, and Scheme scripts for defining PAV semantics.

The use of SGML enables development of a PAV information infrastructure for platform- and tool-independent development of visual objects. It may also facilitate automatic monitoring, analysis, and visualization of globally distributed applications via network-enabled SGML entity managers. The use of Scheme for visual object semantics

*This work was supported in part by DARPA contract No. DABT 63–95–C–0072, NSF grant No. CDA–9529488, and NSF grant No. ASC–9624149.

[1] PG$^{\mathrm{RT}}$ is the acronym of our Performance Gateway to Real-Time project.

enables both rapid prototyping of visual objects and customizing VOs for a wide range of platforms via, for example, Scheme-to-C and Scheme-to-Java VM bytecode compilers. That is, a single VOML specification may be used to generate automatically an X library-based visual object and one that runs within a WWW browser.

In Section 2, we describe the visual-object framework in detail, and show an example of successful use for PAV of a distributed multimedia real-time application. A PAV architecture for high-level visual objects, the markup language based on it, and development environment are presented in Section 3. An example of a VOML specification is given in Section 4. We compare our PAV approach to other work in the area in Section 5, and conclude in Section 6.

2 Visual Object Architecture

The Visual Object (VO) architecture identifies two main software layers apparent in the majority of extant PAV tools, and represents them as two classes: the *high-level VO (HLVO) class* and the *low-level VO (LLVO) class*. In general, the responsibility of an HLVO class is to implement an application-specific semantics, while an LLVO class is platform-dependent while providing a platform-independent interface to the HLVO class. When implementing a VO class, an HLVO class implementation is derived from an LLVO class implementation, as shown[2] in Figure 1. In the following subsections, we describe main characteristics of the LLVO and HLVO class, and show an application to a heterogeneous system.

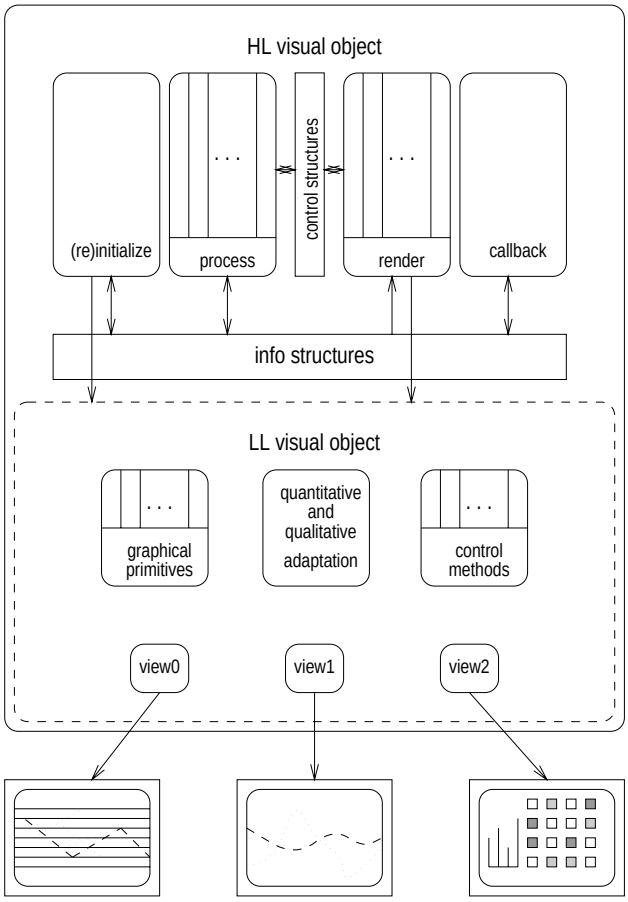

Figure 1. The design of a visual object

2.1 Low-level visual object

The responsibilities of an LLVO class described below illustrate the basic building block of our PAV technology. They have evolved by repeat of substantial experimentation with an X library-based two-dimensional LLVO class that we have implemented in C++.

Multiple views. An LLVO maintains a number of display areas, referred to as *views*. In our implementation of the LLVO class, each display area is supported by a contained object that maintains the state of the corresponding X window.

Graphical primitives. The LLVO class provides methods for rendering simple graphical objects, text and figures in the views. The coordinate system used for the graphical objects' representative coordinates (as arguments to the methods) is a world coordinate system specified by the user at the moment of (re)initializing a view.

[2]Vertical bars in a high-level method denote the presence of multiple peer components.

Display area. A view consists of an internal area surrounded by *margins*, referred to as *scrollable area*. As a visualization progresses, the mapping from the world coordinate system to the view coordinate system may change, at which point only the contents of the scrollable area may be translated or rescaled (zoomed) as a response.

Control methods. Methods such as *scroll*, *resize*, *rescale* and *snapshot* provide explicit control over each view. Combined with the graphical primitives, they allow an HLVO to control explicitly, among other things, what to be drawn and what to be visible at a *point in time*.

Quantitative adaptation. A relation between a view and calls to graphical primitives that draw in the view may be established that causes the view to adapt dynamically by translating or rescaling (zooming) the contents of the scrollable area, thus implicitly controlling what should be visible over an *interval of time*.

Qualitative adaptation. The LLVO class may be portable to multiple graphical platforms that differ at some extent (e.g., different X servers may use different color maps). At run time, it may adapt to the platform capabilities, as well as provide drawing optimization.

An LLVO class implementation may perform bookkeeping about graphical objects being drawn and/or be based on vector graphics, in order to facilitate quantitative and/or qualitative adaptation. However, this is not mandatory and an HLVO class implementation can only assume that the underlying LLVO class is memoryless and raster-based.

The quantitative adaptation of a view in our implementation is initialized by specifying directions (from $\{x+, x-, y+, y-\}$) and types of adaptation (rescaling or scrolling) for these directions. On the other hand, one of the parameters of every graphical primitive is the *adaptation flag* that determines whether the view should adapt before the graphical object is drawn, in order for the graphical object to be visible. In the case of rescaling, the view may also adapt in the opposite way. For example, if the view had to rescale "down" (zoom out) in response to a peak in a temporal line plot[3], it will rescale "up" (zoom in) once the peak has disappeared from the scrollable area. Another parameter for the initialization is the *adaptation quality*. We use it to specify the maximum size of data structures (in our implementation, interconnected red-black trees [6]) used to remember extreme points of graphical objects that have been drawn with the adaptation flag set.

2.2 High-level visual object

Similarly as for the LLVO class in general, we give implementors freedom to define a precise framework for developing HLVOs. In this section, describe our HLVO implementation base, and in Section 3 we present an HLVO development framework. The main components of an HLVO class are the four methods shown in Figure 1.

Event processing. The performance data passed to an HLVO via calls to the processing method are termed *events* (or data events). Based on the events, this method (1) updates performance information referred to as *info structures*, and (2) controls the rendering of this information by updating data structures referred to as *control structures*.

Information rendering. The rendering method may be called, to *map* a portion of the info structures' contents to the LLVO views, either immediately after processing an event (asynchronous rendering mode) or by a thread that may synchronize the rendering of multiple HLVOs (synchronous rendering mode). This method communicates with the processing method by *both reading and writing* the control structures.

Callback processing. An HLVO may also respond to changes in its run-time environment, as well as to the user's commands. This method may, for example, preprocess callback events coming from the LLVO, a GUI, etc., and then forward them to the processing method as if they were data events.

(Re)initialization. In on-line performance visualization, it is desirable to be able to reinitialize partially or reconfigure an HLVO without interrupting the target application and/or instrumentation system that supplies performance data.

In order to allow for rapid prototyping of HLVOs and further PAV research, we have developed a framework based on an implementation of the Scheme language [5] called GUILE [14]. A generic HLVO class inherits the X library-based LLVO class. Both classes have some methods and data wrapped by Scheme procedures within a (run-time) tool integration environment for instrumentation and performance visualization, called $\mathrm{PG}^{\mathrm{RT}}$-TIE [1]. The GUI of a VO (in addition to LLVO callbacks) is implemented separately, using a GUILE interface to Tk [25]. It is possible in Scheme, as a dynamically-typed language, for the event processing method to receive any type of data structure as an event, which allows for easy integration of different performance data sources. Most importantly, this high-level algorithmic language is suitable for easy definition of complex info structures (e.g., association lists serving as micro-databases) and compact expression of updating and querying them in the event processing and information rendering methods, respectively. We have developed a CORBA interface for this framework so that a PAV application may consist of VOs distributed over multiple nodes, while a Scheme-to-C compiler [22] is also available that may speed up the HLVO code.

2.3 Application of visual objects to a heterogeneous system

As part of the $\mathrm{PG}^{\mathrm{RT}}$ environment, two prototype visual objects have been applied to the study of a distributed multimedia real-time application. The target system consisted of a server and a number of heterogeneous receivers of multimedia data streams. The visual objects helped determine the processing demands required to playback different patterns of video frames and to handle different sizes of video frames, as well as the wasted computation due to receiving video frames that cannot be replayed due to time constraints

[3] The contents of the scrollable area is scrolled to the left as the time progresses.

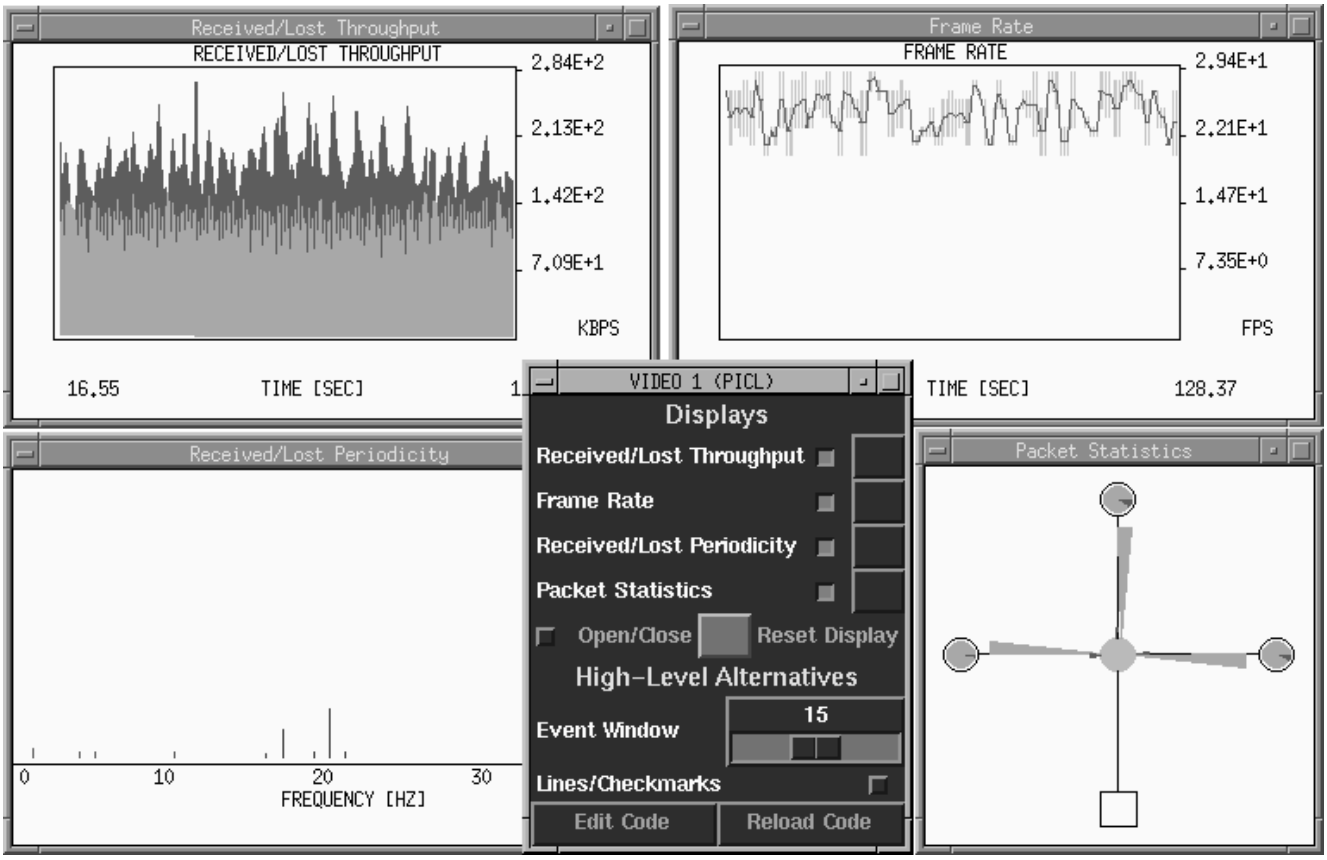

Figure 2. On-line performance visualization of the real-time multimedia application

(e.g., a new video frame arrives before an older video frame can be processed). Furthermore, they helped understand the operation of the application: variations in periodic behavior, and specific points in a network where frame loss occurs, either due to network congestion or individual workstations loading conditions, were displayed. Immense amount of state information, condensed into a set of visual displays, could be used by the feedback control algorithm to make decisions automatically about target bandwidth being requested of the video source.

The snapshot of one visual object is given in Figure 2. Its four views show (1) the throughput of received and estimated throughput of lost video data, (2) the frame rate, (3) the frequency distributions of received and lost video frames over one-second intervals, and (4) a spatial, animated view of all receivers and their connections, depicting the relative volumes of received, lost, used, and dropped video packets. The other visual object has 16 views, divided into four groups: (1) CPU utilization, (2) the periodicity of video frames received, the number of received ATM cells, and (4) the number of lost ATM cells. In each group, there are four related views of the corresponding metric: (1) minimum-average-maximum, (2) sample deviation, (3) aggregate, and (4) per-receiver histogram.

3 Visual Object Markup Language (VOML)

In this section, we describe a framework for semi-automatic design and prototyping of HLVOs. We first define a generic architecture for event processing and performance information rendering that is orthogonal[4] to the VO architecture described in Section 2. Next we present salient characteristics of a very high-level language based on this architecture, called Visual Object Markup Language (VOML), and its compiler that we have designed and implemented. The VOML system allows a performance visualization developer to concentrate on application- and visualization-specific semantics and build HLVOs by combining reusable components.

[4]In this context, where the two architectures coexist, orthogonal means that either architecture can be extended without affecting the other.

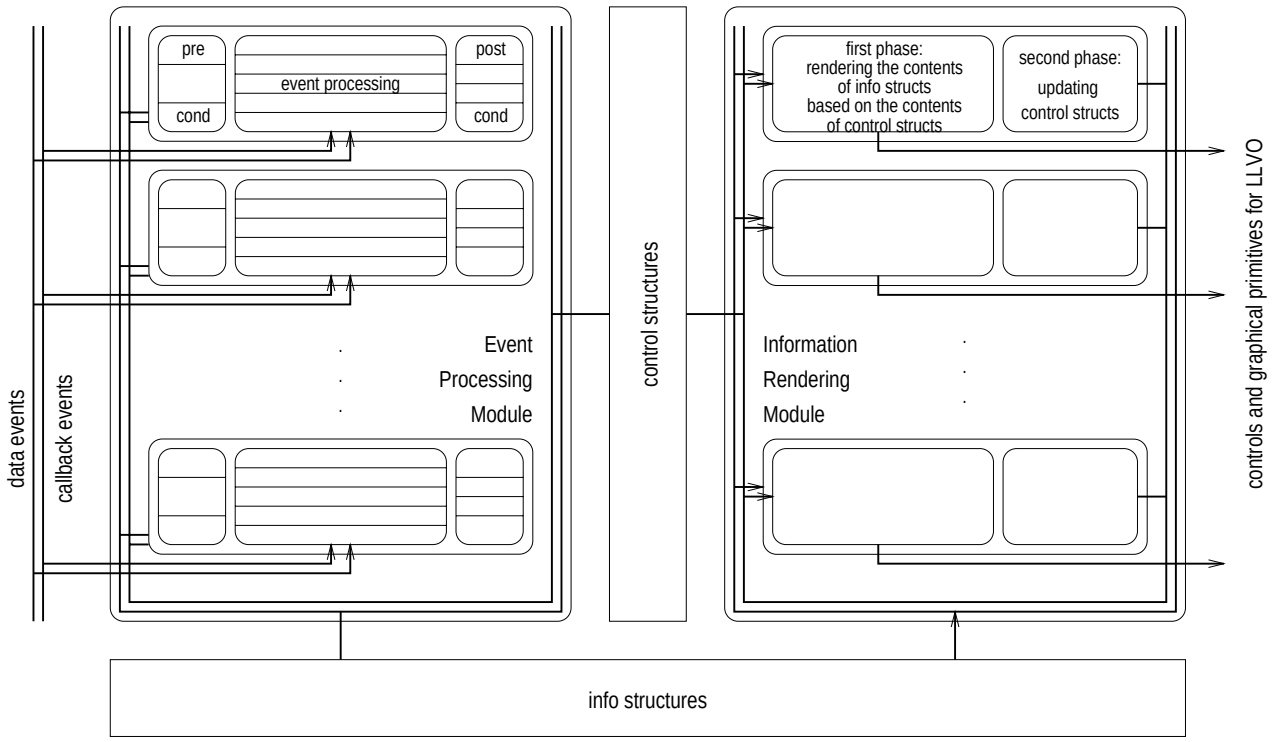

Figure 3. Event Processing and Information Rendering Architecture (EPIRA)

3.1 Event Processing and Information Rendering Architecture (EPIRA)

There are many possible patterns for development of complex HLVOs. For example, one could extend or modify the VO architecture in Figure 1 and build complex HLVOs in a pure object-oriented style, by inheriting from simpler HLVOs. However, since our goal was to develop a framework that could be applicable to target languages that do not have strong support for object orientation (e.g., Scheme and C), we have taken a component-based approach.

Figure 3 shows the *Event Processing and Information Rendering Architecture (EPIRA)*. The architecture specifies the tentative parts of the HLVO architecture, shown in Figure 1, and focuses on the data-driven computation aspect. The two modules in the figure correspond to the event processing and information rendering methods. The events arrive (via method calls) from the two "busses" on the left: they carry the performance data and the changes in the runtime environment. Arrows are drawn to denote unidirectional data flows.

The event processing module may contain a number of *event processing (EP) components*. Each EP component in turn may contain a number of *parts* (separated by horizontal lines in the figure), belonging to one of three classes: (1) event-based ones, shown in the middle and executed upon arrival of a specific event, and condition-based ones, which can be executed (2) before or (3) after the event processing, provided that a specific condition tests true[5]. Since only one event can be received at a time, among all event-based parts (belonging to different EP components) only those for the received event are executed. The conditions corresponding to the condition-based parts are evaluated each time any event is received.

Similarly, the information rendering module contains a number of *information rendering (IR) components*. Each IR component in turn contains two parts, or phases. During the first phase, info and control structures are analyzed and the contents of the info structures is rendered in multiple views appropriately. Once that all first-phase parts of the IR components have been executed, the execution of second-phase parts begins, when the control structures may be updated safely. Since the HLVO assumes that the LLVO has no special rendering support (e.g., a depth buffer), some visualizations may depend on the relative execution order of the first-phase parts.

[5] In VOML, this is referred to as a "condition event."

3.2 The VOML language

We have chosen SGML [9] as the basis for a PAV information infrastructure we plan to build around the VO and EPIRA architectures. For a start, the VOML is an SGML document type definition (DTD) that encompasses the *structure* of HLVOs based on EPIRA. Some of its higher-level elements and example relations among the elements are given in Figures 4a and 4b.

As it can be seen from Figure 4b, VOML attributes are used both to specify certain characteristics of software components described by the elements and to create relations among them, some of which directly correspond to the connections shown in Figure 3. The others are not as "hard-wired," and are described using Figures 4b and 5. Figure 5 defines an example IR component that is used in a visual object defined in Figure 4b.

Although SGML is a very suitable tool for writing structured specifications, it lacks the means for describing semantics of a specification. On the other hand, Scheme is a standardized language with simple syntax and clean semantics that is very suitable for describing the semantics of EP and IR components. Hence, we have decided to imbed Scheme into VOML markup. Combining markup and a programming language, typically Java in WWW-related markup languages, is not a new idea. However, the integration of VOML and Scheme is tighter, as can be seen from the code example in Figure 5. Unlike script-augmented HTML files that are final documents to be "executed," VOML specifications are to be compiled.

Namely, within Scheme code defining the semantics of a component, there may exist references to "formal parameters:" info structures, control structures, events (in EP components) or views (in IR components). The reference notation is `Tn[/m]`, where

- `T` is `$` for info structures, `%` for control structures, and `^` for events and views;
- `n` is the position of the formal parameter in the corresponding parameter list (e.g., `$0` corresponds to `currenttime` in the last line of Figure 4b, because it is the 0-th argument supplied via the `infos` attribute);
- optional `/m` is used for referencing individual fields of an event[6]. For example, an occurrence of `^0/1` within code of EP component `onescalarprocess` in Figure 4b would reference field `value` of data event `onescalar`.

The info and control structures are translated into special global variables by the compiler. Effectively, they are "passed" to EP and IR components by reference when listed in the `infos` and `controls` attributes of the enclosing VOML element. In this way, a reusable component may be written, tested, and placed into a library. In an SGML system, such components may be kept as external SGML entities and used in VOML specifications of different HLVOs by simply referencing them by names.

EP components tend to be application-specific, as they process application-specific event records. To make them more reusable, the element `preprocess-inputs` is provided that allows for specifying "glue logic" (as Scheme expressions) for data events specific to a new application. Namely, before an existing EP component is referenced (i.e., used), any fields of the data events it processes may be arbitrarily preprocessed. For example, an EP component that updates info structures for a simple line-plot visualization (e.g., Scheme code just under `<ep-component name="onescalarprocess"...>` in Figure 4b, updating info structures to be rendered by the IR component code in Figure 5) can be used to visualize the frame rate of a multimedia application. The glue logic in this case could be a function that divides the number of frames received in a time interval[7] by the length of the time interval, whose result would be assigned to the second field (named `value` in the case of the default data event `onescalar`).

Similarly, different library IR components that are parameterized may be combined in interesting ways over a number of views. Additionally, they may be given attributes to determine their higher-level behavior[8]. One such attribute is named `refresh`, which currently can have any combination of values `resize`, `rescale` and `update`. If any of the first two values is used for an IR component, the component will *redraw* its contents if any of the views it draws to get resized or rescaled. This is useful for raster-based LLVO class implementations, where resizing or rescaling an image is lossy. If `update` is used, the component will *undraw* what it drew last time, before proceeding to render the contents of the info structures again. Certain higher-level behavior, which would by default ignore any control structures, can also be controlled by an `enable` attribute that takes a Scheme expression evaluating to a Boolean value. When combining IR components, the HLVO developer may define their execution order.

[6] Currently, VOML only supports PICL [26] compatible events, i.e., lists with the first two elements being integers that determine the record and event type.

[7] Assume that the number of frames is contained in a data event field, and the time interval is kept in an info structure.

[8] Currently, this behavior is supported in our HLVO implementation by auxiliary Scheme code; it might also be supported by an optimizing LLVO class.

```
voml
  head
  body
    visual-object
      event-declarations
        data-event
      info-structures
      control-structures
      utility-code
      view-initializations
        view
      event-processing
        ep-component
          preprocess-inputs
      info-rendition
        ir-component
          line
```

(a) Higher-level elements

```
<event-declarations>
  <data-event name="onescalar" rtype="entry" etype="3000">
    <data-field name="key">
    <data-field name="value">
  ...
<info-structures>
  <variable name="currenttime" type="real">
  <variable name="assoclist" type="list">
  <variable name="palette" type="list">
  ...
<control-structures>
  <variable name="beepcount" type="int" init="0">
  ...
<utility-code>
  (define (beep)
    (display #\Bel))
</utility-code>
<view-initializations>
  <view name="lineplotview" title="Multi-scalar line-plot"...>
...
<event-processing>
  <ep-component name="onescalarprocess" inputs="onescalar.key.value"
    infos="currenttime assoclist">
  ...
<info-rendition>
  <ir-component name="lineplotrender" views="lineplotview"
    infos="currenttime assoclist palette" controls="beepcount">
```

(b) Relations among elements of a VOML specification

Figure 4. A brief description of VOML

```
<description>
  This IR component draws a line-plot of multiple scalars over time, in the supplied
  view (^0). Only lines with the last-update time equal to the current time are drawn.
  Once 10 lines have been drawn, a short sound (beep) is generated.

  The info structures consist of the current time ($0, non-negative real number)
  a multi-scalar association list ($1, indexed by non-negative integer keys),
  and a color palette ($2, a list of strings -- color names).
  Each value in the association list is a 4-element vector:
  #(old-time old-value new-time new-value).

  The key of each value in the multi-scalar association list is used to index the
  color. When all colors are exhausted, the line thickness is increased to distinguish
  between different scalars. A counter is used as a control structure (%0) for
  generating sounds.
</description>
  (let ((palette-len (length $2)))
    (alist-for-each
      (lambda (scalar-id scalar)
        (if (= $0 (vector-ref scalar 2))
          (begin
            (set! %0 (+ %0 1))
            (if (= %0 10)
              (begin
                (beep)
                (set! %0 0)))
            <line view="^0" from="(vector-ref scalar 0) (vector-ref scalar 1)"
              to="$0 (vector-ref scalar 3)" thick="(+ (quotient scalar-id palette-len) 1)"
              color="(nth (modulo scalar-id palette-len) $2)" adapt="yes" clip="margin">)))
      $1))
```

Figure 5. Code of the IR component used as `lineplotrender` in Figure 4b

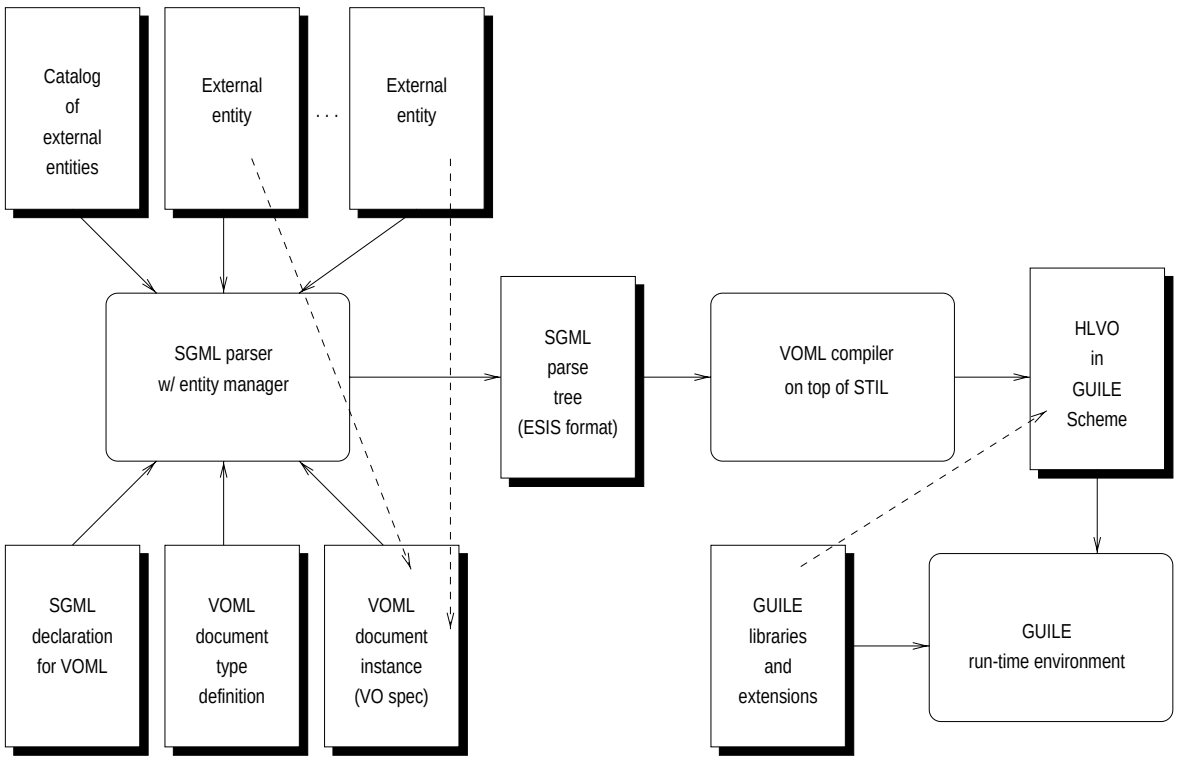

Figure 6. VOML compilation and execution process diagram

3.3 The VOML compiler

The VOML compiler is built on top of an SGML transformation library called STIL [21] and consists of the following components.

SGML parser. The `sgmls` parser [4] is used as the front-end that parses an SGML declaration, VOML DTD, and external entities used in a VOML specification of an HLVO.

STIL library. This library is written for the `clisp` [10] implementation of Common Lisp with CLOS. It allows traversing a parse tree created by the SGML parser, and defining "hooks" (semantic actions) that are called during the traversal.

VOML validating parser. One part of this component consists of the hooks called by the STIL library. The other part consists of CLOS objects that contain code and other information relevant to EPIRA components of the HLVO specification being compiled. The hooks process VOML elements (including the contents in Scheme), their relations and attributes, and build the CLOS objects.

VOML code generator. This component "tangles" the plain, application- and visualization-specific Scheme code from a VOML specification with code that it generates for integration with the run-time environment. The latter includes a graphical user interface for accessing and modifying selected info and control structures, managing views, registering VOs with routines that supply data events, etc.

Figure 6 shows the compilation and execution process of a VOML specification in our GUILE-based environment. Processes are shown as oval rectangles, while input, intermediate and output files are shown as rectangles. Solid lines denote the process of file inclusion, while dashed lines denote references in VOML and Scheme files.

The VOML design allows for extending the compiler to support other Scheme-based run-time environments. Interesting extensions would be for Kawa [2], a Scheme compiler written in Java that generates JVM bytecodes, and Skij [13], a Scheme interpreter that allows rapid prototyping in the Java environment. With a LLVO class implementation in Java, it would become relatively easy to develop PAV applications running in a WWW browser, leveraging platform-independent VOML specifications, and receiving performance data over the Internet. Alternatively, a simpler version of VOML could be defined in XML [15] instead of

SGML, and/or Java could be used instead of Scheme to both compile and execute VOML specifications.

While an SGML parser uses an *entity manager* to find components of a document which are referenced as external entities—such as library EP and IR components—within its virtual storage system, the SGML standard itself does not specify how to implement one. A WWW-enabled entity manager would further enlarge the PAV information infrastructure and automatize monitoring and PAV of globally distributed applications. Figure 7 shows an example of how component definitions could be fetched from a WWW site by the entity manager, to be included for compilation. In the example, the vendor of an imaginary software product, whose performance we want to visualize, keeps the latest implementations of an EP and IR component for the product, ready to be used in our HLVO specification[9].

```
<!DOCTYPE VOML PUBLIC "-//MSU-PGRT//DTD VOML 1.0//EN"
  [
     <!ENTITY SoftwareXYZep
       SYSTEM "http://vendor.com/voml/XYZep.voml">
     <!ENTITY SoftwareXYZir
       SYSTEM "http://vendor.com/voml/XYZir.voml">
  ]
>
<voml>
...
      <event-processing>
        <ep-component name="XYZep"
                      inputs="mydata.f1.f2" ...>
          &SoftwareXYZep;
        </ep-component>
...
      <info-rendition>
        <ir-component name="XYZir"
                      views="myview" ...>
          &SoftwareXYZir;
        </ir-component>
...
```

Figure 7. Sketch of a VOML specification that uses remote component definitions

4 The VOML Specification of a Simple Visual Object

In this section, we present and comment on main parts of the VOML specification of a simple VO with a view similar to the last one of the VO shown in Figure 2. The VO receives performance data events from a distributed application, generated whenever a node is (1) added or (2) removed, and periodically to carry profile data from each node. The event declaration section is shown in Figure 8. The record and event types of the first two events are taken from the PICL specification [26], while the profile event belongs to an extension of PICL. When some field are skipped (i.e., ignored), the `index` attributed is used to specify the position of the next declared field.

```
<event-declarations>
  <data-event name="addnode"
              rtype="pg-entry" etype="-901">
    <data-field name="ts" type="int">
    <data-field name="node-id" type="int">
  </data-event>
  <data-event name="rmnode"
              rtype="pg-exit" etype="-901">
    <data-field name="ts" type="real">
    <data-field name="node-id" type="int">
  </data-event>
  <data-event name="node-prf"
              rtype="entry" etype="3141">
    <data-field name="ts" type="real">
    <data-field name="node-id" index="3" type="int">
    <data-field name="node-type" index="5" type="int">
    <data-field name="rkbps" type="real">
    <data-field name="tb" type="real">
    <data-field name="used" type="real">
    <data-field name="fps" type="real">
    <data-field name="packets" type="int">
    <data-field name="pack-used" type="int">
  </data-event>
</event-declarations>
```

Figure 8. Event declarations

The info and control structure specifications are shown in Figure 9. The info variable `numofnodes` (although redundant) keeps the current number of communicating nodes; `nodes` is an association list that keeps the previous and current profile of each node; `nodeno` keeps the (non-negative) node id from the latest profile event. The control variable `nodechange` indicates whether the number of nodes has changed (meaning that the view has to be updated, as will be seen later).

```
<info-structures>
  <variable name="numofnodes" type="int">
  <variable name="nodes" type="list">
  <variable name="nodeno" type="int" init="-1">
</info-structures>

<control-structures>
  <variable name="nodechange" type="boolean">
</control-structures>
```

Figure 9. Info and control structures

The next is the utility code section, which we omit for brevity. It contains function `getfontname` that returns a font name from a list of available fonts, given some hints. Besides, functions `id-get`, `id-put` and `id-rem`, which are used to manipulate the `nodes` association list, may be defined in this section (in our case, they are defined and exported from another module, available in the run-time environment).

The view initialization section is shown in Figure 10. The (only) `BU-View` view is 700 by 700 pixels, through which a rectangle in the world coordinate system from

[9] It is assumed that we already have the information about the components' interfaces.

$(-10,-10)$ to $(110,120)$ is visible. In this example, the view neither scrolls nor zooms. The control variable `nodechange` is set to true only to trigger the drawing of the switch in the beginning.

```
<view-initializations>
  <view name="BU-View" window="700 700"
    world="-10 110 -10 120" controls="nodechange">
    <description>Switch, nodes and bandwidth
      utilizations</description>
    (set! %0 #t)
  </view>
</view-initializations>
```

Figure 10. View initialization

```
<event-processing>
  <ep-component name="rmnode" inputs="rmnode.ts.node-id"
    infos="numofnodes nodes nodeno" controls="nodechange">
    <description>Remove a node</description>
    <input name="^0">
      (set! $1 (id-rem $1 ^0/1))
      (set! $0 (- $0 1))
      (set! $2 -1)
      (set! %0 #t)
    </input>
  </ep-component>
  <ep-component name="addnode" inputs="addnode.ts.node-id"
    infos="numofnodes nodes nodeno" controls="nodechange">
    <description>Add a node, reset the infos</description>
    <input name="^0">
     (set! $1 (id-put $1 ^0/1
           (cons (vector ^0/0 0 0 0 0 0 0 0 0 0)
                 (vector ^0/0 0 0 0 0 0 0 0 0 0))))
     (set! $0 (+ $0 1))
     (set! $2 -1)
     (set! %0 #t)
    </input>
  </ep-component>
  <ep-component name="nodeprofile"
    inputs="node-prf.ts.node-id.node-type.rkbps.tb.used.fp
s.packets.pack-used"
    infos="numofnodes nodes nodeno">
    <description>Update a node's infos</description>
    <input name="^0">
      (let ((old-info (cdr (id-get $1 ^0/1)))
            (new-info (vector ^0/0 ^0/1 ^0/2 ^0/3 ^0/4
                              ^0/5 ^0/6 ^0/7 ^0/8)))
        (set! $1 (id-put $1 ^0/1
                         (cons old-info new-info))))
      (set! $2 ^0/1)
    </input>
  </ep-component>
</event-processing>
```

Figure 11. Event processing components

There is one EP component for each event, although one could implement, for example, only one for all the three events. They are shown in Figure 11, as updating the info and control structures according to the event declarations. Fields of an event are listed using a notation in which the event name is followed by some of its fields' names, delimited by periods. In the `nodeprofile` EP component, all the event fields declared above are used. It is not necessary to use them all and in the same order as declared; the `^m/n` notation uses the order(s) given in the `inputs` attribute. It can be seen that the field `node-id` is used as the key, and the value field in the `nodes` association list is a pair of vectors keeping the previous and current profile of a node. In this example, only the current profile will be used, but in a more complex VO both the previous and current one may be needed.

```
<ir-component name="nodes-ir" views="BU-View"
  infos="numofnodes" controls="nodechange"
  refresh="update resize" buffer="yes" enable="%0">
  <description>Switch, nodes, connections</description>
  (let* ((viewinfo <view-info view="^0">)
         (width    (list-ref viewinfo 5))
         (height   (list-ref viewinfo 6))
         (size     (inexact->exact
                     (max (/ width 40) (/ height 40))))
         (font     (getfontname "fonttable"
                                "courier" size "bold")))
    <text view="^0" coords="50 107" halign="CENTER"
      font="font" fcolor='"black"'
      content='"Bandwidth Utilization"'>
    <figure view="^0" filename='"bggif/switch.gif"'
      orig-origin="0 0" orig-extents="0 0"
      world-origin="45 45" world-extents="10 10">
    (let* ((nodenum (- $0 1))
           (step    (/ 6.28 nodenum)))
      (if (gt nodenum 0)
        (let loop ((num nodenum))
          (let* ((angle  (* num step))
                 (sine   (sin angle))
                 (cosine (cos angle)))
            <figure view="^0"
              filename='"bggif/node.gif"'
              orig-origin="0 0" orig-extents="0 0"
              world-origin="(+ 45 (* 45 sine))
                            (+ 45 (* 45 cosine))"
              world-extents="10 10">
            <line view="^0" from="(+ 50 (* 6 sine))
                                  (+ 50 (* 6 cosine))"
                            to="(+ 50 (* 39 sine))
                                (+ 50 (* 39 cosine))"
              color='"red"' thick="12">
            (if (gt num 1)
              (loop (- num 1))))))))
  <end-with>(set! %0 #f)</end-with>
</ir-component>
```

Figure 12. Template IR component

Finally, the information rendering section consists of two IR components. The `nodes-ir` IR component, which is shown in Figure 12 and will be executed first[10], writes text and draws the switch and as many "$\mathrm{PG}^{\mathrm{RT}}$ globes" around it as there are active nodes, connected with the switch via thick red lines. The `enable` attribute specifies that this IR component should be executed whenever the number of nodes has changed. The `refresh` attribute adds that everything the IR component drew last time should be redrawn when the view `BU-View` is resized. It also specifies that everything the IR component drew last time has to be undrawn before something new is drawn (whenever the IR component is enabled). The `buffer` attribute is used

[10] In the prototype implementation of the VOML compiler, the execution order of the IR components is opposite of the order they appear in a specification.

```
<ir-component name="bu-ir" views="BU-View"
  infos="numofnodes nodes nodeno" refresh="resize">
  <description> Bandwidth utilization </description>
  (if (gt $2 -1)
    (let* ((angle    (* $2 (/ 6.28 (- $0 1))))
           (sine     (sin angle))
           (cosine   (cos angle))
           (new-info (cdr (id-get $1 $2)))
           (newkbps  (vector-ref new-info 3))
           (newused  (vector-ref new-info 5))
           (mag      (* 25 (/ newused newkbps))))
      <line view="^0" from="(+ 50 (* 6 sine))
                            (+ 50 (* 6 cosine))"
                      to="(+ 50 (* 39 sine))
                          (+ 50 (* 39 cosine))"
        color='"red"' thick="12">
      <line view="^0" from="(+ 50 (* 6 sine))
                            (+ 50 (* 6 cosine))"
                      to="(+ 50 (* (+ 6 mag) sine))
                          (+ 50 (* (+ 6 mag) cosine))"
        color='"blue"' thick="10">
    (set! $2 -1)))
</ir-component>
```

Figure 13. Active IR component

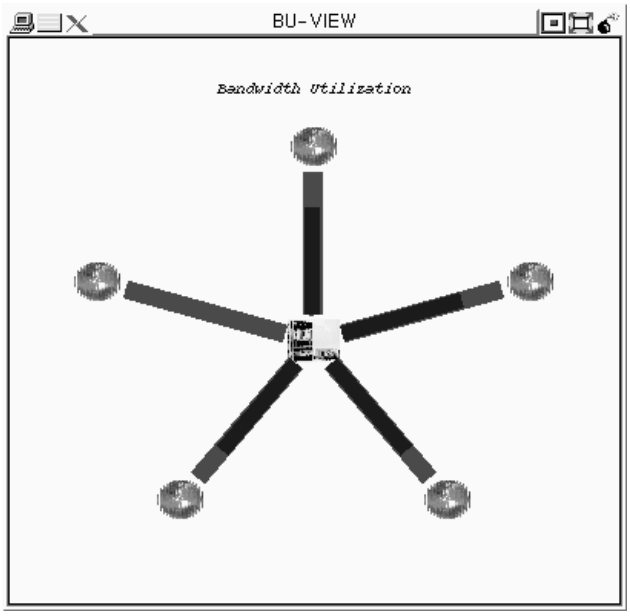

Figure 14. A snapshot of the view

to make the IR component draw in-memory only until it is done, and then flush the contents of the memory to the screen. This is useful to make the rendering smoother and faster when there are many graphical objects to be drawn. This IR component resets the nodechange control variable in the second phase, so that other IR components may be added safely that depend on the value of this variable[11].

The other IR component is executed each time an event is received, and it draws a blue thick line on top of a red thick line (drawn in the same place as the thick red line drawn by the former IR component, so that it can effectively be undrawn), showing the relative bandwidth used by a node (if a profile event was received last that set the nodeno info variable to a non-negative value). Its code is shown in Figure 13. The refresh attribute treats the resizing of the view same as above.

A snapshot of the view is given in Figure 14. We have not shown all the VOML features in this example; for a tutorial, please visit the URL given in Section 6.

5 Related Work

ParaGraph [11] is a PAV tool for parallel programs, based on the PICL communication library. PAV environments are progressing with features to incorporate new analysis and display modules. Visualization environments are not only becoming extensible, but retargetable to different analysis scenarios. Pablo took this research one step further by incorporating support for performance environment prototyping [20]. VIZ continues in this direction by focusing on the visualization technology required for application-specific performance visualizations [12]. Avatar [19] uses Pablo to study two types of high-performance input/output of the Portable Parallel File System (PPFS): parallel scientific codes and WWW servers. The Rivet project [17] integrates new visualization tools into the design and evaluation process of a variety of computer system components, specifically processor and memory systems, multiprocessor architectures, compilers, operating systems, and networks. Lucent Technologies' Visual Insights [24] offers a set of interactive and linked data visualization components for the Microsoft ActiveX developer market that help software developers to create more flexible, animated ways to display trends in vast stores of information.

In Table 1 we compare $\mathrm{PG}^{\mathrm{RT}}$ visual objects with related PAV tools and systems. While some of the latter have gone farther in certain direction, such as the graphical metaphor, our design decisions were primarily based on the requirements stated in Section 1. We have also been concerned that insisting on state-of-the-art, academic software technologies, such as, for example, lazy functional languages, could limit the practicality of our approach. Instead, using technologies that are gaining acceptance, such as SGML (e.g., in the Chemical Markup Language [18]) and use of structure, components and scripting (e.g., in VRML [23]), we hope to contribute to the PAV community.

[11] In Scheme, this second phase is implemented using `delay` and `force`.

Tool/System	On/off-line operation	Graphical metaphor	Underlying graphical technology	View classes	Reusable
ParaGraph	off-line	data-flow	X library	generic	no
Pablo widgets	off-line	data-flow	X library	generic	yes
Avatar	on-line	data-flow	VRML	scattercube only	no
VIZ	on-line	data-reactive	Open Inventor	domain-specific	yes
Rivet	off-line	data-flow	OpenGL	domain-specific	yes
Visual Insights	off-line	n/a	n/a	generic	yes
PG^{RT} VO	on-line	data-flow	low-level VO implementations	domain-specific	yes

Table 1. Performance visualization tools and systems

6 Conclusions

We have presented a novel PAV technology intended to satisfy growing needs by researchers and users of parallel and distributed systems. Salient characteristics of the technology include support for rapid prototyping and automated design of PAV tools, object orientation, distributability, portability, code reuse and flexibility. Tutorials with examples and reference manuals for PG^{RT}-TIE and VOML can be found in the Documents section at `http://www.egr.msu.edu/Pgrt/`.

In the future, we plan to extend and improve the technology, and make it available to the PAV community. We will develop visual objects specific to parallel/distributed real-time applications, but also try to help PAV developers using our technology in other areas by developing domain-specific libraries of EP and IR components.

References

[1] A. Bakić, M. W. Mutka, and D. T. Rover. Real-time system performance visualization and analysis using distributed visual objects. In *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, December 2 1997.

[2] P. Bothner. Kawa - compiling dynamic languages to Java VM. In *Proceedings of the 1998 USENIX Annual Technical Conference, New Orleans*, June 19 1998.

[3] D. Box. *Essential COM*. Addison-Wesley, January 1998.

[4] J. Clark. `sgmls`: A validating SGML parser. WWW, 1993. `ftp://ftp.jclark.com/pub/sgmls/`.

[5] W. Clinger and J. Rees, editors. *Revised(4) Report on the Algorithmic Language Scheme*. IEEE, November 2 1991.

[6] Corman, Lieserson, and Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[7] J. Dongarra et al. MPI: A message-passing interface standard. Technical report, Oak Ridge National Laboratory, June 1995.

[8] G. Giest et al. *PVM 3.0 User's Guide and Reference Manual*. ORNL/TM–12187, February 1993.

[9] C. F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[10] B. Haible. CLISP, a Common LISP implementation, 1997. `http://clisp.cons.org/ haible/clisp.html`.

[11] M. T. Heath and J. E. Finger. Visualizing performance of parallel programs. *IEEE Software*, pages 29–39, September 1991.

[12] H. H. Hersey, S. T. Hackstadt, L. T. Hansen, and A. D. Malony. Viz: A visualization programming system. Technical Report CIS-TR-96–05, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403–1202, April 1996.

[13] IBM alphaWorks. An interactive scripting language for rapid prototyping in the Java environment. WWW, 1998. `http://www.alphaWorks.ibm.com/formula/-Skij/`.

[14] T. Lord. An anatomy of Guile/the interface to Tcl/Tk. *Usenix Tcl/Tk Workshop '95*, 1995. More information available on the WWW, at `http://www.red-bean.com/guile/`.

[15] E. Maler. XML specification DTD. WWW, 1998. `http://www.sil.org/sgml/xmlspec--19980323-dtd.txt`.

[16] T. J. Mowbray and R. Zahavi. *The Essential CORBA—System Integration Using Distributed Objects*. The Object Management Group ISBN 0–471–10611–9, 1997.

[17] T. Munzner. Laying out large directed graphs in 3D hyperbolic space. In *Proceedings of the InfoVis '97*, 1997.

[18] P. Murray-Rust. Chemical markup language. Technical report, Venus, `http://www.venus.co.uk/omf/cml/`, 1997.

[19] D. Reed, K. Shields, W. Scullin, L. Tavera, and C. Elford. Virtual reality and parallel systems performance analysis. *IEEE Computer*, 28(11):55–67, November 1995.

[20] D. A. Reed, R. A. Aydt, T. M. Madhyastha, R. J. Noe, K. A. Shields, and B. W. Schwartz. An overview of the Pablo performance analysis environment. Technical report, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, November 7 1992.

[21] J. Schrod. SGML transformations in LISP. Technical report, Computer Science Department, Technical University of Darmstadt, Kranichweg 1, D-63322 Roedermark, FR Germany, 1995.

[22] T. Tammet. Hobbit: A Scheme-to-C compiler. Technical report, Department of Computing Science, Chalmers University of Technology, University of Göteborg, Sweden, 1995. Available at http://www-swiss.ai.mit.edu/~jaffer/Hobbit.html.

[23] The VRML Consortium Inc. The virtual reality modeling language. WWW, 1997. http://www.vrml.org/.

[24] Visual Insights, Lucent Technologies. Software components. WWW, 1998. http://www.visualinsights.com/components/.

[25] B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall ISBN 0–13–182007–9, 1995.

[26] P. H. Worley. A new PICL trace file format. Technical Report ORNL/TM-12125, Oak Ridge National Laboratory, Mathematical Sciences Section, P.O. Box 2008, Bldg. 6012, Oak Ridge, TN 37831-6367, September 1992.

Biographies

Aleksandar M. Bakić is a research assistant in the Department of Computer Science and Engineering at Michigan State University. His research interests include engineering of complex, distributed real-time systems, integrated instrumentation and performance visualization for parallel and distributed systems, and compiler-based technologies. He received his BS in computer engineering from the School of Electrical Engineering, Belgrade, Yugoslavia, and his MS in computer science from Michigan State University. He is a student member of ACM. Contact him at the Dept. of Computer Science and Eng., 3115 Eng. Bldg., Michigan State Univ., East Lansing, MI 48824-1226; bakicale@cse.msu.edu.

Matt W. Mutka is an associate professor in the Department of Computer Science and Engineering at Michigan State University. His research interests include resource-management issues in distributed and parallel systems, real-time systems, high-speed computer networks, and instrumentation and visualization for the design and evaluation of embedded computing systems. He received his BS in electrical engineering from the University of Missouri-Rolla, his MS in electrical engineering from Stanford University, and his PhD in computer science from the University of Wisconsin-Madison. He is a member of the IEEE Computer Society, IEEE Communication Society, and ACM. Contact him at the Dept. of Computer Science and Eng., 3115 Eng. Bldg., Michigan State Univ., East Lansing, MI 48824-1226; mutka@cse.msu.edu.

Diane T. Rover is an associate professor in the Department of Electrical and Computer Engineering and the director of the Computer Engineering Program at Michigan State University. Her research interests include integrated program development and performance environments for parallel and distributed systems, instrumentation systems, performance visualization, embedded real-time systems, and reconfigurable hardware. She received her BS in computer science and her MS and PhD in computer engineering, all from Iowa State University. She is a member of the IEEE Computer Society, ACM, and American Society for Engineering Education. Contact her at the Dept. of Electrical and Computer Eng., 2120 Eng. Bldg., Michigan State Univ., East Lansing, MI 48824-1226; rover@egr.msu.edu.