Modeling and Analysis of a Virtual Reality System with Time Petri Nets

Rajesh Mascarenhas
EECS Dept.
University of Illinois
851 South Morgan
Chicago, IL 60607
+1 (312) 413-2102
rmascare@eecs.uic.edu

Dinkar Karumuri
Oracle
Corporation
203 N. LaSalle St.
Chicago, IL 60601
+1 (312) 704-4875
dkarumur@us.oracle.com

Ugo Buy
EECS Dept.
University of Illinois
851 South Morgan
Chicago, IL 60607
+1 (312) 413-2296
buy@eecs.uic.edu

Robert Kenyon
EECS Dept.
University of Illinois
851 South Morgan
Chicago, IL 60607
+1 (312) 996-0450
kenyon@eecs.uic.edu

ABSTRACT
The design, implementation, and testing of virtual environments is complicated by the concurrency and real-time features of these systems. Therefore, the development of formal methods for modeling and analysis of virtual environments is highly desirable. In the past, Petri-net models have led to good empirical results in the automatic verification of concurrent and real-time systems. We applied a timed extension of Petri nets to modeling and analysis of the CAVE™[1] virtual environment at the University of Illinois at Chicago. Here, we report on our time Petri net model and on empirical studies that we conducted with the Cabernet toolset from Politecnico di Milano. Our experiments uncovered a flaw in the way a shared buffer is used by CAVE processes. Due to an erroneous synchronization on the buffer, different CAVE walls can simultaneously display images based on different input information. We conclude from our empirical studies that Petri-net-based tools can effectively support the development of reliable virtual environments.

KEYWORDS
Virtual reality environments, time Petri nets, real-time analysis, concurrency models, automated verification.

1 INTRODUCTION

Virtual Reality (VR) systems are becoming increasingly widespread. Projection-based systems, such as the CAVE, consist of several walls that display computer-generated images for the benefit of a human viewer. These images are drawn in real-time on the basis of the viewer's perspective in the virtual world in such a way as to create the impression of a real-life, three-dimensional view of a given scene.

[1]CAVE™ is a registered trademark of the Regents of the University of Illinois.

Several features of VR systems complicate modeling, analysis and testing of these systems. For instance, VR systems usually consist of multiple hardware and software components that operate asynchronously, such as sensors, image computation and rendering processes, and analog-to-digital converters. However, the output of a VR system typically consists of video and audio streams that must be output synchronously to create the impression of a real scene to human eyes and ears. Thus, the computations occurring in a VR system must comply with real-time constraints in order for the system to work convincingly. Moreover, the presence of multiple asynchronous components introduces the possibility of concurrency errors, such as missed updates or inconsistent changes to shared data. When these errors occur, the output of the VR system is often compromised, sometimes resulting in "simulation sickness" for the unfortunate human viewer.

Given the high cost and the timing requirements of VR systems, there is a need for automated tools that can predict the performance of these systems before the systems are deployed. To date, numerous techniques have been defined for the automated analysis of general concurrent and real-time systems [1, 4–11, 14, 16, 19, 22]. However, these techniques and tools have yet to be applied to modeling and analysis of VR systems.

Our goal here is to analyze a specific VR system, the CAVE environment at the University of Illinois at Chicago, using a timed extension of Petri nets for modeling and analysis [12]. We selected a Petri-net-based formalism for many reasons. First, Petri nets can capture quite naturally the main features of VR systems. It is well known that Petri nets can model easily nondeterminism and parallel computation, two essential features of concurrent systems, such as VR systems. In addition, Petri nets can model easily synchronization of asynchronous processes, which is commonplace in VR systems. For instance, VR systems often carry out the computation and the rendering of the images for multiple walls as asynchronous processes. However, the processes must be synchronized with each other (and, when applicable, with processes producing audio streams) before their output is displayed.

An additional advantage is that Petri nets can accommodate quite easily models at different abstraction levels. Although we have only considered high-level models of the CAVE thus far, we plan to use Petri nets also for finer-grained analysis, such as automatic verification of CAVE application code. Other authors have shown that Petri nets can be easily generated from high-level code written, for instance, in the Ada language [5, 13, 24]. Finally, Petri nets have been studied extensively in the past three decades, resulting in the definition of numerous tools and techniques for analysis. In particular, Petri nets are amenable both to formal verification and simulation techniques [5, 13].

To date, many extensions of Petri nets to the timed domain have been defined (see, for instance, [15, 18, 23, 25]). These models differ in terms of their expressive power and their ability to support analysis. In general, the more expressive notations are also more difficult to analyze and vice versa. For our work we chose Merlin and Faber's *time Petri nets* mostly because they were the least expressive notation that could adequately model the properties of interest of VR systems [18]. These nets associate a so-called *firing interval* (i.e., a delay bounded by two constants) with net transitions.

Our work on Petri-net-based analysis of the CAVE environment has led to two main results. First, we built a time-Petri-net model of the CAVE environment. Second, we applied the Cabernet toolset to the simulation and automatic verification of the net model [2]. Here we report on the model we defined and on the results of our simulations with the model.

Our experiments uncovered a flaw in the way a shared buffer is used by CAVE processes. The buffer is written by a process producing sensor information about the current position and orientation of the CAVE viewer. Four additional processes use information from the buffer to compute the images to be displayed simultaneously on each of the CAVE walls. Due to an erroneous synchronization on the buffer, the four processes sometimes use inconsistent information (e.g., by missing an update to the buffer) about the position and the orientation of the viewer. As a result, different walls can simultaneously display images based on different sensor information. This possibility was discovered concurrently but independently from us by another CAVE researcher. Evidently, this flaw could have been detected before CAVE's code was written—and corrected more easily—if tools similar to ours had been used during the CAVE's design stages.

This paper is organized as follows. Section 2 summarizes the CAVE environment. Section 3 introduces time Petri nets. Our Petri net model of the CAVE is discussed in Section 4. We discuss our empirical results in Section 5. Some conclusions and future research directions are discussed in Section 6.

2 THE CAVE ENVIRONMENT

The CAVE is a multi-person, room-sized, high-resolution, 3D video and audio environment [12]. The CAVE is a theater 10x10x9 feet, made up of three rear-projection screens for walls and a down-projection screen for the floor. Electrohome Marquis 8000 or 8500 projectors throw full-color workstation fields (1024x768 stereo) at 96 Hz onto the screens, giving 3000 x 2000 linear pixel resolution to the surrounding composite image. Computer-controlled audio provides a sonification capability to multiple speakers. A viewer's head and hand are tracked with Ascension tethered electromagnetic sensors operating at a 96 Hz sampling frequency for a dual sensor configuration. The tracking system has a valid operating range of 7.5 feet and a delay of about 50–75 ms.

Stereo images are generated by Stereographics' LCD stereo shutter glasses that are used to present the alternating right and left eye images viewed by the subject. The correct perspective and stereo projections are based on values returned by the position sensor attached to the Stereographics shutter glasses. Two SGI Onyxes with Infinite Reality (IR) Engines are used to create the imagery that is projected onto the walls and floor.

The heart of the image generation is the Infinite Reality Engines running on two SGI Onyx hosts with three high-speed graphics pipelines each linked to an independent R10,000 processor. The processors within each Onyx host share a common memory space where variables for the generation of the scenes can be stored and accessed by each processor; however, the two Onyx hosts are connected through a high-speed communication network. Each processor uses input data from the tracker and information stored in a visual database to generate an image. The database stores a 3D representation of the scene on display in the CAVE. Some processors can be used to update visual scene variables while other are used to generate database changes and communicate with other computers over high-speed networks for multi-system collaborative environments.

A typical CAVE application starts by initializing internal graphics and external projection and sensory hardware. An initial scene is generated and displayed on all walls of the CAVE. Next, the application begins reading real-time data from the sensors attached to the viewer moving about in the environment. These data are used to change the generated look-at point and to interact with objects in the scene. The images are either updated at a fixed interval set by the program or run free. In the free-running mode, there is no deadline for the program to finish computing an image. When this happens, the image is displayed. In the fixed-interval mode, there is a strict timing loop whereby the program must display the content of a suitable buffer, whether the image is complete or not, upon expiration of a deadline. After the new image has been sent to the projectors, the program returns to the point where it obtains new input data from the sensors or other devices.

3 TIME PETRI NETS

A *time Petri net* is a 5-tuple $N = (P, T, F, D, M_0)$, where P is a finite set of places, T is a finite set of transitions, F is an arc set, D associates a static delay interval $\tau = [a, b]$ with each transition $t \in T$, and M_0 is an initial marking, that is, an initial assignment of tokens (i.e., markers) to each place $p \in P$. Given an arc f from a place p (a transition t) to a transition t (a place p), p is said to be an input (output) place for t, and t is an output (input) transition for p. A static delay is bounded by two numeric constants, a and b, with $0 \leq a < +\infty$ and $a \leq b \leq +\infty$.

State changes are carried out by firing *fireable* transitions. A transition is said to be *enabled* when all its input places have at least one token. A transition with delay interval $\tau = [a, b]$ is fireable if it is continuously enabled for at least a, but no more than b, time units. Thus, if transition t with delay interval $\tau = [a, b]$ becomes enabled at time θ_0, then transition t *must* fire in the time interval $[\theta_0 + a, \theta_0 + b]$, unless it becomes disabled by the removal of tokens from some input place in the meantime. The *static earliest firing time* of transition t is a; the *static latest firing time* of t is b; the *dynamic earliest firing time* of t is $\theta_0 + a$; the *dynamic latest firing time* of t is $\theta_0 + b$; the *dynamic firing interval* of t is $[\theta_0 + a, \theta_0 + b]$.

A *state* of a time Petri net consists of a *marking* (i.e., an assignment of tokens to each place) and a vector of dynamic firing intervals for each enabled transition. The initial net state is defined by the initial net marking, time $\theta = 0$, and dynamic delays equal to the static delays for all enabled transitions. When a transition t is fired the net moves from a state x to a new state y. The marking of y is obtained by removing a token from each input place of t and adding a token to each output place of t. The dynamic firing delays of transitions enabled in y are computed as follows. If a transition was not enabled in x, its dynamic delay is equal to its static delay. If a transition was enabled in x, its dynamic delay in y is the difference between its dynamic delay in x and $\delta(t)$, the dynamic delay of the transition that fired. An important property of time Petri nets is that their state space (i.e., the set of states reachable from the initial state) can be fully represented as a finite graph [3].

4 PETRI NET MODEL OF THE CAVE

Our first objective was to build a Petri-net-based model of the CAVE. This activity turned out to be more difficult than we had anticipated because there were no documents describing in sufficient detail the interactions among CAVE components and the effect of delays introduced by the components on the overall CAVE behavior. We did have access to some high-level descriptions of the CAVE [12, 17, 20] and to operational specifications for some of the components. We also conducted interviews with CAVE developers when these descriptions proved inadequate. The time Petri net that we defined is the first formal model of the CAVE's operational and timing behavior.

In brief, the CAVE consists of the following three main subsystems. First, the tracker subsystem obtains input data about the position and orientation of the CAVE viewer. This subsystem also calibrates the data in order to reduce noise in the data. The main subsystem uses this data to compute the images to be displayed on the four CAVE walls and renders (i.e., draws) the images. Finally, a monitor subsystem displays the images on four screens. In this section, we first summarize the behavior of each subsystem and then describe how the subsystems are modeled in our time Petri net.

4.1 Cave subsystems

We will now provide a brief description of the functional subsystems of the CAVE. We understand that this organization is fairly common among VR environments.

Tracker subsystem. This subsystem computes the position and orientation of the head and wand of a CAVE viewer. Measured data are sent to two SGI Onyx hosts in the main subsystem, where the images to be displayed are computed. In brief, the tracker transmits a pulsed direct current DC magnetic field that is simultaneously measured by all the receivers in the Ascension sensors. These receivers are located on the viewer's eyeglasses and wand; they provide input data about the position and orientation of the viewer's head and wand. The signal read by the antenna located on the viewer's eyeglasses provides six readings, corresponding to the six degrees of freedom of the viewer's head. This information is important because it allows the VR system to compute the viewer's perspective on the scene being displayed.

An additional antenna located on the wand tracks the wand's position and orientation. This antenna works in a similar way to the antenna on the viewer's head. In addition, the wand has three buttons and a pressure sensitive joystick. The joystick is a two-dimensional device that allows the viewer to enter navigation information.

The buttons allow the viewer to set modes and select options.

The tracking sample is synchronized with the leading edge of the monitor signal coming from the display subsystem. Once a tracker sample is obtained, it is calibrated by electronic filters that reduce the noise present in the input data. The tracker communicates with the rest of the VR system through a 33.6 Kbaud serial line connected to an IBM PC. The PC, which also takes input from the joystick and buttons, is connected to the two Onyx hosts through a high-speed fiber-optic network link.

Main subsystem. This subsystem creates images to be displayed on the walls of the CAVE. The created images are stored in a buffer shared with the display subsystem. Image creation is accomplished in two steps. First, an image computation process defines the geometric features of each image to be displayed. The main purpose of this process is to identify the objects that will appear in each image. Second, an image rendering process defines the full visual representation of the image and stores it in the shared buffer. A rendering process running on one of the Onyx hosts reads the data from the tracker and copies it to the internal memory shared by the Onyx processors. Given that there are four walls in the CAVE and each SGI-Onyx has only 3 graphics pipelines, two Onyxes are required to render the four walls. After reading tracker data, Onyx 1 forks a master process. The master process first forks a network process to communicate with Onyx 2. This system computes and renders the image for the bottom wall of the CAVE. Next, the master process forks two additional processes on Onyx 1 that compute and render the left and right walls. Finally, the master process proceeds to compute and render the front wall.

The CAVE implementation uses double buffering to avoid interference between the main subsystem and the display subsystem. The buffer between these subsystems consists of two components. While the main subsystem is writing into one buffer component, the display subsystem reads from the other component and vice versa.

The processes rendering the four images must be synchronized with each other before the images are displayed on the CAVE walls. Whenever a process finishes an image, it sends a message to the master process and suspends itself while waiting for a response from the master process. Upon completion of all four images, the master process instructs the other processes to swap the double buffer; buffer swapping takes place at the next monitor cycle.

Image displaying subsystem This subsystem consists of a large screen, high resolution, passive (or active) stereo, projection display well-suited for large audiences. The swapping between the front and back buffers is synchronized with the leading edge of the monitor which has a frequency of 48 Hz. When all the four walls are ready, swapping takes place at the next edge of the monitor signal. If any of the walls is not ready to swap, the monitor signal is ignored and a new monitor signal is issued at the next monitor cycle (i.e., after 20.8 ms). Once the buffers are swapped the four images are displayed on the CAVE walls.

4.2 Time Petri net model

Figure 1 shows a time Petri net model of the CAVE virtual environment. This model was entered into Cabernet using Cabernet's graphical editor. Subsequently, we ran numerous experiments on the Petri net, which are discussed in the next section. Except when stated otherwise, assume that transitions appearing in the Petri net have either zero or negligible delays (i.e., because they model synchronization among CAVE processes or short process computations).

The Petri net in Figure 1 models all the subsystems of the CAVE as well as the interactions among the subsystems. This net has 48 places and 35 transitions.

Places *Head*, *Wand*, and *Button_Input* represent input sources from a CAVE viewer. These places are initally marked, meaning that input data is available when an experiment is started. Transition *Head_Wand_Input* fires whenever *Head* and *Wand* are marked. The *Tracker_Obtain_Data* transition has an interval delay of [10.4, 10.4]. Thus, this transition fires every 10.4 ms in order to model a 96 Hz *Monitor* signal. Thus, a token appearing in *Tracker_Got_Data* signifies that the data has been sampled from the tracker sensors.

Transition *Trans_Delay* represents the sending of data from the tracker to the IBM PC. When transition *Trans_Delay* fires, a token is deposited in place *PC_Receive*, meaning that the PC has received tracker data. Transitions *Conversion* and *Calibration* capture computations performed by the IBM PC. Transition *Write_enabled* models synchronization on the line connecting the IBM PC to the SGI Onyx hosts. When place *Read_Write_Lock* is marked, transition *Write_enabled* can fire, meaning that the PC can send data to the Onyx hosts. When this happens, tokens are deposited in places *Rendering_Available* and *Data_Ready.* Tokens in places *Data_Ready* and *Used_Data* (which is initially marked) enable transition *Replace_Old_Data.* The firing of this transition deposits a token in places *New_Data* (meaning that new data are available for drawing a new set of images) and *Read_Write_Lock* (meaning that the lock on high-speed communication link between the PC and the Onyx hosts has been released).

Now transition *Use_New_Data* is enabled. When this transition fires, a token is deposited in place *Tracker_Data_For_Rendering*, signifying that the computation of the images can actually start. The firing of transition *Use_Old_Data* signifies that the tracker fell behind the image computation processes. In this case, old tracker data is reused to perform a new rendering. In the net, this happens when place *Used_Data* is marked and place *Data_Ready* is not marked. As with transition *Use_New_Data*, firing transition *Use_Old_Data* removes the token from place *Rendering_Available* and deposits a token into place *Tracker_Data_For_Rendering*. When this place is marked, transition *Fork_Process* fires and deposits a token in places *Master_Process1*, *Onyx1_Barrier* and *Forked_Network_Process*.

Place *Master_Process1* represents the starting point of the master process. This process spawns two additional processes, which we model by firing two additional transitions labeled *Fork_Process*. The master process and the two spawned processes proceed to compute and render images for the front wall, right wall, and left wall of the CAVE. These computations are captured by the transitions labeled *Comp_Rend_Front_Wall*, *Comp_Rend_Right_Wall*, and *Comp_Rend_Left_Wall*.

When transition *Comp_Rend_Front_Wall* is fired, a token is deposited in places *Swp_Rdy_1* and *Wait_Swap_1*. Place *Swp_Rdy_1* is used for synchronization among the processes computing the four walls. Place *Wait_Swap_1* is used for synchronization between the display subsystem and the main subsystem. The behavior of transitions *Comp_Rend_Right_Wall* and *Comp_Rend_Left_Wall* is similar to the case of transition *Comp_Rend_Front_Wall*.

The image to be displayed on the bottom wall is computed on the second Onyx host in parallel with the other three walls. Place *Rendering_4th_Available* captures a mutual-exclusion lock on the network between the two Onyx hosts. When this place, which is initially marked, and place *Forked_Network_Process* are marked, transition *Shared_Mem_Write_Enabled* is fired, meaning that data is transferred from Onyx 1 to Onyx 2. At this point, Onyx 2 computes and renders the bottom wall, which is captured by the firing of transition *Comp_Render_Bottom_Wall*. When this happens, tokens are deposited in places *Swp_Rdy4* and *Wait_Swap_4*. The token in place *Swp_Rdy4* enables transition *Onyx_Barrier_reached* whose firing adds a token to place *Ready_For_Swap*.

Transition *Swap_Buffer* captures the swapping of the two components in the double buffer between the main subsystem and the display subsystem. This transition is enabled when places *Swp_Rdy_1*, *Swp_Rdy_2*, *Swp_Rdy_3*, and *Ready_For_Swap* are marked, signifying that all four images have been computed. If all these places are marked, transition *Swap_Buffer* is fired as soon as a token appears in place *Mon_Swap*. This place captures the monitor signal that synchronizes buffer swapping. If, however, any of the input places is not marked when a token appears in *Mon_Swap*, the token in place *Mon_Swap* is removed by the firing of transition *Mon_Sync_Sink*. In this case, the token will reappear in *Mon_Swap* after 20.8 ms.

The firing of transition *Swap_Buffer* adds a token to places *Rendering_Available* and *Rendering_4th_Available*. A token in the first of the two places signifies that the main subsystem can start the computation of a new set of images. A token in the other place signifies that the communication network between the Onyx hosts is available again.

Transition *Swap_Buffer* also adds tokens to places *Swap1* through *Swap4*. These places, along with places *Wait_Swap1* through *Wait_Swap4* enable the display processes. The displaying actions are modeled by four sets of transitions, one for each of the display devices used by the CAVE. In particular, when transition *SwapComm1* fires the front wall is able to read from the buffer. Transition *Swap_FW_Complete* captures the actual reading. Transition *Display_FW_Complete* models the displaying on the walls. This transition has a delay interval of [20.8, 20.8] in order to capture the time required by the walls to display the images. The behavior of the transitions modeling the other three walls is similar.

When the images have been displayed on all four walls, transition *Display4WallsComplete* is fired, which adds a token to place *Monitor_Swap*. Note that the total amount of time required to return a token to this place is 20.8 ms.

We observe that the static delay of most transitions in our Petri net is a point, rather than an interval. A transition has a point delay when its earliest and latest static firing times are identical. There are several reasons for this fact. Some transitions model computations that require a negligible amount of time. We defined the static delay of these transitions to be [0, 0], meaning that the transition must always fire as soon as it becomes enabled unless it is disabled by another fireable transition. Other transitions model synchronous events, such as the monitor clock at 48 Hz or the tracker sampling period at 96 Hz. For instance, we capture the beginning of each monitor cycle by firing a transition with a delay interval of [20.8, 20.8], meaning that a new cycle begins exactly every 20.8 ms.

5 EMPIRICAL RESULTS

We conducted numerous experiments with the Cabernet toolset for the analysis of Petri-net models [22]. Our experiments used either simulation or automatic verification techniques on our model of the CAVE. Cabernet performs verification by applying standard reachability analysis techniques. Starting from the initial net state, Cabernet iteratively explores states reached by firing fireable transitions. However, whenever a new state is found, Cabernet does not check whether the state has been visited previously. For this reason, Cabernet can only verify so-called *bounded safety* and *bounded liveness* properties. These properties hold within a time interval starting with the initial net state [22].

We used automatic verification to establish certain bounded safety properties of our net models. For instance, we checked that deadlock cannot occur within 40 ms from the beginning of an experiment. Deadlock is possible whenever the state space contains a nonfinal state without successors. This experiment took less than two hours of CPU time on a Sun Sparcstation IPC with 24 MBytes of memory. However, we were unable to use the verification capabilities of Cabernet for experiments whose duration was greater than 40 ms because of the state explosion problem. In the sequel, we summarize relevant simulation experiments with our Petri-net model.

In general, our simulation experiments differ from each other in the way we associate delay intervals with transitions appearing in our Petri-net model. The first experiment that we discuss is the simulation of the normal behavior of the CAVE. The goal of this experiment is to define a baseline for the timing of CAVE events. The second and third experiments are aimed at observing the effects of delays on the arrival of head data on CAVE behavior. The fourth, fifth, and sixth experiments impose delays on the processes that compute and render images. In practise, such delays can occur when complex images (i.e., images containing many objects) must be drawn. The goal of the seventh experiment is to establish absence of starvation in the CAVE. Starvation is an erroneous condition in which a process cannot make progress because it lacks a required resource, although the resource never becomes permanently unavailable.

All experiments reported below were run on our Sun Sparcstation IPC with 24 MBytes of memory.

5.1 Normal Behavior

This experiment is aimed at observing normal (i.e., correct) CAVE behavior. Relevant transition delays that we used for this experiment are shown in Table 1. The delay on the arrival of tracker data reflects the 96 Hz frequency of the tracking devices. The interval delay on transition *Trans_Delay* models transmission of 224 bits over a 33.6 KBaud serial line between the tracker and the IBM PC. The 224 bits consist of 12 16-bit words giving the position and orientation of the viewer's head and wand. Two additional 16-bit words start and end the transmission of information. The delays on the transitions corresponding to the image computation processes were set to a small amount (e.g., 1 ms), in order to model a simple CAVE application. Finally, the delay on the monitor transition is set to 20.8 ms to capture the standard delay of all display devices.

Transition name	*Time intervals in ms.*
Tracker_Obtain_Data	[10.4, 10.4]
Trans_Delay	[6.7, 6.7]
Comp_Render_Front_Wall	[1, 1]
Comp_Render_Right_Wall	[1, 1]
Comp_Render_Left_Wall	[1, 1]
Comp_Render_Bottom_Wall	[1, 1]
Display4WallsComplete	[20.8, 20.8]

Table 1: Time intervals of key transitions.

The timing of relevant events is shown in Table 2. The computation of all images is completed at 19.4 ms. This is after tracker data is sampled, the sampling is synchronized with the Monitor signal, the sampling is sent to the IBM PC for calibration, and the images are computed by the Onyx hosts. Because these activities are completed before the end of the first monitor cycle at 20.8 ms, the images are displayed, as expected, upon completion of the second monitor cycle (i.e., at 41.6 ms).

Transition name	*Firing time in ms.*
Comp_Render_Front_Wall	19.4
Comp_Render_Right_Wall	19.4
Comp_Render_Left_Wall	19.3
Comp_Render_Bottom_Wall	18.3
Display4WallsComplete	41.6

Table 2: Transition firing time for the four walls.

5.2 Arrival of head data with a delay of 5 ms

In the first experiment the data is immediately available at time zero. In the second experiment, we modify the model to make head data arrive at time 5 ms. Table 3 shows that the drawings on all the walls are again completed by time 20.8 ms, despite the additional delay. This is so because head data must be synchronized in the tracker with a Monitor signal at time 10.4 ms. Thus, the delay that we introduce is absorbed by the tracker subsystem before the main subsystem receives the data. Again, the images are displayed on the four walls at time 41.6 ms.

Transition name	*Firing time in ms.*
Comp_Render_Front_Wall	19.4
Comp_Render_Right_Wall	19.4
Comp_Render_Left_Wall	19.3
Comp_Render_Bottom_Wall	18.4
Display4WallsComplete	41.6

Table 3: Transition firing time for the four walls.

5.3 Arrival of head data with a delay of 15 ms

For this experiment we further increase the delay of the head data to 15 ms. Our goal is to see how a delay of more than 10.4 ms (i.e., the sampling period of the monitor signal) affects the displaying of images. The increased delay causes *Head* and *Wand* data to be read at time 20.8 ms, rather than 10.4 ms. In this case, buffers are swapped at time 41.6 ms and the four images are displayed 62.4 ms after the start of the experiment (see Table 4).

Transition name	*Firing time in ms.*
Comp_Render_Front_Wall	29.8
Comp_Render_Right_Wall	29.8
Comp_Render_Left_Wall	29.7
Comp_Render_Bottom_Wall	28.8
Display4WallsComplete	62.4

Table 4: Transition firing time for the four walls.

5.4 Computation of front wall in 5 ms

In this experiment we increase the time to draw the front wall from 1 ms to 5 ms by changing the interval delay of transition *Comp_Render_Front_Wall* from [1, 1] to [5, 5]. Here we are working on the assumption that the image on the front wall is far more graphics intensive than the images on the other walls, resulting in a long computation time for the front wall. Data at the head and wand is assumed to be available at the start of the simulation, similar to the first experiment.

In this case, the left, right, and bottom walls must wait for the front wall to finish its computation. From Table 5 we observe that the three walls complete their drawing by time 20.8 ms; however, the front wall fails to do so, which causes all the walls to miss a monitor cycle. From time 22.9 ms to time 41.6 ms the processes computing all walls are idle while they wait for synchronization with the next monitor cycle. The interesting result of this experiment is that a relatively small increase in the computation of the front wall is magnified to a delay of 20.8 ms on the displaying of the images.

Transition name	*Firing time in ms.*
Comp_Render_Front_Wall	22.9
Comp_Render_Right_Wall	19.0
Comp_Render_Left_Wall	19.0
Comp_Render_Bottom_Wall	18.0
Display4WallsComplete	62.4

Table 5: Transition firing time for the four walls.

5.5 Computation of front and right walls in 5 ms and 10 ms

This experiment is similar to the previous one, except for the time to compute the right wall being increased to 10 ms. This is achieved by changing the delay on transition *Comp_Rend_Right_Wall* to [10, 10]. Table 6 shows that the added delay on the righT transitions are fired within 1 ms, except for *Comp_Render_Front_Wall*, which fires at 42.7 ms. However, at time 20.8 ms the next tracker input is sampled; this input reaches the image computation processes after a delay of 6.77 ms, approximately at time 28 ms. In our Petri net, this phenomenon is modeled by tokens appearing again in the input places of the four image computation transitions at time 28 ms. Thus, all walls begin computing a new image except for the front wall, which is still working on the old image, as evidenced by multiple tokens accumulating in place *Master_Process3*. At time 62.4 ms the next monitor signal arrives, as signified by the appearance of a token in place *Mon_Swap*. At this time, the left, right and bottom wall have processed the first and third tracker samples; however, the front wall has only processed the first sample.

In this case, the frame drawn by the front wall lags the frame drawn by the other three walls, which can result in simulation sickness on the part of the CAVE viewer. This phenomenon was confirmed by a CAVE developer, who discovered the anomaly independently and concurrently with us [21]. At the time of this writing, the anomaly has been corrected by introducing an additional synchronization between the tracker subsystem and the main subsystem.

We tried to detect this error using the automatic verification capabilities of Cabernet. However, this experiment requires that all states reachable within 62.4 ms from the beginning of the experiment be explored. The state explosion problem prevented Cabernet from completing this experiment. We discontinued our run after three hours of CPU time on our Sun Sparcstation IPC.

Transition name	*Firing time in ms.*
Comp_Render_Front_Wall	42.7
Comp_Render_Right_Wall	19.7
Comp_Render_Left_Wall	18.7
Comp_Render_Bottom_Wall	17.7
Display4WallsComplete	83.2

Table 7: Transition firing time for the four walls.

5.7 Absence of starvation on front wall

For this experiment we used a predicate-checking capability of the Cabernet toolset. In particular, we checked whether the computation of the front wall must be completed before time 20.8 ms under normal operating conditions. Thus, we used the same Petri net as for the first experiment. In this case, Cabernet returns the value true, indicating that we can guarantee the computation of the front wall to be completed within 20.8 ms. The analyzer generates the reachability graph and does a graph traversal in order to determine if this assertion is true or not.

6 CONCLUSIONS AND FUTURE WORK

Our preliminary results indicate that Petri-net-based techniques can effectively support the design and validation of virtual reality environments. To our knowledge, ours is the first comprehensive model of a VR environment. We are also strongly encouraged by our ability to find a flaw in the CAVE version that we studied.

We observe that the synchronous aspects of the CAVE's behavior have a significant effect on our Petri-net model. As we noted earlier, most of our net transitions have point, rather than delay, intervals. We suspect that synchronous aspects will play less of a role in models at lower levels of abstraction than our current model. However, the predominance of transitions with point delays suggests that we should experiment also with less expressive models than Merlin and Faber's time Petri nets. In general, less expressive models are more conducive to automated verification and vice versa.

At present, we are pursuing several additional research directions. First, we wish to "hide" Petri-net models from developers of VR applications. In particular, we are in the process of developing a front-end system with an easy-to-use graphical user interface. Developers would use this interface to enter descriptions of virtual environments. Subsequent2734nets.

[illegible]

toolset. *IEEE Trans. Softw. Eng.*, 17(11):1204–1222, Nov. 1991.

[2] C. Bellettini, M. Felder, and M. Pezzé. Merlot: A tool for analysis of real-time specifications. In *Proceedings of the Seventh International Workshop on Software Specification and Design*, pages 110–119, Redondo Beach, California, Dec. 1993.

[3] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273, Mar. 1991.

[4] G. Bucci and E. Vicario. Compositional validation of time-critical systems using communicating time Petri nets. *IEEE Trans. Softw. Eng.*, 21(12):969–992, Dec. 1995.

[5] U. Buy and R. H. Sloan. Analysis of real-time programs with simple time Petri nets. In *Proc. 1994 Internat. Sympos. on Software Testing and Analysis*, pages 228–239, Aug. 1994.

[6] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verus: A tool for quantitative analysis of finite-state real-time systems. In *Second ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 75–84, June 1995.

[7] S. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *IEEE Real-Time Systems Symposium*, 1994.

[8] R. Cleaveland, W. Elseaidy, and J. Baugh. Verifying an intelligent structure control system: A case study. In *Proc. Real-Time Systems Symposium*, pages 271–275. IEEE Computer Society Press, Dec. 1994.

[9] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng.*, 22(3):161–179, Mar. 1996.

[10] J. C. Corbett. Timing analysis of Ada tasking programs. *IEEE Trans. Softw. Eng.*, 22(7):461–483, July 1996.

[11] J. C. Corbett and G. S. Avrunin. A practical technique for bounding the time between events in concurrent real-time systems. In T. Ostrand and E. Weyuker, editors, *Proc. 1993 Internat. Sympos. on Software Testing and Analysis*, pages 110–116, Cambridge, Massachusetts, June 1993. ACM.

[12] C. Cruz-Neira. *Virtual Reality Based on Multiple Projection Screens: The CAVE and its Applications to Computational Science and Engineering.* PhD thesis, University of Illinois at Chicago, May 1995.

[13] S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz. Application and experimental evaluation of state space reduction methods for deadlock analysis in Ada. *ACM Trans. Software Engineering and Methodology*, 3(4):340–380, Oct. 1994.

[14] R. Gerber and I. Lee. A layered approach to automating the verification of real-time systems. *IEEE Trans. Softw. Eng.*, 18(9):768–783, Sept. 1992.

[15] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzé. A unified high-level Petri net formalism for time-critical systems. *IEEE Trans. Softw. Eng.*, 17(2):160–172, Feb. 1991.

[16] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

[17] E. Kennan and S. Bolton. Tracker latency analysis document. Electronic Visualization Lab, the University of Illinois at Chicago, Mar. 1996.

[18] P. M. Merlin and D. J. Farber. Recoverability of communication protocols—implications of a theoretical study. *IEEE Trans. Communications*, COM-24(9):1036–1043, Sept. 1976.

[19] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE Trans. Softw. Eng.*, 18(9):794–804, Sept. 1992.

[20] D. Pape. Cave user's guide. Electronic Visualization Lab, the University of Illinois at Chicago, Dec. 1996.

[21] D. Pape, Apr. 1997. Personal communication.

[22] M. Pezzé. Cabernet: A customizable environment for the specification and analysis of real-time systems. Technical report, Dipartimento di Elettronica e Informazione, Politecnico di Milano, June 1994.

[23] C. Ramchandani. Analysis of asynchronous concurrent systems using timed Petri nets. Research Report MAC-TR 120, Massachusetts Institute of Technology, Feb. 1976.

[24] S. M. Shatz and W. K. Cheng. A Petri net framework for automated static analysis of Ada tasking behavior. *Journal of Systems and Software*, 8:343–359, 1988.

[25] W. M. Zuberek. Timed Petri nets: Definitions, properties, and applications. *Microelectronics and Reliability*, 31(4):627–644, 1991.

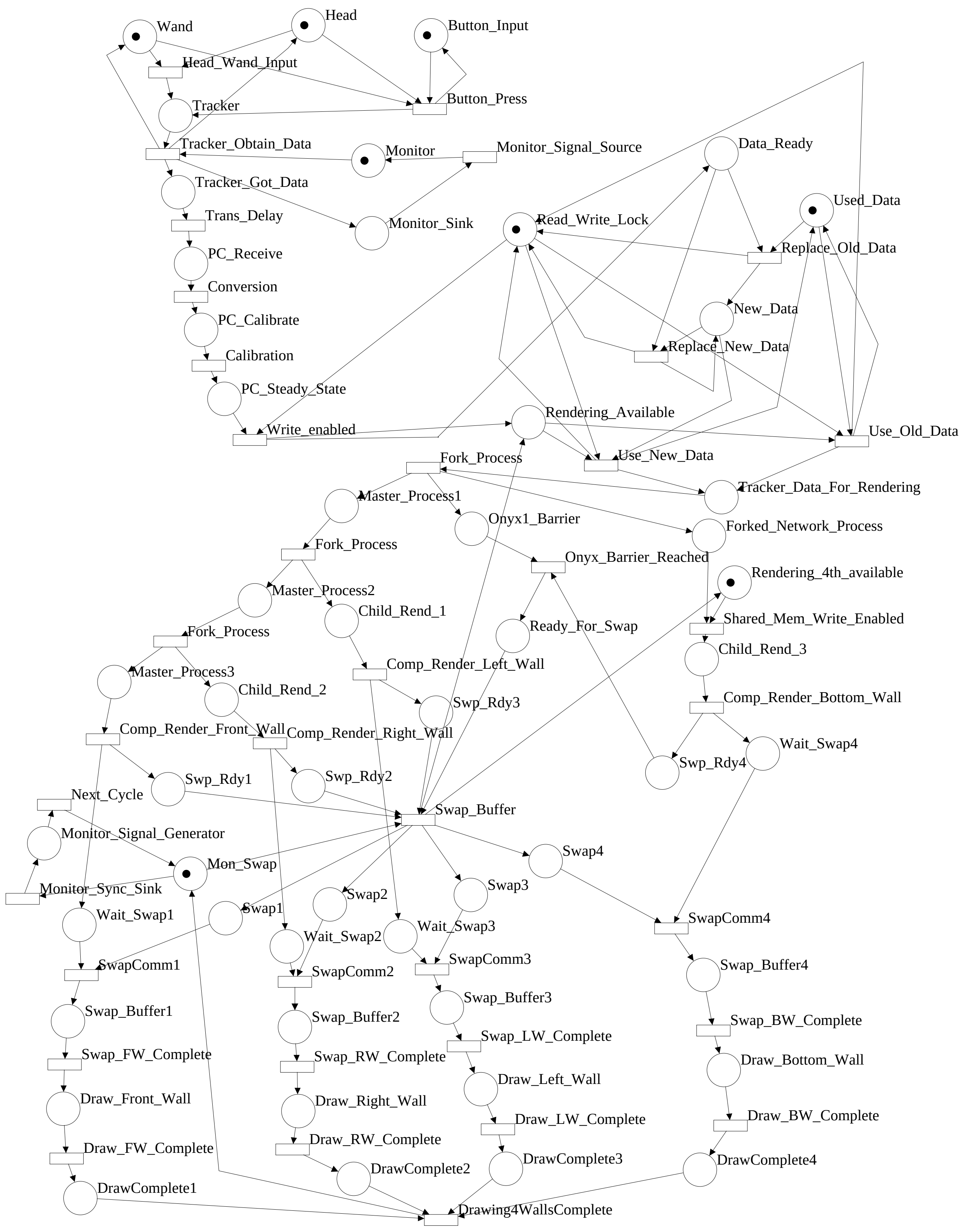

Figure 1: Time Petri net model of the CAVE virtual environment.