μC++ Monitoring and Visualization

Reference Manual

Version 1.1

University of Waterloo

Peter A. Buhr

February 4, 1999

Contents

1 Introduction

The toolkit is built around μC++ [BS96], which is an extended version of C++ providing light-tasking facilities using a shared-memory model. Most of the ideas presented here are language independent, and thus, are applicable to a large number of concurrent and non-concurrent languages. Finally, this work is based in UNIX[1], using X/Intrinsic/Motif because of their current popularity and availability.

Monitoring is the process of asynchronously (or synchronously) collecting information about a program's execution, which can be displayed during the program's execution or afterwards. When monitoring a sequential program, it is generally true that the monitoring does not change the data values generated or the order that the values are generated. However, the non-determinism and temporal aspects of concurrency make it impossible to monitor a concurrent program without affecting its execution. Hence, as soon as a probe is attached to monitor behaviour, the probe affects the system it is measuring so that the system may not behave as it did before the probe was attached, called the probe effect. At best, a designer of monitoring tools can only strive to minimize probe effects.

Both statistical and exact monitoring capabilities are needed. Statistical monitoring is done by periodically probing a target program's memory from a separate task. This form of monitoring provides imprecise information about a program, but has a lower probe effect than exact monitoring. Exact monitoring requires that the target program generate events whenever interesting operations occur. This form of monitoring provides precise information about the program, but has a much higher probe effect.

Monitoring may be on data or events, where data monitoring show a variable as it changes and event monitoring show when an event has occurred. Data monitoring can be considered as event monitoring, where the event is assignment to the variable; however, it is the value that is important and not the fact that the assignment occurred.

Both statistical and exact monitoring can be done implicitly and explicitly, i.e., the monitoring is performed automatically by the underlying runtime system or the user explicitly is involved in specifying the monitoring.

Visualization is the process of displaying monitored information in a concise and meaningful way. The raw monitored data can be presented directly as a sequence of data values; however, there is usually too much data for people to interpret and comprehend. Instead, the raw data is transformed into another form that allows large amounts of data to become comprehendable. For example, displaying data in the form of a graph may allow large amounts of information to be understood at a glance. Just as some graphs can represent multiple dependent and possibly independent variables [Tuf83, p. 40], so can a visualization display represent multiple values and events. As more information is represented by a single display, the more complex the display becomes, and usually, the more difficult it is for a user to understand the display and comprehend its meaning. Colour is often an essential element of visualization because it provides yet another mechanism for condensing and differentiating information in a display.

2 Statistical Monitoring

In general, a program generates too many data values and too many events occur during execution to be of interest to or understood by a programmer. A programmer can often tell if a program is working according to its design simply by examining its general behaviour. For example, are approximately the right number of tasks created, blocked, or destroyed at approximately the right time? A program's general behaviour can be shown by statistically sampling its execution. For example, program profilers, like gprof [GKM82], provide information about execution "hot spots" by sampling the current execution location. Execution hot spots can then be examined as possible locations for optimization to improve program performance. Statistical

[1]UNIX is a registered trademark of AT&T Bell Laboratories

information may be valuable for debugging a new algorithm or simply to understand the differences among algorithms (that are already debugged). Statistical sampling can often be done without having to insert code in an application so sampling can be done on existing execution modules. Furthermore, the probe effect may be lessened on multiprocessor shared-memory architectures as the sampling can occur on a separate processor (special sampling hardware can also be used). The drawback to statistical sampling is that important events may occur between sampling, and hence, be missed.

In a concurrent program, a statistical examination of tasks that form a program, can tell a programmer about the level of concurrency that is provided by a particular algorithm or by different data manipulated by an algorithm. Statistical feedback about the communication among tasks can also be extremely informative.

3 Implicit/Explicit Monitoring and Visualization

In general, it does not seem possible to provide totally automated monitoring and visualization. That is, no visualization system can anticipate all the different ways that users will want to collect and display their information. Therefore, tools must be directly available to users so they have the freedom to experiment with different approaches. On the other hand, some predefined facilities can be provided so that casual users do not have to become experts to produce simple animation components. Therefore, a monitoring/visualization system is best designed as a toolkit, which provides capabilities at a variety of levels.

This toolkit allows users to participate in the design activity by providing their own tools. Users can easily build their own monitoring and displaying tools from existing tools. Furthermore, these same tools are available to the system implementors to build low-level monitoring and visualization tools for the runtime kernel.

4 μC++ MVD Toolkit

The μC++ Monitoring, Visualization and Debugging (MVD) toolkit supports the following facilities:

- There is a general facility to construct statistical monitoring tools. These monitoring tools can be connected to a (currently small) set of Motif visualization tools to display the monitored results. The statistical monitoring and visualization can occur in real-time or monitored data can be written to a file for post-processing visualization. A user may explicitly indicate in a program the variables that are to be monitored and possibly how they are visualized. There is also an implicit visualization capability to monitor the μC++ runtime kernel.

- There is an exact monitoring facility that is connected to a powerful post-processing event-replay facility [TB96]. All μC++ objects can have certain critical events traced by simply compiling the program with an appropriate flag.

- There is a symbolic debugger that understands multiple shared-memory execution states [BKS96]. It concurrently presents sessions for an arbitrary number of tasks in a μC++ program and allows independent control of each task. As well, it provides facilities to look up local data relative to any task's execution stack. Currently, it has limited support to control non-shared memory applications.

5 Working with X Window/Motif

X Window/Motif were chosen for the basic visualization platform because of their current popularity and availability. Most of the ideas presented here would work equally well with other visualization toolkits, e.g., the xview toolkit or the athena widget set.

6 Structure of X

Version 11, release 6 of the X Window System is the first release intended to support client applications that work in a multi-threaded environment. See “Xlib - C Language X Interface” and “X Toolkit Intrinsics - C Language Interface”, which are both distributed with the X Window System, for a detailed description on how to create concurrent client applications. X provides an abstract locking interface that can be implemented by any system- or user-level thread library. In general, locks in X are owner locks, i.e., they can be re-acquired by the thread of control that currently holds the lock, but they have to be released exactly as often as they were acquired.

At the *Xlib* level, multiple connections to the X server can be created and used concurrently, and multiple threads of control can access the same connection. Automatic fine-grain locking is done internally by the *Xlib* library. At the *X Toolkit Intrinsics* level (Xt), multiple application contexts can be created, each of which has internal display connections. In general, locking at this level is done on the basis of application contexts. Additionally, a global lock for the whole application can be used by widget developers, if global data has to be protected.

The Xt is designed to use an event loop and callbacks. That is, an application program passes control to X (by calling XtAppMainLoop), after it has built its windows and installed its callback routines. X then waits for events either from the user’s terminal or other sources, such as timers, and it either invokes internal code, e.g., puts down a menu, and/or it interacts with the application code by invoking the specified callback routines.

The new X locking mechanisms can be used in various ways but the most common approach is the following. A dedicated thread of control is used to receive events from the X server and dispatches them to other objects. Concurrently, other threads can call X/Xt library routines to change application values.

7 X Window System and μC++

While the X client libraries are configured to work with the thread libraries of multiple vendors, some additional changes were required to make them compatible with μC++. As a first step, wrapper routines are planted, so that the X libraries, which are programmed in C, can internally use the uLock and the uCondition classes of μC++. This enables cooperation with the basic scheduling and lightweight-blocking mechanisms of μC++.

When using a vendor’s kernel thread package, blocking I/O only blocks the thread of control that calls the appropriate operating system routine, for example select. Since μC++ provides user-level threads, its special lightweight-blocking I/O library must be used to achieve the same effect. Therefore, calls to the uSelect member routine of the uIOCluster are implemented in the X source code (by using wrappers from μC++ to C) at places where usually select is called. A dedicated uIOCluster object (see [BS96] for a description) is created in an X application, because the X libraries use data that is private to each UNIX process, like file descriptors, etc. A class uXwrapper is provided which migrates a task to this cluster when an object is created, and migrates it back when the object is destroyed. This class can be used for automatic migration in every routine that calls X library routines, like in:

```
#include <uXlib.h>

Display *dpy;

void createInterface() {
      uXwrapper dummy;                // automatic migration
      dpy = XOpenDisplay( NULL );
      // . . .
}                                     // automatic migration back on destruction of dummy
```

In a μC++ application, timer interrupts are used to realize preemptive scheduling. Certain UNIX system calls return a failure value and set the error number, if a timer interrupt occurs while the system call is executed. This is partly handled in the Xlib, except for initialization of the socket connection with the X server. To prevent obscure error messages, preemption of μC++ is turned off, whenever a connection is established. Again, this mechanism is planted into the *Xlib* using wrappers from μC++ to C.

Unfortunately, to this end, the Motif widget library is not reentrant. Therefore, every call that accesses a Motif widget has to be explicitly made mutual exclusive for the whole application and the internal locking mechanisms of the Intrinsics library are largely obsolete. This can be seen in the the following example:

```
#include <uC++.h>
#include <uXlib.h>
#include <X11/Intrinsic.h>

Widget my_widget;
XtApplicationContext app;

void changeValue( int x ) {
      XtAppLock( app );
      XtProcessLock();
      XtVaSetValues( my_widget, XmNvalue, x, NULL );
      XtProcessUnlock();
      XtAppUnlock( app );
}
```

8 X Window Callbacks and C++

The following coding convention was developed for working with X callbacks in C++. Callbacks are C routines (C linkage) that are dynamically registered with X and subsequently invoked by X when particular X-window events occur, such as a button press on the mouse or screen. This is the mechanism that X uses to interact with an application. The following coding conventions allow C++ applications that interact with X to still be written in an object-oriented style.

The conventions are as follows:

- All widget definitions should be defined as a class and that widget's callback routines should be private static members, as in:

```
class widget {
      int foo;
      void bar() { . . . }
      static xxxCB( Widget widget, widget *This, XmAnyCallbackStruct *call_data ) {
            This->foo = 3;
            This->bar();
      }
  public:
      . . .
};
```

Making the callback routines static members reduces global name-space pollution and a static member routine is treated as a non-member routine so its address can be passed to X routines that are written in C (i.e., static member routines do not have an implicit this parameter).

- When installing a callback routine, as in:

```
widget w;
XtAddCallback( w, XmNactivateCallback, xxxCB, this );
```

the last argument is a pointer to client data that is passed to the callback routine's second parameter. The convention requires that the last argument must always be the value of this, and hence, the second parameter to a callback routine is always a pointer to the type of class that contains the callback routine. Furthermore, the name of the second parameter should be This. Following this convention allows the callback routine to access all the data and routines in the containing widget class as if it was actually a member routine of the widget class (albeit explicitly through the This pointer). For example, in callback routine xxxCB, members foo and bar are both accessed through the This pointer.

Figure 1 shows a program that uses the coding conventions to handle its callback routines.

9 The μX Package

A collection of classes is available to simplify the task of interacting with X and Motif under μC++. By including file:

```
#include <uXlib.h>
```

after <uC++.h>, the basic cooperation is enabled and the class uXwrapper can be used for automatic migration (see Section 7). Additionally, the file

```
#include <uXmlib.h>
```

can be included to access classes to support using Motif: uXmwrapper and uXmCBwrapper. As stated previously, Motif is not thread-safe and mutual exclusion must be acquired before any operation can safely by invoked on Motif widgets. Additionally, if a task calls any X routine to send a request to the X server, this request is buffered in the library. If changes shall become visible immediately, the buffer has to be flushed. All this complexity is hidden within uXmwrapper, as well as migration. That is, if an object of uXmwrapper is created at the beginning of a routine, the task is migrated, the necessary locks are acquired and on destruction of the object the library's event buffer is flushed. The above example then looks like:

```
#include <uC++.h>
#include <uIOStream.h>
#include <uXmlib.h>
#include <uXtShellServer.h>

#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/PushB.h>

class uPushMeWidget {
	static void uQuitCB( Widget widget, uPushMeWidget *This, XmAnyCallbackStruct *call_data ) {
		XtDestroyWidget( This->shell );				// destroy parent and all its subwidgets
	} // uPushMeWidget::uQuitCB

	static void uPushMeCB( Widget widget, uPushMeWidget *This, XmPushButtonCallbackStruct *call_data ) {
		uCout << "Please don't tread on me" << endl;
	} // uPushMeWidget::uPushMeCB

	Widget shell;
  public:
	uPushMeWidget( Widget shell ) : shell( shell ) {
		Widget mainWindow = XtVaCreateManagedWidget( "main", xmMainWindowWidgetClass, shell,
							XmNwidth, 150, XmNheight, 75,
							NULL );
		XmString quitTitle = XmStringCreateSimple( "Quit" );
		Widget menuBar = XmVaCreateSimpleMenuBar( mainWindow, "menuBar",
							XmVaCASCADEBUTTON, quitTitle, 'Q',
							NULL );
		XmStringFree( quitTitle );
		XtManageChild( menuBar );
		XtAddCallback( XtNameToWidget( menuBar, "button_0" ), XmNactivateCallback,
							(XtCallbackProc)uQuitCB, this );
#		define uPushMeTitle "Tread on me"
		Widget uPushMe = XtVaCreateManagedWidget( "uPushMe", xmPushButtonWidgetClass, mainWindow,
							XtVaTypedArg, XmNlabelString, XmRString, uPushMeTitle, sizeof(uPushMeTitle),
							NULL );
		XtAddCallback( uPushMe, XmNactivateCallback, (XtCallbackProc)uPushMeCB, this );
	} // uPushMeWidget::uPushMeWidget
}; // uPushMeWidget

void uMain::main() {
	Widget shell = uServerXtShell->XtCreateApplicationShell( "uPushMeTest" );
	{
		uXmwrapper dummy( shell );
		uPushMeWidget pmw( shell );
		XtRealizeWidget( shell );
	}
	uServerXtShell->XtMainLoopBlock();
	uCout << "Quitter!" << endl;
} // uMain::main

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++-work -g uPushMe.cc -I/u/pabuhr/software/MVD-1.1/inc -I/u/pabuhr/software/MVD-1.1/X11R6/include -I/opt/SUNWmotif/include -L/u/pabuhr/software/MVD-1.1/lib -L/u/pabuhr/software/MVD-1.1/X11R6/lib -R/u/pabuhr/software/MVD-1.1/X11R6/lib:/opt/SUNWmotif/lib /u/pabuhr/software/MVD-1.1/lib/uVisual-d.a -lXm -lXt -lX11 -luX -lSM -lICE -lgen" //
// End: //
```

Figure 1: C++ Callback Coding Style

```
#include <uC++.h>
#include <uXlib.h>
#include <uXmlib.h>

Widget my_widget;

void changeValue( int x ) {
      Xmwrapper dummy( my_widget );
      XtVaSetValues( my_widget, XmNvalue, x, NULL );
}
```

When a callback routine is called, the lock for the application context and the global lock are acquired. If a callback contains a call to a mutex member of a task, this can lead to deadlock situations, if the task also tries to acquire the X locks to perform any changes in the interface. To handle this situation, the uXmCBwrapper class can be used to release the owner locks temporarily and re-acquire them on destruction of the object as in the following example:

```
#include <uC++.h>
#include <uXlib.h>
#include <uXmlib.h>

uTask fred {
      void main();
public:
      void request( int from );
}

fred f;

class MyWidget {
      int id;
      static void anyCallback( Widget w, MyWidget *This, XmAnyCallbackStruct *call_data) {
            XmCBwrapper dummy( w );
            f.request( This->id );
      }
```

One should be aware that between creation and destruction of a callback wrapper no lock is held and therefore, no call can safely be made that accesses Motif data. Additionally, a routine uXmAppMainLoop is provided that performs the necessary locking when Motif is used and replaces the usual XtAppMainLoop. In traditional X programming, the single thread of control passes control to X after it has initialized X and built a series of widgets. There are many kinds of problems that are *not* amenable to the event-loop programming style. In fact, virtually all concurrent applications are not amenable to this style because the application does not want to block waiting for X events. Instead, the concurrent application has other work to do and the X interface is only one component of this work. Therefore, the routine Uxmappmainloop can be called from a dedicated task object.

For convenience, the μX toolkit provides an additional server, called uServerXtShell, which is an instance of:

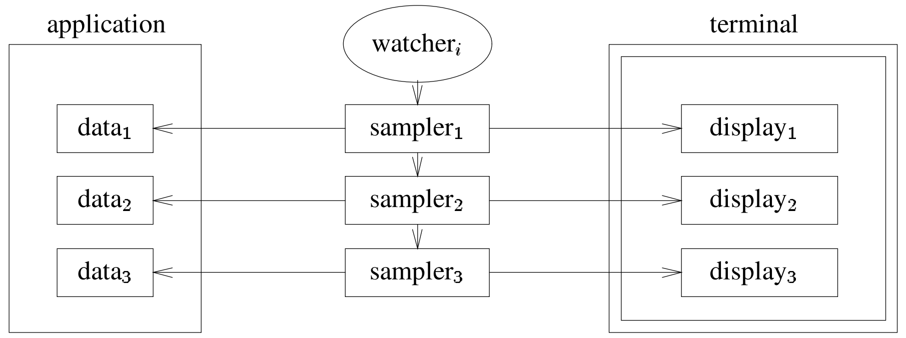

Figure 2: Watcher/Sampler Structure

```
class uXtShellServer {
public:
    uXtShellServer();
    ~uXtShellServer();

    Widget XtCreateApplicationShell( const char *title, String fallBacks[] = NULL );

    void XtMainLoopBlock();
    void XtMainLoopNonBlock();
    void XtMainLoopWait();
}; // uXtShellServer

extern uXtShellServer *uServerXtShell;
```

This server is globally available to μC++ programs that interact with X. The server shell operations are:

XtCreateApplicationShell – creates a new shell widget under which a new widget hierarchy can be created. This routine handles the special case of initializing X/Intrinsic on the first call.

XtMainLoopBlock – acts similarly like the usual XtAppMainLoop. Control is passed to X and the routine eventually returns, when the user interface is destroyed.

XtMainLoopNonBlock – create a dedicated task that executes the main loop. The caller task continues execution.

XtMainLoopWait – waits for the task created with XtMainLoopNonBlock to finish execution. This can be used to synchronize with the destruction of the user interface.

10 Explicit Statistical Monitoring

Explicit statistical monitoring requires a detailed understanding of the low-level structure provided by the μC++ MVD toolkit. Explicit monitoring is performed by two kinds of objects: watcher and sampler objects. Figure 2 shows how watchers and samplers cooperate to produce a display. Both the watcher and sampler objects need to be fairly small and fast because they may be probing an application at a high frequency.

11 Watcher Object

A *watcher* is a task that manages an event list of user-specified sampler objects. At specified intervals, the watcher invokes a sampler object on its event list, and that sampler then inspects part of the application and possibly displays the inspected data. Multiple watchers can be created if the number of samplers is large and/or the sampling frequencies are high; normally a single watcher is sufficient. A watcher is least invasive on machines with multiple processors where the watcher can execute on a separate processor independent of the application. On a single processor machine, a watcher can only be expected to produce a coarse-grained view of the application. Nevertheless, this still can be useful in many cases. The watcher is provided by the μC++ monitoring toolkit and its interface is:

```
uTask uWatcher {
  public:
    void add( uBaseSampler &sampler );        // add sampler to event list
    void remove( uBaseSampler &sampler );     // remove sampler from event list
};
```

The member routines add and remove a sampler to/from the watcher's event list, respectively. Normally, these routines are called by the constructor and destructor of a sampler, respectively.

12 Sampler Object

A *sampler* inspects data values in an application and possibly displays the inspected data. A sampler has two basic operations: poll memory in a particular way to determine some relevant information, and display that information in some useful format. These two operations are invoked by the watcher object that a sampler is associated with. In detail, a sampler indicates to the watcher the frequency in microseconds at which its polling and display operations are invoked. These two frequencies may be different. A sampler must be derived from class uBaseSampler to ensure it has the necessary member routines that are called by the watcher:

```
class uBaseSampler {
  protected:
    int pollFreq, displayFreq;                // poll and display frequency
  public:
    uBaseSampler( uWatcher &w, const int pollFreq, const int displayFreq );
    virtual void poll() = 0;                  // called by watcher
    virtual void display() = 0;               // called by watcher
    void pause();                             // stop sampling temporarily
    void resume();                            // restart sampling
};
```

The sampler is associated with watcher w, which calls the sampler's poll member at frequency pollFreq and display member at frequency displayFreq.

The following generic sampler preforms simple sampling of a numeric memory value:

```
template<class T> class uNumSampler : public uBaseSampler { // T requires: 0, =, >, <, +, /, <<
  public:
    uNumSampler( uWatcher &w,            // watcher where sampler resides
        int pollFreq,                    // polling frequency
        int displayFreq,                 // display frequency
        T &locn,                         // location to be sampled
        char *name,                      // widget title
        T dialMin,                       // absolute minimum
        T dialMax,                       // absolute maximum
        T zero,                          // zero value for type T
        uNumGaugeTL::GaugeType gauge // initial gauge type
    );
    void poll();
    void display();
}; // uNumSampler<T>
```

This sampler examines a numeric object at memory location locn. To make the numeric sampler generic it is necessary to pass a value corresponding to zero for the type T.[2] This simple sampler maintains a running average of the sampled values at location locn as well as the minimum and maximum values. Appendix 19 shows the complete generic numeric sampler.

The generic sampler uNumSampler is used in the following way to sample numeric values in a program:

```
void uMain::main() {
    uWatcher w;                     // create watcher
    int i;                          // target variable
    uNumSampler<int> sd( w, 50000, 500000, i, "i", 0, 50000, 0, uNumGaugeTL::digital ),
    sb( w, 50000, 500000, i, "i", 0, 50000, 0, uNumGaugeTL::bargraph ),
    ss( w, 50000, 500000, i, "i", 0, 50000, 0, uNumGaugeTL::speedometer );

    for ( i = 0; i <= 50000; i += 1 ) uDelay();         // delay necessary on a uniprocessor
} // uMain::main
```

In this example, a watcher is created with three samplers for variable i each displaying the value of i in a different format. (On a multiprocessor, the watcher is automatically created on a separate processor to reduce the probe effect.) When the program is run, the variable i is displayed in three different formats: digitally, as a bar graph, and as a speedometer, as illustrated in Figure 3. In the bargraph, the minimum, average and maximum values appear in the left column, while the current value appears in the right column. The spacing of the values between the top and bottom of the window shows the relative location of the values between the absolute minimum and maximum (0-50000). In the speedometer, there are pointers to the minimum, current, and maximum values and the current value is also displayed digitally at the bottom-centre of the speedometer (the average value is not displayed). Since, the current and maximum values are the same for i, the current and maximum pointer are displayed on top of one another, and hence, appear as a single pointer line. Normally, it is unnecessary to create three samplers to obtain three views of a variable because the pull-down menu, `Display Format`, allows any of the three display forms to be changed dynamically to digital, bar graph or speedometer.

A sampler can be written to sample any data structure and generic samplers can be constructed for common types of data structures so many users will never have to write a sampler. For example, uNumSampler samples any type that has the operations, = > < + / <<, and a 0 (zero) constant. One important aspect of this design is that the sampling frequency and the display frequency are independent. This feature allows filtering of large numbers of values while still being able to capture peak values. For example, the minimum and maximum values can be sampled at a high rate, but writing to the display may only be done every

[2]This is an artifact of the C++ type system.

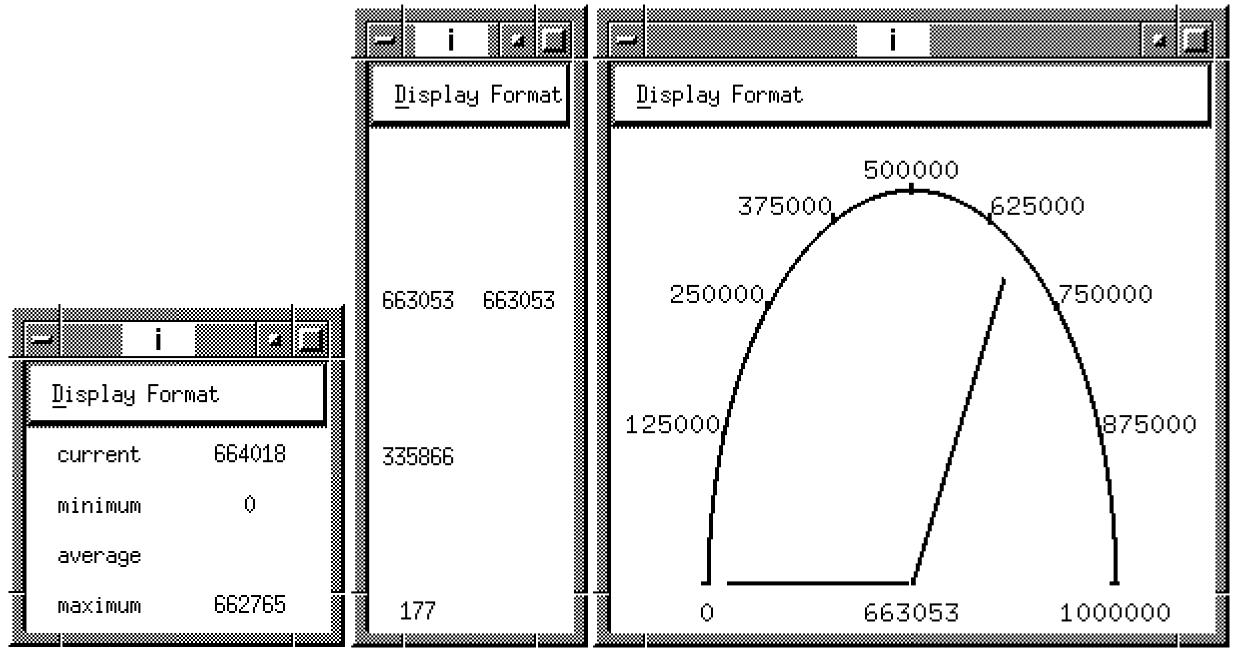

Figure 3: Different Numeric Displays: digital, bar graph, speedometer

1/30 of second because that is the refresh rate of the display device. ([] report that a refresh rate as slow as 1/10 of a second is sufficient for people to perceive smooth motion.) A sampler can also greatly reduce the display updates by not sending an event if the value(s) have not changed. Furthermore, a sampler can dynamically control its poll and display frequency by adjusting the values of the protected variables pollFreq and displayFreq of the base class uBaseSampler. For example, the longer it has been since a change occurred in a value being probed, the less urgent it is to probe again. Finally, it is a sampler's responsibility to deal with non-atomic data structures. For example, when updating a binary tree, there may be a short time interval when the tree is disconnected and a sampler that is probing the tree must be prepared to deal with this. Such samplers must be very pessimistic about the object they are examining and very robust in their probing algorithms. For example, a simple precautionary measure when sampling is to check that a pointer is within the address space before dereferencing it.

Thus, users can write their own samplers or use one of the the predefined samplers. We hope to enlarge the current suite of samplers and displayers as time permits.

13 Coding Conventions

Additional coding conventions are used in the construction of samplers and their display widgets. These conventions follow directly from the X/C++ conventions discussed in Chapter 5. A sampler passes its address to its display widget, just as and the widget does to its callbacks, and the callbacks to X. This approach allows the object receiving the address to communicate back to its creator. In Figure 4, these pointers allow the top communication lines from right to left. (even though the pointers go the other direction). As well, each creator maintains a pointer to the object it created to allow the bottom communication lines from left to right. Hence, there is bidirectional communication among the objects used as follows: output flows from the sampler to the widget to X; input flows from X to the callback to the widget to the sampler and possibly to the watcher.

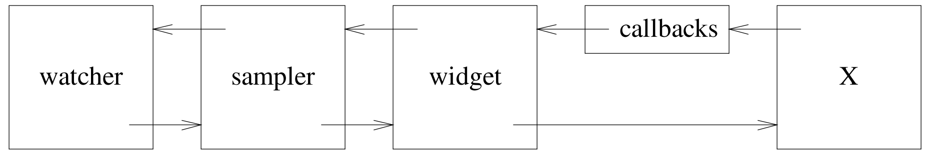

Figure 4: Coding Conventions

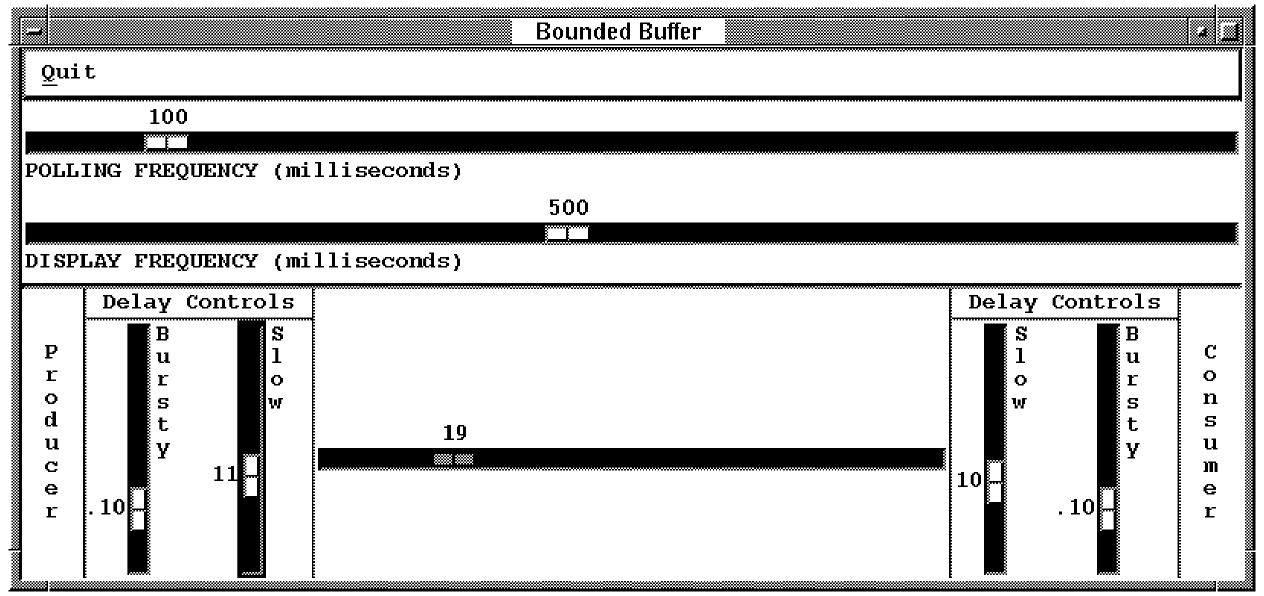

Figure 5: Animated Bounded Buffer

14 Animated Bounded Buffer Example

The following example visualizes the effect of inserting a bounded buffer between a producer and consumer task using synchronous communication (see Figure 5). The buffer is size 100 and the number of elements in the buffer is displayed by a horizontal slider (0 on the left). Feedback controls are provided to adjust the speed of the producer and consumer tasks. These controls adjust random delay cycles, with the random number generated around an average with a particular standard deviation. By playing with the delay controls, it is possible to see that the buffer fills or empties if the speed of the producer and consumer varies only slightly or their speed fluctuates significantly. There are 4 basic components to the animated bounded buffer: display widget, buffer sampler, visual bounded buffer, driver; each is discussed separately.

14.1 Display Widget

While the display widget is the most complex of the components, this results from all of the calls to X to build the display. Since this manual is not an X manual, the details of the display only appear in Appendix 19 and are not discussed. Only the interface to the display widget is Figure 6 is relevant to this discussion.

There are 5 callback routines corresponding to the quit button and 4 sliders. While the quit button halts the program, the slider values have to be available to the driver so it can adjust its execution. Each callback routine has a This parameter, whose value is an instance of the class that contains them. The constructor receives the title for the display and the size of the buffer, which is used to set the maximum size of the buffer slider (horizontal slider). There are 4 get routines to retrieve the values returned from the sliders to

```
#include "uXserver.h"

class uProdConsWidget {
    static void QuitCB( Widget widget, uProdConsWidget *This, XmAnyCallbackStruct *call_data );
    static void prodStdWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void prodAvgWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void consStdWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void consAvgWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );

    Widget parent, bufScaleWD;
    const int NumDecPts = 2;
    float prodStd, prodAvg, consStd, consAvg;
  public:
    uProdConsWidget( const char *title, int bufSize );
    virtual ~uProdConsWidget();

    float getProdStd() {
        return prodStd;
    } // uProdConsWidget::getProdStd

    float getProdAvg() {
        return prodAvg;
    } // uProdConsWidget::getProdAvg

    float getConsStd() {
        return consStd;
    } // uProdConsWidget::getConsStd

    float getConsAvg() {
        return consAvg;
    } // uProdConsWidget::getConsAvg

    void setValue( int val );
}; // uProdConsWidget
```

Figure 6: Display Widget

adjust the producer and consumer tasks and one set routine to move the buffer slider to the current number of items in the buffer.

14.2 Buffer Sampler

Figure 7 shows the sampler that periodically samples the size variables in the monitor and updates the display widget accordingly. The sampler's constructor has the same first 5 parameters as the numeric sampler, uNumSampler, followed by the size of the monitor's buffer to set the horizontal slider. The constructor uses these parameters to initialize the base sampler, location to be sampled, and display widget, respectively. The constructor and destructor add and remove the sampler to/from the watcher in the appropriate manner. The poll routine makes a copy of the buffer's current size and the display routine displays this value in the widget's horizontal slider. Finally, the 4 get routines return the values of the sliders directly from the display widget.

```
#include "uBaseSplr.h"
#include "uProdConsWD.h"

class uProdConsSampler : public uBaseSampler {
        const int &locn;
        int curr;
        uProdConsWidget pcWD;
    public:
        uProdConsSampler( uWatcher &w,          // watcher where sampler resides
                        const int pollFreq,     // polling frequency
                        const int displayFreq,  // display frequency
                        const int &locn,        // location to be sampled
                        const char *title,      // widget title
                        int bufSize ) :         // buffer size
                        uBaseSampler( w, pollFreq, displayFreq ), locn( locn ), pcWD( title, bufSize ) {
                w.add( *this );                 // add sampler to watcher's event list
        } // uProdConsSampler::uProdConsSampler

        ~uProdConsSampler() {
                if ( !paused ) {                // check if already removed from watcher's event list
                        w.remove( *this );      // remove sampler from watcher's event list
                } // if
        } // uProdConsSampler::~uProdConsSampler

        void poll() {
                curr = locn;                    // get current value
        } // uProdConsSampler::poll

        void display() {
                pcWD.setValue( curr );          // display current value
        } // uProdConsSampler::display

        float getProdStd() {
                return pcWD.getProdStd();
        } // uProdConsSampler::getProdStd

        float uProdConsSampler::getProdAvg() {
                return pcWD.getProdAvg();
        } // uProdConsSampler::getProdAvg

        float uProdConsSampler::getConsStd() {
                return pcWD.getConsStd();
        } // uProdConsSampler::getConsStd

        float uProdConsSampler::getConsAvg() {
                return pcWD.getConsAvg();
        } // uProdConsSampler::getConsAvg
}; // uProdConsSampler
```

Figure 7: Buffer Sampler

```
#include "/u/usystem/software/collection/src/uBoundedBuffer.h"

template<class ELEMTYPE> uMonitor VisualBoundedBuffer : public uBoundedBuffer<ELEMTYPE> {
    uWatcher w;
    uProdConsSampler pcsampler;
  public:
    VisualBoundedBuffer( const int size = 10 ) :
            uBoundedBuffer<ELEMTYPE>( size ),
            w(), pcsampler( w, 50000, 100000, count, "Bounded Buffer", size ) {
    } // VisualBoundedBuffer::VisualBoundedBuffer

    uNoMutex float getProdAvg() {
        return pcsampler.getProdAvg();
    } // VisualBoundedBuffer::getProdAvg

    uNoMutex float getProdStd() {
        return pcsampler.getProdStd();
    } // VisualBoundedBuffer::getProdStd

    uNoMutex float getConsAvg() {
        return pcsampler.getConsAvg();
    } // VisualBoundedBuffer::getConsAvg

    uNoMutex float getConsStd() {
        return pcsampler.getConsStd();
    } // VisualBoundedBuffer::getConsStd
}; // VisualBoundedBuffer
```

Figure 8: Visual Bounded Buffer

14.3 Visual Bounded Buffer

Figure 8 shows the enhancements to an existing generic bounded buffer implemented as a monitor. The visual bounded buffer inherits the insert and remove routine from the base monitor, and all the variables needed to manage the buffer. The base monitor is extended with watcher and sampler variables, along with 4 get routines to obtain the values from the display sliders. The visual buffer's constructor initializes the base buffer, widget and sampler, respectively. (Notice that the watcher is initialized in the constructor's initialization list even though it does not require any initialization arguments. This is because the sampler depends on the watcher, and hence, the watcher must be initialized first. By appropriately ordering the variables in the constructor's initialization list, it is possible to control the ordering of initialization.) The monitor variable count is passed to the sampler, pcsampler, as the variable to be sampled. This variable contains the number of items in the bounded buffer.

14.4 Driver

Figure 9 shows the driver program using the visual bounded buffer between a producer and consumer task. The only additions over a non-visual bounded buffer are the two calls to uDelay before and after the calls to insert and remove, respectively. These two delay calls read the current values of the appropriate sliders associated with the buffer display and then delay the producer or consumer task for some appropriate period of time.

```
#include <math.h>

inline double UniformRand( void ) {
    return (double)( rand() % 10000 + 1 ) / (double)10000;
} // UniformRand

inline double ExpRand( double Average ) {
    return - (double)Average * log( UniformRand( ) );
} // ExpRand

inline double HypExpRand( double Average, double Std ) {
    if ( UniformRand() < (double)Std ) {
        return ExpRand( 1.0 ) * ( (double)Average / (2 * (double)Std ) );
    } else {
        return ExpRand( 1.0 ) * ( (double)Average / ( 2 * ( 1 - (double)Std ) ) );
    } // if
} // HypExpRand

uTask producer {
    VisualBoundedBuffer<int> &buf;
    void main() {
        uFloatingPointContext fpcxt;
        for ( ;; ) {
            uDelay( (int)HypExpRand( buf.getProdAvg(), buf.getProdStd() ) ); // spend time producing
            buf.insert( rand() % 100 + 1 );         // insert item into queue
        } // for
    } // producer::main
  public:
    producer( VisualBoundedBuffer<int> &buf ) : buf( buf ) {
    } // producer::producer
}; // producer

uTask consumer {
    VisualBoundedBuffer<int> &buf;
    void main() {
        uFloatingPointContext fpcxt;
        for ( ;; ) {
            int item = buf.remove();                // remove from front of queue
            uDelay( (int)HypExpRand( buf.getConsAvg(), buf.getConsStd() ) ); // spend time consuming
        } // for
    } // consumer::main
  public:
    consumer( VisualBoundedBuffer<int> &buf ) : buf( buf ) {
    } // consumer::consumer
}; // consumer

void uMain::main() {
    srand( getpid() );                              // set random number seed
    VisualBoundedBuffer<int> buf( 100 );            // create a buffer monitor
    {
        consumer cons( buf );                       // create a consumer tasks
        {
            producer prods( buf );                  // create producer tasks
        }
    }
} // uMain::main
```

Figure 9: Animated Bounded Buffer: Driver

15 Implicit Statistical Monitoring

By including file:

```
#include "/u/usystem/software/MVD/src/uKernelSplr.h"
```

after <uC++.h>, implicit monitoring and visualization of the μC++ kernel is presented. The runtime monitor is built using the same facilities detailed in Chapter 10. The only additional capability of the μC++ runtime sampler is that it is allowed to reference variables internal to the runtime system, which are normally hidden.

16 Kernel Display

The μC++ runtime system groups tasks and processors together into entities called *clusters* (see "μC++ Annotated Reference Manual"). When the application begins execution, a list of clusters in displayed (left window in Figure 10). On a uniprocessor there is only one cluster, called the uSystemCluster. The slider at the top of the window dynamically controls the sampling and display frequency of the information in the window. The polling and display frequencies are combined because there is no state that is saved on each poll; the cluster list simply traversed and displayed.

17 Cluster Display

By selecting a cluster from the list of clusters, detailed information about that cluster's execution is presented. For example, cluster uSystemCluster has been selected (indicated in reverse video in the left window) and its detailed information is shown in the right window in Figure 10. The lower portion of the right window shows the tasks currently executing on the cluster (left column) and which tasks are actually executing on processors (right column). The first five tasks are administrative tasks and tasks fred, mary, and john are user tasks. Task uWatcher is currently executing. (The example was run on a uniprocessor). The slider at the top of the window dynamically controls the sampling and display frequency of the information in the window. The polling and display frequencies are combined because there is no state that is saved on each poll; the cluster list simply traversed and displayed.

18 Task Display

By selecting a task from the list of clusters, detailed information about that task's execution is presented. For example, tasks uMain and uWatcher have been selected (indicated in reverse video in the left column) and their detailed information is shown in the right window in Figure 11. The sliders at the top of the window dynamically controls the sampling and display frequency of the information in the window. The polling and display frequencies are separate because there is state, stack high-water mark and task status information, that is saved on each poll. By polling at a high frequency, more precise stack high-water mark and task status information is acquired.

A task display shows the status of the task's stack and execution status; more information may be added. The stack display shows the size of the stack the last time the task performed a context switch (bottom number in the column). This value is displayed in a range from 0 to the task's maximum stack size. As well, there is a high-water mark indicator showing the maximum amount of stack space observed thus far (top number in the column). This information is useful for setting the amount of stack space for a task. If a task is using most of its stack, it could be increased. A task can exceed its stack limit between samples so just because a task's high-water mark has not exceeded the maximum do not assume the stack size is sufficient.

The status display shows the percentage of time a task spends in the running, ready and blocked states, respectively. This information indicates which tasks are performing most of the work and can be used to determine if a group of tasks are achieving expected levels of concurrency. Task uMain is spending

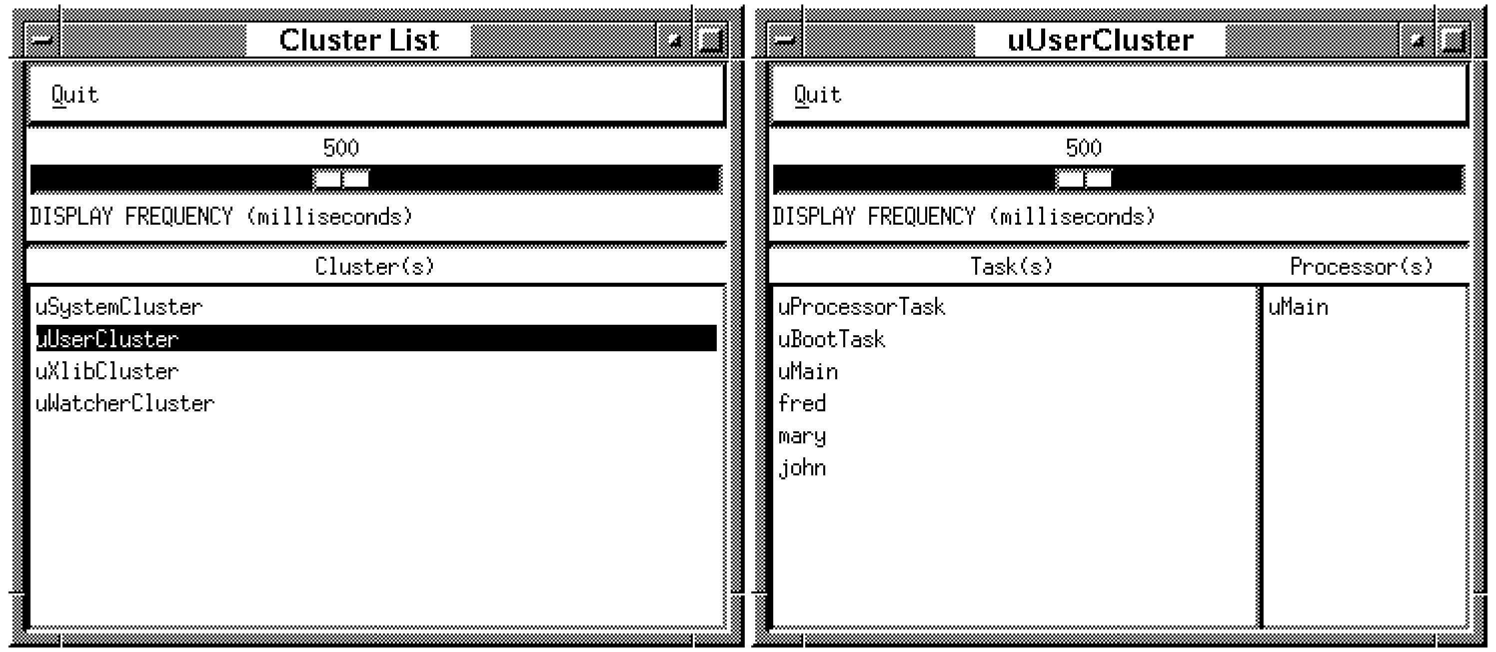

Figure 10: Cluster Displays

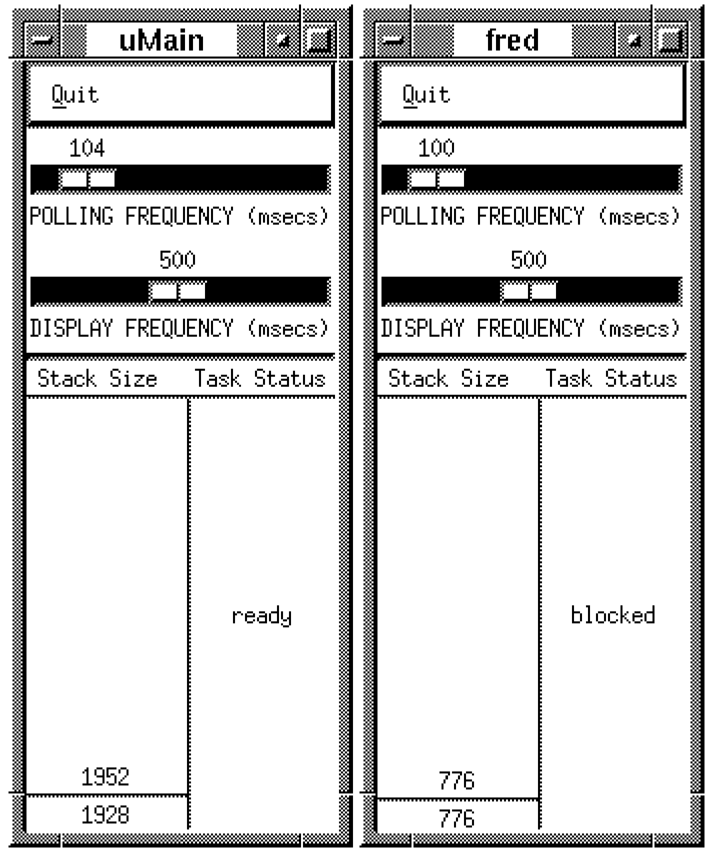

Figure 11: Task Displays

approximately half of its time in the ready and blocked state. Task uWatcher is spending all of its time in the running state. On a uniprocessor, all tasks except uWatcher will have zero running state because the kernel watcher is always running when it invokes the task samplers so a task sampler cannot see its task running. On a multiprocessor, where the kernel watcher is on a separate processor, task displays will show all three states.

19 Contributors

While many people have made numerous suggestions, the following people were instrumental in turning this project from an idea into reality. Peter Buhr got μC++ and X to speak on intimate terms and forced the coding conventions on others; he wrote the first statistical sampling code, the documentation, and did other sundry coding as needed. Thomas Pflum worked on the second version of the statistical sampling code. Martin Karsten wrote the visualization of the dining philosophers. Peter Buhr wrote the visualization of the bounded buffer.

Numeric Sampler

```
// -*- Mode: C++ -*-
//
// Visualization 1.0, Copyright (C) Peter A. Buhr 1992
//
// uNumSplr.h – generic sampler for numeric types
//
// Author : Peter A. Buhr
// Created On : Sun Feb 9 20:34:28 1992
// Last Modified By : Peter A. Buhr
// Last Modified On : Wed Nov 27 19:12:37 1996
// Update Count : 149
//


#ifndef __U_NUMSPLR_H__
#define __U_NUMSPLR_H__


#include "uBaseSplr.h"
#include "uNumGauge.h"


// T must have operations: 0, =, >, <, +, /

template<class T> class uNumSampler : public uBaseSampler {
    const T &locn;
    int count, firstMin, firstMax;
    T curr, min, avg, max, sum;
    uNumGauge<T> numgauge;                                  // displayer object
  public:
    uNumSampler( uWatcher &w, const uDuration pollFreq, const uDuration displayFreq, const T &locn, const char *name,
                const T dialMin, const T dialMax, const T zero, uNumGaugeTL::GaugeType gauge = uNumGaugeTL::digital ) :
                uBaseSampler( w, pollFreq, displayFreq ), locn( locn ), numgauge( name, gauge, dialMin, dialMax ) {
        firstMax = firstMin = 1;
        sum = zero;
        count = 0;
```

```
        // The sampler must be added to the watcher' s event list at the end of the derived
        // sampler because the moment it is added to the event list, the poll and display
        // routine can be invoked by the watcher; therefore, the derived class must be
        // completely initialized. Furthermore, the sampler cannot be added to the watcher' s
        // event list in the base sampler' s constructor because the base sampler' s poll and display
        // routine would be invoked instead of the derived sampler' s routines, which is incorrect.

        watcher.add( *this );                       // add sampler to watcher' s event list
    } // uNumSampler<T>::uNumSampler

    ~uNumSampler() {
        watcher.remove( *this );                    // remove sampler from watcher' s event list
    } // uNumSampler<T>::~uNumSampler

    void poll() {
        curr = locn;                                // get current value

        if ( firstMin ) {                           // maintain minimum value
            min = curr;
            firstMin = 0;
        } else {
            if ( curr < min ) {
                min = curr;
            } // if
        } // if

        count += 1;                                 // calculate running average
        sum += curr;
        avg = sum / count;

        if ( firstMax ) {                           // maintain maximum value
            max = curr;
            firstMax = 0;
        } else {
            if ( curr > max ) {
                max = curr;
            } // if
        } // if
    } // uNumSampler<T>::poll

    void display() {
        numgauge.display( curr, min, avg, max );
    } // uNumSampler<T>::display
}; // uNumSampler<T>


#endif __U_NUMSPLR_H__


// Local Variables: //
// compile-command: "dmake" //
// End: //
```

Visual Bounded Buffer Widget

```
// -*- Mode: C++ -*-
//
// Visualization 1.0, Copyright (C) Peter A. Buhr 1992
```

```
//
// uProdConsWD.h --
//
// Author : Peter A. Buhr
// Created On : Tue Sep 29 13:30:45 1992
// Last Modified By : Peter A. Buhr
// Last Modified On : Thu Jan 23 15:44:25 1997
// Update Count : 42
//


#ifndef __U_PRODCONSWD_H__
#define __U_PRODCONSWD_H__


#include <uXmlib.h>
#include "uXtShellServer.h"


class uProdConsSampler;                                   // forward declaration


// The widget structure for a ProdConsWidget:
//
// * Top-level Shell
// * Main Window Widget
// * Menubar Widget
// * Cascade Widget (quit)
// * Form Widget
// * Label Widget (Producer)
// * Separator Widget
// * Scale Widget (Producer Average)
// * Separator Widget
// * Scale Widget (Producer Std Dev)
// * Separator Widget
// * Label Widget (Delay)
// * Separator Widget
// * Scale Widget (Consumer Std Dev)
// * Separator Widget
// * Scale Widget (Consumer Average)
// * Separator Widget
// * Label Widget (Consumer)


class uProdConsWidget {
    static void QuitCB( Widget widget, uProdConsWidget *This, XmAnyCallbackStruct *call_data );
    static void pollingFreqCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void displayFreqCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void prodStdWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void prodAvgWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void consStdWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void consAvgWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );

    uProdConsSampler &prodconsSampler;
    Widget shell, pollingFreqWD, displayFreqWD, bufScaleWD;
    const int NumDecPts;
    float prodStd, prodAvg, consStd, consAvg;
  public:
```

```
    uProdConsWidget( uProdConsSampler &prodconsSampler, const char *title, int bufSize );
    virtual ~uProdConsWidget();

    float getProdStd() {
        return prodStd;
    } // uProdConsWidget::getProdStd

    float getProdAvg() {
        return prodAvg;
    } // uProdConsWidget::getProdAvg

    float getConsStd() {
        return consStd;
    } // uProdConsWidget::getConsStd

    float getConsAvg() {
        return consAvg;
    } // uProdConsWidget::getConsAvg

    void setValue( int val );
}; // uProdConsWidget


#endif __U_PRODCONSWD_H__


// Local Variables: //
// compile-command: "dmake" //
// End: //
```

References

[BKS96] Peter A. Buhr, Martin Karsten, and Jun Shih. KDB Reference Manual, Version 1.1. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, December 1996. Available via ftp from plg.uwaterloo.ca in pub/MVD/KDB.ps.gz.

[BS96] Peter A. Buhr and Richard A. Stroobosscher. μC++ Annotated Reference Manual, Version 4.7. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, July 1996. Available via ftp from plg.uwaterloo.ca in pub/uSystem/uC++.ps.gz.

[GKM82] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a Call Graph Execution Profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982. Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, June 23–25, 1982, Boston, Massachusetts, U.S.A.

[TB96] David Taylor and Peter A. Buhr. POET with μC++. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, December 1996. Available via ftp from plg.uwaterloo.ca in pub/MVD/Poet.ps.gz.

[Tuf83] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 1983.

Index