POET with μC++

Reference Manual

University of Waterloo

David Taylor and Peter A. Buhr ©*1996

July 23, 2006

Contents

1 Introduction

This document describes the use of the Partial-Order Event Tracer (POET) with μC++. POET is a general tool for collecting and displaying event data from concurrent and distributed applications, and is used to understand the behaviour of these programs in order to improve performance or detect errors. This document does not attempt to describe all the features of POET; only a subset is presented for use with small to medium size μC++ programs. The manual "Partial-Order Event Tracer—User Documentation" describes all the features of POET in a manner independent of a particular application/programming environment.

This document assumes a basic understanding of μC++ ; see "μC++ Annotated Reference Manual" [Buh06] for details.

2 Before Starting POET

In order to use POET with a μC++ program, *all* program components must be compiled with the compilation flag -trace on the u++ command line. Currently, it is impossible to trace individually compiled parts of a μC++ program, i.e., all translation-units are traced or nothing. The -trace flag causes events to be generated for several fundamental μC++ operations (see Section 8, p. 6). These events are implicitly generated from the user's perspective because no code is written to cause their generation. If a program compiled *without* -trace is run under POET, the event display simply remains blank.

3 Accessing POET

To access POET, the path:

```
/u/poet/bin
```

must be added to the PATH environment variable. This variable is usually initialized in the .cshrc file in a user's home directory. Usually, there is a line in .cshrc that looks like:

```
setenv PATH `/bin/showpath $HOME/bin standard`
```

This line can be augmented to:

```
setenv PATH `/bin/showpath /u/poet/bin $HOME/bin standard`
```

A symbolic link or alias to /u/poet/bin/poet is insufficient because the bin directory contains several other executables needed to run POET.

As well, a POET resource file must be copied to your home directory:

```
% cp /u/poet/Poet ~/Poet
```

The name of the file must be Poet. The resource file contains commands that control the behaviour of POET. Changing this file customizes POET, e.g., changing the line:

```
Poet.quitConfirm:     True
```

to:

```
Poet.quitConfirm:     False
```

turns off prompting with a quit dialogue when POET is terminated. For colour or grey-scale displays, adjusting the colours or grey-scales for various output is also a common change. However, caution is required, particularly when changing colours, because some changes can render POET unusable.

If you run POET without a resource file, the following error message appears:

```
******No target-system directories have been specified.
******Impossible to continue.
```

4 User Interface

The user interface for POET is based on the *X Window System*, the *X Toolkit Intrinsics* and the *Motif* widget set. All windows in the interface can be resized from the window manager's border and the size of components in a window adapt automatically. At times, window buttons or menus may appear “greyed out”, which means that the button is *inactive* (insensitive), so pressing it does nothing (see Figure 2(b), e.g., Start CHIT is inactive). Inactive buttons are meaningless in that particular context or mode; a button becomes active (sensitive) again when an appropriate mode change occurs.

5 Starting POET

POET operates as a server at which an application registers as a client (see Figure 1). Both server and client can be started together using the command:

% poet –r *application-name*

If the application program requires arguments, enclose the program name and arguments within quotes:

% poet –r "*application-name [application-argument-list]*"

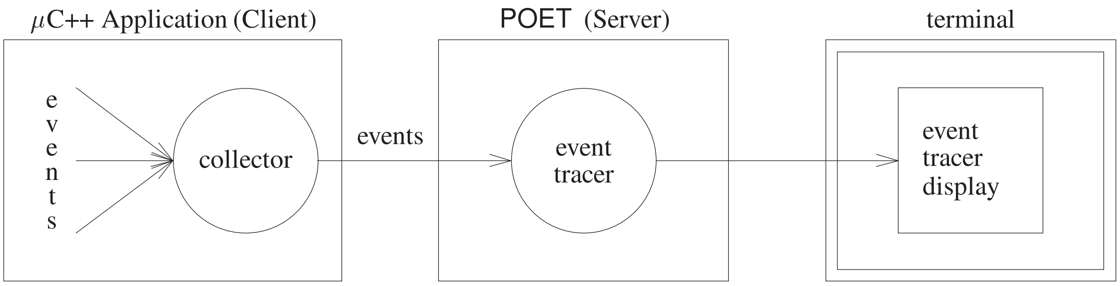


Figure 1: Event Tracer Structure

POET’s main display window appears at startup (see Figure 2(a)). The copyright notice is replaced by an event trace as soon as events are received from the application (client). Connecting the client (application) to an already running POET server is discussed in Section 7.

6 Terminating POET

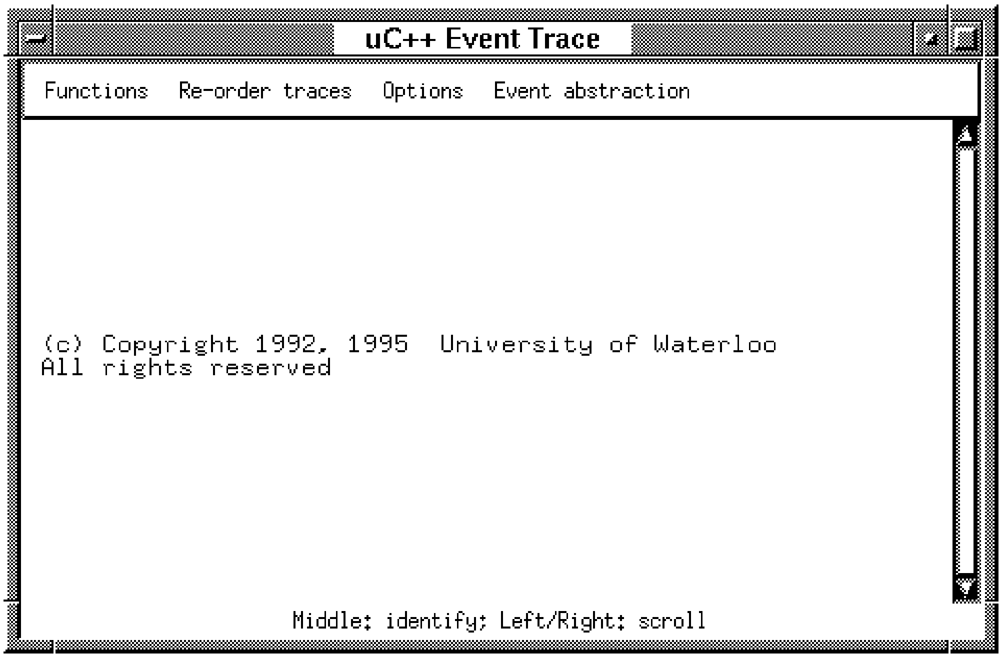
To terminate POET, move the mouse cursor over the Functions button in the menu at the top of the main window, and press the left mouse button, which pulls down the Functions menu (see Figure 2(b)). While continuing to hold down the left mouse button, slide the mouse cursor down the menu until it is over the Quit button, and release the mouse button. Notice as the cursor is moved over a menu button, the border changes to indicate it is ready for selection. Alternatives to selecting the Quit button are typing ctrl–C anywhere in the main window or closing the main window using the window manager. A dialog box pops up requesting confirmation of the termination (unless turned off in the resources file, see Section 3). If execution of the μC++ program is not complete, a confirmation dialog is always displayed with a message indicating the application event-data is incomplete.

7 Reusing POET

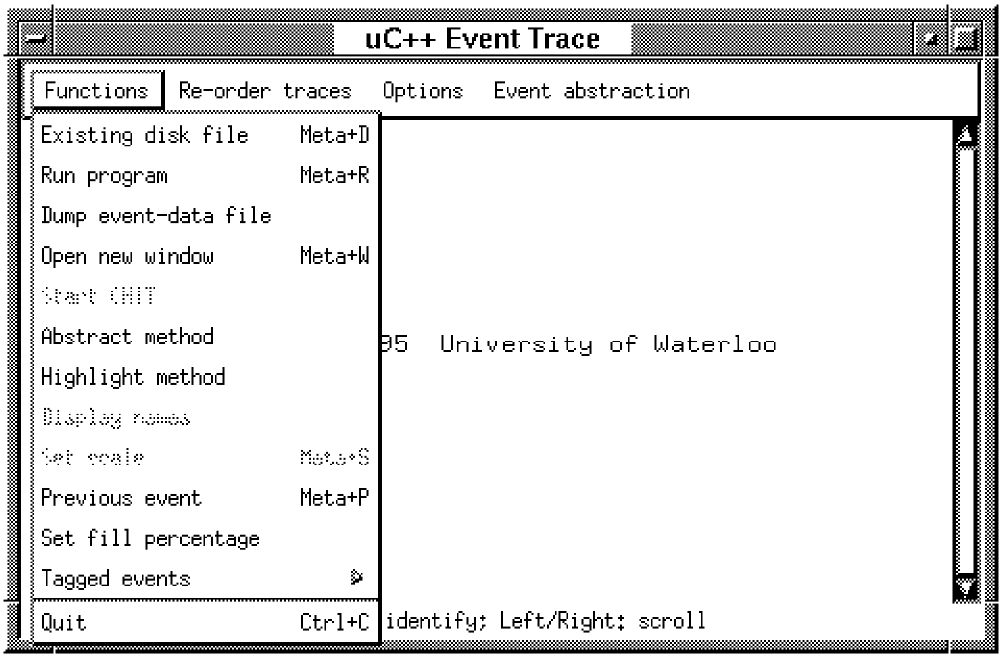
In certain situations, it is desirable to start POET without specifying an application, and subsequently start the application. For example, after an application finishes execution, POET is still running and can proceed with a new event trace for the same or a new application. In fact, because the cost of starting POET is fairly large, it is efficient to use the same instance for multiple tracing sessions. (Currently, POET does not provide a mechanism to clear the display between event traces, but scrolling the previous trace off the window is sufficient.)

If POET is not running, start POET using the command:

% poet



(a) Startup



(b) Termination

Figure 2: POET main window

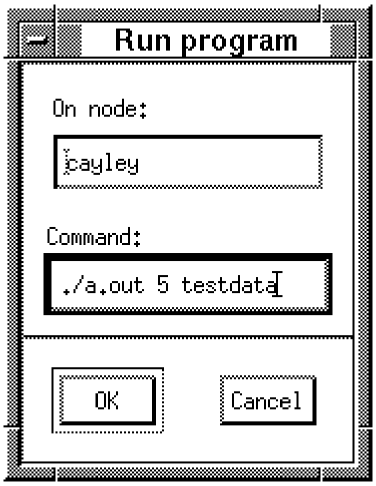


Figure 3: Run Program Dialog

From POET's main window, pull down the Functions menu at the top of the main window, and select Run program (see Figure 2(b)), which pops up a dialog box containing a field in which an application command may be entered (see Figure 3). Enter the program name in the Command field of the dialog box followed by any arguments, e.g., ./a.out 5 testdata. (Leave the On node field unchanged.) Clicking the OK button or typing <Return> in the command field starts the program.

8 μC++ Tracing Model

To understand the event display in the main window, it is necessary to understand the μC++ tracing model. Everything that happens in a μC++ program is done by the *threads* of tasks; the current tracing model focuses on the activities of the threads within a program, and how these threads interact with certain kinds of objects. Currently, the following objects are traced in μC++: coroutines, semaphores, monitors, coroutine-monitors, and tasks. Only certain operations on these objects generate events: for coroutines, all interactions with the coroutine body are traced; for objects with mutual exclusion, all interactions with the members that acquire mutual exclusion (called *mutex members*) are traced. The following calls do not generate events: to free routines (i.e., routines not members of objects), to members of a coroutine that do not resume the coroutine, to non-mutex members. Generating events for all calls provides too much information in an object's trace, much of it unrelated to coroutine or concurrent execution.

Figure 4 shows the trace produced by POET for the execution of a *sequential* μC++ program using coroutines. First, the figure is monochrome, but as mentioned, POET uses colour if available to convey information more readily. Second, each traceable object has its own trace-line (horizontal line) in the event display. In the simple example, there are four trace-lines: uBootTask, uMain and two instances of a fibonacci coroutine. In this manual, the trace line for uBootTask can be ignored, as it is used solely to execute the global constructors and destructors. Third, a sequential program has only one thread, in this case starting in object uMain. (As mentioned, the thread for uBaseTask can be ignored.) This thread is seen moving among the traced objects. Finally, time flows from left to right, although the diagram does not reflect real time (see Section 9). A vertical line reflects interactions between traced objects. In this case, the thread of uMain enters the first fibonacci coroutine, which returns to uMain. The thread then enters the second fibonacci, which returns to uMain. This pattern is repeated multiple times beyond the right edge of the window (Section 14.2, p. 19 discusses scrolling the display).

Figure 5 shows the trace produced by POET for the execution of a *concurrent* μC++ program using

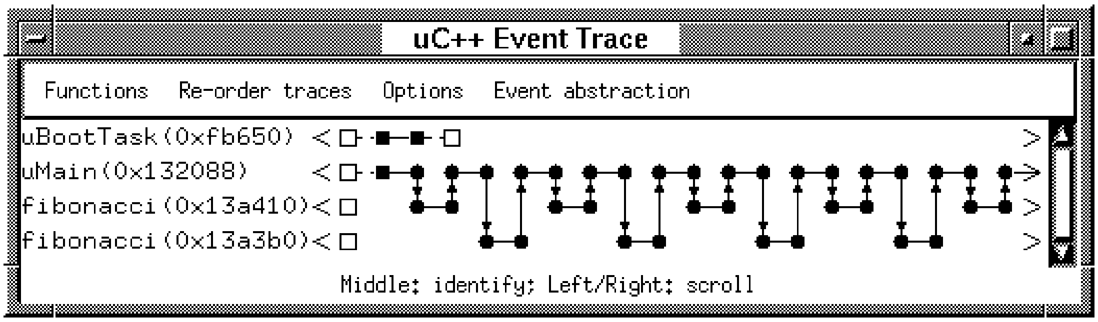


Figure 4: Sequential μC++ Program using Coroutines

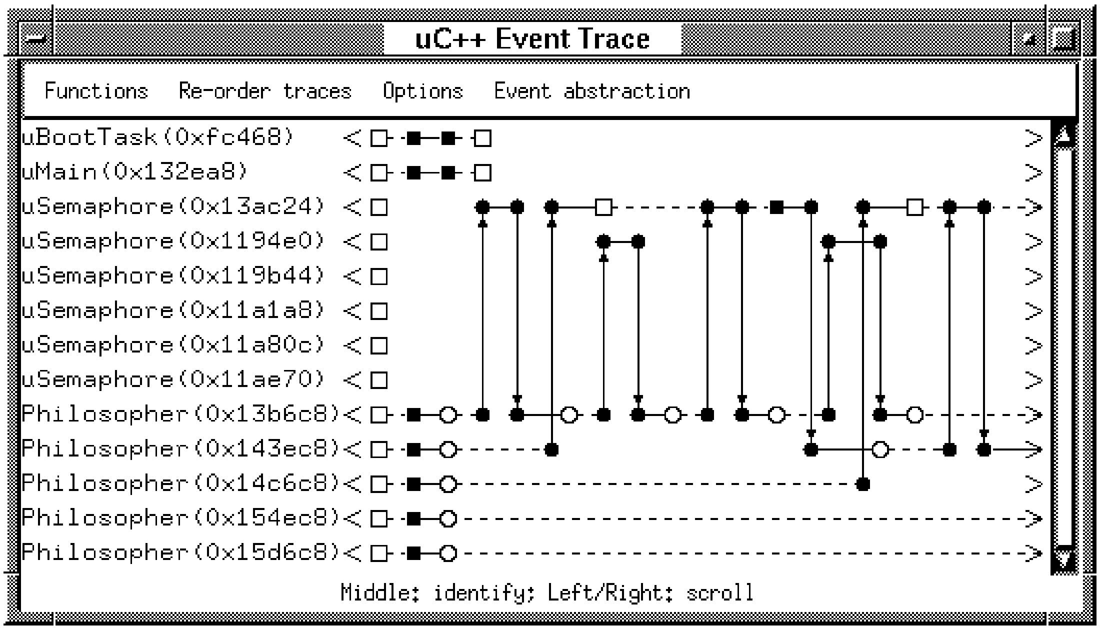


Figure 5: Concurrent μC++ Program using Semaphores and Tasks

semaphores and tasks. This example shows six tasks, uMain and five Philosophers, and six semaphores. It also shows many different kinds of lines and symbols not shown in the previous example. Each of these is discussed in detail in following sections. The main point is the additional thread lines from the Philosopher tasks moving among the traced objects. The complexity introduced by these additional threads is obvious from the complexity of the display. POET helps understand this complexity by allowing easy manipulation of the trace information.

9 Event Ordering

Events are places along a trace line based on relative, not absolute time. That is, events to the left of a particular event along a trace line occur before it and events to the right occur after it. In general, the events are evenly spaced along a trace line but the distance between them does *not* reflect the amount of time between the events. Similarly, events on different trace lines have no time relationship. For example, in Figure 5, task uMain appears to terminate before all other traced objects, when in fact it is normally the last object to terminate. By imagining the trace lines as being elastic and the events as beads along the elastic lines, it is possible to stretch the trace line for uMain so that it is always longer than any other trace line.

Partial orderings among trace lines occurs because of synchronization events (● event and a vertical arrow between the trace lines). At a synchronization point, all events to the left of that point along *both* trace lines occurred before the synchronization. Similarly, all events to the right of a synchronization point along *both* trace lines must occur after the synchronization.

10 Producer-Consumer Example

The following programs illustrate tracing of coroutines, semaphores, monitors, coroutine-monitors and tasks, respectively. The examples, and in particular their explanations, are intended to be read in the order in which they are presented. All the example programs are solutions to a simple producer-consumer problem. In all the examples, a single datum is generated by the producer and given to the consumer.

11 Coroutine Trace

The first example program in Figure 6(a) illustrates interactions of coroutines, which is visualized in Figure 6(b). A coroutine has an *execution-state* but not a thread. When a thread interacts with a coroutine, the thread may transfer to and from a coroutine's execution-state, called a *context switch*, which is visualized for coroutines. *One exception is when a coroutine context switches to itself, which does not generate an event so it is not visualized.*

A context switch from a task's or coroutine's execution-state to another coroutine's execution-state occurs when a coroutine's member-routine performs a resume and the thread transfers to the coroutine's last suspension point (often in the coroutine's main member). The thread then continues executing from that point using the coroutine's execution-state. Another context switch occurs when the thread executing the coroutine performs a suspend; control returns back to the point of the last resumption. Each context switch between execution-states is visualized by an arrow between trace-lines. For example, an arrow from an object to the trace-line of a coroutine indicates that the coroutine has been restarted and a context switch has occurred.

The four trace lines of Figure 6(a) visualize the coroutine producer-consumer program. Notice that the trace-lines are named according to object types rather than according to variables, because an object's type name is the default name assigned to an object in μC++; hence, multiple instances of the same traceable type have the same trace-line name. μC++ allows individual objects to be named using members setName (see Section 12.3, p. 14 for an example of naming objects). To aid in debugging, the address of each object is printed after it. These addresses can be used with a debugger to examine the individual instances.

In uMain::main, after the coroutines are created, the start member of coroutine p is called, which resumes the coroutine (indicated by an arrow from the uMain trace-line to the prod trace-line). p's main member calls the member delivery in coroutine c, which resumes coroutine c (indicated by an arrow from the prod trace-line to the cons trace-line).

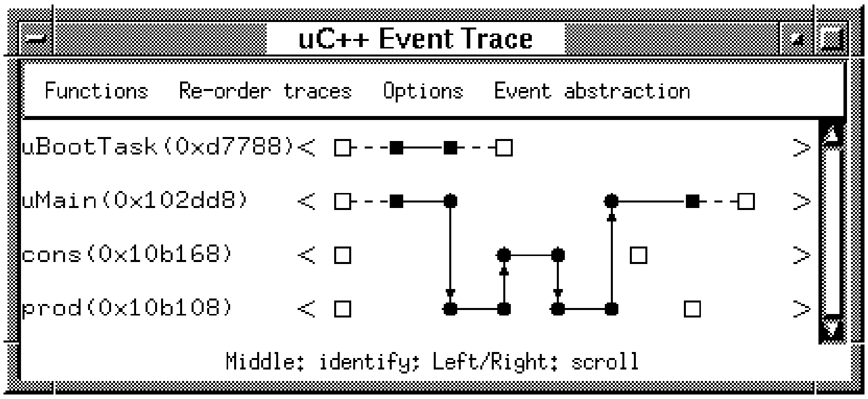
When c's main member ends, it implicitly suspends back to the point of initial resumption (indicated by an arrow from the cons trace-line to the prod trace-line), and coroutine c terminates (indicated by the □). When p's main member ends, it implicitly suspends back to the point of initial resumption (indicated by an arrow from the prod trace-line to the uMain trace-line), and coroutine p terminates (indicated by the □). Routine uMain::main ends, which terminates task uMain (indicated ultimately by the □).

12 Concurrent Trace

In the remaining programs, the producer and consumer are changed from coroutines to tasks (except in the coroutine-monitor program), and a bounded buffer is used between the producer and consumer to provide some asynchrony (except in the coroutine-monitor program). The producer and consumer tasks are shown only once, in Figure 7, p. 10; only the buffer changes in the following concurrent programs.

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

_Coroutine cons {
    int elem;

    void main() { /* consume */ }
  public:
    void delivery( int e ) {
        elem = e;
        resume();                       // restart cons::main
    }
};

_Coroutine prod {
    cons &c;

    void main() { c.delivery( 5 ); }
  public:
    prod( cons &c ) : c( c ) {}
    void start() { resume(); }          // restart prod::main
};

void uMain::main() {
    cons c;                             // create consumer
    prod p( c );                        // create producer
    p.start();
}
```

(a) Program



(b) Trace

Figure 6: Coroutine

```
_Task prod {
    buffer &buf;                    // reference to shared buffer
    void main() {
        buf.query();                // check status of buffer
        buf.insert( 3 );            // insert data
    }
  public:
    prod( buffer &b ) : buf( b ) {}
};

_Task cons {
    buffer &buf;                    // reference to shared buffer
    void main() {
        buf.remove();               // remove data
    }
  public:
    cons( buffer &b ) : buf( b ) {}
};

void uMain::main() {
    buffer b;                       // create bounded buffer task
    cons c( b );                    // create consumer task
    prod p( b );                    // create producer task
}
```

Figure 7: Producer-Consumer Tasks

The following discussion applies to any concurrent trace. As mentioned, a thread's execution moves strictly from left to right along the trace graph, called a *thread-line*. Along any particular section of a trace-line, the form of the thread-line indicates the status of a thread, if any, within that traceable object. When a traceable object contains a ready thread, its trace line is solid. When a traceable object contains only threads that are blocked, its trace line is dashed. A trace line is blank if there is no thread currently in that object. In Figure 5, p. 7, notice the Philosopher threads move through multiple transitions between ready and blocked (solid and dashed line). uSemaphore objects do not have threads, so there are sections of trace-line that are blank, i.e., no thread in the object.

In general, events that cause a thread to be blocked are displayed with an empty square □, and events that cause a thread to be made ready are displayed with a filled square ■. Special cases exist for the creation and destruction of traceable objects; both creation and destruction events are indicated with an empty square, which is consistent for a task because its thread is blocked at these times. Also, the initial starting and final stopping of a task's thread are indicated with a filled square, stating that the thread is ready to execute its main member or its destructor member, respectively. In Figure 5, p. 7, notice a solid thread-line starts with a filled square (thread ready) and a dashed line starts with an empty square (blocked thread).

A thread moves from one object to another by invoking a member of the other object; for *mutex objects*, i.e., objects with mutex members, a thread may block at the call to a mutex member because another thread is using the object. Whenever a call is made to a mutex member, a petition to enter is indicated with an empty circle ○. The calling thread is blocked (dashed line) until it acquires mutual exclusion to the mutex object. Once a thread acquires mutual exclusion, the mutex member call proceeds, and the thread is seen to transfer from one trace line to another; it transfers back when the mutex member returns. *One exeption is when a mutex member is called by a thread in the same mutex object that has already been acquired; in this case, only a petition to enter is indicated in the trace.* Thread transfers are indicated by an arrow perpendicular to the trace-lines indicating the direction of the thread transfer. An arrow originates and ends with filled circles ●. In Figure 5, p. 7, notice each Philosopher task petitioning for entry to the uSemphore mutex objects (actually the P and V mutex members of the semaphore).

In addition, detailed information about an event can be obtained by moving the mouse cursor over an event and clicking the middle mouse button (see Section 13, p. 17).

12.1 Semaphore Trace

The example program in Figure 8(a) illustrates interactions of tasks with counting semaphores, which is visualized in Figure 8(b). In this case, tasks c and p make calls to a class b using semaphores to provide mutual exclusion and synchronization. As before, note that traceable objects are labelled with their class types rather than their variable names.

The creation of the tasks, and the starting of their threads, is indicated by the empty and filled squares at the beginning of the trace-lines. As mentioned, the trace-line is dashed between these two events, indicating the thread is blocked, and the trace-line is solid after the filled square, indicating the thread is ready. Class buffer is not a mutex object so it does not have a trace-line; instead, the two semaphores *within* it have trace-lines. When a task calls the P or V members of a semaphore, it petitions for entry into the semaphores because these members are mutex members; the petition is indicated by the empty circle on the cons and prod trace-lines. Following the empty circle, the line is dashed, indicating that the thread is blocked waiting for entry. Notice that prod::main makes a call to a non-mutex member, buffer::query (see Figure 7). Because query is a non-mutex member, no event appears in the trace for it.

Task c calls the P member of semaphore full, semaphore 0x10accc, (indicated by an arrow from the cons trace-line to the uSemaphore trace-line), executes in the P member (solid trace-line), and blocks because the semaphore count is 0 (indicated by the □). Task p calls the P member of semaphore empty, semaphore 0x10b330, (indicated by an arrow from the prod trace-line to the uSemaphore trace-line), executes in the P member (solid trace-line), returns (indicated by an arrow from the uSemaphore trace-line to the prod trace-line), and continues execution (solid trace-line). Because semaphore empty is non-zero, task p does not block. Next task p calls the V member of semaphore full (indicated by an arrow from the prod trace-line to the uSemaphore trace-line), executes in the V member (solid trace-line), returns (indicated by an arrow from the uSemaphore trace-line to the prod trace-line), and continues execution (solid trace-line). Because the V member never blocks, calls to it always have this form. Task p then stops and is destroyed (indicated by the ■ and □, respectively).

Task c now continues execution after task p has unlocked the semaphore full on which it was blocked. This continuation is indicated by first restating that the task is blocked (dashed line along the uSemaphore trace-line), then the restart (indicated by the ■) and finally execution continues (solid trace-line). Task c now returns (indicated by an arrow from the uSemaphore trace-line to the cons trace-line), and continues execution (solid trace-line). Next task c calls the V member of semaphore empty (indicated by an arrow from the cons trace-line to the uSemaphore trace-line), executes in the V member (solid-trace-line), returns (indicated by an arrow from the uSemaphore trace-line to the cons trace-line), and continues execution (solid-trace-line). Task c then stops and is destroyed (indicated by the ■ and □, respectively).

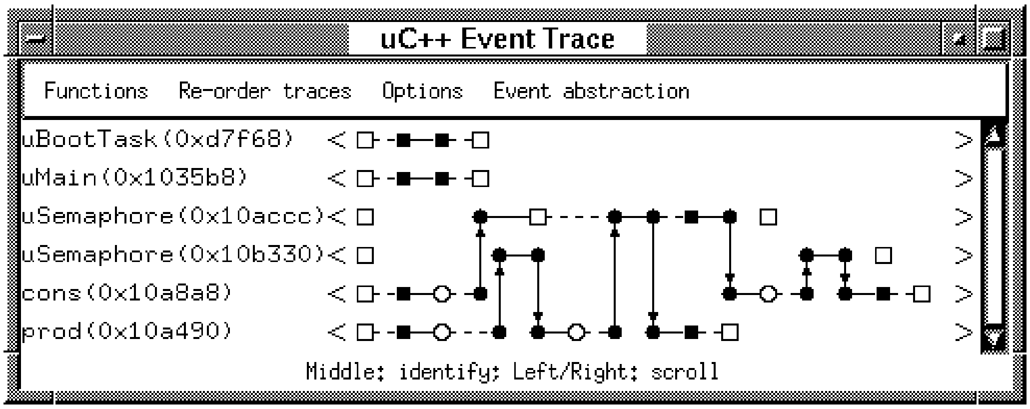
Note that the tasks uBootTask and uMain is created, started, stopped and finally destroyed, as indicated by the top trace-lines in the display, but has no other interactions.

12.2 Monitor Trace

The example program in Figure 9(a), p. 13 illustrates interactions of tasks with a monitor, which is visualized in Figure 9(b), p. 13. In this case, tasks c and p make calls to a monitor b to provide mutual exclusion and synchronization. As before, note that traceable objects are labelled with their class types rather than their variable names.

```
#include <uC++.h>
#include <uSemaphore.h>

class buffer {
    int front, back;                // position of front and back of queue
    int count;                      // number of used elements in the queue
    uSemaphore full, empty;         // synchronize for full and empty buffer
    int elems[5];
  public:
    buffer() : full( 0 ), empty( 5 ) { front = back = count = 0; }
    int query() { return count; }
    void insert( int elem ) {
        empty.P();                  // wait if queue is full
        elems[back] = elem;
        back = ( back + 1 ) % 5;
        count += 1;
        full.V();                   // signal a full queue space
    }
    int remove() {
        int elem;

        full.P();                   // wait if queue is empty
        elem = elems[front];
        front = ( front + 1 ) % 5;
        count -= 1;
        empty.V();                  // signal empty queue space
        return( elem );
    }
};

#include "ProdConsTasks.cc"
```

(a) Program

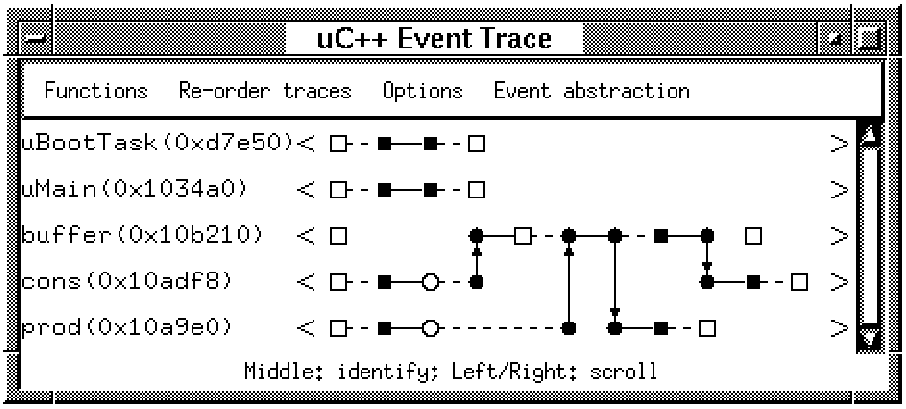


(b) Trace

Figure 8: Semaphore

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

_Monitor buffer {
    int front, back;                    // position of front and back of queue
    int count;                          // number of used elements in the queue
    int elems[5];
  public:
    buffer() { front = back = count = 0; }
    _Nomutex int query() { return count; }
    void insert( int elem );
    int remove();
};

void buffer::insert( int elem ) {
    if ( count == 5 ) _Accept( remove );      // no calls to insert
    elems[back] = elem;
    back = ( back + 1 ) % 5;
    count += 1;
}

int buffer::remove() {
    if ( count == 0 ) _Accept( insert );      // no calls to remove
    int elem = elems[front];
    front = ( front + 1 ) % 5;
    count -= 1;
    return elem;
};

#include "ProdConsTasks.cc"
```

(a) Program



(b) Trace

Figure 9: Monitor

The creation of the tasks, and the starting of their threads, is indicated by the empty and filled squares at the beginning of the trace-lines. However, a monitor does not have a thread so its creation is marked with an empty square and there is a blank trace-line and no starting indicator. Both tasks c and p petition to enter the monitor buffer, but the mutual exclusion property allows only one thread in the monitor at a time.

Task c gains entry first (indicated by an arrow from the cons trace-line to the buffer trace-line), checks for data (indicated by the solid line), and then blocks waiting for a call to insert because there is no data in the buffer (indicated by the □). Task p now gains entry (indicated by an arrow from the prod trace-line to the buffer trace-line), drops off data (indicated by the solid line), leaves the monitor (indicated by an arrow from the buffer trace-line to the prod trace-line), stops execution and is destroyed. Task c now continues in the monitor, which is indicated by first restating that the task is blocked (dashed line along the buffer trace-line), then the restart (indicated by the ■) and finally execution continues (solid trace-line), where task c picks up the data left by task p. Finally, task c leaves the monitor (indicated by an arrow from the buffer trace-line to the cons trace-line), stops execution and is destroyed.

12.3 Coroutine-Monitor Trace

The example program in Figure 10(a) illustrates interactions of tasks with a coroutine-monitor, which is visualized in Figure 10(b). Like the previous coroutine program, this program does not use a buffer. As well, in this example, the producer is a task, while the consumer is the coroutine-monitor. Two producer tasks, p1 and p2, make calls to the same coroutine-monitor c to introduce concurrency. Therefore, the consumer requires mutual exclusion to serialize receiving and consuming of data.

The creation of the tasks, and the starting of their threads, is indicated by the empty and filled squares at the beginning of the trace-lines. However, both tasks make calls to setName to set specific task names, which causes the trace lines to have the task variable name rather than the default class-type name. The name change is indicated in the event trace by the next filled square along the trace lines of both tasks (see the end of Section 13, p. 17 for more information). Because a coroutine-monitor does not have a thread, its creation is marked with an empty square and there is a blank trace-line and no starting indicator. Both producer tasks petition to enter the coroutine-monitor, but the mutual exclusion property only allows one thread in the coroutine-monitor at a time. While the petition to enter the mutex coroutines is indicated in the event trace (empty circle), only the context switch is indicated by an arrow between trace lines. Hence, if the coroutine's mutex member does not resume the coroutine, the trace shows a petition to enter a mutex object but no transfer of thread to the particular mutex coroutine. Currently, there is no obvious way to display both a mutex entry call and a coroutine resumption without having coroutine members as traceable entities.

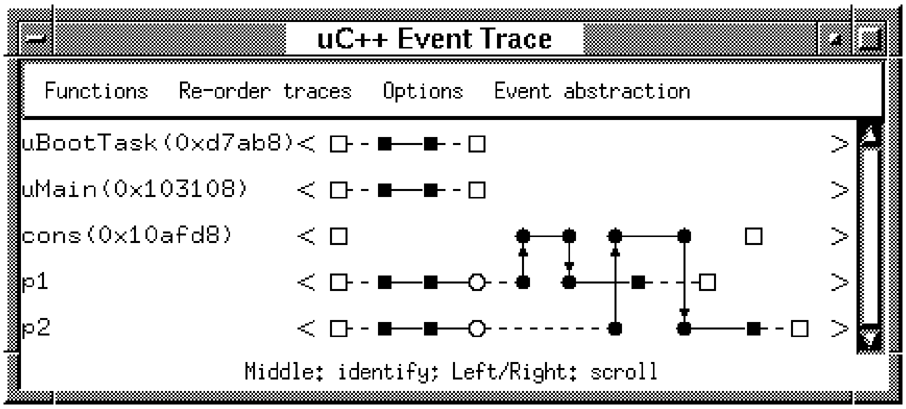
Task p1 petitions first for entry into the mutex member delivery (indicated by the open circle), which resumes coroutine c (indicated by an arrow from the p1 trace-line to the cons trace-line). c's main member consumes the data (indicated by the solid line) and suspends back to the point of resumption (indicated by an arrow from the cons trace-line to the p1 trace-line), and task p1 terminates. Similarly, task p2 petitions for entry into the mutex member delivery (indicated by the open circle), which eventually resumes coroutine c (indicated by an arrow from the p2 trace-line to the cons trace-line). c's main member consumes the data (indicated by the solid line) and suspends back to the point of resumption (indicated by an arrow from the cons trace-line to the p2 trace-line), and task p2 terminates. Routine uMain::main ends, which terminates task uMain and the coroutine-monitor c.

12.4 Task Trace

The example program in Figure 11(a), p. 16 illustrates interactions among tasks, which is visualized in Figure 11(b), p. 16. In this case, tasks c and p make calls to a task b to provide mutual exclusion and synchronization. As before, note that traceable objects are labelled with their class types rather than their variable names.

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

_Mutex _Coroutine cons {
    int elem;

    void main() { /* consume */ suspend(); /* consume */ suspend(); }
  public:
    void delivery( int e ) {
        elem = e;
        resume();                      // restart cons::main
    }
};

_Task prod {
    cons &c;
    const char *name;

    void main() {
        setName( name );
        c.delivery( 5 );
    }
  public:
    prod( cons &c, const char *name ) : c( c ), name( name ) {}
};

void uMain::main() {
    cons c;                            // create consumer
    prod p1( c, "p1" ), p2( c, "p2" ); // create producers
}
```

(a) Program

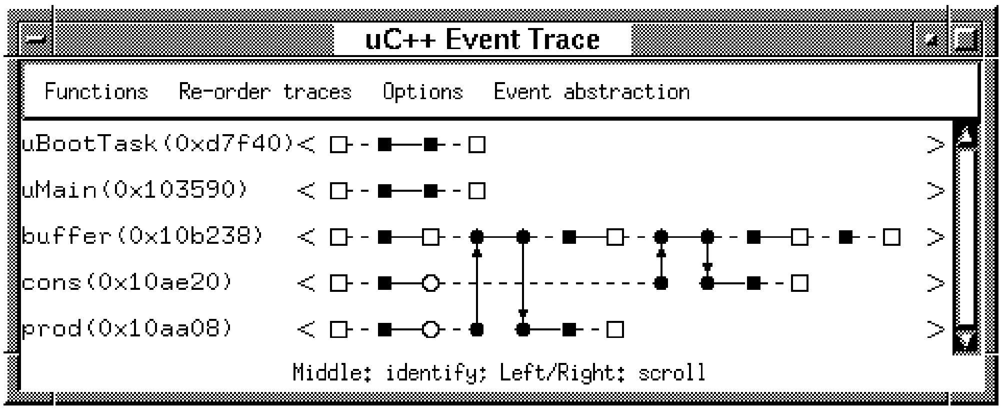


(b) Trace

Figure 10: Coroutine Monitor

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

_Task buffer {
    int front, back;                    // position of front and back of queue
    int count;                          // number of used elements in the queue
    int elems[5];
  public:
    buffer() { front = back = count = 0; }
    _Nomutex int query() { return count; }
    void insert( int elem ) { elems[back] = elem; }
    int remove() { return elems[front]; }
  protected:
    void main() {
        for ( ;; ) {
            _Accept( ~buffer ) {
                break;
            } else _When( count != 5 ) _Accept( insert ) {
                back = ( back + 1 ) % 5;
                count += 1;
            } else _When( count != 0 ) _Accept( remove ) {
                front = ( front + 1 ) % 5;
                count -= 1;
            } // _Accept
        } // for
    }
};

#include "ProdConsTasks.cc"
```

(a) Program



(b) Trace

Figure 11: Task

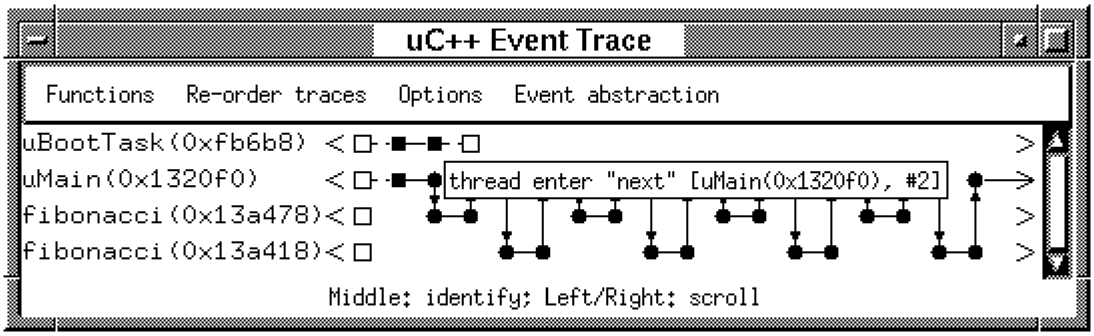


Figure 12: Event Information

The creation of the tasks, and the starting of their threads, is indicated by the empty and filled squares at the beginning of the trace-lines. Since the buffer is a task it has both a trace-line and a thread. Both tasks c and p petition to enter the task buffer, but the mutual exclusion property allows only one thread in the task at a time.

Task b's first action is to accept a call to its members insert and remove. (The accept of ~buffer is discussed later.) The acceptance is indicated by the empty square on the buffer trace-line, and following the acceptance event, the trace-line is dashed since b's thread is blocked waiting for an appropriate call.

Unlike the other examples, task p gains entry first (indicated by an arrow from the prod trace-line to the buffer trace-line) because the **_When** clause prevents a consumer from entering while the buffer is empty, drops off data (indicated by the solid line), leaves the task (indicated by an arrow from the buffer trace-line to the prod trace-line), stops execution and is destroyed. Task c now gains entry (indicated by an arrow from the cons trace-line to the buffer trace-line), checks for data (indicated by the solid line), and picks up the data left by task p. Finally, task c leaves the task (indicated by an arrow from the buffer trace-line to the cons trace-line), stops execution and is destroyed.

Task b now continues execution, which is indicated by first restating that the task is blocked (dashed line along the buffer trace-line), then the restart (indicated by the ■) and finally execution continues (solid trace-line). Task b now accepts a call to its destructor (made implicitly at the end of the block in which b is declared), and it stops execution and is destroyed.

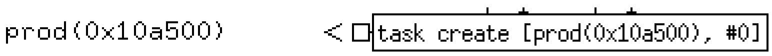
13 Event Information

It is possible to determine more information about an event by moving the mouse cursor to an event and depressing the middle mouse button, which is referred to as *identifying* the event. In fact, the meaning of the mouse buttons is always printed at the bottom of a window (see any previous image of the main window). (The left and right mouse buttons usually "scroll" the display, as discussed in Section 14.2, p. 19.)

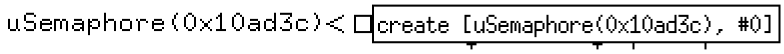
Figure 12 shows the display produced when the third event along the trace line for task uMain is "identified". The box that pops up contains four pieces of information. The first, thread enter, indicates the event type, in this case an event representing a call to a coroutine member that resumes the coroutine. The second, next, is additional information about the event, in this case the name of the coroutine member called. This information is only present for certain kinds of events. The third, uMain(0x132088), gives the name of the entity generating the event, which can also be read at the left edge of the display. The fourth, #2, gives the number of the event, with the first event in each trace line numbered 0. (Note: The box containing the event text is normally positioned to the right of the selected event, unless it extends beyond the right edge of the window, in which case, it may cover the event when it pops up.)

In detail, for each event-display element (□, ■, ○, ●), the following event-specific information is available (all examples are generated from Figure 8(b), p. 12, except the last, which is generated from Figure 10(b), p. 15):

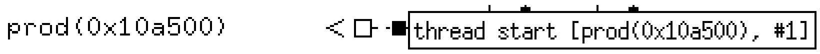
- task create [□]: The initial event of a task object.



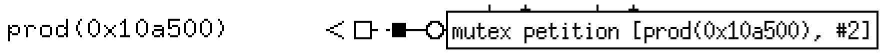
- create [□]: The initial event of a non-task object.



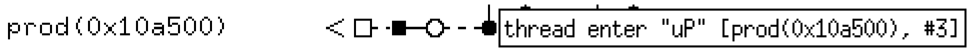
- thread start [■]: The beginning of execution of a thread within a task.



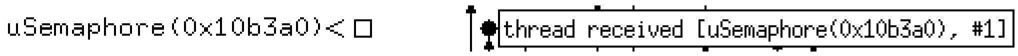
- mutex petition [○]: Attempt to invoke a mutex method.



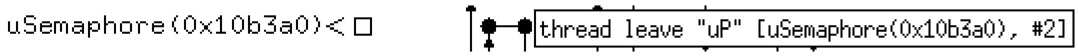
- thread enter [●]: Invocation of a member that context switches to another object or a mutex member. The event states the member called, and the object that the thread is about to leave.



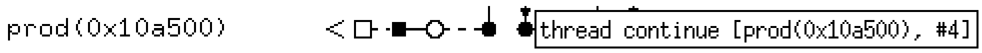
- thread received [●]: Entry of a thread into an object. This event pairs with a thread enter event.



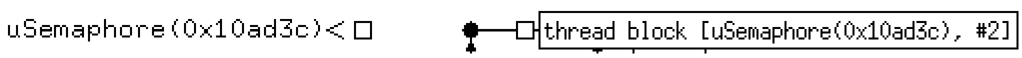
- thread leave [●]: Return from a method back to the call site. The event states the member that is returning, and the object that the thread is about to leave.



- thread continue [●]: Continuation of the thread at the call site. This event pairs with a thread leave event.

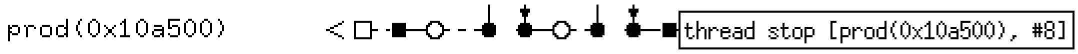


- thread block [□]: Blocking a thread, e.g., on a condition variable.

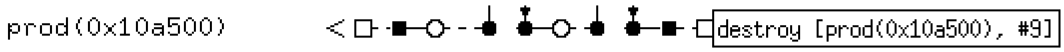


- thread ready [■]: Unblocking a thread, e.g., because of a signal operation. The event states the name of the unblocking task, and the object that blocked the thread.

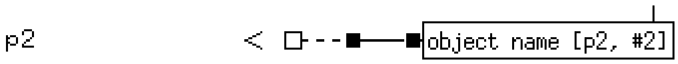
- thread stop [■]: The end of execution of a thread within a task.



- destroy [□]: The final event for any object.



- object name [■]: An object has changed its name (by calling setName). See the example in Section 12.3, p. 14 for details. The associated text is the argument to setName.



14 Manipulating the Event Display

14.1 Precedence Information

On colour or grey-scale terminals, POET also indicates precedence relationship for an identified event (see Section 13, p. 17) and other events on the display. Events that are predecessors of an identified event are coloured in one colour and successors in another colour. In addition, arrowheads are coloured to indicate whether there are any predecessor events to the left of the display or any successor events to the right of the display. If, in Figure 5, p. 7, the fourth event of the first philosopher is identified, the first three events of the remaining philosophers are not predecessors of it, but the remaining displayed events of the second and third philosophers are successors. This means that although the first three events of all philosophers are drawn in the same horizontal position, there is no need for all to occur before the fourth event of the first philosopher. Furthermore, the precise real-time sequencing of the events should not have any effect on program behaviour. Selecting the fourth event of the first philosopher also reveals that all entities other than uMain have successor events beyond the right edge of the display.

14.2 Scrolling Displayed Events

For very short programs, like the previous producer-consumer programs, or programs that crash just after starting, all events appear in the main display window, but in most cases it is necessary to move among the displayed events. First, it should be noted that the window may be resized to display more information, i.e., making it horizontally wider allows more events to be displayed. Somewhat less obvious, making the window vertically narrower may also display more events. Within limits, POET moves the horizontal lines closer together as the window is shrunk vertically and farther apart as the window is expanded vertically. Since it also uses roughly the same spacing horizontally and vertically, squeezing the window vertically may also move events closer together in the horizontal direction.

If the window is not big enough vertically to display all required horizontal lines, a standard scrollbar at the right edge of the window may be used to move the displayed events up and down.

Movement in the horizontal direction is somewhat more complex, because the events do not have fixed relative positions. The mechanism used is to pick an event and move it to the left or right edge of the window. Clicking with the left mouse button moves an event to the left edge of the window; clicking with the right

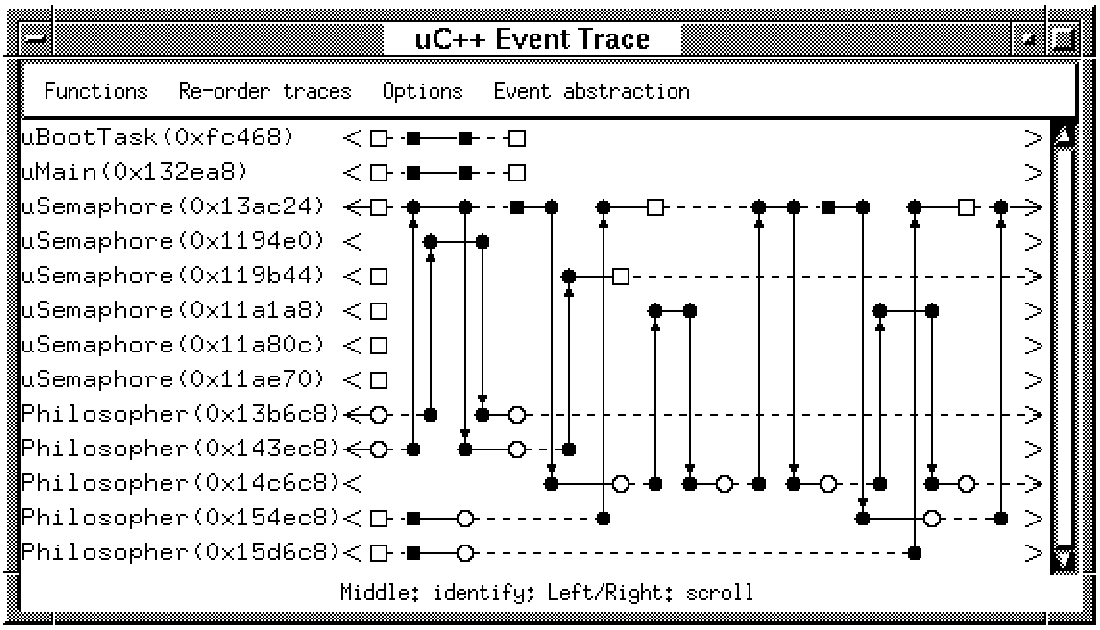


Figure 13: Scrolling Events

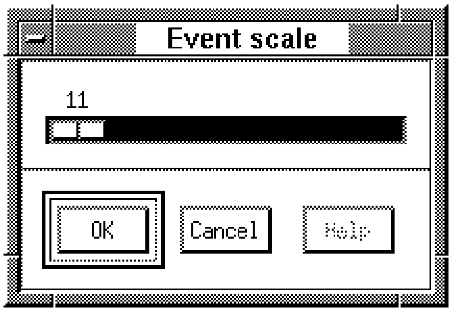


Figure 14: Far Scroll Dialog

mouse button moves an event to the right edge of the window. For example, clicking the left mouse button on the rightmost open square of the first semaphore in Figure 5, p. 7, results in Figure 13. Notice that while the events of the first two semaphores and the first three philosophers have all moved to the left, a uniform left shift of events has not occurred. The case of uMain is quite simple: its events are not connected to any other displayed events, so it has not changed. The other relevant situation is explained by examining the last two philosophers. They each have three events displayed in Figure 5, p. 7. These three events are still displayed in Figure 13, but additional events are added to the display (three for the second-last philosopher, just one for the last philosopher). The reason is that the additional events can be displayed without pushing the existing events out of the display. As mentioned, it is best to imagine movement of events in the horizontal direction like sliding beads along wires, not like moving a large event diagram left and right behind a viewing window. The algorithm used is intended to be helpful in understanding program behaviour. For example, in this case it allows the mutex-petition and thread-enter events for the last two philosophers to be seen simultaneously, which would not have occurred if all events had been moved uniformly to the left.

To move long distances, clicking on the arrowheads that begin and end the visible trace line pops up a dialog box containing a slider (see Figure 14). The slider is used to select an event number anywhere from

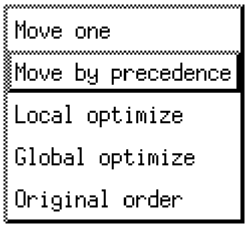


Figure 15: Reorder Traces Menu

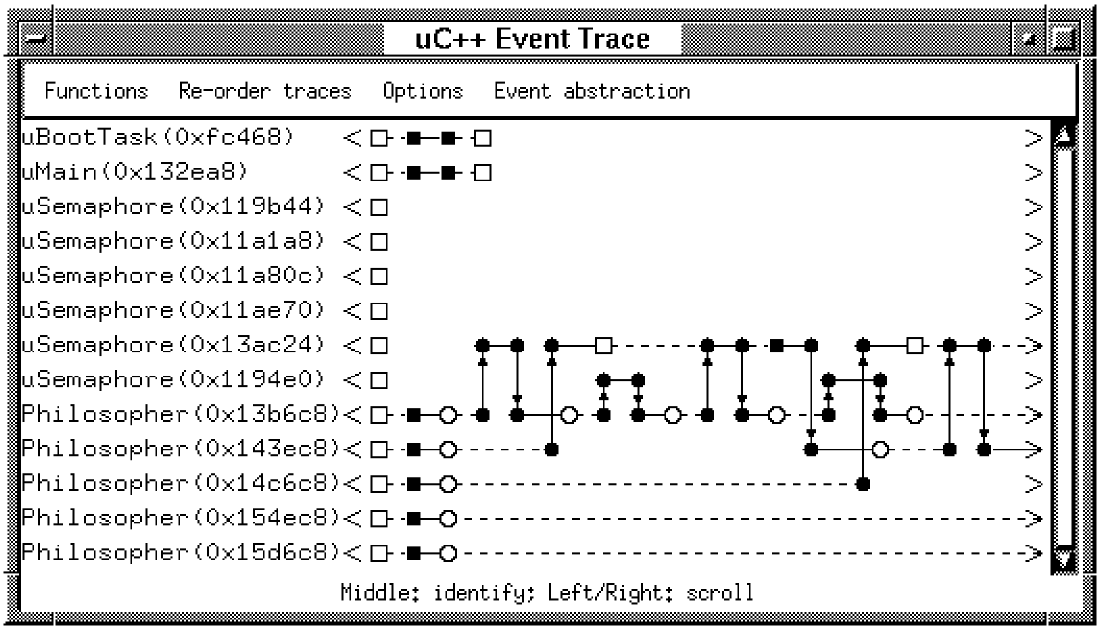


Figure 16: Move by Precedence

the next event (events to the right of the event after the arrowhead) to the last event of the trace line. For the right arrowhead, the range is from event 0 to the previous event (events to the left of the event after the arrowhead). In most cases, exact knowledge of event numbers is not known, nevertheless, this capability does provide an easy means for quickly scanning through the trace or reaching the very end of the trace.

14.3 Changing the order of horizontal lines

The displayed order of the horizontal lines may be changed to suit user preferences. Such reordering may be particularly useful if the window is not large enough to display all horizontal lines simultaneously. All such operations are performed using the Re-order traces menu (see Figure 15) available from the main window.

Move one is used to change the position of a single line. After selecting the menu entry, the prompt message at the bottom of the main window becomes Middle: select trace to be moved. Position the mouse cursor to the desired line, depress the middle mouse button, the prompt message at the bottom of the main window becomes Release at desired new position, move the mouse cursor to the desired new position, and then release the mouse button. The concept is that of picking up a trace line and dragging it to a new location.

Move by precedence is used to select a set of lines and re-organize them so they are closer together. After selecting the menu entry, the prompt message at the bottom of the main window becomes Middle: select event for re-ordering. Position the mouse cursor to the desired line, de-

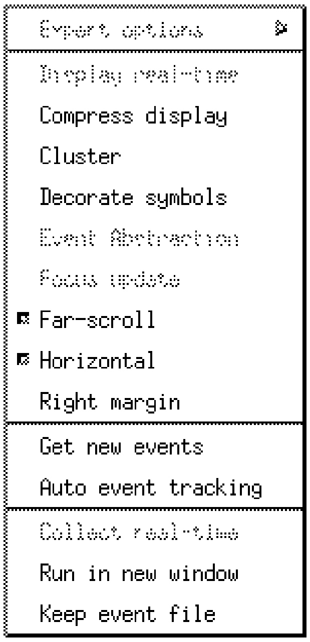


Figure 17: Options Menu

press the middle mouse button, the prompt message at the bottom of the main window becomes Release at desired new position, move the mouse cursor to the desired new position, and then release the mouse button. Like Move one, the concept is that of picking up a trace line and dragging it to a new location. The difference is that, in addition to moving the trace line, all displayed lines connected by precedence to the moved line are also moved closer to the moved line, with other lines shifted to the top and bottom of the display. For example, selecting the first philosopher trace line in Figure 5, p. 7, without moving the mouse cursor before releasing the middle button, moves up the first two semaphores and the first three philosophers so they are juxtaposed (see Figure 16).

The Local optimize and Global optimize options from the Re-order traces menu attempt to minimise the lengths of the vertical connecting lines—on the current display for Local optimize and throughout the execution for Global optimize. For small programs, these options are not particularly useful.

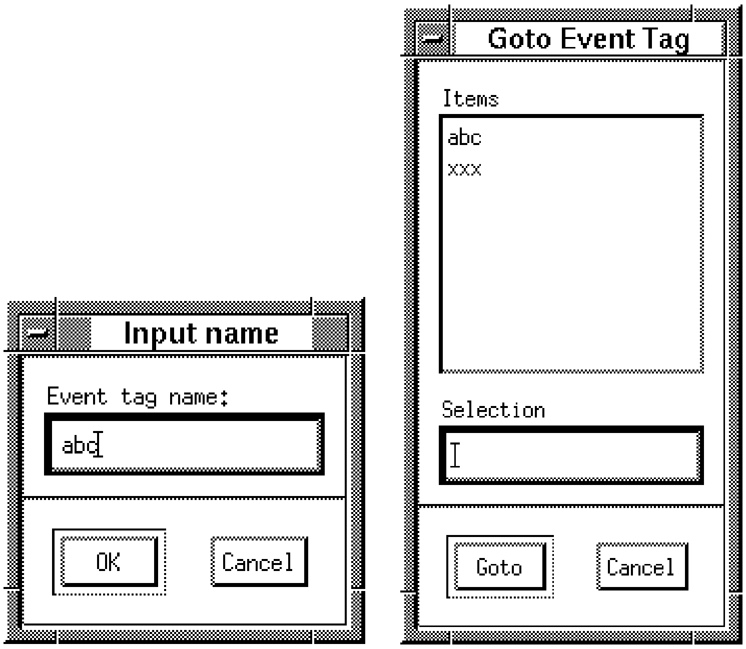
Finally, the Original order option from the Re-order traces menu puts the trace lines back to their original ordering.

15 Other features

15.1 Automatically Updating the Event Display

Normally, once the initial event display is created, the display changes only in response to a scroll command. If the program continues to run, it may be desirable to have new events added to the display automatically. Setting the Get new events option from the Options menu (see Figure 17) causes periodic polling to check for new events followed by a display update when new events arrive. (Note: events are saved into the event-data file continuously in all cases, it is simply the updating of the display that is affected by this option.) The poll interval is set in the resource file. Poet.getEventDelay gives the value in seconds; 5 seconds is the default value. With this option, new events are added until the right edge of the window is reached.

To keep the end of the trace continuously in view, the Auto event tracking option available from the Options menu may be used. In addition to performing the functions described for Get new events, when



(a) Tag Name Dialog (b) Goto Event Tag Dialog

Figure 18: Tagging

events spill over the right edge of the window, the display is scrolled to the right, attempting to fill half the window. Because new events may be arriving as the display is redrawn, there is no guarantee that the window is only half filled after the display is shifted. The percentage of the window to (attempt to) fill may be changed from the default value of 50 using the Set fill percentage entry on the Functions menu.

The above description may appear to indicate that Auto event tracking is the strictly superior choice. The difficulty is that since it can cause major, spontaneous changes in the event display, all user interaction is suppressed when the option is on. All menu buttons are desensitized, event details may not be displayed using the middle mouse button, and manual event-scrolling operations may not be performed. The prompt message reflects this. It changes to Any: cancel auto–tracking mode, indicating that clicking any mouse button in the event-display area turns off auto-tracking mode.

15.2 Tagging events

To work extensively with a particular set of events, it is convenient to be able to move back and forth repeatedly among specific sections of the trace. This capability can be accomplished using the far-scroll facility described above, in conjunction with a record of the event numbers that are in the areas of interest. A convenient method to remember locations of interest in a trace is to attach *tags* to events. Selecting Tagged events on the Functions menu displays a sub-menu with three entries: Goto tagged event, Tag an event, and Delete an event tag. When Tag an event is selected, the prompt message at the bottom of the main window becomes Middle: select event to be tagged. After an event is selected by clicking on it with the middle mouse button, a dialog appears (see Figure 18(a)) in which an arbitrary name for the event may be entered.

Tagging has two effects on an event. First, a halo is displayed around the event to indicate that it is tagged. Second, when the event is selected with the middle mouse button, the tag name is included in the

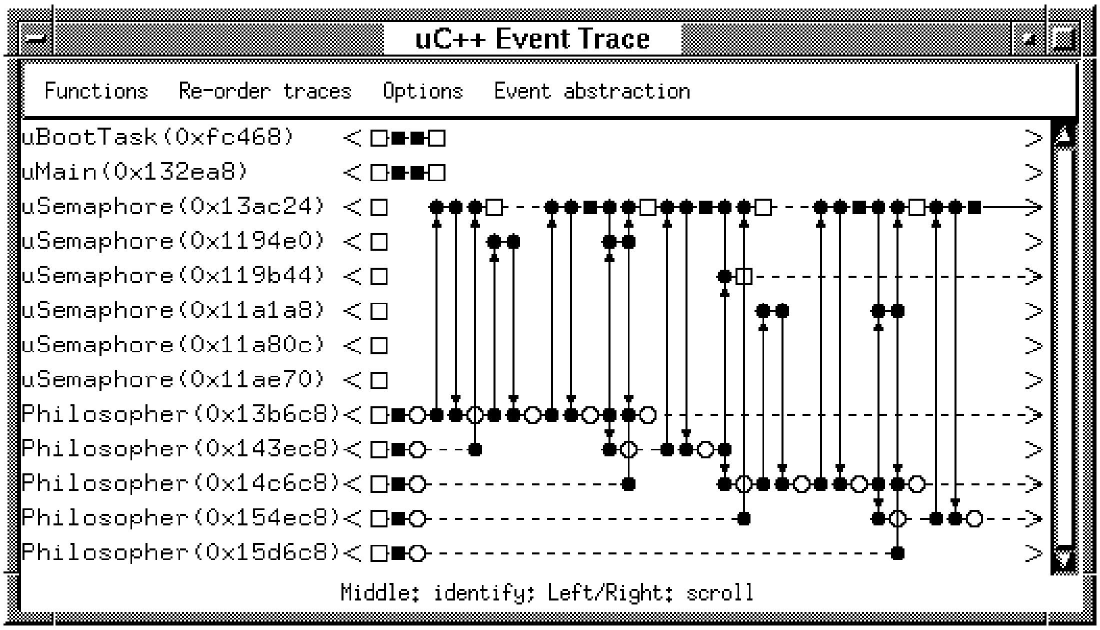


Figure 19: Compressed Events

information displayed.

To scroll directly to a tagged event, select the Goto tagged event from the sub-menu of Tagged events, which pops up a dialog box containing the list of tags (see Figure 18(b)). Select a tag name to move to by clicking on a name in the list, or typing the tag name into the Selection field of the dialog box and clicking the Goto button or entering <Return> in the command field.

If a tag is no longer needed, it can be deleted by selecting Delete an event tag from the sub-menu of Tagged events, which pops up a dialog box containing the list of tags (similar to Figure 18(b)). Select a tag name to delete by clicking on a name in the list, or typing the tag name into the Selection field of the dialog box and clicking the Delete button or entering <Return> in the command field.

15.3 Multiple windows

It may be useful to look at multiple parts of a trace simultaneously. Another event-display window may be created using Open new window from the Functions menu. A new window behaves exactly like the initial window, except that functions related to selecting an event-data file and running a program are disabled. POET only terminates when all event-display windows are closed, so windows may be closed in any order. However, once the initial window is closed, it is impossible to select an event-data file or run a new program.

15.4 Compressing the Display

A summary view of an execution is possible by compressing the trace; compression is accomplished by reducing the distance between events. Turn on the Compress display option from the Options menu, which uses the Set scale value from the Functions menu to control the number of pixels between display events. (Compress display must be turned on before the Set scale option becomes sensitized.) Figure 19 shows a compressed display of Figure 5, p. 7 using a spacing of 10 pixels. POET has a small upper limit on the scale value, so Compress display cannot be used to expand the display. Turning on Compress display also disables the logic that normally prevents the drawing of a vertical line through an unrelated event. Notice how the vertical lines overlap and run through events in Figure 19.

15.5 Saving and re-using event files

In most cases, POET is used to run a program, examine the event trace to help understand a problem, and then the event data is of no further use. Occasionally, it is desirable to save the event data for future use. If Keep event file is set on the Options menu, the event-data file is not deleted when POET terminates. Instead, the events are saved in a file named T*digits*.ef in the current directory, or if the current directory is not writable, in directory /tmp. It is best to rename the file to something more meaningful, but retain the suffix .ef to allow easy selection of the file later.

To run POET using a saved event trace, the Existing disk file entry on the Functions menu is used, which pops up a dialog allowing selection of an event file (see Figure 20). The standard resource file configures the dialog to display files with the .ef suffix. Alternatively, poet -f *filename* can also be used to access a saved file.

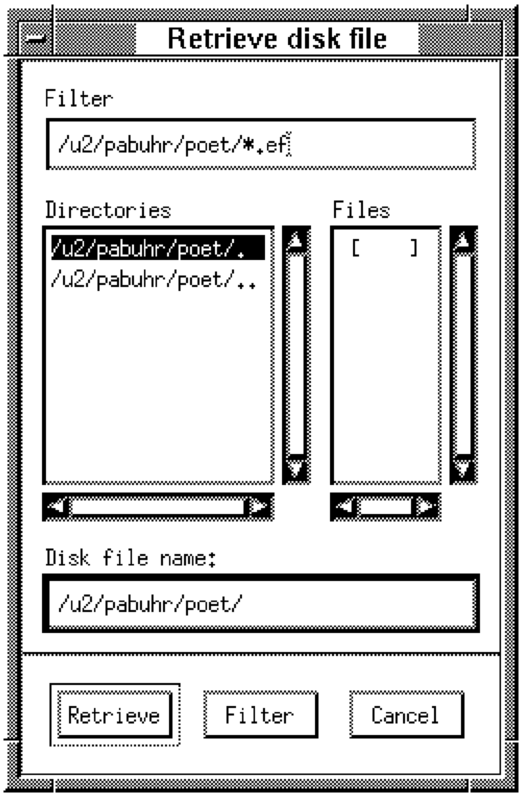


Figure 20: Retrieve Disk File Dialog

16 POET and KDB

To use POET and KDB together, start POET with a command like:

% poet -r "kdb *command*"

or enter kdb *command* in the Command field of the Run program dialog (see Section 7, p. 4). As explained in the KDB documentation [BKSS00], it is possible to run several programs under kdb without shutting down and restarting the KDB server. To connect to an existing kdb server, simply use the command kdb_target *command* from the command line where POET was started. (This may require putting POET into the background so it and KDB run concurrently.)

17 Contributors

While many people have made numerous suggestions, the following people were instrumental in turning this project from an idea into reality. David Taylor wrote most of POET and made many changes as we whined and complained. Scott Zinn and Rob Good kept the event tracer going on plg. Johan Larson wrote the first version of the event generation code for μC++, and Peter Buhr wrote the second version.

References

[BKSS00] Peter A. Buhr, Martin Karsten, Jun Shih, and Oliver Schuster. KDB Reference Manual, Version 1.1. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, July 2000. ftp://plg.uwaterloo.ca/pub/MVD/KDB.ps.gz. 25

[Buh06] Peter A. Buhr. μC++ Annotated Reference Manual, Version 5.3.0. Technical report, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, January 2006. ftp://plg.uwaterloo.ca/pub/uSystem/uC++.ps.gz. 3

Index