Squeak: a Language for Communicating with Mice

Luca Cardelli
Rob Pike

ABSTRACT

Graphical user interfaces are difficult to implement because of the essential concurrency among multiple interaction devices, such as mice, buttons, and keyboards. *Squeak* is a user interface implementation language that exploits this concurrency rather than hiding it, helping the programmer to express interactions using multiple devices. We present the motivation, design and semantics of *squeak*. The language is based on concurrent programming constructs but can be compiled into a conventional sequential language; our implementation generates C code. We discuss how *squeak* programs can be integrated into a graphics system written in a conventional language to implement large but regular user interfaces, and close with a description of the formal semantics.

CR Categories: I3.6 Graphics languages, Interaction techniques D3.1 Formal semantics
CR General Terms: Algorithms, Theory, Languages
Additional Keywords: Concurrency, User Interfaces

Introduction

User interface implementation languages ([Buxton 83], [Thomas 83]) usually address the construction of a user interface from building blocks such as menus, scroll bars and freehand curves. Although it is worthwhile to automate the building of programs from such building blocks, there is an underlying level that these languages do not address: the implementation of the building blocks themselves. Moreover, the procedures that provide menus, graphical potentiometers and other user interface modules tend (in our experience) to be more difficult to write or modify and clumsier in execution than one would expect. The primitives never seem complex in principle, but the programs that implement them are surprisingly intricate.

Providing a suitable graphical display is not especially difficult; what causes problems is the complicated flow of control required to deal with all the possible sequences of user actions with the input devices. One might consider a scrolling menu, for example, as a finite state automaton reading an input token for each event generated by the user: buttons up and down, entering and leaving the scroll bar rectangle, etc. Interaction primitives would probably be

simpler to write and understand if they were implemented as state machines. A translator that converted state machine descriptions into regular programs would make the job even easier.

There are a couple of factors that limit the usefulness of this technique, however. First, the presence of multiple input devices invalidates the notion of a single stream of tokens driving the state machine; for example, the procedure implementing a menu should not worry about characters typed on the keyboard, even those typed while the user is using the menu. Second, the passage of time is often important in user interfaces. Some pairs of events are only meaningful when the individual events occur sufficiently near in time. Consider clicking a mouse button twice: if the clicks are nearly simultaneous, they might be construed as the single event 'double click.'

A more powerful structure is a set of communicating finite state machines, each of which implements the actions associated with some set of user events. If the individual machines execute concurrently, each may be enabled when an event is available for it, so the user interface need never 'lock up' waiting for a specific event. Another concept from concurrent programming, the *timeout*, can be used to encode time-sensitive procedures.

In contrast to approaches based on parsing a single input stream [van den Bos 83], the language we present here, called *squeak*, is an explicitly concurrent language, resembling CSP [Hoare 78] and CCS [Milner 80], and with the passage of time built rigorously into the semantics as in SCCS [Milner 82] and ESTEREL [Berry 84]. Processes in *squeak* communicate by exchanging simple messages on multiple channels. A predefined channel is used for communicating with each device.

The concurrency in a *squeak* program must be expanded out for *squeak* to be of practical value in a conventional programming environment. Our implementation generates an open-coded (as opposed to table driven) state machine, written in C [Kernighan 78], that expresses all possible execution paths of the set of processes in the program. The sequence of input events controls the path taken by the single-stream sequential execution of the program. In practice the relatively simple programs needed to describe user interfaces are well-behaved, although in general the state space of a set of concurrent processes can explode.

Tutorial introduction to *squeak*

The following sections will explain *squeak* in detail; this tutorial introduces and motivates the basic ideas.

Squeak programs are composed of processes executing in parallel. A process, or perhaps a few processes, typically deal with a particular action or external device; the composition of processes then handles the set of actions and device events relevant to the program. Communication between processes is achieved by sending messages on *channels*. There are two classes of channels: primitive and non-primitive. Primitive channels are pre-defined, and provide access to external devices. Non-primitive channels are for ordinary message-based communication. The syntax *c*!*exp* sends the value of the expression *exp* on channel *c*; *c*?*var* reads the message on channel *c* into the variable *var*.

Our implementation of *squeak* defines the primitive channels *DN* and *UP*, which report mouse button transitions; *M*, mouse cursor position (*M!p* sets the cursor position, *M?p* reads it);† *K*, characters typed on the keyboard; *T*, the current absolute time; and *E* and *L*, the mouse entering and leaving certain rectangles. The primitive channels return appropriate values; *M* for example returns a point data structure. *UP* and *DN* return no value; if the mouse had several buttons, they might return the mouse button number, or there could be a separate channel for each button. *E* and *L* return the appropriate rectangle. *Squeak* does not specify how the program announces its interest in rectangles on the display; our implementation provides C-callable functions to push and pop sets of rectangles to be watched. Events come in meaningful order, so that *UP* and *DN* events must alternate, as must *E* and *L* of a given rectangle.

Here is a simple *squeak* program that places typed text on the display at points indicated by the mouse:

```
proc Mouse = DN? . M?p . moveTo!p . UP? . Mouse

proc Kbd(s) = K?c .
        if c==NewLine then
            typed!s . Kbd(emptyString)
        else
            Kbd(append(s, c))
        fi

proc Text(p) =
        < moveTo?p . Text(p)
        :: typed?s . {drawString(s, p)}? . Text(p) >

type = Mouse & Kbd(emptyString) & Text(nullPt)
```

Process structure in the example program

The last line states that the generated C procedure *type* is the result of the parallel execution of three processes. The *Mouse* process waits for the mouse button to be depressed. When it is, the mouse coordinates are sent on channel *moveTo*, where they will be read by the *Text* process. *Mouse* then waits for the mouse button to be released, and restarts. (The precise semantics of these actions are discussed in the following sections.) *Kbd* waits for a character to be typed. If the character is a newline, it sends the complete line on channel *typed* and restarts; otherwise it appends the character to the line. The *append* function is a C routine defined elsewhere; *squeak* treats its invocation as a literal expression. Note that because *Mouse* and *Kbd* are processes, not functions, their recursive invocations do not stack; they are *goto*'s, not subroutine calls.

† Primitive events are special: the sender of *M?* and the receiver of *M!* are always external to the program.

Finally, the *Text* process waits for a message on channel *moveTo* or *typed*, and records the mouse position or draws the string on the display, as appropriate. The code in brace brackets { } is a C expression evaluated at that point in the execution of *Text*. Typical *squeak* programs implement only the flow of control; the actual work at each state of execution is done by such calls to external code.

This simple example is artificial, but illustrates the basic ideas of *squeak*. Most important, a process monitors each input device, and each such process is independent. If a mouse button is held down while typing continues, the text will still be displayed when a newline is typed. This works because of the concurrent execution of the *Mouse* and *Kbd* processes.

Syntax and informal semantics

A *squeak* program is a set of process declarations followed by a main process, which may use the declared processes.

```
prog ::=   decl id = main
decl ::=   ε | proc pid formals = prcs decl
main ::=   prcs rename | main \ id | main & main | ( main )
prcs ::=   pid actuals |
           action . prcs |
           wait [ exp ] prcs || prcs |
           if exp then prcs else prcs fi |
           < prcslist > |
           ( prcs )
action     ::=  id ? | id ! | id ? id | id ! exp
prcslist   ::=  ε | prcs | prcs :: prcslist
formals    ::=  ε | ( idlist )
actuals    ::=  ε | ( explist )
rename     ::=  ε | [ id / id ] rename
idlist ::= id | id , idlist
explist    ::=  exp | exp , explist
exp  ::=   id | num | exp op exp | ( exp )
op   ::=   + | – | * | / | = | == | < | > | <= | >= | !=
```

The simplest process is <> (also called *stop* or *deadlock*), which cannot perform any action.

A process of the form *a*!*exp.p* is ready to output the value of *exp* on the channel *a*, and then execute *p*. The value can be read by the process *a?x.p*, which binds the input value to the identifier *x*, with *x* available in (and local to) the continuation *p*. If no value is passed during a communication, we can simply write *a*! or *a?*. These are all instances of *simple processes*, which consist of an *action* (*a*!*exp* or *a?x*) and a continuation.

The action *a*!*exp* cannot execute until there is a matching *a?x*, and *vice versa.* If more than one input is active on a channel, only one will receive the value; the others remain

suspended until the next input.

A process may wait for input or output simultaneously on several channels: this is a *non-deterministic choice* operation among processes. For example $<a?x.p_1 :: b!y.p_2 :: c?z.p_3>$ is waiting for input on a and c, and for output on b. Communication may happen on any available channel, say a, and in that case p_1 becames the process continuation (the other continuations p_2 and p_3 are discarded). A choice between two processes may also be written $p+q$; this is not part of the syntax, but is a convenient notation when discussing the semantics. Choice is associative, so that $<p :: q :: r>$ can be written $((p + q) + r)$, or $(p + (q + r))$. A choice with a single alternative $<p>$ is equivalent to p. A choice of zero alternatives is the deadlock $<>$, which is the identity in sums, i.e. $<> + p$ is p.

Some actions can have a *timeout* condition: the simple process $wait[3]a?x.p \;||\; q$ will wait for input on channel a for a maximum of three time units. If an a communication happens within that time, p will be executed. If communication is not achieved in time, the process will time out and execute q; $wait[0]a?x.p \;||\; q$ is equivalent to q.

Conditional flow of control is achieved by an **if-then-else-fi** construct. A boolean condition is used to decide between two possible process continuations. If the condition is true, the **then**-part continuation will become the current process, otherwise the **else**-part will.

Processes can be defined recursively:

```
proc p = <c!0 . <>   ::   a!3 . p   ::   b?x . p>
proc q = <c?z . <>   ::   a?y . b!(y+1) . q>
example = p & q
```

The third line executes the processes p and q in parallel. Initially the two processes can exchange a c action, in which case they both stop, or an a action, in which case p goes back to the initial state, while q gets into a state where it can only do a b action (which can now be absorbed by p) and then go back to its initial state. Note that when a process calls another process, it is a process replacement, not a subroutine call; processes never return to the calling process.

Every execution path of a process must encounter an action before it encounters an execution of itself. This rules out pathological cases like $p = p$ and $p = <a?x.q :: p>$.

A process may have parameters, which are available as local variables within that process. For example, consider the following counter process, which may receive an *increment* signal or a *tellContents* signal. It has a local parameter n, which is the current count:

```
proc counter(n) =
      < increment?m . counter(n+m)
      ::  tellContents!n . counter(n) >
```

The process would be created by running $counter(0)$. Although from appearance the $tellContents!n$ message can be emitted at any time, the meaning of communications is such that there must be a matching action $tellContents?v$ to receive the message in some other process

before the sending action may be executed. Similarly, the *increment?m* is only executed when another process emits an *increment*!*exp*. Therefore, most of the time the *counter* process is suspended waiting for a matching message to choose which path of the selection to take.

A complete *squeak* program is a parallel composition of processes, possibly with channel renamings to facilitate the reuse of process definitions. A complete program can perform external communications on predefined channels, or internal communications on user-defined channels. Communications on user-defined channels must all be satisfied internally, or a deadlock will result.

Example 1: Simple menus

Our implementation of *squeak* compiles a program into a single C function that executes the combined state machines of the processes in the program. The passing of arguments and return values is handled by two special primitive channels, *ARG* and *RES*. The action *ARG?x* stores in *x* the program's actual argument list. The variable *x* will in general be a data structure to implement the passing of sets of values to the program. The action *RES*!*exp* returns the expression to the caller of the program. The way these must be implemented, of course, is as a call and return from the function, so *ARG* must be the first communication received by a program, and *RES* the last emitted.

To handle more complicated interactions with C code, our implementation of *squeak* interprets text enclosed in brace brackets { } as literal C expressions (except that *squeak* process variables may be renamed for uniqueness). Such an expression is valid wherever an ordinary expression is valid, or in place of an input channel in an action, in which case the value of the expression is assigned to the variable (if any) in the action. This allows a reasonably clean connection to the outside world, and keeps *squeak* independent of the generated language.

Here is a complete example that uses *ARG* and *RES* to implement simple menus:

```
proc Roam(m, sel) =
        < E?r . {highlight(r)}? . Roam(m, rtosel(r))
        :: L?r . {lowlight(r)}? . Roam(m, –1)
        :: UP? . {erasemenu(m)}? . RES!sel . Menu >
proc Menu = ARG?menu . {drawmenu(menu)}? . Roam(menu, –1)


simpleMenu = Menu      # The generated function is called 'simpleMenu'
```

A menu is an array of labeled rectangles. The external function *rtosel*(*r*) maps a rectangle to its label, *drawmenu* and *erasemenu* create and destroy the menu's display, *highlight* highlights a rectangle and *lowlight* undoes the highlighting. Part of the task of drawing and undrawing the menu is identifying to the event manager the rectangles that tile the menu, one per element. Conceptually, the *Menu* process is always running, but blocked on receipt of an argument *menu* to draw. (In reality, of course, *Menu* is not started until *simpleMenu* is called.) A higher-level process invokes *Menu* when it detects the mouse button depressed for a

significant time. *Menu* then draws the menu and invokes *Roam*, which highlights the appropriate rectangles as the mouse roams across the menu. The return result, generated when the mouse button is released, is the label of the rectangle the mouse is in when the button is released, or –1 if the mouse is outside the menu, indicating no selection. Its final action is to restart the *Menu* process, but this is done only for clarity; *RES* terminates the invocation of *Menu*. Note that the *ARG* and *RES* channels must be handled specially by the compiler so that a program bracketed by *ARG* and *RES* actions behaves like a conventional C function.

Example 2: Double clicking

As an example of a *squeak* program using timeouts, consider the problem of detecting clicks (mouse button down and up again in a short time) and double clicks (two clicks separated by a longer but finite time) without losing any button transitions. Here is a *squeak* process that detects single clicks on a one-button mouse:

```
Click =
    DN? . wait[clickTime] (UP? . click! . Click)  ||  (down! . UP? . up! . Click)
```

When a mouse button is depressed, *Click* receives a *DN* event and waits for a corresponding *UP*. When the *UP* is received, a *click* event is generated and the process restarts. If the *UP* event is not received within *clickTime*, *Click* emits a non-primitive *down* event to indicate to another process that the mouse button is being held down. Then it waits for the corresponding *UP* and re-emits it as an *up* signal.

Here is a process that detects clicks and double clicks:

```
DoubleClick =
    DN? .
        wait[clickTime] UP? .
            wait[doubleClickTime] DN? .
                wait[clickTime] UP? . doubleClick! . DoubleClick
                || click! . down! . UP? . up! . DoubleClick
            || click! . DoubleClick
        || down! . UP? . up! . DoubleClick
```

If *DoubleClick* receives two clicks with the proper timing, it emits a *doubleClick* event; otherwise it emits *click*, *down* and *up* events so another process can receive them.

If clicks and double clicks did not have timeouts, *DoubleClick* could call *Click* to interpret the single clicks. Because two timeouts are involved, though, the processes can get out of step. Consider the following erroneous implementation of *DoubleClick*:

```
DoubleClick =
    click? . wait[doubleClickTime] (click? . doubleClick! . DoubleClick)
                || (click! . DoubleClick)
```

If the timeout occurs, the *click*! action must be emitted to preserve the events, but it may

appear after a *down* event emitted by *Click*. The two independent timeouts on the same stream of events have reordered the events. *DoubleClick* is therefore written as a single process with nested timeouts. If timeouts are not involved (and in practice they rarely are), constructing *squeak* programs hierarchically works well.

Compilation

A *squeak* program is compiled by analyzing all the possible execution sequences of the program, and expanding them into C code. There is no scheduling on user channels: scheduling and communications are 'compiled away,' producing efficient sequential code segments interleaved with random choices and calls to the underlying primitive event manager. This is made practical by two properties of the language. First, there are restrictions on its expressive power, primarily that the syntax only allows a fixed number of processes, and all the channels are statically known. Second, most practical programs focus their activity on the external device channels rather than on inter-process communication. The special nature of the primitive events in *squeak* are essential to its usefulness and practicality.

Primitive events are handled by three C functions that monitor the mouse, buttons, clock and other I/O devices of the system. The event types are button transitions, mouse motion, mouse entering or leaving a rectangle, keyboard characters typed, and clicks of the 60Hz clock. (Our display is a Teletype DMD-5620 terminal running a simple non-preemptive multiprogramming system similar to that described in [Pike 83].) The function *waitevent*(*elist*) suspends the calling process until one of the events in the list is pending. The return value is the name of one of the pending events. The event remains pending until *event*(*e*) is called with an argument naming the desired event. *Event* returns a structure describing the event, including information such as, for example, which rectangle was entered. *Event* will call *waitevent* if no event is pending. *Waitevent* also allows a timeout to be specified for each of the events being awaited. Finally, *testevent*(*e*) tests whether any of the named events are pending. The split structure of the event code simplifies the implementation of processes awaiting multiple events: a C **switch** statement selects, based on the return value of *waitevent*, which event to read and which variable should receive the event's return value. Because the type of the return value depends on the event and two values are returned per event, it is clumsy to read events in a single call. An event called *alarm* is enabled by a separate function, and is generated when the specified number of clock ticks have elapsed.

Device interrupts place event descriptors on queues. There is one queue for each device — keyboard, mouse button, etc. — so *waitevent* simply examines the head of all the queues to see what events are pending. Each event has a time stamp which is compared with the current time when timeouts are activated on a queue. If the program examines the queues often, timeouts are straightforward to implement. But since the program may compute for a significant time between successive calls to *waitevent*, timeouts in the past must make sense. The algorithm is this: When an event is returned to the program, its time stamp is recorded. When the program enables a timeout, *waitevent* decrements the timeout period by the interval

between the last event returned and minimum of the present time and the time of the next event (if any) in the queue being timed out. If the timeout period becomes negative, a timeout is generated. Otherwise the next event is returned if it exists, or the regular timeout code is executed if not. It is the decrement of the timeout period that lets the program catch up with real time.

A communication on a user channel is transformed into a simple assignment. A matching pair of actions *a?x* and *a*!3 becomes $x=3$. A nondeterministic choice between primitive events is compiled to a call to the underlying event code. As soon as one of the events is available, control is returned to the *squeak* program, which selects the appropriate process continuation for that event. A nondeterministic choice between user communications becomes a random choice between the possible execution paths. When a choice must be made between primitive events and user communications, *testevent* is called to check which primitive events are pending, and the choice made dynamically among the possible paths.

A parallel composition of processes is compiled into all the possible interleavings of primitive actions and communications of the component processes. This is done by advancing one of the processes one step, and considering all the possible continuations of that and all other processes. The state of the entire system is then restored to the initial state, and another path considered, advancing another process or the same process by a different action. This procedure is repeated until all possible executions have been considered. When more than one execution path is possible at a point, the set of possible communications is pruned and flattened to eliminate all the avoidable deadlocks and redundant nested execution paths, according to the laws $p+<> = p$ and $((p+q)+r) = (p+(q+r))$. The remaining available paths are compiled as a dynamic random selection of which path to take. A process identifier is simply expanded into the corresponding definition.

There are some optimizations that can be made during code generation. Note that any legal interleavings of the actions of parallel straight-line processes that do not access primitive events are equivalent. It is therefore unnecessary to generate all possible interleavings; one will do. The same applies within all subsequences of selections. The compiler therefore ‘pushes’ all processes as far as they can legally go, without accessing any primitive events, until the system is deadlocked. At this point, some processes will probably be blocked on primitive events, so the code is generated to access the event and choose subsequent execution depending on which event is received. For this to be successful, of course, the program must access primitive events, but a *squeak* program whose execution does not depend heavily on external inputs is probably pathological. To avoid loops in the compilation and to keep the generated code small, at each step of the compilation the translator detects states that have already occurred in the translation process, and generates jumps back to them, thereby folding the executions paths together at common states.

Here is a simple example, followed by the output of the translator:

```
proc p  = DN? . <a?x .  <c?z . p :: d?k . UP? .  p> :: b?y . p>
proc q  = <a!1 .  d!2 . q :: b!3 . UP? . q>
proc r  = c!4 . UP? . a!5 . r
example = p & q & r
```

```
example(){
        int x, y, z, k;
        Lab0:event(DN);
        Lab1:switch(nrand(2)){     /* 'a' or 'b' */
        case 0:     /* 'a' */
                x=(1);
        Lab2:switch(nrand(2)){     /* 'c' or 'd' */
                case 0:     /* 'c' */
                        z=(4);
                        switch(waitevent(DN|UP)){
                        case DN:
                                event(DN);
                                event(UP);
        Lab5:                   x=(5);
                                goto Lab2;
                        case UP:
                                event(UP);
                                event(DN);
                                goto Lab5;
                        }
                case 1:     /* 'd' */
                        k=(2);
                        event(UP);
                        goto Lab0;
                }
        case 1:     /* 'b' */
                y=(3);
                switch(waitevent(DN|UP)){
                case DN:
                        event(DN);
                        event(UP);
                        goto Lab1;
                case UP:
                        event(UP);
                        goto Lab0;
                }
        }
}
```

Initially, nothing can execute until *p* receives a *DN* event. It can then exchange with *q* either an *a* message, setting *x* to 1, or a *b* message, setting *y* to 3. It is instructive to follow through

the rest of the execution tree. Note particularly the state folding at *Lab*1, and where *p* and *r* exchange an *a* message, setting *x* to 5. The assignment to *x* occurs in two different execution paths that are folded together at *Lab*5. The innermost switch could actually be compiled into better code, since the order of receipt of the *DN* and *UP* events is irrelevant, but detecting situations like this requires looking at the states of processes after actions not yet compiled (that is, looking into the future), which our implementation does not do.

Use of squeak for complex interfaces

Although *squeak* was designed to program the lowest levels of a user interface, it can be used effectively to construct the higher levels by combining *squeak* programs hierarchically, treating larger events such as menu selections in the same manner as primitive events.

Consider the implementation of a hypothetical paint program on a bitmap display with a three-button mouse. A pair of *Click*-like processes monitor the left and middle buttons. The left button sets bits, the middle button clears them. When a click is received, a single instance of the brush is placed in the picture, with boolean combination function depending on which button was clicked. If *Click* generates a *down* event, multiple copies of the brush are laid out along the path traced by the mouse until an *UP* event is generated by the mouse. A *Menu* process is invoked whenever the right button is depressed, to select commands to change brushes, read and write files, and so on. Some action, perhaps a menu selection or a double click, invokes a high-resolution paint program that operates on individual pixels in a magnified portion of the picture.

By coding a *squeak* program that takes as arguments functions to call for the left and middle buttons, and a menu for the third button, the user interface can be made nearly identical in both painting modes: the regular paint program is instantiated with procedures to draw the brushes and the main menu, and the action that invokes the high-resolution program calls the same program recursively, but with arguments appropriate to painting individual pixels. Only one user interface need be written.

Of course, it may be possible to apply these ideas to the operating system itself. The concurrency in a *squeak* program is compiled out because processes are fairly expensive in a conventional operating system. If process scheduling is sufficiently fast, however, as in many real-time operating systems, it may be feasible to run *squeak programs* (not processes) as operating system processes. If the primitive events are known to the scheduler, it is possible to write a *squeak* program to read events from each input device and emit higher-level events. The higher-level events can then enter the scheduler as ‘primitive’ events to be dispatched to other processes. For example, the *Click* and *DoubleClick* processes above could interpret mouse button transitions for a set of independent user-level programs sharing the mouse, much as in the Blit operating system **mpx** [Pike 83].

Formal semantics: Concurrency and time flow

The interrelationships of the parallel processes and communications and timeouts lead to intricate flows of control. We defined the formal semantics of *squeak* as a tool for understanding the detailed behavior of *squeak* programs. In fact, our first attempt at a compiler failed because we underestimated the complexity of the behavior of parallel communicating processes. Once we had specified the formal semantics, our understanding was good enough that the second compiler was easy to write.

The semantics of *squeak* is given in a language called *formal squeak*. The two languages are very similar, but not identical. The major difference is that in *formal squeak* all delays between actions are explicit. To give the semantics of a *squeak* program, we translate it into *formal squeak*. First, all *squeak* actions *a?x.* or *a!v.* are converted to *formal squeak* actions *a?x*: or *a!v*:. The latter means ''do the action immediately, and at the next time unit do the rest of the process (immediately).'' To preserve the meaning of the original *squeak* program, we then introduce explicit delays between actions where they are needed.

A process is called *urgent* if all its immediate actions have timeouts, and is called *patient* if all its immediate actions do not have timeouts. Otherwise it is called *sloppy*. If the process following an action is urgent, no delay is introduced. If the process following an action is patient, a delay operator (δ) is introduced after the action. Finally, the top-level processes in the main program are examined, and the patient ones are prefixed by a delay. If a sloppy process is found, an error is reported.

We use operational semantics [Plotkin 81] to describe the meaning of *formal squeak* programs. A process in a state p can transfer to a state p' by a transition λ. In our case a transition can be an input action *a?v*, an output action *a!v*, a silent action (passage of one time unit), written 1, or several simultaneous actions.

The possible state transitions are expressed by a set of *inference rules*, listed below. There are two kinds of rules. In some situations a process can autonomously change state: these *ground rules* have the form $p \xrightarrow{\lambda} p'$. In other situations a process can change state only if a part of it can change state according to the inference rules; these *conditional rules* have the form $p \xrightarrow{\lambda} p' \Rightarrow q \xrightarrow{\lambda'} q'$. The implication sign is also written as a fraction line, with the condition above it and the consequence below.

A process δ*p* can spend some time doing 1 actions and then do whatever action p can do.

A simple output process, like *a!v*:*p*, can autonomously do an *a!v* transition and become p. As mentioned above, *a!v*:*p* means ''do *a!v* immediately, then at the next time unit do p.'' Hence *a!v.p* is equivalent to *a!v*:δ*p*, if p does not have immediate timeouts.

If there is a timeout, such as *wait*[3]*a!v*:*p* || *q*, and a silent action is performed, then the passage of time decrements the timeout period: *wait*[3]*a!v*:*p* || $q \xrightarrow{1}$ *wait*[2]*a!v*:*p* || *q*. If the *a!v* action is not selected in time, the process will degenerate into *wait*[0]*a!v.p* || *q* which can

perform only q. Input timeouts are treated similarly.

A process $a?x:p$ can receive any value on a, hence it can perform *all* the actions $a?v$ for any possible input value v. Therefore, $a?x:p$ is allowed to make autonomously any $a?v$ action, but only one of those v will be the right one — the one which is produced by a matching output action. Communication therefore occurs as pairs of actions; this is discussed in detail below.

A nondeterministic choice of processes can perform any action allowed by any of its component processes. As soon as a component process is chosen, the others are discarded.

A parallel composition of processes can perform an action only if all its components perform an action. The resulting action is a composite *product* action of all the component actions. For example, in $p \ \& \ q$, p may produce a $a?v$ action and q may produce a $b!w$ action. The resulting action for $p \ \& \ q$ is $a?v \ \& \ b!w$, the simultaneous occurrence of $a?v$ and $b!w$. Note that if a component of a parallel composition deadlocks, the whole composition deadlocks.

There are rules for simplifying these action products. A product of the form $a?v \ \& \ a!v$ reduces to 1, which models the exchange of a value v on channel a between exactly two processes. Moreover, the silent action is absorbed in products: $a?v \ \& \ 1$ is $a?v$. Because two complementary actions reduce to 1, the named channel has been used for communication, and the matching two actions are no longer available to other processes.

How does communication happen? According to the rules for input and output actions, it seems that inputs and outputs on a channel can happen independently and need not happen simultaneously, or transmit the same value. However, as one of many possible situations, input and output actions *may* match.

The restriction rule, labeled [Restr] in the list of rules below, is used to prune those situations in which inputs and outputs do not match: communications which *may* happen are *forced* to happen. When two communications match, the resulting action for the whole system is a 1. Hence, to force possible internal communications to happen, a subsystem is forced to exhibit only 1 transitions, or external communications. The notation $p \setminus R$, where p is a process and R a set of actions, prevents p from emitting those actions not contained in R, although such actions may still be reduced to 1 within p. The notation used in the syntax is $p \setminus a$, which is equivalent to $p \setminus R$ where R is the complement of the set containing all the single or composite actions having an a component; that is, $p \setminus a$ prevents p from exporting any action containing a.

For semantic purposes, a *main* program p in the syntax should be intended as $p \setminus Prim$, which can perform only primitive actions in the set $Prim$, which by definition always contains 1. All the other user-defined actions that p may want to perform are *inhibited* by $p \setminus Prim$; note that this is stronger that just filtering them away. Hence all the user-defined actions that components of p may perform must be matched by other components of p and reduced to 1; otherwise a deadlock will occur.

The following are the operational semantic rules for interpreting *formal squeak*. There are no rules for reducing expressions; we simply assume that expressions are already reduced to their final value wherever they occur. The letter v will be used to denote values.

[Delay] $\delta p \xrightarrow{1} \delta p \qquad \dfrac{p \xrightarrow{\lambda} p'}{\delta p \xrightarrow{\lambda} p'}$

[Wait] $\mathrm{wait}[n+1]p \;||\; q \xrightarrow{1} \mathrm{wait}[n]p \;||\; q$

$$\frac{p \xrightarrow{\lambda} p'}{\mathrm{wait}[n+1]p \;||\; q \xrightarrow{\lambda} p'} \qquad \frac{q \xrightarrow{\lambda} q'}{\mathrm{wait}[0]p \;||\; q \xrightarrow{\lambda} q'}$$

[Input] $a?id:proc \xrightarrow{a?v} proc\{v/id\}$

[Output] $a!v:proc \xrightarrow{a!v} proc$

[If] $$\frac{proc_0 \xrightarrow{\lambda} proc_0'}{\text{if true then } proc_0 \text{ else } proc_1 \text{ fi} \xrightarrow{\lambda} proc_0'}$$

$$\frac{proc_1 \xrightarrow{\lambda} proc_1'}{\text{if false then } proc_0 \text{ else } proc_1 \text{ fi} \xrightarrow{\lambda} proc_1'}$$

[Choice] $$\frac{proc_0 \xrightarrow{\lambda} proc_0'}{proc_0+proc_1 \xrightarrow{\lambda} proc_0'} \qquad \frac{proc_1 \xrightarrow{\lambda} proc_1'}{proc_0+proc_1 \xrightarrow{\lambda} proc_1'}$$

[Par] $$\frac{proc_0 \xrightarrow{\lambda_0} proc_0' \quad proc_1 \xrightarrow{\lambda_1} proc_1'}{proc_0 \,\&\, proc_1 \xrightarrow{\lambda_0 \& \lambda_1} proc_0' \,\&\, proc_1'}$$

[Rename] $$\frac{proc \xrightarrow{\lambda} proc'}{proc\{id/id'\} \xrightarrow{\lambda\{id/id'\}} proc'}$$

[Restr] $$\frac{proc \xrightarrow{\lambda} proc'}{proc \setminus R \xrightarrow{\lambda} proc' \setminus R} \quad \text{if } \lambda \varepsilon R$$

[Def] $$\frac{proc\{actuals/formals\} \xrightarrow{\lambda} proc'}{pid(actuals) \xrightarrow{\lambda} proc'} \quad \text{where } pid(formals)=proc \;\varepsilon\; \text{Defn}$$

where *Defn* is the set of process definitions for a particular *squeak* program.

A simple example may clarify how the semantics works. Consider the following process (where we have taken some syntactic liberties to match the semantic rules):

```
proc p = a?x : <>
proc q = a!3 : <>  +  b?y : <>
simple = (p & q) | Prim
```

All possible actions of the components from the bottom up must be computed to determine the actions of the whole. The *p* component can do all actions of the form *a?v*. The *q* component can perform *a*!3 and all actions *b?w*. Their parallel composition can perform all the possible products of *a?v* with *a*!3 and of *a?v* with *b?w*. These product actions are: (*a?*3 & *a*!3) = 1, (*a?v* & *a*!3) for $v \neq 3$, and (*a?v* & *b?w*) for all *v* and *w*. But of all these actions, only 1 is in the Prim set. Hence (*p* & *q*) | *Prim* can do only a 1 action, which corresponds to the communication of 3 on channel *a* between *p* and *q*.

Conclusions

Squeak is a concurrent language for specifying interactive user interfaces. It can express complex time-dependent interfaces in a compact notation. Although *squeak* could be developed into a full-blown language, we use it to express subroutines which are then integrated in larger programs written in a conventional sequential language (C).

The integration of concurrent subsystems in sequential programs is achieved by compiling concurrency into sequential code whose execution is controlled by the sequencing of external device events. It is interesting that in the restricted domain of *squeak* programs, the context switches between concurrent processes can be compiled out.

The real-time behavior of *squeak* is subtle, and we have found it helpful to express the language's semantics formally, using the methods of operational semantics.

References

[Berry 84] Berry, G., ''The ESTEREL synchronous programming language and its mathematical semantics,'' *Proc. of the NSF/SERC workshop on concurrency,* CMU, 1984.

[van den Bos 83] van den Bos, J., Plasmeijer, M.J. and Hartel, P.H., ''Input-Output Tools: A Language Facility for Interactive and Real-Time Systems,'' *IEEE Trans. Soft. Eng.,* **SE-9**(3), pp. 247-259, 1983.

[Buxton 83] Buxton, W., Lamb, M. R., Sherman, D. and Smith, K.C., ''A User Interface Management System,'' *USENIX Conf. Proc.*, June 1983, pg. 177.

[Hoare 78] Hoare, C.A.R., ''Communicating Sequential Processes,'' *Comm. ACM* **21**(8), pp. 666-678, 1978.

[Kernighan 78], Kernighan, B.W. and Ritchie, D.M., *The C Programming Language,* Prentice-Hall 1978.

[Milner 80] Milner, R., ''A Calculus of Communicating Systems,'' *Lecture Notes in Computer Science, nr.92,* Springer-Verlag, 1980.

[Milner 82] Milner, R., ''Four combinators for concurrency,'' *ACM SIGACT-SIGOPS Symp. on Princ. of Distributed Computing,* Ottawa, Canada, 1982.

[Pike 83] Pike, R., ''The Blit: A Multiplexed Graphics Terminal,'' *AT&T Bell Labs Tech. J.,* **63**(8), part 2, pp. 1607-1631

[Plotkin 81] Plotkin, G.D., ''A Structural Approach to Operational Semantics,'' *Internal Report DAIMI FN-19,* Computer Science Department, Aarhus University, September 1981.

[Thomas 83] Thomas, J.J. and Hamlin, G., ''Graphical Input Interaction Technique Workshop Summary,'' *Computer Graphics,* January 1983, pp. 5-30.