Synchronous/Reactive Programming of Concurrent System Software

Bruce R. Montague and Charles E. McDowell
Computer and Information Sciences
University of California, Santa Cruz

UCSC-CRL-95-51
November 28, 1995

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

Synchronous languages are intended for programming reactive systems. Reactive systems, which include real-time systems and key operating system components, interact continually with their environment. This paper considers the applicability of imperative synchronous/reactive languages to the development of general system software, that is, to the implementation of operating system kernels, file systems, databases, networks, server architectures, device drivers, etc.. The languages Esterel and Reactive C (RC) receive special attention as Esterel is the oldest and most developed such language and RC is specifically designed for compatibility with C systems programming. An alternative *soft-instruction* software architecture is described which is well suited to real-world system programming.

keywords: reactive systems, synchronous language, concurrent programming, system software, operating systems, threading, real-time systems, soft-instructions.

0.1 Introduction

The imperative synchronous languages Esterel and Reactive C (RC) were developed to address concurrent programming difficulties associated with the reactive systems commonly encountered in real-time and embedded programming [BdS91] [Bou91]. These languages have been called *synchronous/reactive* languages. This paper considers the applicability of such languages to the closely related problem of general system software development, and then describes an alternative software architecture intended specifically for system software development.

The remainder of this paper is organized as follows: The concurrent system programming problem is briefly described and the importance of establishing software architectures suited to this problem is noted. Reactive systems and synchronous languages are then described and existing work briefly surveyed. Source code is examined from the viewpoint of the system programmer, and conclusions drawn regarding applicability to implementation of generic system software. The *soft-instruction* software architecture is described, an example considered, and relevant historical and current work noted.

0.1.1 Concurrent System Software

System software consists of components such as operating system kernels; file, database, and network systems; device drivers; and server architectures for I/O, transaction processing, and multimedia. These components react to external *service requests* generated by applications, other components, and hardware. Each component is generally capable of servicing multiple concurrent requests, some of which may be of long-duration, and some of which may have real-time constraints. Thus, operating system software constitutes a significant concurrent programming problem, and it is widely agreed that development of such software remains difficult [Atw76] [Sch86] [Wat90].

There are many ways to view concurrency. The remainder of this section describes concurrent programming considerations germane to the design and programming of concurrent system software, that is, concurrency from the viewpoint of the system programmer.

We can identify three orthogonal attributes that affect the systems programmer:

Competitive vs Cooperative Are concurrent requests competing for the same resource or are they cooperating to satisfy a single, higher level request? The answer to this may depend on the component that is being used as a frame of reference.

Heavyweight, Lightweight, or Featherweight How expensive is a context switch?

Internal vs External Are concurrent requests being managed by a single component?

These are explained further below.

Cooperative concurrency either increases a single request's performance or simplifies multiple event coordination pertaining to a single request. Performance is increased by techniques such as issuing multiple overlapped I/O operations on behalf of a single service request. Coordination provided by cooperative concurrency simplifies activities such as request cancelation and error handling. Cooperative concurrency typically reduces the latency of a single request.

Competitive concurrency maintains the *timesharing illusion*. This permits a given service request to be considered 'invisible' to other unrelated service requests currently active within the same system component. Competitive concurrency occurs when a system component can execute two or more concurrent and unrelated service requests generated external to the component. Competitive concurrency mechanisms typically increase overall throughput of the component.

The amount of context copied during a context switch dominates performance. Modern hardware register sets are divided into non-privileged and privileged registers. Non-privileged registers are accessible to applications while privileged registers are protected from normal application access. A *heavyweight* context switch completely switches both privileged and non-privileged registers. This may require updating memory structures associated with both the privileged registers and with the software process model implemented by the system. A heavyweight

context switch typically crosses protection boundaries.

Switching only the non-privileged registers provides *lightweight* context switching. Typically, one of these registers is the application stack pointer, so there is usually a stack associated with each *thread* defined by a set of non-privileged registers. A lightweight context switch is often an order of magnitude faster than a heavyweight context switch [Laz91].

High-performance systems occasionally are designed which switch only a subset of the non-privileged registers. This approach is used in some transaction systems, such as IBM's TPF, and is often used within the interrupt handling component of operating systems [Mar90]. Context switching a subset of the non-privileged registers will be called *featherweight* in the remainder of this paper. In the extreme, featherweight context switching involves switching only a single register, which usually is the base register of a data context describing a transaction or service request.

In this paper we are concerned with concurrency explicitly controlled by a single component to at least some extent. We call this *internal* concurrency. All other concurrency is *external*. More specifically, from the viewpoint of a systems programmer, all concurrent activities occur in response to some request. Given two concurrent activities, the activities represent internal concurrency if at least one activity is being handled by the component under consideration and the following two conditions are true. All other cases represent external concurrency.

1. Both activities are being handled by the same component.
2. The data structures representing the concurrent activities are explicitly represented within the component.

An example that satisfies the first condition above but not the second would be conventional re-entrant code with no shared data structures. One or more external components could initiate distinct activities within the component that were completely unrelated.

Due to differing internal concurrency requirements, different components often implement different internal concurrency mechanisms. Both competitive and cooperative concurrency may be supported internal to the component.

A system component itself typically executes in a concurrent environment in which it competes with other components for external resources. A common example of such external concurrency is competition between components for the CPU. Another example of important external concurrency is the mechanism by which the component can initiate and manage multiple asynchronous external requests, for instance, multiple asynchronous I/O operations.

The degree that concurrent contexts internal to a system component are simultaneously exposed to the system programmer differs, often reflecting the required degree of cooperation between requests. At one end of the spectrum are classical threading models which typically do not expose to a given thread the internal state of other threads, that is, a thread's stack contents are private. At the other extreme, as exemplified by some windowing systems, multimedia servers, and simulation-action games, all concurrent context may be equally visible and expressly managed by the programmer.

In thread-based approaches context tends to be encapsulated in a single data structure, such as a stack, which is explicitly scheduled by a lower-level scheduler. The implementation of this scheduler is often not visible to the programmer. In these systems, thread scheduling provides for concurrent behavior of processes, threads, transactions, requests, etc.. The programmer gets concurrency 'for free' because implementation of the underlying threading system is not the programmer's responsibility.

At the other extreme, where expressly managed concurrent behavior is inherent in the program implementation, the programmer has complete responsibility for locating and scheduling activities. The programmer is in control and effectively defines custom context and concurrency mechanisms. Context may be distributed throughout program data structures. Locating relevant context upon the occurrence of each significant event may require considerable run-time logic. Such expressly managed approaches tend to be used when concurrent activities are highly interrelated and it is advantageous for the programmer to have a 'gods eye' view of the entire concurrent state. Sometimes, as with the X-windows system, the

code that expressly manages activities is placed into a standard run-time system. In general, modern windowing systems have used approaches based on explicit application event dispatch work-loops and callbacks rather than thread-based concurrency. This concurrency style, convenient in highly interactive cooperative environments, is sometimes called *faux* concurrency [Rep95].

Concurrent service requests for file, network, database, and transaction systems often have a high degree of independence. Unlike message and window servers, two arbitrary service requests to this type of server are usually competitive. Optimally, the degree of internal concurrency within such a component is determined dynamically by the rate at which the external environment generates service requests. Lightweight or featherweight competitive internal concurrency mechanisms are thus very important for these servers.

The system programmer is responsible for considering all of these issues. The internal concurrency mechanisms in many system components are only used by the system programmer, and never directly by external code. The system programmer is therefore free to implement custom concurrency mechanisms as well as program using those mechanisms.

Thus, when implementing a system component, one often programs with at least 3 different viewpoints in mind: the perspective of the individual service request, the perspective of the custom internal concurrent programming mechanism through which all individual service requests are controlled within the component, and the external concurrent environment that must be used but over which one may have no direct control. Optimally, the operating system kernel implementation itself supports these viewpoints, thus facilitating component implementation. The kernel is often considered simply another component that has primary responsibility for CPU allocation and interrupt dispatching.

0.1.2 Software Architecture

Software architecture has been defined as *'The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time'* [GP95]. A software architecture is distinguished by *'a shared repertoire of methods, techniques, patterns and idioms for structuring complex software systems'*.

Software architectures provide an abstract conceptual approach to complex systems more specific than the models often inherent in a general purpose language but more general than a single design. A software architecture provides an identifiable approach or framework often implicitly framed by the adopted tools, languages, and development environments [GTP95] [Gar95].

Good system software invariably adopts a well defined software architecture. The architectures used for system software are usually based on some form of explicitly coded *critical sections* [Atw76] [FP88] [Her91]. However, explicit critical sections introduce nondeterminism and are a root cause of many concurrent programming difficulties [Her90]. Since synchronous languages do not contain explicit critical sections, software architectures based on such languages may have advantages over traditional architectures.

0.2 Reactive Systems

A *reactive system* is event-driven, maintains a permanent interaction with its environment, and executes at a rate determined by the environment [HP85]. Reactive systems are assumed to execute with performance sufficient to ensure they are never overdriven by their environment. Since under normal circumstances a reactive system never terminates, reactive systems cannot be characterized as a simple function from a single initial input state to a single final output state. *Real-time* systems are reactive systems with the addition of *timing constraints*. Operating systems are inherently reactive and provide the archetypical example of large reactive systems [JS89] [MK93].

The term *reactive* is more specific than the informal term *event-driven*, which is widely used and overloaded. For instance, an event-driven program may calculate a simple transformation and terminate. The term reactive is more general than *soft real-time* and *near real-time*, because a reactive system does not address any real-time constraints but only correct causality ordering [BB91].

```
state light_off {                   /* State 1 -- Light is Off           */
    when (volt > 5.0) {             /*    When in this state and voltage  */
        light = TRUE;               /*    exceeds 5 volts, turn on the    */
        pvPut(light);               /*    light, and then                 */
    } state light_on                /* Enter State 2, Light On.           */
}
                  GUARDS
state light_on {                    /* State 2 -- Light is ON             */
    when (volt < 3.0) {             /*     When in this state and voltage */
        light = FALSE;              /*     falls below 3 volts, turn the  */
        pvPut(light);               /*     light off, and then            */
    } state light_off               /* Enter State 1, Light Off.          */
}
```

Figure 0.1: A SNL Code Fragment

Simple reactive systems are often programmed as explicit finite state machines, with external events driving the machine through state transitions. Explicitly coded state machines work well for problems with fewer than around 10 states. Above this size, explicitly programming a single state machine becomes difficult. Nonetheless, state machines are often used for device drivers, as this size suffices for many driver architectures. In this case a component such as an I/O supervisor usually has responsibility both for executing the state machines and instantiating the state machines needed to deal with physical concurrency due to multiple devices.

Programmers are often provided with special languages for integrating state machines into their program source. State Notation Language (SNL) is typical of these languages [Koz93] [KKW94]. SNL is used for I/O intensive control systems and is compatible with C system programming.

An example SNL program fragment is shown in Figure 0.1. This code turns a light on when a voltage exceeds 5 volts and off when the voltage falls below 3 volts. Examination of this code fragment illustrates how large programs developed in this manner suffer from hidden 'spaghetti gotos'. Statement <`state label_1 {...}state label_2`> effectively terminates in a <`goto label_2`>. The `when()` clauses guard the following code-block whenever the appropriate state has been entered, that is, the code-block will not execute until the corresponding guard is true.

SNL uses a 'run-time sequencer' responsible for evaluating all the guards and serializing execution of ready code-blocks. SNL provides very good integration of the state machine, I/O, and conventional C code. Kozubal, et al., note that 'extremely complex' SNL applications have between 10 and 20 states. However, an SNL sequencer may control execution of multiple independent state machines, on occasion controlling concurrent execution of as many as 10 state machines. SNL is considered easy to understand and use.

For slightly larger applications, such as standalone industrial controllers, some means of providing hierarchical structuring of state machines is required. A typical industrial system, intended to support up to around 150 states, is the *Action-State diagram* [KVK91]. This approach resembles coupling decision tables and state transition diagrams. The decision tables are compiled into hierarchical tables of action-routine addresses. A work-loop then sequentially executes action routines in response to events and the current state. This approach works well for controllers or drivers that do not require internal competitive concurrency.

Larger reactive systems that must support both competitive and cooperative concurrency are usually based on conventional operating system kernels providing multithreading and critical sections. This approach introduces nondeterminism and its associated concurrent programming problems. For large *hard real-time* systems that must guarantee predictable

performance, *cyclic executives* are arguably still the preferable architecture. A cyclic system uses pre-computed deterministic schedules designed for the worst case. While effective, the overhead of such pessimistic systems can be high, as code executes on a rigid table-driven timeline even if not needed. Under sustained near worst case conditions, however, nondeterministic systems that must expend run-time overhead scheduling their activities are less efficient than cyclic systems [Kop91].

0.3 Synchronous Languages

Recently, *synchronous* software architectures have been proposed specifically for reactive systems [BB91] [Hal93]. The resulting synchronous/reactive systems include data-flow and declarative approaches as well as more traditional imperative languages. None of these approaches have proven intrinsically more powerful than the others, and an effort is currently underway to provide a common 'back-end' for a number of these systems. The typical *reactive kernel* for which the synchronous languages are intended has many properties of general system software. Many such kernels resemble *stand-alone* device drivers, that is, drivers running directly on the machine hardware without additional systems support.

The *synchrony hypothesis* assumes that all computation occurs in discrete atomic steps during which time is ignored. This is often stated as the assumption that all program code executes in zero time. Time only advances when no code is eligible for execution. During a single step, all output is considered to occur at the same time as all step input, that is, output is synchronous with input. The notion of continuous time is thus replaced by an ordered series of discrete steps between which discrete changes occur in global state. The code executed at each step is called a *reaction*.

The fundamental advantage of this approach is that internal cooperative concurrency can be handled deterministically. Concurrent asynchronous events are manifested only within global state 'snapshots'. No explicit critical sections occur in the source code, not even in the guise of monitors or concurrent objects, because all code can be considered inside implicit critical sections executed as required via *guarded command* style programming.

The synchronous/reactive languages focus on the internal cooperative concurrency commonly found in drivers and controllers. Synchronous languages essentially 'compile away' all internal cooperative concurrency by producing a single deterministic state machine that manages all required activities. Nondeterministic external events are handled by the reaction dispatch or guard evaluation mechanism, and do not directly propagate into the body of the program [BGJ91]. Since the single state machine into which concurrent programs are compiled cannot deadlock, the need for multiple threads, critical sections, and nondeterminacy is eliminated. However, synchronous languages that compile to a single state machine must sacrifice recursion and dynamic data allocation to obtain determinism.

Synchronous architectures somewhat resemble cyclic systems executing code repetitively using pre-computed schedules. Synchronous approaches, however, see time as merely another discrete global state variable and only execute required reactions. Synchronous languages provide *non-blocking* and *wait-free* internal concurrency. A concurrent system is non-blocking if some member is guaranteed to complete an operation in a finite number of system steps. A system is *wait-free* if it can be guaranteed that each member completes an operation in a finite number of system steps [Her91] [Her90]. Investigating alternatives to process-based concurrency is a current research area. For instance, Lamport notes that processes are an artifact and need not be adopted as a fundamental primitive in theories of concurrency [Lam94].

0.4 Current Studies

Selected efforts relating to synchronous/reactive programming are examined in this section, primarily with respect to their source code and potential applicability to system programming.

0.4.1 Meta/NPL

The ISIS system is a reliable distributed system developed at Cornell [BJ87]. While attempting a

large distributed ISIS application, the need arose for a distributed reactive toolkit because ISIS lacked tools for distributed control [MW91]. This motivated the development of Meta, a toolkit for building non-real-time reactive systems. Meta provides a state machine language called NPL. NPL is a guarded command language providing a globally consistent view of distributed state. Guarded commands are interpreted and have atomic action semantics. Performance is considered adequate for systems in which timing is not crucial.

Figure 0.2, based on an example by Marzullo and Wood, illustrates NPL code. NPL uses simple stack-based expressions, with arguments preceding a postfix operator. The general format of a statement is <predicate GUARD actions [ALTERNATE predicate GUARD actions]*>. The actions are executed atomically whenever the corresponding guard predicates become true. Figure 0.2 consists of one such statement. The action consists of a single statement with operator NPL which takes the 2 preceding strings as arguments. The second string argument is itself an NPL statement which contains an alternate guard.

Three guards exist in Figure 0.2. The first guard executes the NPL statement whenever load exceeds 5. The NPL statement takes two arguments, a ‘context’ (in this case, server), and a program fragment to execute in that context. The second guard exits the action if the load falls below 5. In this case, the first guard remains enabled and the action will be re-executed if load again exceeds 5. The third ‘alternate’ guard starts a timer whenever the first guard fires. If 20 seconds passes, TIMER returns true and the third guard executes idle-server. The LEAVE statement then causes the entire statement in Figure 0.2 to be removed from further possibility of execution. Thus, Figure 0.2 has the effect of executing idle-server if the load exceeds 5 for 20 seconds.

NPL is implemented as an interpreter driven by guard evaluation. NPL uses a non-deterministic Least Recently Used (LRU) policy to select which ready action to execute first. Although typical of guarded command languages and designed explicitly for reactive environments, NPL does not assume the synchrony hypothesis.

0.4.2 Esterel

Esterel is the oldest synchronous/reactive language and the best documented [BC84] [BdS91] [BG92] [Hal93] [Edw94]. The design of Esterel was motivated by an effort to develop a semantics of parallel and real-time programming following Robin Milner’s theories of synchronous process algebras [Mil93].

A complete Esterel module is shown in Figure 0.4 and code fragments in Figures 0.3 and 0.5. Figure 0.3 is from an example by Murakami and Sethi, and Figures 0.4 and 0.5 follow an example by Halbwachs [MS90] [Hal93]. Esterel is not a complete programming language. It is a program generator used to describe the reactive kernel of reactive programs, that is, it provides a deterministic reactive control harness which calls routines written in a conventional programming language.

Program execution forms a discrete sequence of *instants*. The only global state consists of instantaneously broadcast *signals*, with broadcast synchronous with the instant in which a signal occurs. Signals last the entire current instant, that is, from the start of the instant in which they are emitted to the end of that instant. The state of all signals emitted during a given instant is altered synchronously. Time is treated as a signal identical to any other signal. A *pure* signal is emitted using the syntax <emit s;>, where s is a signal name. Statement <emit s(N);> associates the integer value N with signal s. The statement <present s then s1 else s2 end> executes statement s1 if signal s is present and statement s2 otherwise. The value of signal s is obtained by ?s.

The statement <s1||s2> indicates that statements s1 and s2 react in the same instant. Potentially all routines in a program can react during the same instant. All reactions are atomic and execute to completion within the current instant. While one reaction is executing, no other reactive routine can execute. A reaction can block ‘in place’ until the instant in which signal s is present via <await s;>. To wait for the third instant in which signal s has occured, for example, await takes the form <await 3 s;>.

The fundamental control construct is the *watchdog*, with syntax <do s1 watching s timeout

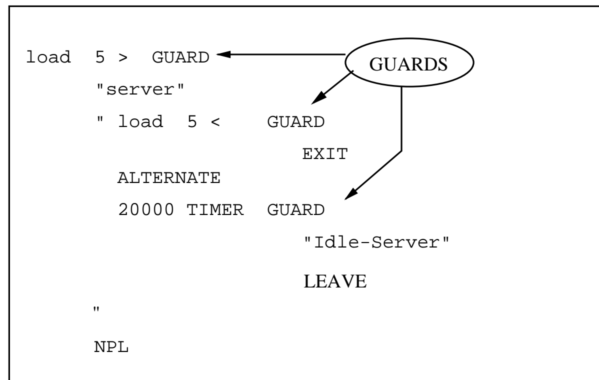

Figure 0.2: An NPL Routine

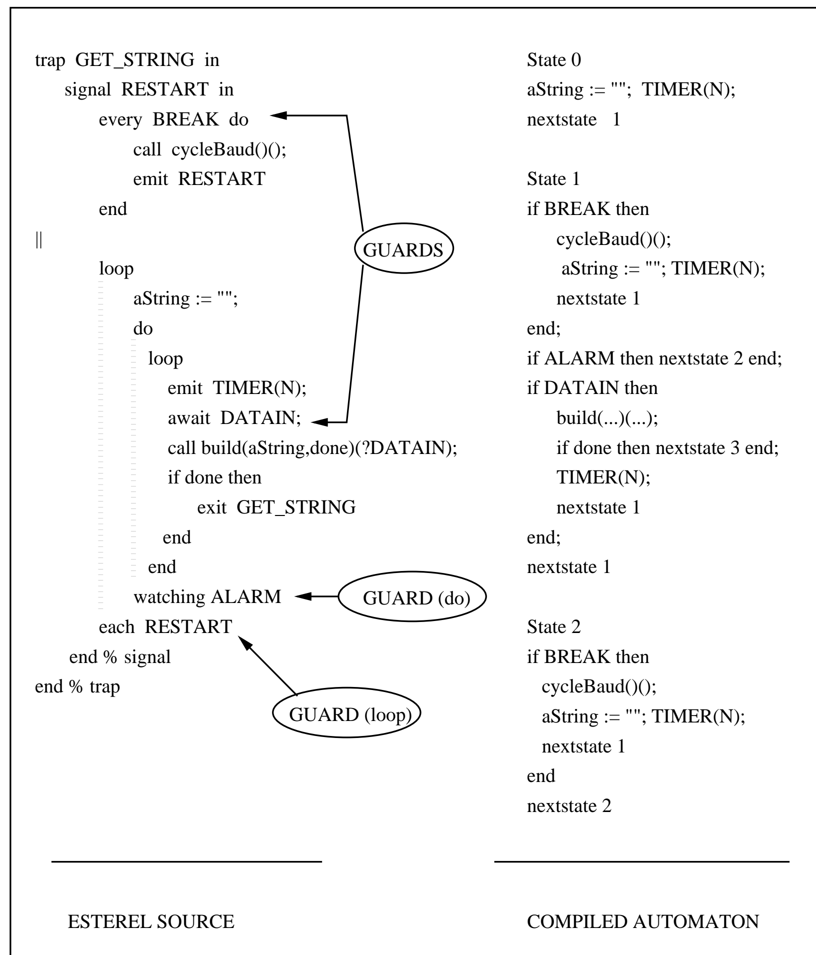

Figure 0.3: An ESTEREL Routine

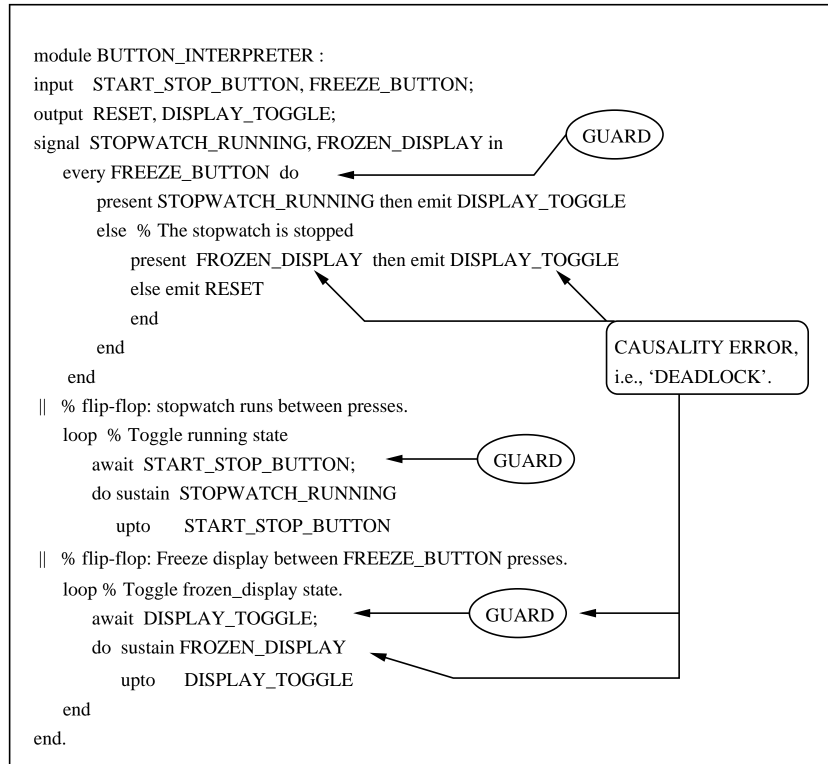

Figure 0.4: A Causality Error in ESTEREL

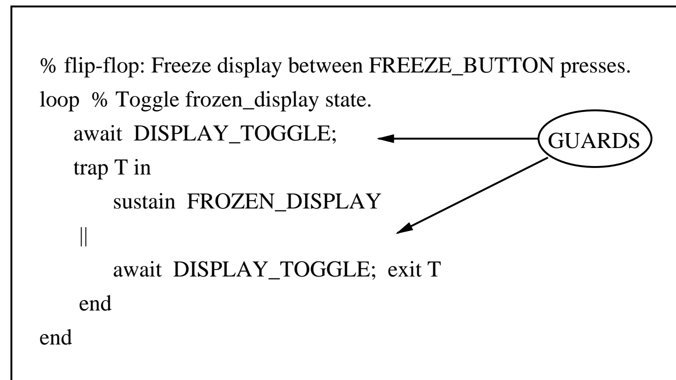

Figure 0.5: Interval Sequencing Solves a Causality Error

s2>, where s is a signal and s1 and s2 are compound statements, that is, sequences of statements separated by semicolons. Body s1 is executed unless signal s occurs, in which case the body is terminated and the timeout executed. The timeout is optional. Statement s1 is not executed in the instant signal s occurs. Since statement s1 may contain await statements, a do statement may span many instants.

The general form of a trap statement is <trap T in s1 end>, where T specifies the name of the trap block defined by compound statement s1. Traps can be nested. The trap can be exited from within the trap block by a statement of the form <exit T;>, where T indicates which enclosing trap to exit. The statement <trap T in await s; s1; exit T end> executes s1 until and including the instant in which signal s occurs. Note that the previous do statement does not execute statement s1 within the

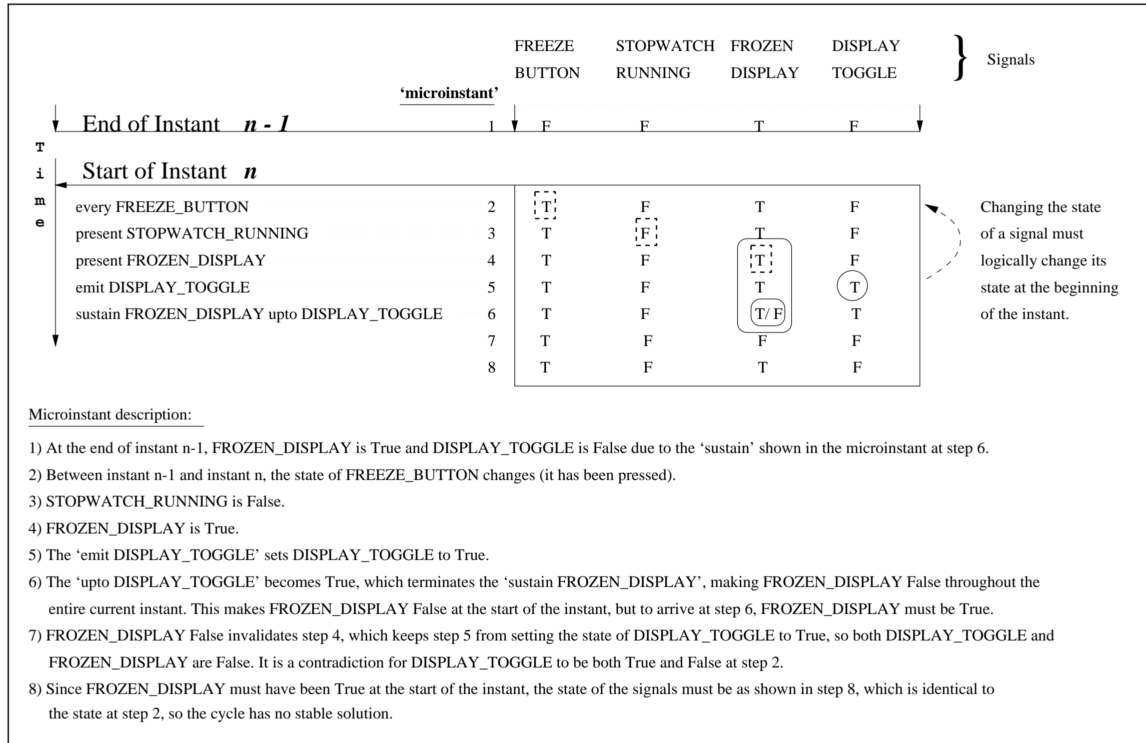

Figure 0.6: A Causality Error Trace

first instant signal `s` occurs, while the given `trap` statement does execute statement `s1` within the first instant signal `s` occurs. Many Esterel statements are effectively constructed as macros using the `do` and `trap` statements as primitives.

Figure 0.3, from an example by Murakami and Sethi, contains a code-block that assembles a string of input characters into a buffer, subject to a time-out [MS90]. If a `BREAK` occurs, the baud rate of the input line is re-determined and buffer assembly restarted. This is a typical small systems programming problem.

The `<trap GET_STRING>` in Figure 0.3 enables `<exit GET_STRING>` to exit the code-block. Signal `RESTART` is declared with local name scope. In any instant that external signal `BREAK` occurs, routine `cycleBaud()()` executes and `RESTART` will reset the loop executing in parallel. Note Esterel routines take 2 argument lists. The first specifies parameters passed by reference and the second parameters passed by value.

The loop collects input into buffer `aString`. The loop starts a timer by emitting signal `TIMER(n)`. At the instant corresponding to an elapsed time of `n`, the timer module will raise signal `ALARM`. The timer counts signal `TICK`, which implicitly occurs every instant. The `await` command blocks execution until the instant that `DATAIN` contains data, at which time its value is obtained by `?DATAIN`. Since all code executes atomically and global state is updated synchronously with the instant, there is never a possibility of `DATAIN` changing value due to a race condition.

Figure 0.3 also illustrates Esterel compilation. Output is a directly compiled finite state machine which does not require an interpreter for execution. The compiled code is extremely fast since there is no need for concurrent threads, messages, or interpreters. However, there is considerable redundant code. For example, the first code-blocks in State 1 and State 2 are the same.

Figure 0.4, based on an example by Halbwachs, illustrates a module in a stopwatch controller [Hal93]. The stopwatch has 2 buttons. One button starts and stops the timer. The other button freezes and unfreezes the stopwatch display when the stopwatch is

running. The start/stop button can stop the running stopwatch when the display is frozen, after which the freeze button can be used once to display the final time. When the stopwatch is stopped and the final time is displayed, pressing the freeze button again resets the stopwatch.

The example contains 3 concurrent code-blocks. The signal `start_stop_button` simply toggles the signal `stopwatch_running` as specified by the loop in the second code block. The signal `display_toggle` likewise toggles the signal `frozen_display` in the loop contained in the third code block. Signal `display_toggle` does not simply reflect the state of `freeze_button` because of the modal operation of this button, which is interpreted by the first code-block.

This example contains a *causality error*, which is the synchronous equivalent of deadlock. Unlike run-time deadlock, causality errors in principle can be detected at compile time due to the synchrony hypothesis. These errors arise because it is possible to write reactions of the form *'emit signal s if and only if it is absent'* or the undetermined *'emit signal s if and only if it is present'*. In Figure 0.4 it arises because if signal `frozen_display` is present, signal `display_toggle` is emitted, but if signal `display_toggle` is emitted then signal `frozen_display` is not present due to the `upto` at the bottom of the module. The `<do s1 upto s>` is defined as `<do s1; halt watching s>`, that is, statement `s1` is not executed the instant that signal `s` occurs and `s1` will not be executed in future instants.

Figure 0.6 illustrates the causality error. The synchronous nature of Esterel is also illustrated in this figure because the `emit` of signal `display_toggle`, which occurs if signal `frozen_display` is true, must cause the signal `frozen_display` to be false *at the beginning of the instant*, leading to a contradiction.

The solution to the causality problem, following Halbwachs, is shown in Figure 0.5. The code-block at the bottom of the module in Figure 0.4 is replaced with a trap-based sequencer. Since the trap continues to execute its body the first instant that `display_toggle` is true, signal `frozen_display` remains true for the entire instant, becoming false in the next following instant.

0.4.3 Reactive C

Reactive C (RC) is designed to provide extensions to C supporting synchronous/reactive programming based on the Esterel model [Bou91] [Bou92]. RC is implemented as a preprocessor generating C source code. RC does not compile to a finite state machine, but rather provides the C programmer additional statements for expressing Esterel-style reactive control flow. RC is more general than Esterel. Guard conditions can be boolean expressions and reactions executing within the same instant can coordinate and synchronize via *micro instants* and the *suspend* statement.

Example RC statements described by Boussinot are shown in Figure 0.7. The `stop` statement in reactive procedure `Hello()` is considered the basic RC reactive statement. `Stop` halts execution for the remainder of the current instant. The next instant, control resumes at the immediately following statement.

The condition argument of the `select` statement in Figure 0.7 is evaluated every instant. If the condition is true, routine `P1()` is executed, otherwise `P2()` is executed. In the statement shown, `x` alternates between true and false, so execution of `P1()` and `P2()` will alternate each instant.

The `par` statement in Figure 0.7 specifies multiple reactive statements to be executed in the same instant. It takes only 2 other statements as arguments. Unlike the Esterel `||` syntax, the order of execution of these 2 statements is fixed, with the first argument executing first within the instant, and the second executing next within the instant. Typically, execution would block within each branch of the `par` statement during the first instant. At the second instant, control proceeds from the 2 blocked control points in the same order, that is, the first argument executes before the second.

RC abandons Esterel's purity with respect to logical instants providing the only event ordering. A single instant can be divided up into micro instants. This is illustrated by the 2 statements on the right side of Figure 0.7. In the first `par` statement, the

```
rproc void Hello(){
        printf( "Hello, world\n" );
        stop;
        printf( "I repeat: hello, world\n" );
}
```

```
par
   {  suspend;  printf("1"); }
   {                printf("2"); }
```

```
select( x = !x )
    exec  P1();
    exec  P2();
```

```
close
    par
        { suspend; printf("1"); }
        {              printf("2"); }
```

```
par
    exec  Hello();
    exec  Bye();
```

Figure 0.7: Reactive C Statements

`suspend` in the first argument suspends execution till the next instant. Thus, a `'2'` is output the first instant and a `'1'` the second instant. The `close` statement forces execution to be restarted during the current instant, that is, after the second argument of the `par` statement completes execution, the suspended first argument is resumed. The output of the `close` statement is `'21'` at the end of the first instant. The `suspend` and `close` statements are useful when multiple routines have to monitor each others state, wait for initialization to be complete, etc..

0.4.4 Evaluations

A group at Bell Labs evaluated 6 different reactive specification and development systems with respect to a reactive coding problem originally implemented in C in the ATT 5ESS system [ACJ+95]. Esterel was included but not RC. The evaluation considered real-world applicability, compatibility, and software engineering concerns such as language learning curves. This study concluded that Esterel required more expertise than the other approaches. For this group of experienced evaluators, however, overall learning curves were not a significant problem for any of the systems. Esterel was found to be the most expressive language and scaled best to large application domains. The most notable result of this study was that maintainability was not a strength of any of the evaluated approaches. The group recommended that maintainability must be addressed before any of the evaluated methods would be suitable for large-scale industrial use.

A German research effort evaluated 18 different reactive programming systems amenable to formal analysis by funding implementations of the same 'toy' manufacturing cell controller in each system [LL95]. As with the ATT evaluation, Esterel was included but not RC. The resulting 400 line Esterel program provided a reactive control harness for conventional C routines. The Esterel program had to be split into 5 separate reactive kernel modules with independent state machines in order to be tractable for the tools and the code generator. No proofs were attempted because the complete system was too large, and the modularized system introduced asynchronous communication between modules. It was noted that a large number of signals overwhelm the designer, and that small changes in an Esterel program can result in significant changes in state machine size, making requirements unpredictable. Although believing the approach simplified the design of the controller, the group implementing the Esterel controller concluded that *'the advantages are limited if either many kernels are loosely coupled or if the*

data structures used are complex' [Bud95]. The resulting executable program was 46 Kilobytes, with 26 Kilobytes resulting directly from Esterel and the remainder primarily resulting from the low-level C routines.

0.5 System Programming Considerations

High-performance concurrent reactive systems play a central role in the future of operating systems. This is especially true for servers requiring a very high degree of competitive internal concurrency, for instance, servers for transaction processing, databases, networks, and multimedia. Context switch overhead bounds the performance of conventional multithreading approaches based on heavyweight and lightweight concurrency [Ous89] [MB91] [ALBL91]. Context switch code paths through general purpose systems are lengthy and context switches destroy locality of reference assumptions upon which high performance systems rely. Featherweight context switch techniques that can provide a high degree of both competitive and cooperative internal concurrency are thus of interest. Historically, such approaches have long been used in areas such as real-time avionics and high performance transaction processing. Example systems that have been described include Boeing's Rex and IBM's TPF [BS86] [Mar90]. This section examines whether the synchronous/reactive techniques considered in the previous section are appropriate for high performance generic system software.

0.5.1 General Observations

As indicated by the 'guard' balloons in the figures, all the languages described enable programming large state machines using parallel variations of Dijkstra's *guarded commands* [Dij75] [Hal77]. To understand the source code in these systems, one identifies the guard locations and determines the conditions under which a guard will activate. For instance, in reading Esterel or RC, a productive first step is to identify all `emits` and all corresponding guards.

Guarded command programming resembles dataflow programming in that the programmer has no direct control over the selection process, that is, the order in which multiple ready reactions are activated. This is most clearly visible in NPL, which uses an LRU policy to select what executes next. Although synchronous languages are deterministic, the specific order in which code executes may not be immediately obvious from the source code.

The synchronous languages resemble a dataflow approach to programming cyclic systems with all computation based on a traditional timeline divided into *major* and *minor* cycles. The similarity is most obvious in the case of RC, where instants may be considered major cycles and micro instants correspond to minor cycles. In both Esterel and RC instants can be considered major cycles and the sequence of all reactions that execute within the instant corresponds to a minor cycle sequence. In RC these minor cycles are guaranteed to occur in a fixed order. Unlike cyclic systems, however, the synchronous languages only execute needed reactions.

It is not clear whether the synchronous/reactive languages are truly intended for programming general operating systems software. Operating systems drivers, protocol stacks, user interfaces, and real-time process controllers are mentioned as example applications. Halbwachs specifically notes that most system software is based on reactive kernels embedded within more traditional architectures. This split corresponds roughly to the distinction between cooperative and competitive internal concurrency. Although the synchronous languages make internal cooperative concurrency deterministic, traditional means of managing competitive concurrency must be used.

Previous efforts to eliminate operating system nondeterminism, most notably associated with declarative language research, have not been particularly successful. For instance, in describing the design of a functional operating system based on streams, Jones and Sinclair note that *'operating systems are inherently reactive'*, but that *'it is not clear that operating systems can be written without modeling nondeterministic behavior'* [JS89]. They discuss such efforts in the functional language community, including an effort to provide *implicit time determinism*, *timestamps*, and *clock oracles*, and note that *'The effect of this proposal is to place non-determinism* entirely *outside the software..'*. They remark:

> 'In recent years, however, research activity on the functional operating systems front has been rather quiet, possibly because the experiments reported above showed that... it does not always lead to greater elegance and clarity in the detailed coding of programs' [JS89].

Logic programming approaches have fared little better. Concurrent Prolog was expressly designed for systems programming and provided dataflow style synchronization via nondeterministic guarded-commands as its basic control mechanism [Sha86a] [Sha87]. Concurrent Prolog served as the basis for KL1, the Kernel Language used to write the operating system for the Japanese Fifth Generation project [Fur92]. Regarding efficiency of Concurrent Prolog implementation efforts, Shapiro reports:

> ' ... It was a deathblow to the implementability of Concurrent Prolog, at least for the time being, since it showed that implementing Concurrent Prolog efficiently is as hard, and probably harder than, implementing OR-parallel Prolog. As we all know, no one knows how to implement OR-parallel Prolog efficiently, as yet' [Sha86a].

0.5.2 Implementation Concerns

It is not possible to ignore software engineering issues such as performance, size, and maintainability when evaluating system software. There are a number of means by which one can implement synchronous/reactive languages. Esterel, for example, has been implemented both as a compiler producing a finite state machine and as a compiler producing a set of routines called by a reaction dispatch work-loop interacting with the environment and then calling required reactions. A conventional thread-based implementation appears quite possible for languages such as RC, as RC's fundamental `stop` statement can be considered a coroutine call to a coroutine coordinator which evaluates conditions and issues a coroutine resume to the next ready coroutine.

When compiling a language such as Esterel into a single finite state machine, the number of resulting states is potentially exponential in the size of the Esterel program source, as the state machine must correctly support all legal orderings in which reactions can occur. These orderings correspond to all possible input sequences. Observations such as these motivated the Esterel compiler of Edwards, which resembles a traditional compiler [Edw94]. Source code is translated to assembler, and the resulting routines dispatched by a run-time reaction dispatch work-loop. This compiler is reported to have roughly linear compile time, output size, and output execution time.

Edwards attempted to compile a 1000 line Esterel program using both compiler approaches. The generated finite state machine contained over 200 states and was output as a C source file of over 230 Megabytes, which could not be compiled. This same 1000 line Esterel program was compiled by the work-loop compiler and produced 21 thousand lines of assembler and a 128 Kilobyte executable.

A 600 line Esterel program compiled to a state machine of 32 states. This was output as a 19 Megabyte C source file which, somewhat amazingly, compiled into a 12 Megabyte executable. The 600 line program, when compiled with the work-loop compiler, produced 13 thousand lines of assembler code resulting in a 96 Kilobyte executable.

These executables are exceptionally large by conventional system programming standards, especially considering that the 1000 line program consisted of the controller for a 4 button digital stopwatch. While not a trivial application, some device drivers are certainly more complex. Edwards notes that this is a 'large' Esterel application, and also notes that causality errors can be arbitrarily subtle but that it is impractical to have the compiler perform exact causality checking due to excessive compile times.

0.5.3 Programming Implications

None of the synchronous/reactive languages are general purpose. For instance, Esterel lacks data structures and its signals consist only of integer values. The languages presented here all require some form of guard evaluation which requires either evaluation of global state between each instant or redundant code. Overhead, either time or space, can become excessive.

These languages have been designed to provide primarily cooperative concurrency. None except for NPL address the needs of systems which must deal with both cooperative and competitive concurrency. Esterel and RC explicitly preclude internal competitive concurrency because dynamic thread creation introduces a dynamic degree of concurrency that cannot be eliminated by the compiler and thus is not allowed. These languages have no mechanism whereby a single routine can be executed concurrently with itself within the same instant.

Debugging large state machines is a significant problem, especially when the state machine has been automatically generated. Halbwachs notes that *'the correspondence between the source code and the generated code is far from being obvious. The slightest change in the Esterel program can involve a complete modification of the automaton'*. This also illustrates why handcoding state machines for large problems is unreasonable.

Although the compiler can in principle detect indirect causality errors, in large systems it may not be obvious how to fix a causality error. Causality errors complicate program development and impact modularity as the compiler needs to analyze all program source.

Programming RC's micro instants, which require explicit source code coordination, resembles conventional concurrent programming. Micro instants introduce 'invisible gotos' and their attendant programming difficulties. Additionally, Huizing and Gerth note that the semantics of such micro instants *'turned out to be too subtle and non-deterministic to be of practical use'* [HG91].

The problem of distributed mental state is not solved by any of these languages. Understanding the interaction of cooperative code scattered throughout a program presents a serious cognitive challenge to program comprehension [LS86]. Programs written in a guarded command dataflow style can require considerable study before overall program behavior is deduced. Since every routine potentially executes every instant, and since control can be located in the midst of each routine, the programmer potentially must keep the entire program state in mind. For instance, consider the following quote from Halbwachs in describing a 36 line Esterel program:

> 'Initially, the control is stopped at..., lines 3, 12, and 24... line 12 is interrupted and ... stopped by ... line 16... The new global state is... lines 3, 16, and 24... line 3 is interrupted. ... comes back to ... line 3. ... control is stopped at lines 3, 16, 28, and 32' [Hal93].

It is thus difficult to program Esterel or RC without drawing timelines, mapping source statements to the timeline, and mentally executing code fragments. Without significant study, the source is insufficient to understand the program.

0.5.4 Summary

In summary, the advantages of the synchronous/reactive approach with respect to programming system software are:

- Programs become deterministic and deadlocks at run-time impossible. Internal cooperative concurrency and communication overheads are compiled away and inherently handled by one state machine.
- Atomic reactions with essentially basic block granularity, coupled with discrete time steps that change global state in 'snapshot' fashion, simplify concurrent programming.
- Because explicit register-set based context switch is not required, implemented systems can be extremely fast, indeed, potentially optimum with respect to time.
- Since explicit critical sections are eliminated, so are the associated maintenance, debugging, and design problems.

Disadvantages of synchronous/reactive programming with respect to system software are:

- Guarded-command dataflow-style programming is difficult.
- Distributed mental state makes programming-in-the-large difficult. Debugging large state machines is hard.
- There is little provision for competitive internal concurrency. In general, all the problems relating to competitive concurrency remain.
- The compilers produce code that is considerably larger than desirable for real systems, and thus the approach does not scale to large systems.

- If state machines are not produced, the overhead of the guard interpreter can become excessive.

The first 2 disadvantages relate directly to mental programming difficulty. The sample source code illustrates that basic psychological complexity and program comprehension effort appears similar to traditional concurrent programming approaches.

0.6 An Alternative Proposal

Is there a software architecture for conventional systems programming which retains synchronous/reactive advantages but not the disadvantages identified in the previous section? The developers of Esterel have noted the similarity between the synchrony hypothesis and clocked digital circuits in which all 'reactions' take one clock cycle [BB91]. Indeed, it is natural to consider, as an alternative to dataflow-flavored approaches, a software architecture based on a sequential instruction model similar to that found in conventional hardware.

0.6.1 The Soft-Instruction Architecture

A conventional computer architecture advances a program counter each 'instant', thereby executing instruction sequences. The current instruction may alter the value of the program counter, thus transferring control to another location in the instruction stream. The processor keeps a large amount of local state which participates in instruction stream execution.

The software architecture in Figure 0.8 is called a *soft-instruction* architecture by analogy with the discrete atomic instructions implemented in conventional hardware. Programs are developed at 2 levels. A *reactive* level is concerned primarily with concurrency, high-level control flow, and the reactive logic of the program. This level consists of control constructs and soft-instructions. It is programmed as if a custom instruction set, one instruction per required type of reaction, is available. The control constructs can be more elaborate than those in hardware instruction sets and can resemble those of any high-level programming language. The reactive level provides for *programming-in-the-large* and exposes one view of the *deep structure* of the program [DK76].

A *synchronous* lower program level consists of soft-instruction implementations. Soft-instruction implementations perform the *programming-in-the-small* tasks required of the program. Soft-instructions are written in a traditional system programming language such as C. These routines are written in a stylized manner defined by each specific implementation of the soft-instruction architecture. With respect to the soft-instruction program, soft-instructions execute atomically, similar to the manner in which most normal hardware instructions execute atomically with respect to the CPU. As with hardware instruction implementation, soft-instruction duration must be bounded by the implementation.

In conjunction with the design of the two program levels, a data structure defining instruction stream state must be developed. This data structure is called a *context structure* and is an implicit argument processed by all soft-instructions. The actual context structure can be produced automatically by soft-instruction programming tools.

0.6.2 The Soft-Instruction

Figure 0.8 shows a single system component implemented using a soft-instruction architecture. The single system component in Figure 0.8 could be a driver, I/O subsystem, file system, network server, etc.. This component could be either included in a larger system or running stand-alone. If implementing a conventional server within a conventional operating system, the entire system component shown in Figure 0.8 could be implemented internal to a single server process.

The soft-instruction program in Figure 0.8 is partitioned completely into soft-instructions, *S1()*, *S2()*, etc.. Soft-instructions often correspond to basic blocks. All soft-instructions must be implemented so that they never block. Thus, as with hardware I/O instructions, all I/O requests internal to a soft-instruction implementation are asynchronous, with the I/O activation typically followed by a return from the soft-instruction. Again, the size of a given soft-instruction may be bounded by the desired latency of the system, as well as by I/O requirements.

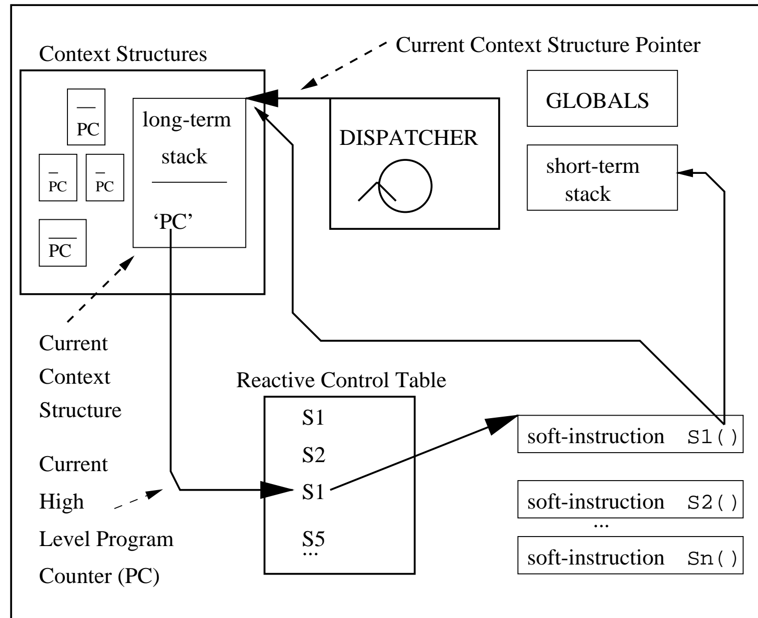

Figure 0.8: Soft-Instruction Software Architecture

0.6.3 The Scheduling and Dispatch Loop

A reactive dispatch loop drives the reactive level. It selects a ready context structure using any operating system scheduling technique. The dispatcher in Figure 0.8 then obtains a *reactive control table* program counter (PC) from within the selected context structure. This high-level logical PC is used in conjunction with the control table to locate the soft-instruction implementation to execute. The dispatcher typically bumps the control table PC within the context structure to point to the next entry in the control table, thus pointing indirectly to the next soft-instruction to be executed on the context structure's behalf after completion of the current soft-instruction. The dispatcher invokes the soft-instruction with the context structure specified, in some fashion, as an argument to the soft-instruction.

The dispatcher can transfer control to a soft-instruction in many ways, including a direct or indirect jump or call. Whatever the mechanism, the dispatch loop invokes the routine implementing the next soft-instruction, and upon completion of the routine control returns to the dispatcher. Successive soft-instruction dispatches may advance the state of different context structures, thus providing featherweight internal concurrency.

The dispatcher is a small threaded-code dispatch loop that can easily be modified by the programmer to implement custom reactive policies. The overhead of this dispatcher can usually be reduced to a few instructions. In threaded-code terminology, such a loop is an *inner interpreter* or *address interpreter*. On some architectures such interpreters can be reduced to a single instruction, and as such they do not have the negative performance connotations of high-level *outer interpreters* [Kog82].

0.6.4 Short-term and Long-term Stacks

Each valid context structure provides an explicit context for an internally competitive concurrent soft-instruction stream. A single program run-time stack provides space for short-term variables with lives lasting the duration of the current soft-instruction execution. This short-term stack usage is identical to that found in any modern language, for instance, it can be the normal C run-time stack.

The soft-instruction model does not have conventional per-thread run-time stacks. Rather, the 'per-thread' context structures contain fixed-size stacks used for both reactive-level control flow and for long-term variables with lives spanning soft-instruction executions. Typical system components, such as

file servers, often require only around 1 Kilobyte for such long term variables and reactive level control. The bounded 'per-thread' stack space in each context structure is thus usually quite small. This can be important when serving hundreds of concurrent requests.

Variables within the scope of a single soft-instruction implementation can thus be found on 2 stacks, one the standard run-time stack containing the soft-instruction's short-term variables, and the other containing long-term variables specified as arguments, perhaps implicitly, to the soft-instruction. Dynamically allocated global resources, such as I/O buffers, are accessed by long-term pointer variables.

0.6.5 The Control Table

The control table represents the reactive program. This data structure can take many forms depending on the dispatcher design. However implemented, the control table determines the execution path of each context structure. A typical implementation consists of a table of addresses pointing to soft-instruction implementations. This is similar to a traditional sequence of machine instructions.

Any number of context structures can be actively traversing the control table, that is, in general the design of the control table is completely independent of the system's required degree of internal concurrency. Soft-instructions that define control constructs within the reactive table evaluate context state and may alter the value of the context structure's program counter. The implementation of such soft-instructions can be included directly within the dispatcher or implemented in the same manner as any other soft-instruction.

Control tables can be generated and analyzed by tools and utilities ranging from simple macros to complete compilers.

0.6.6 Concurrent Programming, I/O, and Context Switch

Soft-instructions are *serially reusable*. The dispatcher executes a single soft-instruction at a time, and each soft-instruction runs to completion, thus implicitly placing all the 'synchronous' code in the program within critical sections.

Each context structure concurrently traversing the table represents a unique soft-instruction stream executing the common reactive control table logic. When an activity of unknown duration, such as an asynchronous I/O request, is activated on behalf of a context structure, the context structure is blocked. This blocks the corresponding soft-instruction stream.

Context structures are blocked by eliminating the structure's eligibility to be selected by the dispatcher for execution. In Figure 0.8, the current context structure could be blocked by pointing the current context structure pointer at another ready context structure after removing the current context structure from the dispatcher's ready queue. Although such blocking and scheduling logic may be implemented in many ways, it is often useful to place an explicit concurrency instruction in the control table, for instance, a `SUSPEND` or `AWAIT_IO_COMPLETION` soft-instruction. Conversely, if a certain class of soft-instructions exist that all initiate I/O requests, each of these instructions can suspend instruction stream execution, as in the usual manner of hardware I/O instructions.

As with any soft-instruction, the soft-instruction that activates an I/O request or other asynchronous activity runs to completion and then returns to the dispatcher. Note that it is the soft-instruction stream described by the context structure, not the soft-instruction itself, that becomes blocked.

A blocked context structure is unblocked by some mechanism associated with completion of the concurrent asynchronous activity. This reactive event logic and associated unblocking mechanism varies widely. For instance, logic to react to event completion can be located directly in the dispatch loop, handled by interrupt routines that interact with the scheduler, or performed by special soft-instruction streams that examine status locations associated with event completion. The specific mechanism will usually depend on the lower layer mechanisms which support external concurrency with respect to the system component. For example, a hardware or software interrupt routine could change the status of the context structure and put the context structure on

```
CONTEXT              *cs;              /* Current Context Structure pointer.  */
typedef void          instruction();  /* A soft-instruction is a void func.  */
typedef instruction  *ins_adr;         /* A pointer to a soft-instruction.    */
ins_adr               si_adr;          /* Current soft-instruction address.   */
cs_adr               *fork_input;      /* 'producer' location in ring-buffer. */
cs_adr               *fork_output;     /* 'consumer' location in ring-buffer. */
…
fork_output = fork_input = fork_ring; /* Initialize fork ring.               */
for(;;) {                             /* Execute soft-instructions forever.  */
   while(  fork_input != fork_output ) {        /* If anything is in the     */
       unblock( *fork_output++ );               /* fork ring, put it on      */
       if( fork_output >= END_RING )            /* the ready list and        */
            fork_output = fork_ring;            /* wrap the fork ring, if    */
   }                                            /* needed.                   */
   cs       = null_cs.cs_flink;                 /* Get the head of the       */
   si_adr   = (ins_adr)(*cs->cs_next++);        /* ready list and fetch      */
   (*si_adr)( cs );                             /* the next soft-instruction */
}                                               /* address and execute it,   */
                                                /* bumping the table pointer.*/
…
```

Figure 0.9: A Simple Dispatcher

a dispatcher ready queue. Conversely, if completion status can be checked with minimal overhead, it may be desirable for the dispatcher loop to directly check all required completion status at the end of every soft-instruction. Whatever the mechanism, when it is determined that the activity invoked by the blocked soft-instruction stream has completed, the corresponding context structure is placed in a state that will again result in the dispatcher executing soft-instructions on its behalf.

The range of possible reactive dispatch mechanisms, and the small amount of code required to implement alternatives, is one of the advantages of the soft-instruction architecture. The dispatcher in many ways resembles a single work-loop merging both the work-loop of a hardware microcode instruction interpreter and the inner work-loop of an operating system kernel. The dispatcher implementation is under complete control of the system programmer and provides a mechanism by which soft-instruction systems can readily be made compatible with particular architectures and environments into which a soft-instruction system is inserted.

Context switching between concurrent soft-instruction streams uses the same mechanism as that used to sequence through the soft-instructions in a single soft-instruction stream. Discounting the overhead of the scheduler, which typically only runs as the result of a significant I/O event, context switching is accomplished by low-overhead operations such as changing a single base register pointing to the current context structure and then performing a table-directed jump or call.

0.6.7 A Simple Dispatcher

Figure 0.9 shows a simple dispatcher implementation. This dispatcher code fragment is designed assuming hardware or software interrupt routines execute upon external asynchronous request completion. The simplest assumption, given this implementation, is that the interrupt routines are real hardware device interrupts. In this case, the 'driver' routines that manage the device are simply special soft-instructions.

A ring buffer called the 'fork ring' provides a producer-consumer style data structure used to communicate between the interrupt level and the soft-instruction level. The fork ring is simply an array of addresses. The fork ring contains addresses of blocked context structures that need to be unblocked because the external request on which they were waiting has completed. In this implementation, it is assumed that pointers are incremented atomically by hardware without any need for interrupt masking,

```
/*-- Called from within a Control soft-instruction. */
block() {   /* Block the current Context Structure  */
CONTEXT   *cs;   /* by removing it from the head    */
                 /* of the dispatch queue.          */
   cs                  = null_cs.cs_flink;
   null_cs.cs_flink    = cs->cs_flink;
   if( cs == null_cs.cs_tail )
       null_cs.cs_tail = NULL_CS:
}


/*-- Called by the dispatcher's work-loop.          */
unblock( CONTEXT  *cs )  {    /* Puts a ready       */
CONTEXT  *old_tail;           /* Context Structure  */
                              /* on the dispatcher's */
   old_tail  = null_cs.cs_tail;  /* ready queue.    */
   old_tail->cs_flink  = cs;
   cs->cs_flink        = NULL_CS;
   null_cs.cs_tail     = cs;
}


   ...


/*-- Called by an interrupt routine to put the      */
/*   address of a Context Structure now ready to    */
/*   proceed into the 'fork' ring.                  */
add_context_to_ring( CONTEXT *cs ) {
   *fork_input++ = cs;
    if( fork_input == END_RING )
        fork_input =  fork_ring;
}
```

Figure 0.10: Dispatcher Support Routines

which is true on many, but not all, hardware architectures. The size of the fork ring in this implementation limits the degree of concurrency, although it can clearly be very large.

If the fork ring contains anything, the dispatch loop 'moves' each context structure from the fork ring to the dispatcher's ready queue by simply chasing the interrupt level's `fork_input` pointer with the `fork_output` pointer and calling `unblock()` for every address it encounters. The `unblock()` routine simply links the context structure onto the end of the ready list. This implementation assumes that a null context structure, `null_cs`, anchors the dispatcher ready list. Context structure element `cs_flink` is the forward link and pointer `cs_tail` in the null context block points to the end of the ready list. The `cs_flink` pointer of the last context structure in the list points back to the null context block.

When all newly ready context structures have been unblocked, this implementation simply selects the first context structure on the ready list as the soft-instruction stream to execute. Element `cs_next` contains this stream's high-level program counter which points to the stream's current location in the control table. The address of the soft-instruction implementation corresponding to the current control table location is obtained and stored in `si_addr`. The stream's program counter, `cs_next`, is incremented to point to the next table location. The soft-instruction implementation is then called with the address of the context structure itself passed as the single argument to the soft-instruction implementation.

Given this dispatch loop, and a typical interrupt routine design, the maximum penalty between activation of any 2 successive soft-instructions is determined by the number of external devices, each of which may correspond to a single interrupt routine execution, plus the time for the dispatch loop to call

unblock() on each corresponding context structure. This is a fixed worst case overhead that will only occur if all devices have outstanding requests which complete in the course of the first soft-instruction execution or the dispatcher's unblock() loop. In most implementations, external interrupts will not be 'rearmed' on individual interrupting devices until specific soft-instructions in the stream handling that device perform I/O completion operations.

Figure 0.10 shows this implementation's unblock() routine and 2 other support routines of interest. The block() routine is called internal to the implementation of a control soft-instruction, such as AWAIT_IO_COMPLETION. It simply removes the current context structure from the head of the ready list. When the soft-instruction containing this call returns to the dispatcher, the next iteration of the dispatcher's loop will no longer be able to advance the soft-instruction stream corresponding to the blocked context structure. When using hardware interrupts, the address of the blocked context structure is often stored within a control block corresponding to the device performing the pending operation. When using software interrupts or signals, arguments can usually be passed to the external service request. These values are then passed back by the external environment to the software interrupt or completion routine associated with the asynchronous request. In either case, the interrupt routine can trivially locate the context structure corresponding to the event completion. The interrupt routine then puts the address of this context structure into the fork ring using routine add_context_to_ring().

0.7 An Example

The soft-instruction program get_string shown in Figures 0.11 and 0.12 is similar to the Esterel GET_STRING program in Figure 0.3. Figure 0.11 contains the high-level 'reactive' code and Figure 0.12 contains the low-level 'synchronous' C code. The equivalent of the Esterel program's hidden build call is included in the soft-instruction source. The soft-instruction get_string()() takes an additional argument specifying maximum buffer size and is also passed an I/O handle so it can be used concurrently by any number of instruction streams, that is, unlike the Esterel program the soft-instruction version supports an arbitrary degree of internal competitive concurrency within the component in which it is located.

In this example, keywords defined by the software architecture implementation are in upper case, while lower-case names denote items in the particular example program written using this architecture. Names pertaining to external items defined by the external system are in lower case with upper case leading characters.

Routine get_string()() is declared REACTIVE. Its arguments and automatic variables will be located in context structure long-term stacks, not the conventional C run-time stack. The soft-instructions shown here, following Esterel, use 2 argument lists, the first for arguments that can be modified by the soft-instruction, often called in-out arguments, and the second for arguments that cannot be modified by the soft-instruction, often called in arguments. Both parameter lists are call-by-reference, providing a window of selective exposure between the reactive and synchronous levels that can be treated as a merged single level.

A REACTIVE routine contains only declarations, soft-instructions, control-flow statements, and concurrency control statements. Limited expressions resulting in boolean values can be used in control-flow conditionals. Except for iteration initialization, assignment is not allowed within a REACTIVE routine.

In this example, soft-instruction init_buffer readies the buffer for I/O, after which a loop is entered in which soft-instruction post_read issues a single-character I/O request to read the next character into the buffer. After each asynchronous read request is issued, the AWAIT_IO_COMPLETION soft-instruction blocks the instruction stream if the I/O request has not yet completed. I/O request completion triggers a software interrupt routine which calls add_context_to_ring(), thus resuming the execution of the soft-instruction stream. Soft-instruction check_read then checks for I/O errors, maintains the buffer, and sets the done flag if needed. Routine get_string()() completes normally when either max_buf characters have been received, a newline encountered, or a timeout occurs.

```
REACTIVE   get_string(  int   done       )
                     (  int   io_handle,
                        char *buf,
                        int   max_buf    )
{
Status_Block  sb;
char         *p;
int           i;

   init_buffer( p, done )( buf );

   for(i=0;i<max_buf;i++) {

       post_read( p )( io_handle, sb );
       AWAIT_IO_COMPLETION;

       check_read( p, done )( sb );
       if( TRUE == done )   break;
   }
} ON_ERROR {
   get_string_io_err()( sb );
}



//------------------------------------------
INSTRUCTION( get_string_io_err )()( Status_Block sb )
{
   if( sb.ret_stat == Ctrl_Break )  RESTART( get_string );
   if( sb.ret_stat == Timeout    )  RETURN;

   ERROR_THROW( sb.ret_stat );
}
```

Figure 0.11: get_string – Reactive Level

The `status` variable in `post_read` is the only variable declared in the entire `get_string()()` program that uses any space on the C run-time stack. Long-term data structure `Status_Block` receives I/O completion status when primitive `Read_Io` completes. The `Status_Block` structure and `Read_Io` function are not part of the soft-instruction architecture but are primitives of the implementation in which `get_string()()` is included. These primitives are typical of an environment that supports asynchronous requests. I/O request completion updates the status block and readies the corresponding context structure. The `Read_Io` call specifies a timeout after which the I/O request completes with a timeout status.

The `ON_ERROR` statement specifies a reactive error handler for `get_string()()`. The `ON_ERROR` block is effectively a reactive level interrupt routine, that is, a trap handler. In Figure 0.11, this interrupt routine executes only 1 soft-instruction, `get_string_io_err()()`. Although in principle the logic included within the implementation of this soft-instruction could be included directly within the reactive level interrupt handler, it is placed in the soft-instruction implementation to illustrate a custom soft-instruction that alters context structure location within the soft-instruction stream. Using the machine instruction analogy, this soft-instruction corresponds to the implementation of a privileged machine instruction, for instance, a return-from-interrupt or trap instruction.

The `get_string_io_err()()` soft-instruction source is included with the source for the `REACTIVE get_string()()` routine because an error han-

```
//--------------------------------------
INSTRUCTION   init_buffer( char   *p,
                           int    done  )
                         ( char   *buf   )
{
   done = FALSE;

   p    = buf;
  *p    = '\0';
}

//--------------------------------------
INSTRUCTION   post_read( char          *p   )
                       ( int           io_handle,
                         Status_Block  sb )
{
int  status;

   status = Read_Io( io_handle, p, 1, Timeout, &sb );
   if( !status ) ERROR_THROW( status );
}

//--------------------------------------
INSTRUCTION   check_read( char          *p,
                          int           done )
                        ( Status_Block  sb   )
{
   if( !sb.ret_stat ) ERROR_THROW( sb.ret_stat );

   if( '\n' == *p ) {

       *p   = '\0';
       done = TRUE;

   } else {

       p++;
       done = FALSE;
   }
}
```

Figure 0.12: get_string – Synchronous Soft-Instruction Level

dler can potentially be invoked at any point in `get_string()()` execution. Placing the source in this location emphasizes the unique relationship of the trap handler and its custom soft-instructions to the code in which it is enabled. The `ERROR_THROW` statement propagates an error to the next highest `ON_ERROR` handler. The `RESTART` statement in Figure 0.11 causes the instruction stream to restart execution of the `get_string()()` function. The `RETURN` results in an immediate return from `get_string()()`. In this case control transfers back to the reactive routine which called `get_string()()`. Reactive routines can call other reactive routines, thus providing modularity within the control table, that is, at the reactive level. No such call is shown here.

Finally, soft-instructions from many instruction streams may be executing in interleaved fashion, thus providing an arbitrary degree of internal competitive concurrency. By design, this concurrency is not visible in the `get_string()()` source.

0.8 Soft-Instruction Advantages

The soft-instruction architecture illustrated in the example implementation has the following advantages:

- **Separation of Concerns and Programming-in-the-Large**. The soft-instruction architecture is based on a separation of concerns between the reactive high-level and the synchronous instruction low-level. Concurrent programming,

overall organization, and gross control flow are treated as high-level programming-in-the-large that reflects the deep structure of the overall program. Low-level issues such as data structure maintenance are programming-in-the-small issues cleanly encapsulated within synchronous soft-instructions. Logical concurrency results from interleaving at the soft-instruction level of granularity, rather than explicit P and V semaphore coding.

A reactive subset of C is used for programming-in-the-large. The reactive level is programmed in familiar sequential procedure-oriented fashion, rather than using a nonprocedural or nonsequential approach, such as results from using guarded commands or path expressions [And79] [AS83]. Programming-in-the-small uses a minor variant of C, an ordinary systems programming language.

- **Single Locus Concurrency**. Concurrency is explicitly visible when reading the source 'from the top down'. Concurrency is visible at one program level, the reactive level, and can be understood in a reading confined to the source of all the reactive routines. There is no 'hidden' special-case concurrency lurking at the bottom of long call chains initiated by arbitrary C routines. Rather, it is clear that interleaving can occur between every soft-instruction.

  Soft-instruction program source is 'structured' with respect to concurrency. Control flow obscured by `goto`s is considered problematic. If so, hidden concurrency in many conventional multithreading implementations is worse because *nothing* about the potential concurrency event is necessarily visible at the point of invocation in the program source and any subroutine call can result in numerous concurrency events.

  There is minimal low-level 'bookkeeping' code in the high-level reactive code. This single locus of concurrency information makes concurrency explicit in the entire architecture of the program, not just an implicit side effect of some system call. Thus, it may be fair to say there is no invisible 'spaghetti concurrency'. To the systems programmer, this means that all code does not need to be read 'bottom-up', searching for 'buried' concurrency constructs.

  Critical sections protecting main memory data structures have been subsumed by soft-instructions. As with monitors, external device or request serialization is typically performed by blocking context structures on a wait queue from which they are later unblocked. For example, `AWAIT_IO_COMPLETION` could simply link the context structure onto the correct device waiting list.

  However, it is not the case that all critical section and semaphore-based coding has been irrevocably banished. The programmer can easily construct any desired concurrent programming soft-instructions, and can thus program at the reactive level with concurrent program constructs tailored to the internal concurrency requirements of the system component. For instance, custom soft-instructions `P()(resource)` and `V()(resource)` can readily be implemented. The effect of a critical section in the control table defined by these soft-instructions would be to cause a sequence of soft-instructions to be treated as atomic with respect to some logical resource. Even in this case, all concurrency remains visible at the reactive control table level. The reactive level in a real system is usually considerably smaller than the synchronous level. In practice, the use of 2 layers seems to significantly reduce both the demand for explicit critical sections and the amount of code that must be studied to understand the concurrency aspects of the program. One finds oneself studying a dozen pages of code instead of hundreds. Quantifying the reduction in explicit concurrent programming and the relative sizes of the reactive and synchronous levels requires additional study.

- **No Concurrent C Programming**. No explicit concurrent programming is possible in the compiled C code, that is, in the synchronous soft-instruction implementations. Potential concurrent programming bugs in the low-level C code are thus eliminated. Deadlock requires mutual exclusion, wait-and-hold, no preemption, and

circular wait. Soft instructions prevent deadlock from occuring since there is no wait-and-hold (strong) and there is preemption (weak) in that any correctly written soft-instruction is required to terminate.

The synchronous C code is more understandable without embedded concurrency primitives. When writing C code for soft-instructions one can concentrate on data structure bookkeeping. Because each soft-instruction executes to completion, globals (such as hardware registers) cannot be updated nondeterministically during the same 'instant', that is, one can think of the soft-instructions as executing atomically at discrete points in time during which 'invisible' events cannot happen.

- **Context-switch**. Context switch is featherweight, with no copying of register sets required. Creation of a concurrent service thread is also very low overhead. Context switch overhead is less than for a system using lightweight concurrency, and significantly less than the heavyweight overhead found in a general purpose operating system. Since soft-instructions run to completion, context does not need to be saved upon featherweight context switch because context is saved 'on-the-fly' in the current context structure.
- **Stack Space**. An entire soft-instruction program, no matter what the degree of internal concurrency, requires only 1 conventional run-time stack. Short term variables all reuse this same short term stack, while long term variables use the bounded size stacks embedded in context structures. If soft-instruction implementations never nest, that is, never call another soft-instruction implementation or recursively call themselves, maximum short-term run-time stack depth is simply the deepest conventional stack usage of any single soft-instruction implementation.
- **Reusability**. Because soft-instructions have arguments, soft-instructions are general routines that can be reused. They are not statically bound to specific transaction or object block elements. Thus, as with a number of threaded environments, application development results in the evolution of a special application-oriented 'vocabulary' at the reactive level.

This soft-instruction architecture provides practical advantages to the practicing systems programmer. The soft-instruction architecture does not preclude being used in conjunction with other conventional techniques and can be used within components of existing systems. Because the reactive dispatcher is small enough to be routinely customized, the architecture leaves the system programmer in control at all levels, unlike approaches which preempt the programmer's design prerogatives [SW92].

The synchronicity model adopted by soft-instructions is weaker than the strong synchronicity model assumed by synchronous/reactive languages such as Esterel, but it is intuitive to systems programmers familiar with hardware instruction sets. Since systems programmers are responsible for 'disguising' the low-level machine hardware, they must be intimately familiar with low-level hardware instruction set details. Pragmatic advantages result from using familiar cognitive models at multiple programming levels.

0.9 Related Work

Some historical and current work of interest is briefly described in this section. Variations of the soft-instruction architecture are perhaps among the oldest architectures used for concurrent real-time programming. The generic features of the abstract architecture, however, do not appear widely appreciated. Soft-instruction techniques have not been organized, analyzed, and presented so that the features and advantages of the generic software architecture are clear. General purpose toolsets are not available supporting concurrent systems programming using soft-instructions.

0.9.1 Historical Work

Systems implemented using techniques similar to those described here have usually suffered from low-level ad-hoc connotations often confounded with low-level assembler implementation concerns. Real-time cyclic executives, for example, have been implemented essentially using soft-instruction techniques.

Each such system has developed its own custom reactive language, table compiler/assembler, and means for the synchronous instructions to interact with the reactive layer.

The advantages of 'traditional' soft-instruction flavored cyclic executives, as used in real-time avionics, have previously been enumerated and contrasted with conventional multithreading techniques using preemptive scheduling of lightweight threads [Mac80] [Gla83] [Sha86b] [BS86]. MacLaren, in discussing one such system, notes that *'the efficiency of a cyclic executive derives from its minimal scheduling property, and from the very small implementation cost'* [Mac80].

Shaw classifies real-time software as either based on concurrent interacting processes, or based on table-driven soft-instruction style approaches, which he terms *slice-based* following that usage in BBN's Pluribus IMP Arpanet communication processor [OCK+75] [Sha86b]. Shaw notes that soft-instruction equivalents have been called *slices*, *chunks*, and *strips*. He contrasts the two real-time software architectures, and calls for research in including time as a first class programming object. Current real-time and concurrent programming research tends to emphasize replacing, rather than modernizing, soft-instruction related approaches.

Baker and Scallon describe a family of systems implemented at Boeing using a soft-instruction architecture [BS86]. They note that its lineage includes the executives of the 1965 Apollo Range Instrumentation Ship and the 1970 Safeguard ballistic missile defense effort. They point out that a soft-instruction architecture provides a high-level virtual machine language, reduces the need for explicit synchronization, and provides the benefits of splitting the system into 2 levels, one intended for high-level programming, and one for low-level programming. They describe a member of this family as follows:

> 'Rex's software architecture is characterized by its view of the executive as an independently programmable machine that executes application procedures written in conventional programming languages as if they were individual instructions of a higher level program.
>
> ... Programming in the large is concerned with producing a program in the machine language of the executive, while programming in the small is concerned with extending its instruction set.
>
> ... Application procedures are coded in a conventional programming language and compiled into machine code of a physical machine. A plan for the management of these procedures' executions to form a system is expressed in a separate system-specification language, and is separately translated into tables (agendas) used by the Rex virtual machine. These tables are in effect a high-level machine code interpreted by the executive' [BS86].

They claim the resulting system provides a virtual machine for the system specification, rather than simply a virtual machine on which the application program executes. Baker and Scallon trace the origins of these systems at least back to the AgPrep system developed by DBA Systems for the Apollo range ship [Joh70]. This system apparently functioned as a special linker that processed tables of entry points and scheduling requirements called *agendas*.

The soft-instruction approach has many similarities to the *threaded-code* architectures used by some Microsoft applications and languages such as Forth and UCSD Pascal [Kog82]. These systems are usually single threaded, that is, have a single context block, and thus concurrent programming is not integral to the basic threading system. Threaded-code is often used as an implementation technique in memory constrained environments. The size of the compiled control table, and thus the resulting executable, depends on the control table data structure design. Well designed control tables can take considerably less space than the equivalent assembler code. Thus, compilers generating threaded-code were originally introduced to implement high-level languages, such as Fortran, on small address-space minicomputers where program size was critical [Bel73] [Bre78]. Besides not directly supporting concurrent programming, most existing threaded-code architectures implement a rather fixed set of low-level primitives, with the bulk of the program occuring at the control table level.

Allworth provides a short description of real-time threaded-code soft-instruction architectures with only a single high-level program counter [All81]. He notes the flexibility and space savings provided by what is now called *token-threaded* code and simply calls the dispatcher an 'interpreter'. Token-threaded code compresses control tables by using indices smaller than the machine address size to access an intermediate table. Allworth notes the analogy with machine hardware instructions:

> 'A compiler translates... a high-level-language program into an equivalent machine code program. ... hardware reads an instruction, interprets the instruction to mean that it must carry out a certain action, then executes that action. A piece of *software* that acts in this way, reading, interpreting and executing a sequence of coded instructions, is called an interpreter.
>
> ... The *action code address table* contains the value of the start location of the program code that implements each possible action... Each action is given an instruction code... The instruction pointer indicates which code is to be interpreted next. On each cycle of the interpreter it is incremented to point to the next code in sequence. Non-sequential jumps within the interpreted code can be implemented by allowing action programs to manipulate the instruction pointer' [All81].

The view of non-concurrent programs as abstract layers of virtual instructions is an old one. For instance, Dijkstra writes:

> 'I want to view the main program as executed by its own, dedicated machine, equipped with the adequate instruction repertoire operating on the adequate variables and sequenced under control of its own instruction counter, in order that my main program would solve my problem if I had such a machine. I want to view it that way, because it stresses the fact that the correctness of the main program can be discussed and established regardless of the availability of this (probably still virtual) machine....
>
> ... this ideal machine will turn out not to exist, so our next task – structurally similar to the original one – is to program the simulation of the 'upper' machine. ... we have to decide upon data structures to provide for the state space of the upper machine; furthermore we have to make a bunch of algorithms, each of them providing an implementation of an instruction assumed for the order code of the upper machine. Finally, the 'lower' machine may have a set of private variables, introduced for its own benefit and completely outside the scope of the upper machine... until finally we have a program that can be executed by our hardware' [DDH72].

Perhaps because Dijkstra is explaining layered architecture and step-wise design in a book on structured programming, he does not propose literally implementing such a multiple-level instruction interpreter, but rather is motivating an abstract model of structured programming languages.

Early operating systems often treated software implemented instructions similar to hardware instructions with respect to sequencing and concurrency. Operating system services were considered simply special assembler instructions implemented in software instead of hardware, that is, instructions in the program instruction stream to be executed interpretively by the operating system. Purser and Jennings summarize this viewpoint:

> 'The basic instruction code of the computer is frequently supplemented with virtual instructions (VIs). These are subroutines which are available for performing certain critical operations: they are coded as such for the usual reason of writing a subroutine (saving repetition of code) but also, more importantly, to incorporate them into the executive. VIs perform operations on executive and other data on behalf of processes, with the result that processes do not have to operate on such data (including their own PCBs) directly. Many VIs can be constructed...
>
> ... In general, therefore, a VI is not entered in parallel and hence is non-reentrant'

[PJ75].

Mixed hardware/software instruction stream interpretation, with custom user written routines extending the machine instruction set, is an old idea [Bra82]. Such approaches are often seen today on RISC architectures, for instance PALcode on the DEC Alphas provides a mechanism for operating system programmers to develop privileged 'hardware' instructions specific to the support of their system. Similarly, in many CISC processor families, software instruction execution has been used to provide compatibility between low-end and high-end members of a processor family.

0.9.2 Current Work

Huizing and Gerth have proposed a 2 level semantics with separate modularity and causality levels that is suggestive of the 2 level soft-instruction architecture. In their semantics 'global' time is more abstract than 'local' time. This proposal is of special interest as it is intended to overcome problems in the semantics of Esterel [HG91].

Discrete-event simulations often use architectures similar to soft-instructions but without direct realtime application. These environments are usually not intended for development of large system programs. An example of such a programming environment is *Reactive-C*, not to be confused with the RC described earlier in this paper [Su90].

There have been efforts in the debugging community to analyze sequential programs and determine information similar to that needed to automatically generate soft-instructions [Wei82]. These *program slicing* algorithms are currently impractical for real programs [GL91] [Hu93].

In parallel and distributed programming research, *coordination frameworks* support subroutine-level parallelism [Pan93]. Examples of such systems are Parallel Virtual Machine (PVM), STRAND88, Program Composition Notation (PCN), and Express. These systems have many soft-instruction characteristics:

> 'The programmer codes subroutines in some standard programming language... the tool automatically generates a source code 'wrapper' for each subroutine, as well as a driver... The coordination language is not really 'compiled'; rather, it is transformed into calls to runtime libraries in a preprocessing step before compilation' [Pan93].

These systems are intended for writing distributed applications on top of existing operating systems. The level of granularity of these systems is often not appropriate for systems programming.

A large amount of work has been performed on multithreaded runtimes and parallel programming environments for massively parallel machines. A system of particular interest is Cilk, a multithreading parallel programming C run-time influenced by dataflow research [BJK+95]. It is used to program parallel MIMD machines for a particular class of computationally intensive distributed computations. Cilk provides high-performance cooperative concurrency in a distributed environment.

As with soft-instruction architectures, individual Cilk routines are atomic units of computation which always run to completion. Cilk calls such routines *threads*. All Cilk threads participating in a given computation on a single machine share the same stack locations and return to a common scheduling and dispatching loop. No lightweight context switching is required as context is explicitly saved 'on-the-fly' in data structures called *closures*. A unique closure corresponds to each thread. Closures resemble custom context structures created dynamically before each thread's activation and deleted upon thread completion.

A Cilk thread's argument list effectively can specify guards corresponding to locations within its closure. When all arguments become valid, the guard fires, closure arguments are copied to automatic variables, and the thread executes, processing the 'arguments'. Cilk threads cannot return values to their parents. Rather, the programmer uses explicit calls to place data into other closures using descriptors indicating specific closure locations. Cilk calls such descriptors *continuations*. The parent thread passes its children all needed continuations.

This programming model leads to an *explicit continuation* style of programming in which a parent thread never waits for a child, but rather explicitly spawns a successor thread which blocks until

all its arguments are provided by the children of the first thread. Programming in this style raises programming-in-the-large issues and can result in a 'chained' style of programming with a sense of high-level control that can be characterized by 'next do' and 'after goto' control logic.

The applicability of this model is described by its implementors as follows:

> 'Although Cilk offers performance guarantees, its current capabilities are limited, and programmers find its explicit continuation passing style to be onerous. Cilk is good at expressing and executing dynamic, synchronous, tree-like, MIMD computations, but it is not yet ideal for more traditional parallel applications that can be programmed effectively in, for example, a message-passing data-parallel, or single-threaded shared-memory style' [BJK+95].

Cilk is of interest as its developers note and study the performance advantages resulting from various soft-instruction related techniques, for instance, the use of atomic routines, a single scheduler/dispatch loop, and featherweight context switch provided by a thread model based on a linguistic abstraction. Through studied in a different context, many of the motivations for such techniques are equally applicable to concurrent systems programming.

Also of direct relevance are the observations noting that, although well suited for the class of computations for which it is intended, the programming style is questionable for general purpose programming-in-the-large. Such concerns are similar to those motivating the explicit reactive level of the soft-instruction architecture.

0.10 Conclusions

The *soft-instruction* software architecture has been described. This architecture supports implementation of general purpose concurrent system software. Soft-instructions provide many of the benefits of the synchronous/reactive languages that assume the strong synchrony hypothesis, while providing program source that is easier to read, comprehend, and maintain. In addition, the soft-instruction architecture can coexist with existing systems, can flexibly adapt to many environments, and has many similarities to the hardware instruction set architectures with which systems programmers are very familiar.

References

[ACJ+95] Mark A. Ardis, John A. Chaves, Lalita Jategaonkar Jagadeesan, Peter Mataga, Carlos Puchol, Mark G. Staskauskas, and James Von Olnhausen. A framework for evaluating specification methods for reactive systems. In *Proceedings of the 17th International Conference on Software Engineering*, pages 159–168, April 1995.

[ALBL91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, April 1991.

[All81] Steve T. Allworth. *Introduction to Real-Time Software Design*. Springer-Verlag, 1981.

[And79] S. Andler. Predicate path expressions. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 226–236, 1979.

[AS83] G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[Atw76] J. William Atwood. Concurrency in operating systems. *IEEE Computer*, 9(10):18–26, October 1976.

[BB91] Albert Benveniste and Gerard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[BC84] Gerard Berry and Laurent Cosserat. The Esterel synchronous programming language and its mathematical semantics. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science, 197, Seminar on Concurrency*, pages 389–448. Springer–Verlag, July 1984.

[BdS91] Frederic Boussinot and Robert de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, June 1973.

[BG92] Gerard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[BGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the Signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[BJ87] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 123–138, November 1987.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Priciples and Practice of Parallel Programming*, pages 207–216, July 1995.

[Bou91] Frederic Boussinot. Reactive C: An extension of C to program reactive systems. *Software – Practice and Experience*, 21(4):401–428, April 1991.

[Bou92] Frederic Boussinot. RC reference manual. Technical Report CMA 92–16, Ecole Nationale Superieure des Mines, 1992.

[Bra82] James Brakefield. Just what is an opcode? or a universal computer design. *Computer Architecture News*, 10(4):31–34, June 1982.

[Bre78] Ronald F. Brender. Turning cousins into sisters: An example of software smoothing of hardware differences. In C. Gordon Bell, J. Craig Mudge, and John E. McNamara, editors, *Computer Engineering: A DEC View of Hardware Systems Design*, pages 365–378. Digital Press, 1978.

[BS86] Theodore P. Baker and Gregory M. Scallon. An architecture for real-time software systems. *IEEE Software*, 2(5):50–58, May 1986.

[Bud95] Reinhard Budde. Esterel. In *Formal Development of Reactive Systems: Case Study Production Cell*, pages 75–100, 1995.

[DDH72] O. J. Dahl, Edsger W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, 1972.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[DK76] Frank DeRemer and Hans H. Kron. Programming in the large versus programming in the small. *IEEE Transactions on Software Engineering*, 2(2):80–86, June 1976.

[Edw94] Stephen Edwards. An Esterel compiler for a synchronous/reactive development system. Technical Report ERL 94–43, University of California, Berkeley, June 1994.

[FP88] Stuart R. Faulk and David L. Parnas. On synchronization in hard-real-time systems. *Communications of the ACM*, 31(3):274–287, March 1988.

[Fur92] Koichi Furukawa. Logic programming as the integrator of the fifth generation computer systems project. *Communications of the ACM*, 35(3):82–92, March 1992.

[Gar95] David Garlan. First international workshop on architectures for software systems: Workshop summary. *ACM Software Engineering Notes*, 20(3):84–89, July 1995.

[GL91] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–760, August 1991.

[Gla83] Robert L. Glass. *Real-Time Software*. Prentice-Hall, 1983.

[GP95] David Garlan and Dewayne E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, April 1995.

[GTP95] David Garlan, Walter Tichy, and Frances Paulisch. Summary of the Dagstuhl workshop on software architecture. *ACM Software Engineering Notes*, 20(3):63–83, July 1995.

[Hal77] Horst Halling. Steps towards the implementation of a parallel code executor. In *Proceedings of the IFAC/IFIP Workshop*, pages 55–63, June 1977.

[Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[Her90] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, March 1990.

[Her91] Maurice Herlihy. Wait-free synchronization. *Communications of the ACM*, 11(1):124–149, January 1991.

[HG91] C. Huizing and R. Gerth. Semantics of reactive systems in abstract time. In *Real-Time: Theory in Practice*, pages 291–314, June 1991.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. *Logics and Models of Concurrent Systems, NATO ASI Series*, 13:477–498, 1985.

[Hu93] Shibin Hu. *Automatic Generation of Language-Based Program Slicer*. PhD thesis, Wayne State University, 1993.

[Joh70] Frederick C. Johnson. Real-time data processing and orbit determination on the Apollo tracking ships, NASA-CR-111576. In *AGARD Conference Proceedings No. 68 on the Application of Digital Computers to Guidance and Control, AGARD-CP68-70*, pages 22–32. Harford House, June 1970.

[JS89] S. B. Jones and A. F. Sinclair. Functional programming and operating systems. *The Computer Journal*, 32(2):162–174, 1989.

[KKW94] Andrew J. Kozubal, Debora M. Kerstiens, and Rozelle M. Wright. Experience with the State Notation Language and run-time sequencer. *Nuclear Instruments and Methods in Physics Research A*, 352(1,2):411–414, 1994.

[Kog82] Peter M. Kogge. An architectural trail to threaded-code systems. *IEEE Computer*, 15(3):22–32, March 1982.

[Kop91] H. Kopetz. Event-triggered versus time-triggered real-time systems. In A. Karshmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, pages 87–101. Springer–Verlag, July 1991.

[Koz93] Andy Kozubal. *State Notation Language and Run-time Sequencer Users Guide*. Los Alamos National Laboratory, September 1993.

[KVK91] Ilkka Kuuluvainen, Mika Vanttinen, and Perttu Koskinen. The action-state diagram: A compact finite state machine representation for user interfaces and small embedded reactive systems. *IEEE Transactions on Consumer Electronics*, 37(3):651–658, August 1991.

[Lam94] Leslie Lamport. Processes are in the eye of the beholder. Technical Report 132, Digital Systems Research Center, December 1994.

[Laz91] Edward D. Lazowska. Operating system support for high-performance architectures. In A. Karshmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, pages 40–43. Springer–Verlag, July 1991.

[LL95] Claus Lewerentz and Thomas Linder. *Formal Development of Reactive Systems: Case Study Production Cell*. Springer-Verlag, 1995.

[LS86] Stanley Letovsky and Elliot Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41–49, May 1986.

[Mac80] Lee MacLaren. Evolving toward ADA in real-time systems. In *Proceedings of the ACM SIGPLAN Symposium on the ADA Language*, Boston, December 1980.

[Mar90] R. Jordan Martin. *Transaction Processing Facility: A Guide for Application Programmers*. Yourdon Press, 1990.

[MB91] J.C. Mogul and A. Borg. The effect of context switches on cache performance. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, 1991.

[Mil93] Robin Milner. Turing award lecture: Elements of interaction. *Communications of the ACM*, 36(1):78–89, January 1993.

[MK93] K. R. Mayes and J. A. Keane. Levels of atomic action in the Flagship parallel system. *Concurrency: Practice and Experience*, 5(3):193–212, 1993.

[MS90] Gary J. Murakami and Ravi Sethi. Terminal call processing in Esterel. *ATT*, January 1990.

[MW91] Keith Marzullo and Mark D. Wood. Tools for constructing distributed reactive systems. Technical Report TR 91-1193, Cornell, February 1991.

[Neh91] Jurgen Nehmer. The immortality of operating systems, or is research in operating systems still justified? In A. Karshmer and J. Nehmer, editors, *Operating*

Systems of the 90s and Beyond, pages 77–83. Springer–Verlag, July 1991.

[OCK+75] S.M. Ornstein, W.R. Crowther, M.F. Kraley, R.D. Bressler, A. Michel, and F.E. Hart. Pluribus – a reliable multi-processor. In *Procedings of the AFIPS 1975 Conference*, pages 551–559, 1975.

[Ous89] John Ousterhout. Why aren't operating systems getting faster as fast as hardware? Technical Report TN-11, Digital Western Research Laboratory, October 1989.

[Pan93] Cherri M. Pancake. Multithreaded languages for scientific and technical computing. *Proceedings of the IEEE*, 81(2):288–304, February 1993.

[PJ75] W. F. C. Purser and D. M. Jennings. The design of a real-time operating system for a minicomputer. *Software – Practice and Experience*, 5:147—167, 1975.

[Rep95] John H. Reppy. First-class synchronous operations. In *Proceedings of the First International Workshop on Theory and Practice of Parallel Programming*, pages 235–252, February 1995.

[Sch86] Fred B. Schneider. Concepts for concurrent programming. In *Current Trends in Concurrency*, pages 669–236. Springer–Verlag, 1986.

[Sha86a] Ehud Shapiro. Concurrent Prolog: A progress report. *IEEE Computer*, 19(8):44–58, August 1986.

[Sha86b] Alan C. Shaw. Software clocks, concurrent programming, and slice-based scheduling. In *Proceedings of the 1986 Real-Time Systems Symposium*, pages 14–18, December 1986.

[Sha87] Ehud Shapiro. Systems programming in Concurrent Prolog. In *Concurrent Prolog*, volume 2, pages 6–27, 1987.

[Su90] Wen-King Su. *Reactive-Process Programming and Distributed Discrete-Event Simulation*. PhD thesis, California Institute of Technology, October 1990.

[SW92] Mary Shaw and William A. Wulf. Tyrannical languages still preempt system design. In *Proceedings 1992 International Conference on Computer Languages*, pages 200–211. IEEE Computer Society Press, April 1992.

[Wat90] David A. Watt. *Programming Language Concepts and Paradigms*. Prentice Hall, 1990.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, pages 446–452, July 1982.