Web-Scripting Languages for Free *

N. Raja and R.K. Shyamasundar
Tata Institute of Fundamental Research
School of Technology and Computer Science
Mumbai 400 005, India
{raja, shyam}@tifr.res.in

Abstract

Web-scripting languages are languages for programming the World-Wide Web. Cardelli and Davies [6] have described programming primitives which would be useful in web-scripting languages. Their model of the Web, relies predominantly on two observables, namely the notions of failure and the rate of communication. We demonstrate that the well developed discipline of reactive programming languages, which unify asynchrony and perfect synchrony, are well suited for the development of web-scripting languages. In particular we show that the paradigm of ESTEREL *[4] and Communicating Reactive Processes [5] provides ready-made features and constructs for easily prototyping web-scripting languages.*

1. Introduction

Web-scripting languages are languages for programming the World-Wide Web. One of the most widespread, but manually carried out activities on the Web, is the act of browsing the Web for resources. Programs written in web-scripting languages are meant to simulate the process of Web browsing as would normally be carried out by a human being.

Cardelli and Davies [6] have described programming constructs which would be useful in web-scripting languages. The constructs proposed by them are based on certain underlying assumptions about the basic model of computation of the Web. In particular their model of the Web relies predominantly on two observable properties, namely the notions of failure and the rate of communication. Programming constructs for web-scripting make rather specialized demands on the basic error-handling primitives, and also on the concurrency primitives of the underlying language. Common languages are ill suited for such situations.

In contrast to common languages, the well developed discipline of reactive programming languages provides a convenient launching pad for web-scripting languages. The reason for this is the fact that these languages have advanced error-handling primitives, and also sophisticated concurrency primitives. They have orthogonal features such as preemption, concurrent execution, and exception handling. A specially noteworthy feature of such languages is the way they unify the notions of asynchronous and synchronous interaction. Another distinct advantage is that a precise semantics has been formulated for these languages, taking into consideration aspects of reactivity, priority, and other novel mechanisms. Such languages are ideally suited for the development of web-scripting languages. In this paper, we demonstrate that the paradigm of ESTEREL [4] and Communicating Reactive Processes (CRP) [5] provides ready-made features and constructs for easily prototyping web-scripting languages. In particular, we show that the various service combinators for Web computing [6] may be easily realized [8] in ESTEREL and CRP.

The rest of this paper is organized as follows: Section 2 introduces the paradigm of ESTEREL and CRP; Section 3 demonstrates that service combinators for Web computing can be easily programmed in CRP; Section 4 discusses the implementation of the service combinators; Section 5 outlines further work; and Section 6 concludes the paper.

2. Communicating Reactive Processes

Communicating Reactive Processes (CRP) proposed in [5] unifies the capabilities of asynchronous and synchronous concurrent programming languages.

A CRP program consists of a network $M_1//M_2\cdots//M_n$ of ESTEREL [4] reactive programs or *nodes*, each having its own input/output reactive signals and its own notion of an *instant*. The network is asynchronous, and each node M_i is locally reactively driving

*Proc. International Conference on Software Engineering Applied to Networking and Parallel/ Distributed Computing (SNPD'00), Reims, France (2000) pages 289-296.

a part of a complex process which is handled globally by the network. Asynchronous communication between nodes is achieved through the `exec` primitive and `channel` declarations.

2.1. Basic ESTEREL

The execution of an ESTEREL program associates a sequence of output events with a sequence of input events. The program repeatedly receives an input event E_i from its environment and reacts by building an output event E_i'. The events E_i and E_i' are synchronous in the sense that the external observer observes the input/output event as a single event $E_i \cup E_i'$. The production of an output event from an input event is called a *reaction*. The flow of time is defined by the sequence of reactions and hence, *reaction* is also termed an *instant*. Note that at any instant there is at least one input signal. A signal is present at an instant if and only if it is either received as input from the environment or it is emitted by the program itself at that instant. At each instant, each input or local signal is consistently seen as present or absent by all statements – thus ensuring determinism. By default, signals are absent. The notion of an instant leads to consistent definitions of temporal expressions.

In order to provide the ability to quantify the progress of computation as the system evolves, we assume that every input event or instant consists of a special clock signal of type *real* referred to as `tick`.

The list of ESTEREL kernel statements is given below. (We have described and used the old syntax of ESTEREL in this paper. Though our current implementation uses the latest syntax, the ESTEREL compilers still provide backward compatibility with the old syntax):

```
nothing
halt
emit S
stat1; stat2
loop stat end
present S then stat1 else stat2 end
do stat watching S Timeout Alarm end
stat1 || stat2
trap T in stat end
exit T
signal S in stat end
```

The kernel statements are imperative in nature, and most of them are classical in appearance. Instantaneous control transmission appears everywhere. The `nothing` statement is purely transparent: it terminates immediately when started. An `emit S` statement is instantaneous: it broadcasts `S` and terminates right away, making the emission of `S` transient. In `emit S1; emit S2`, the signals `S1` and `S2` are emitted simultaneously. In a signal-presence test such as `present S ...`, the presence of `S` is tested for right away and the `then` or `else` branch is immediately started accordingly. In a `loop` *`stat`* `end` statement, the body *stat* starts immediately when the loop statement starts, and whenever *stat* terminates it is instantaneously restarted afresh; to avoid infinite instantaneous looping, it is required that the body of a loop should not terminate instantaneously when started.

The `do-watching` and `trap-exit` statements deal with behavior preemption – the most important feature of ESTEREL. In the watchdog statement `do` *`stat`* `watching` *`S`* `Timeout` *`Alarm`* `end`, the statement *stat* is executed normally up to proper termination, or up to future occurrence of signal *S* called the guard. If *stat* terminates strictly before *S* occurs, so does the whole `watching` statement; then the guard has no action; otherwise, the occurrence of *S* provokes immediate preemption of the body *stat* leading to the execution of *Alarm* statement, and immediate termination of the whole watching statement. Note that nesting watching statements provides for priorities. For details on `trap` and other derived constructs, the reader is referred to [3, 4].

2.2. Asynchronous tasks

In a very general way, asynchronous tasks are those tasks which do take time; that is, the time between initiation and completion is observable. In the terminology of ESTEREL this can be interpreted to mean that there will be at least one instant between initiation and completion. The `exec` primitive provides the interface between ESTEREL and asynchronous tasks.

Task Declaration

An asynchronous task is declared as follows:

```
task task_id (fparlst)
return signal_nm (type);
```

where

- `task_id` is the name of the task;
- `fparlst` gives the list of *formal* parameters (reference or value);
- the signal returned by the task is given by `signal_nm` with its type after the keyword `return`; it is possible to have multiple return signals.

Task Instantiation

Instantiation of the task is done through the primitive `exec`. For example, the above task can be instantiated from an ESTEREL program as follows:

```
exec task_id (aparlst);
```

where

- `task_id` is the name of the task;
- `aparlst` gives the list of *actual* parameters (variables/expressions corresponding to reference/value parameters);
- There is no explicit need to specify the `return` signals as it is the same as in the task declaration.

For example, a typical task declaration appears as

```
task ROBOT_move (ip, fp)
return complete;
```

and the call appears as

```
exec ROBOT_move (x,y);
```

The execution of the above statement in some process starts task `ROBOT_move` and awaits for the return signal `complete` for it to proceed further. Of course, the number and type of arguments, and the return signal type should match the task declared. In other words, `exec` requests the environment to start the task and then waits for the return signal (which also indicates the termination of the task).

Since there can be several occurrences of `exec T` in a module for the same task `T`, several simultaneously active tasks having the same name can coexist. To avoid confusion among them one can assign an explicit label to each `exec` statement; a labeled `exec` statement is of the form `exec L:T`. The label name must be distinct from all other labels and input signal names. An implicit distinct label is given to `exec` statements.

The primitive `exec` provides an interface between ESTEREL and the asynchronous environment that can be seen by the following interpretation. Given an `exec` statement labeled L, the asynchronous task execution is controlled from ESTEREL by three implicit signals `sL` (output signal), `L` (input signal), and `kL` (output signal) corresponding to starting the task, completion, and killing the execution of the task respectively. The output *start signal* `sL` is sent to the environment when the `exec` statement starts. It requests the start of an asynchronous incarnation of the task. The input *return signal* `L`, is sent by the environment when the task incarnation is terminated; it provokes instantaneous termination of the `exec` statement. The output *kill signal* `kL` is emitted by ESTEREL if the `exec` statement is preempted before termination, either by an enclosing watching statement or by concurrent exit from an enclosing trap. The return signals corresponding to the `exec` label can be used for declaring incompatibility with other input signals (this becomes handy in declaring channels).

2.3. Interpretation of a global clock in terms of `exec`

Consider a task CLOCK which accesses a global clock and sends the alarm signals at the times requested. Declaration of the task CLOCK in CRP takes the form:

```
task CLOCK(d)
return alarm (real);
```

Now, an alarm after `l` units can be instantiated by

```
exec CLOCK(l);
```

Since the `exec` statement is not instantaneous, it is necessary that `l` > 0. The semantics of `exec` permits the use of multiple instantiations on the same and different nodes. This feature makes it possible to use CLOCK as a global clock for different components. That is, different components can instantiate CLOCK to give them alarm at appropriate times without any interference.

2.4. Timed statements

The primitive timed statements of CRP are given below where `tick` denotes the clock signal.

a. *await* with ticks:
`await tick(d)`: This statement delays the next reaction till the instant `t+d` is crossed where `t` is the value of CLOCK at the current instant. In other words, it corresponds to a delay of `d`. `await tick(0)` is not valid as it instantaneously terminates. In ESTEREL `await tick` corresponds to awaiting for the next input signal or the instant.

b. *watching with* ticks:
`do` *stat* `watching tick(d)`: This statement gives a *time limit* to the execution of its body *stat*. Let us assume `d` > 0. In this case, the body starts as soon as the `watching` statement starts. If the body terminates or exits a trap strictly before a timeout of `d` from the current instant, so does the `watching` statement; otherwise the `watching` statement terminates as soon as the value of `tick` signal reaches `t+d`. As the `watching` statement is active at the current instant, `d` $= 0$ corresponds to the instantaneous termination of the body without being executed.

Restrictions: As `tick` is always present, the following restrictions are placed:

1. It is not permitted to use `tick` in expressions of `await immediate` and `watch immediate` statement constructs.
2. The expressions in the `present` statement constructs are not allowed to reference clocks explicitly.

2.5. Using timed statements

1. *Specifying Time Bounds*

   (a) `await tick(l)` specifies a delay of `l`.

   (b) Specifying timeout (maximum duration) of `u` can be done by:
   ```
   do S watching tick(u)
   ```

   (c) Enforcing a time interval of $[l, u]$ for producing a *good_part* in the usual producer-consumer problem is given below:
   ```
   do
     [exec good_part || await tick(l)]
   watching tick(u)
   ```

2. Actual time for the transition/task with a timeout of `d` is specified below:
   ```
   x:=0;
   do
    exec T (* Let us assume that rT *)
            (* is the return signal *)
   watching tick(d)
   timeout abort (* aborted due to *)
                 (*  timeout *)
   end
   present rT then x := ?tick
   ```
   In the above program, `x` would have the actual value of time taken on proper termination; if aborted due to timeout then `x` will have value zero and also signal `abort` will be emitted.

3. Specifying time for multiple rendezvous and interrupt servicing can be done in the same manner.

2.6. Illustrative examples

In this subsection, we provide a couple of illustrative examples [7] of programming in CRP.

As a first example, consider the specification of an air-traffic control system which should provide final clearance for a pilot to land within 60 seconds after clearance is requested. Otherwise, the pilot will abort the landing procedure. Assuming that there is no global clock, the program to achieve the above activity is given below.

```
emit req_clearance;
do
trap T in
 [exec CLOCK(60) || await get_clearance;
 exit T]
end trap
watching time_up
timeout abort_landing
end
```

where the asynchronous task `CLOCK` is declared as follows:

```
task CLOCK(l) return time_up;
```

with the interpretation that the return-signal `time_up` will be sent from the asynchronous medium after l units of time.

Assuming that there is a global clock which sends `tick` signals at appropriate settings, the program takes the form:

```
emit req_clearance;
do
  await get_clearance
watching  tick(60)
timeout abort_landing
end
```

As a second example, consider a telephone network in which switches periodically monitor other switches to detect node failures. Each switch sends an *I am alive* message to a subset of switches in the network. This subset is referred to as *cohorts* of a switch. If a switch does not receive an *I am alive* message once every period of duration p seconds, from each of its cohorts, it suspects that the cohort may be down and initiates fault detection and recovery. If for any reason a switch is shutdown, a `SHUTDOWN` message is sent to all its cohorts. Each switch creates processes, one per cohort, to monitor the cohorts. These processes execute a code segment of the form:

```
do
  await p;
  trap T in
      loop
        do
          exec cohort(id);
         watching p
         timeout  exit T;
        end
      end %loop
  end %trap
watching SHUTDOWN
```

where emitting `p` corresponds to the cycle given by,

```
every p_secs do
   emit p
end
```

The asynchronous task `cohort` is given by

```
task cohort (id) return ‘‘I AM ALIVE’’;
```

whose activity is to send the return signals periodically.

Service	Combinator
Basic Service	$url(String)$
Gateways	$index(String, String_1)$
	$gateway\ get(String,$ $Id_1 = String_1, \ldots$ $Id_n = String_n)$
	$gateway\ post(String,$ $Id_1 = String_1, \ldots$ $Id_n = String_n)$
Sequential	$S_1\ ?\ S_2$
Concurrent	$S_1 \mid S_2$
Time Limit	$timeout(t,\ S)$
Rate Limit	$limit(t, r, S)$
Repetition	$repeat(S)$
Non Termination	$stall$
Failure	$fail$

Figure 1. Service combinators for the Web

3. Service combinators for Web computing

In this section, we demonstrate that the well developed discipline of reactive programming languages provides a convenient launching pad for web-scripting languages. In particular, we show that the paradigm of ESTEREL [4] and Communicating Reactive Processes [5] provides ready-made features and constructs for easily prototyping web-scripting languages.

One of the most predominant activities on the Web, which is carried out manually, is the act of browsing the Web for resources. Cardelli and Davies [6] have proposed a number of primitives which would be useful in developing programs for automating the act of browsing the Web. Figure 1 contains a complete list of these primitives. They call these primitives as *services* and *service combinators*, and define them as follows.

Definition 3.1 *A service is an HTTP information provider wrapped in error-detection and handling code.*

Definition 3.2 *A service combinator is an operator for composing services, both in terms of their information output and of their error output; and possibly involving concurrency. The error recovery policy and concurrency are thus modularly embedded inside each service.*

In the following subsections, we demonstrate that the service combinators for Web computing, may be easily translated to ESTEREL and CRP.

3.1. Basic service

Service: *url(String)*
Specification: The service *url(String)* fetches the resource associated with the URL indicated by the string. The service fails if the fetch fails, and the rate of the service while it is running is the rate at which the data for the resource is being received, measured in kilobytes per second.
Translation:

```
% Comments begin with a percentage symbol
% Task Invocation
exec url(String);

% Where the Task Declaration is as given below
task url(String)
return {success(contents), failure};
```

Interpretation: The translation consists of two steps. There is a declaration of `task url(String)` which fetches the content of the URL `String`. This `task` may `return` two different kinds of signals, namely `success(contents)` or `failure`. The invocation of the declared task is achieved using the `exec url(String)` command.

3.2. Gateways

Services: $index(String, String_1)$
$gateway\ \ get(String, Id_1 = String_1, \ldots Id_n = String_n)$
$gateway\ \ post(String, Id_1 = String_1, \ldots Id_n = String_n)$
Specification: Each of these services is similar to the service $url(String)$, except that the URL String should be associated with a CGI gateway having the corresponding type ($index$, get, or $post$). The arguments are passed to the gateway according to the protocol for this gateway type.
Translation:

```
% Task Invocations
exec index(String, String1);
exec gateway get(String, Id1=String1,... Idn=Stringn);
exec gateway post(String, Id1=String1,... Idn=Stringn);

% Task Declarations follow
task index(String, String1)
    return {success(contents), failure};
task gateway get(String, Id1=String1... Idn=Stringn)
    return {success(contents), failure};
task gateway post(String, Id1=String1... Idn=Stringn)
    return {success(contents), failure};
```

Interpretation: Similar to the translation of the service `url(String)`, there is a declaration part followed by the invocation part.

3.3. Sequential execution

Service: $S_1 \ ? \ S_2$
Specification: The combinator "?" allows a secondary service to be consulted in the case that the primary service fails for some reason. Thus, the service $S_1 \ ? \ S_2$ acts like the service S_1, except that if S_1 fails then it acts like the service S_2.
Translation:

```
do
  exec S1
watching S1.failure
timeout S2
end
% where S1.failure is the failure signal
% returned by service S1
```

Interpretation: The translation begins with invoking service S1. If S1 succeeds then the rest of the statements are not executed. However, if S1 fails (corresponds to the emission of S1.failure) then service S2 is invoked.

3.4. Concurrent execution

Service: $S_1 \mid S_2$
Specification: The "|" combinator allows two services to be executed concurrently. The service $S_1 \mid S_2$ starts both services S_1 and S_2 at the same time, and returns the result of whichever succeeds first. If both S_1 and S_2 fail, then the combined service also fails. The rate of the combined service is always the maximum of the rates of S_1 and S_2.
Translation:

```
do
  exec S1
watching (S2.success and not S1.success)
% where S2.success is the success signal
% returned by service S2, and
% not S1.success denotes that the
% signal S1.success is not currently present

||

do
  exec S2
watching (S1.success)
% where S1.success is the success signal
% returned by service S1
```

Interpretation: Both S1 and S2 are begun concurrently, and if at most one of them succeeds then the result confirms to the specification. However, on the rare occasions when both S1 and S2 succeed at the same instant, S1 is accorded higher priority and is chosen as the resultant service.

3.5. Time limit

Service: $timeout(t,\ S)$
Specification: The $timeout$ combinator allows a time limit to be placed on a service. The service $timeout(t,\ S)$ acts like S except that it fails after t seconds if S has not completed within that time.
Translation:

```
do
  exec S
watching tick(t)
```

Interpretation: The `tick(t)` denotes the passage of time duration `t` on the Clock. The value of the time duration t has to be greater than zero.

3.6. Rate limit

Service: $limit(t, r, S)$
Specification: The $limit$ combinator provides a way to force a service to fail if the rate ever drops below a certain limit r. A start-up time of t seconds is allowed, since generally it takes some time before a service begins receiving any data. The service $limit(t, r, S)$ acts like service S, except that each physical connection is considered to have failed if the rate ever drops below $r\ Kbytes/sec$ after the first t seconds of the connection. Physical connections are created by invocations of $url,\ index$ and $gateway$ combinators.
Translation:

```
do
    exec S
    ||
    [await tick(t);
    emit Start-Limit]
watching LIMIT
||
await Start-Limit;
trap T in
    loop
          await rate;
          if ?rate < r
          then [ emit LIMIT;
                exit T ]
    end % end loop
end % end trap
```

Interpretation: A utility that is added to CRP is the ability to compute the rate of a basic service, namely the rate of arrival of the incoming data stream on fetching an URL. As suggested by Cardelli and Davies [6], the data-rate is computed using an ad hoc but nevertheless effective mechanism of taking the average over the previous two seconds as calculated by samples done five times a second. The statement

`?rate` provides information about the current rate of arrival of input data. The above translation forces the service to fail if (after an initial start-up time of t seconds) the rate ever drops below the desired rate. Given the ability of CRP to count the number of occurrences of events, it is easily possible to have variants of the above scenario, such as forcing the service to fail, not the first time the rate drops below r (after initial start-up time), but the third (say) time the rate drops below r:

```
do
    exec S
    ⋮
watching 3-rate-limit
⋮
```

3.7. Repetition

Service: $repeat(S)$
Specification: The $repeat$ combinator provides a way to repeatedly invoke a service until it succeeds. The service $repeat(S)$ acts like S, except that if S fails, $repeat(S)$ starts again.
Translation:

```
trap T in
    loop
        exec S;
        if ?S.success then exit T
    end % end loop
end % end trap
```

Interpretation: In a `loop`, the body should not terminate instantaneously. This means that the `exec S` should consume at least one instant of time for execution.

3.8. Non-termination

Combinator: $stall$
Specification: The combinator $stall$ never completes or fails and always has a rate of zero.
Translation: `halt`
Interpretation: The `halt` command continues forever, and the only way to terminate it is through preemption.

3.9. Failure

Combinator: $fail$
Specification: The $fail$ combinator fails immediately.
Translation:

```
do
    await tick(1)
watching immediate tick(0)
```

Interpretation: The translation of `fail` involves awaiting `tick(1)` on the clock, but is preempted if `tick(0)` occurs earlier. As `tick(0)` does always precede `tick(1)`, the entire construct fails at the same instant.

4. Implementation

We have implemented a laboratory prototype of the service combinators for Web computing in the language of Communicating Reactive Processes (CRP) [9] . The combinators have been implemented on top of an existing compiler of CRP [9], which in turn is based on an ESTEREL [4] compiler.

In order to implement the service combinators in CRP, a few enhancements had to be made to the existing CRP implementation. In particular we had to enrich the CRP implementation with capabilities for interaction on the World-Wide Web [1]. In other words we had to endow it with functions for supporting the HTTP [2] protocol.

Another important utility that was added to CRP was the ability to compute the rate of a basic service, namely the rate of arrival of the incoming data stream on fetching an URL. As suggested by Cardelli and Davies [6], the data-rate is computed using an ad hoc but nevertheless effective mechanism of taking the average over the previous two seconds as calculated by samples done five times a second.

5. Further work

From our translation it can be seen that we have realized the scripting language with ESTEREL [4] and *tasks*, where CLOCK is a special task. We intend to use the `rendezvous` feature of CRP to show the possible interactions that one could have between various agents. In implementing the service combinators, we have so far utilized only a small fraction of the interaction power of CRP. As mentioned earlier, a CRP program consists of a number of independent ESTEREL reactive programs or nodes which interact with each other through asynchronous operations. The implementation of the service combinators makes use of only one ESTEREL node, which interacts with its environment, namely with the rest of the world wide Web.

The next logical step would be to allow the migration of CRP programs over the Web to other sites, and exploit the interaction between the CRP agents at the local site and the external sites. So far we have modeled the interaction of a single CRP program with its environment, using the notions of asynchronous `tasks` and the `exec` primitive.

In the case where there are multiple nodes, interaction between them would be based on the more powerful mechanism of `rendezvous` on `channels`, using the `send` and `receive` constructs for asynchronous operations. The basic service of fetching the contents of an URL by the local agent from a remote agent would be translated to CRP as follows.

```
module local-agent:
        output channel url: string;
        input channel contents: string;

        rendezvous url(?Address);
        rendezvous contents(?File)
end module

module remote-agent:
        input channel url: string;
        output channel contents: string;

        rendezvous url(?Address);
        rendezvous contents(?File)
end module
```

We are in the process of developing a package for use in distant education courses, which makes use of such autonomous CRP agents interacting over the world wide Web. The presence of such agents helps in tailoring the education programs to the specific needs of every individual learner, and also for easy monitoring of the learners' progress. It also helps in retaining the flexibility of the teaching process itself by tailoring further lessons depending on the detailed feedback obtained from the agents.

6. Conclusion

We have shown that the well developed paradigm of reactive programming provides an ideal base for launching web-scripting languages. We have shown that service combinators for Web computing can be easily implemented in the language of ESTEREL and Communicating Reactive Processes.

A number of features which are unique to reactive languages, support the embedding of primitives suitable for Web programming. Among these features of reactive programming, are the unification of synchronous and asynchronous interaction provided by them; their concurrency capabilities, and also the rich set of error-handling primitives.

From the experience obtained so far, we have good reasons to believe that further exploration of the reactive paradigm for web-scripting languages could prove to be a fruitful endeavor. Such web-scripting languages would help in the construction of novel applications over the Web. They would also help in exploiting the resources available over the Web by migrating autonomous agents over the Web to other sites, and thus assist in tailoring the Web to suit the needs and requirements of each individual user of the World-wide web.

References

[1] T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret. The world-wide web. *Commun. ACM*, 37(8):76–82, August 1994.

[2] T. Berners-Lee and D. Connolly. *Hypertext Markup Language – 2.0*. RFC 1866, MIT/W3C, 1995.

[3] G. Berry. Preemption in concurrent systems. In *Proc. FSTTCS'93, Lecture Notes in Computer Science 761, Springer*, pages 72–93, December 1993.

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

[5] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 85–99, January 1993.

[6] L. Cardelli and R. Davies. *Service Combinators for Web Computing*. SRC Research Report 148, Digital Equipment Corporation Systems Research Center, June 1, 1997.

[7] N. Gehani and K. Ramamritham. *Real-Time Concurrent C: A Language for Programming Dynamic Real-Time Systems*. Technical Report, University of Massachusetts, 1991.

[8] N. Raja and R. K. Shyamasundar. *Web-Scripting Languages for Free*. TIFR Research Report, Computer Science Group, Tata Institute of Fundamental Research, 1998.

[9] B. Rajan and R. K. Shyamasundar. An implementation of communicating reactive processes. In *Proc. ICPDCN*, 1997.