Linguistic Support for Controlling Protocol Execution *

Yen-Min Huang

Experimental Systems
IBM Corporation
R.T.P, NC 27715

Chinya V. Ravishankar

Department of EECS
The University of Michigan
Ann Arbor, MI 48109-2122

Abstract

Implementing efficient communication protocols is an important task in building distributed systems, but is complicated by the difficulties of dealing with complex multi-thread interactions and timing-related bugs. This paper describes Cicero, a set of language constructs designed to alleviate these difficulties. Cicero uses the notion of event patterns [1] to help programmers build robust protocol implementations. Event patterns provide structure for controlling synchrony, asynchrony, and concurrency in protocol execution, and also allow implementors to exploit parallelism of varying grains. Event patterns can be translated into other formal models, so that existing verification techniques may be used. Our prototype implementation indicates that the total overhead imposed by event patterns accounts for less than 5% of the overall latency for protocols above the transport layer on single-processor implementations.

1 Introduction

New protocols are often useful, but hard to implement well. In the future, more advanced functionality will likely be moved down to protocol implementations to save application development effort. This trend is evident from the recent development of RPCs to include semantics for supporting group communication, transactions, fault-tolerance, etc [2, 3, 4].

As new distributed applications continue to emerge, protocol developers will also be challenged to provide correct and efficient implementations managing concurrent I/O channels, and to increase the protocol throughput to meet real-time requirements. Such requirements demand better language support to facilitate precise control of multiple-thread interactions, and aggressive exploitation of parallelism in protocol execution, but without complicating the protocol verification process. Cicero is a set of language constructs designed to meet these challenges.

Difficulties are most likely to arise in correcting bugs dealing with synchrony, asynchrony and concurrency in protocol execution, especially when multi-threaded execution is involved. A typical challenge is to detect and correct a timing-related bug which happens non-deterministically in 5% of all test runs. Such difficulties will become common in future protocol implementations, which may use multiple-thread support to implement multicast protocols, or exploit parallelism in protocol execution to increase throughput.

We designed Cicero as a language veneer extending existing languages, since we believe the difficulties above can be alleviated by a small set of language constructs, mostly responsible for controlling protocol execution. These control constructs can be designed to coexist with most traditional programming languages, freeing programmers from having to learn an entirely new language. We have chosen C as our present target language for Cicero because it is portable and has efficient compilers.

Cicero is designed to facilitate *hybrid* protocol implementation strategies. Protocols can either be implemented by hand or be synthesized from specifications [5, 6, 7, 8]. These two approaches represent opposite strategies, and are effective in different situations. We argue that they can be combined, given a good protocol construction language that enables the representation of execution aspects of protocol implementations, but which does not complicate verification. Our work attempts to combine these two approaches by giving programmers an executable specification language to specify control aspects of protocol execution as well as implementation details.

We have implemented Cicero and used it with Nestor [9] and the URPC toolkit [10] for describing

*This work was partly supported by the Consortium for International Earth Sciences Information Networking

RPC protocols. Nestor is an agent synthesis and management system for synthesizing cross-RPC gateway agents [11, 9]. URPC (Universal RPC) is a toolkit for prototyping new RPC systems rapidly [10]. Experience in these projects has shown that Cicero is effective in implementing a variety of RPC protocols. Cicero offers a better protocol implementation paradigm with little overhead, amounting to less than 5% for protocols above transport layer, even for non-parallel implementations. Cicero is designed to address a variety of issues common to protocol implementations, and not specifically for RPC. It can be effective in implementing protocols in other layers.

2 Design Rationale

2.1 Basic Language Abstractions and Semantics

A natural abstraction for protocols is an event-driven paradigm, which views a protocol as a machine reacting to internal/external events or messages [12, 13, 14, 15, 16]. However, an event-driven model is insufficient because complex relationships between events must be expressed in a structured way. For this purpose, we have borrowed the notion of *event patterns* from POST [1]. Event patterns use *event combinators* to recursively describe relationships between events. Cicero uses the three event combinators "$\wedge$", ",", and "$\sim$", to express synchronous, asynchronous, and sequential relationships between events respectively.

Event patterns in Cicero have active semantics [1, 17]. They behave as safeguards, locally guaranteeing relationships between events, rather than passively detecting these relationships. Our event patterns ensure specified relationships between events before performing actions, rather than detecting some relationships between events and then performing actions as in many other event-driven languages [14, 16]. For example, in Cicero, the event pattern $(a \sim b)$ sequences the order of events a and b, delivering a first. If event b actually occurs first, its delivery is delayed until event a has first been delivered. In contrast, passive semantics trigger actions only when an occurrence of event a is followed by an occurrence an event b.

We believe active pattern semantics enable programmers to construct more robust protocol implementations than do passive semantics. With passive pattern semantics, if some expected actions are not executed, it is up to the programmers to locate the reasons for mismatched patterns. When such mismatches are caused by subtle timing problems, finding the bug can be extremely hard. In contrast, with active pattern semantics, the problem can often be avoided or corrected by patching in a few extra patterns. Active pattern semantics can make implementations more robust by ensuring correct behavior, and also reduce the possibilities of inadvertently introducing timing-related bugs during performance tuning.

2.2 The Execution Model

Multi-processor systems are becoming common, so it is important to exploit coarse-grain parallelism in protocol execution. We have found a restricted dataflow model [18] helpful. An obvious analogy can be drawn between events and data tokens in a dataflow model, where token arrival triggers/fires actions [1]. We simply associate event patterns with code segments of the proper granularity. By changing the code granularity, we can change the granularity of parallelism in Cicero. This capability allows programmers to experiment with different granularities in tuning performance.

The dataflow model also has other advantages. First, it is mathematically well-defined [19] and well-understood. Second, it can be translated to/from other formal models (e.g. Petri nets [19]), making it possible to use existing protocol verification methods/tools, and easier to construct tools for generating protocol implementations from existing protocol specifications. These capabilities can further automate protocol implementation and improve the quality of implementation. The translation of event patterns to Petri nets is described in [10].

2.3 The Communication Primitives

Several design choices present themselves for communication primitives in Cicero: remote events, new communication constructs, or library calls. However, our goal is to design a language for implementing protocols, not a language for general distributed computing. Remote events are restrictive since they impose their semantics on every protocol constructed. Also, programmers will have no control over messages passed over the network. We reject new communication constructs for similar reasons. Besides, once they become a part of Cicero, it will be difficult to replace them in the future. Thus, Cicero provides communication primitives through library calls because they are flexible and can be easily replaced. This approach allows developers to customize Cicero for implementing different classes of protocols. For example, we have used Cicero to construct heterogeneous RPC mechanisms to facilitate the interconnection between the client and server programs speaking different RPC protocols [11]. In our case, a Cicero communication library implementing transport layer services is provided as a part of the package, and developers are allowed to construct different RPC protocols on top of the library provided.

3 Cicero Concepts

The Cicero language model is based on the notions of *events*, *event instances*, and *event patterns*. Events and event instances will be introduced first.

3.1 Events and Event Instances

Events are unbounded sequences of event instances. An event instance is an object modeling the occurrence of a real event. For example, if timeouts are modeled as events, then the third occurrence of a timeout event is represented by the third timeout event instance. Instances of an event may occur at several places in a program, but all instances of the same event are globally ordered and delivered in order by the Cicero runtime. No ordering is defined between instances of different events. Delivery order between different events can only be controlled by patterns. Following the previous example, if several timeout instances occur concurrently, they will all be globally ordered, and the order of instances will be identical for all observers of the timeout event. Such ordering helps programmers coordinate tasks among different threads.

An event instance can be generated (emitted), observed, and consumed. An event instance can be explicitly emitted by programmers, or can be implicitly generated by the Cicero language runtime. One copy of an emitted event instance is made available to each of its observers. Actions may be triggered when an event instance is observed, and the event instance copy is consumed when these actions are finished. An instance is not available after it is consumed.

An event instance is a structured object containing three fields representing attributes of the event instance: an *instance number* field, a *priority* field, and a *value* field. The instance number reflects the global ordering among instances of a given event. The priority affects the execution priority of the action triggered by the event instance, but not the order of instance delivery. Priority is used to facilitate the implementation of out-of-band and exception events, which may require actions to be executed immediately. The value field is used to associate a value or a data structure (via a pointer) with an event instance. The value field is used primarily for associating extra states with events, so that programmers can customize them for different situations.

The instance number field is read-only, but the priority and the value fields can be read and set by programmers. The settable fields can be set only when a new instance is generated, and become read-only after the generation of the instance. Thus, no concurrency control is necessary to read field values. If these field values are not set, the newly-emitted instance inherits field values from the previous instance. Default values for the first instance are provided.

3.2 Event Patterns

An event pattern specifies the precise relationships between event instances that trigger actions in a protocol. Patterns are based on three basic notions: *event combinators*, *event pattern instances*, and *actions*.

3.2.1 Event Combinators

Event combinators are operators describing the relationships between event instances that must be ensured before actions can be triggered. These relationships can be synchronous, asynchronous, or sequential, and the corresponding event combinators are "$\wedge$", ",", and "$\sim$", respectively. Event combinators may be used to combine simpler event patterns into more complex ones to express complex relationships. For example, the code segment "**when** ($x \wedge y$): **emit** z; **end**" specifies that when event instances x and y are both observed, instance of z is to be emitted.

Because relationships between events often exhibit repeating behavior, the repeat operator "$*$" is introduced, and specifies that actions are to be executed each time the pattern conditions are met. All operators/combinators can be recursively applied to construct complex event patterns. The basic event patterns in Cicero and their informal semantics are described in Section 3.2.4.

3.2.2 Event Pattern Instances

An instance of event pattern comes into existence when the specified event instance relationship is observed, and the event pattern becomes *active*, and triggers the associated actions. Only the activating instances are accessible (available) in the code triggered by the pattern. The pattern instance and its activating instances are consumed when the triggered actions terminates. Overflow actions associated with finite repeating patterns are triggered when the number of occurrences of pattern instances exceed the specified limit.

3.2.3 Actions

To encourage coarse-grain parallelism, an action is executed as a thread (light-weight process), which may subsequently create more threads (actions) in execution. To meet different implementation requirements, we allow threads to be invoked either synchronously or asynchronously through different mechanisms. Asynchronous invocation is accomplished by emitting event instances, while synchronous invocation is accomplished using **bundle** calls (see Section 4). An invoking thread is blocked until all the synchronously invoked threads are terminated, but may continue execution without waiting for any asynchronously invoked threads to finish. Thus, the termination of a thread is defined as the termination of all its synchronously invoked threads plus the termination of itself.

A triggered action is scheduled and executed according to its execution priority, which is simply the highest of the priorities of the activating event instances. Scheduling in Cicero is preemptive; however, the scheduling quantum is unspecified.

3.2.4 Event Pattern Semantics

Here we describe the semantics of event patterns informally. The formal description can be found in [10]. The syntax "P : *actions* : *overflow-actions*" denotes an event pattern, associated actions and *overflow* actions respectively. Let E_1 and E_2 be two event patterns comprising P. The semantics of P are described as follows:

1. $(E_1 \wedge E_2)$: *actions*
   This pattern requires that the associated action be triggered only when instances of both E_1 and E_2 come into existence.
2. $(E_1 \ , E_2)$: *actions*
   This pattern requires that the associated action be triggered when an instance of either E_1 or E_2 comes into existence. The action instances triggered by instances of E_1 and E_2 may execute concurrently.
3. $(E_1 \sim E_2)$: *actions*
   This pattern requires the actions to be triggered separately by instances of E_1 and E_2, and in that sequence. No action may be triggered by an instance of E_2 unless the action triggered by the corresponding instance E_1 is finished.
4. $(E_1 \ ,)*$: *actions*
   This pattern has the same semantics as the pattern (E_1). The associated action is triggered when each instance of E_1 comes into existence, and triggered action instances may execute concurrently.
5. $(E_1 \sim)*$: *actions*
   The associated action is triggered and executed sequentially when each instance of E_1 comes into existence. No action may be triggered by the i^{th} instance of E_1 unless the action triggered by the $(i-1)^{th}$ instance of E_1 is finished.
6. $(E_1 \wedge)*$: *actions*
   This pattern will never trigger its associated action because it must wait infinite number of instances of E_1.
7. $(E_1 \wedge)*N$: *actions* : *overflow-action*
   This pattern requires that the associated action be triggered only when N instances of E_1 come into existence. The overflow action is triggered by all subsequent instances of E_1. The description for overflow semantics is omitted from here on because they are identical for all finite repeating event patterns.
8. $(E_1 \ ,)*N$: *actions* : *overflow-actions*
   This pattern requires that the associated actions be triggered each time an instance of E_1 comes into existence, up to N times.
9. $(E_1 \sim)*N$: *actions* : *overflow-actions*
   This pattern requires that the associated action be triggered each time an instance of E_1 coming into existence (up to N times), provided that the action triggered by the previous E_1 instance has terminated.

4 Language Constructs in Cicero

Cicero includes five language constructs: **emit**, **when**, **cond**, **bundle**, and **escape**.

- The **emit** construct is used to generate or signal event instances. Two examples of the use of the **emit** construct are:

  emit $e1$:(val=1, pri=2);
  emit $e1$;

The first **emit** generates an instance of $e1$ with its value and priority field set to 1 and 2 respectively. The second **emit** simply generates an instance of $e1$, whose value and priority will be inherited from the previous instance.

- The **when** construct uses event patterns to control the execution of associated target code, which determines the granularity of parallelism in Cicero. Each **when** construct has its own thread of control. Since many **when** constructs may name the same event in their event patterns, one event instance may trigger many **when** constructs. These threads will all run concurrently.

  A **when** construct consists of three parts: an event pattern, a list of target statements, and possibly an overflow statement. The syntax and an example of the **when** construct are illustrated below.

when	*event_pattern*:	**when**	($e1$, $e2$,)*3:
	action_stmts		**emit** $e3$;
end:	*overflow_stmt*	**end**:	**emit** $e4$;

  In this example, an instance of event pattern ($e1$, $e2$) goes active to trigger actions when an $e1$ or $e2$ instance occurs. The overflow code is executed when the number of the event-pattern instances created exceeds three.

- The **cond** construct implements conditional statements, and is reminiscent of the LISP **cond** construct. When several conditionals evaluate to **true**, the statements associated with all of them are executed in order. This approach helps increase the granularity of parallelism and reduce concurrency control overhead. An example of a **cond** construct can be found at line 35 in Figure 2.

- The **bundle** construct defines the extent of visibility (scope) for event instances, and provides an environment for sharing variables among a group of **when** constructs. It encourages a modular programming style by factoring out multi-threaded subproblems. For example, a **bundle** may consolidate a class of the general producer/consumer problem within a protocol construction, and may contain two **when** constructs (one for the producer and one for the consumer) sharing a common data structure. The **bundle** call semantics are synchronous, i.e., the **bundle** caller is blocked until the completion of the **bundle**. The completion of the **bundle** is indicated by emitting a special event RETURN, and the return codes can be set in the value field of RETURN. These synchronous call semantics are designed to make programming easier, because no concurrency control between the **bundle** and its caller is necessary. **Bundles** can be nested or recursive; therefore, all the activated threads of a **bundle** form a thread hierarchy.

- The **escape** construct allows programmers to include statements in the base language, and is used to include implementation details. Such inclusion is accomplished by enclosing these statements within '{' and '}'. This block of target language statements is called an *escape component*. Within the braces, programmers can declare local variables, access data structures, and call procedures as in ordinary C programs. See Figure 2, line 10 for an illustration of its use.

5 An Example

In this section, we will construct an RPC protocol with at-most-once semantics by mirroring the extended finite state machine (EFSM) specification in Figure 1 in Cicero code. The protocol described here is based on the client-server model, and only the client code segments are presented. In brief, the protocol works as the following: after sending the data to the server, the client will ping the server periodically until either the result is received, or the number of unacknowledged ping messages exceeds the predefined limit.

Although the Cicero specification can be made more compact, we present a somewhat longer version to make the implementation easier to understand. There is a one-to-one mapping of events between the EFSM specification in Figure 1 and the Cicero code segment. The correspondence between the Cicero code segment and the original specification is shown in the comments within the code segment. The library functions used in the code segment are also briefly described in Table 1.

The Cicero code segment is shown at Figure 2. Initially, two events, *send_data* and *recv_data*, are emitted to trigger two **when** constructs to send and receive RPC messages (line 14 to 15, line 18 and 24). After the sending out the RPC message (line 20), a timer **when** construct (line 29) is triggered for pinging the server every 60 seconds. If the result is received, the **bundle** returns (line 26). If the number of unacknowledged ping messages exceeds the limit Max_Retry, the **bundle** returns with an error (line 36). It is possi-

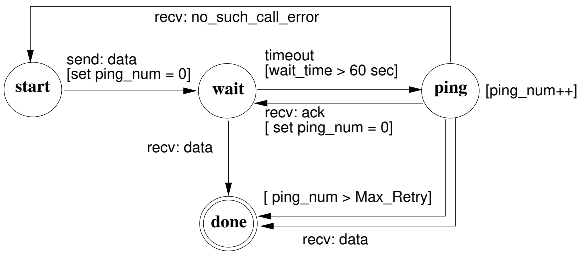

Figure 1: An Extended FSM Diagram For At-Least-Once Semantics

Function	Description
CC_send_udef_msg	sends out an RPC message.
CC_recv_udef_msg	waits for an RPC reply message.
CC_send_ctrl_msg	sends out a control message.
CC_recv_ctrl_msg	waits for receiving a control message.
CC_wait	pause for a period of time before continuing.
CC_ioctl	set input/output control options (similar to UNIX *ioctl*()).
CC_set_udef_sendmsg	associates an RPC message with the communication handle, so that it can be sent out later.

Table 1: Description of Functions Used In Bundle client_rpc()

```
 1 bundle client_rpc(CC_handle_t handle, CC_msg_t *msg)
 2 {
 3     int        err_code, ping_num, msg_type;
 4     long       wait_time, value;
 5     event      recv_data, recv_ctrlmsg;
 6     event      send_data, wait, ping;
 7
 8     when (INIT):  /* FSM: start -> wait */
 9         {
10         wait_time = 60;  /* wait for 60 sec. */
11         CC_set_udef_sendmsg(handle, msg);
12         CC_ioctl(handle, RECVBLOCK, TRUE); /* block at recv */
13         }
14         emit send_data;
15         emit recv_data;
16     end;
17
18     when (send_data): /* FSM: start -> wait */
19         { ping_num = 0;
20           CC_send_udef_msg(handle); }
21         emit wait;
22     end;
23
24     when (recv_data): /* FSM: wait -> done */
25         { err_code = CC_recv_udef_msg(handle); }
26         emit Return:(val=err_code);
27     end;
28
29     when (wait):    /* FSM: wait -> ping */
30         { CC_wait(wait_time); }
31         emit ping;
32     end;
33
34     when (ping): /* FSM: ping -> wait */
35         cond (ping_num > Max_Retry):
36                emit Return:(val=E_RPCFAIL);
37             otherwise:
38               { ping_num++;
39                 CC_send_ctrl_msg(handle,PING); }
40               emit recv_ctrlmsg;
41               emit wait;
42        end;
43     end
44
45     when (recv_ctrlmsg): /* FSM: ping -> wait, start */
46         { msg_type = CC_recv_ctrl_msg(handle); }
47         cond (msg_type == ACK):
48                    { ping_num = 0; }
49                     emit wait;
50              ((msg_type == ERROR) && (value == NO_SUCH_CALL_ERR)):
51                     emit send_data;
52               otherwise:
53                     emit Return:(val=E_RUNTIME);
54         end;
55     end;
56 }
```

Figure 2: Cicero Code Segment For At-Most-Once Semantics

ble that the original RPC message never reaches the server. In this case, a NO_SUCH_CALL_ERR error message is returned, and the original RPC message is resent (line 50 to 51).

6 Related Work

The most novel aspect of Cicero is its approach to the integration of existing notions and abstractions to fulfill its design goals. We refer the reader to [10] for an expanded discussion of material in this section. Cicero integrates the following ideas:

• ***Event Patterns***: The semantics of event patterns are borrowed from POST [1], but POST is designed as a general pattern-driven dataflow language, while Cicero is designed for protocol construction. Event patterns are also reminiscent of *path expressions* [17] and *data path expressions* [16], but have different usage and semantics. Event patterns are used to specify when to execute a piece of code, while path expressions are used to specify the synchronization constraints on how procedures can be executed. For example, it is hard using path expressions to indicate that some target code be triggered only when both event $e1$ and $e2$ occur. Data path expressions [16] provide passive pattern-matching semantics.

• ***Event-Driven Abstraction***: Our event-driven model is similar to the notion of *event flows* in ESTEREL [14], and to some concepts in Actor [15]. However, neither system includes event patterns, which express complex relationships between events for controlling the execution. Also, we globally order the instances of the same event to facilitate the coordination among multiple threads.

• ***Communication Mechanisms***: Communication within the same address space is accomplished by emitting/observing events, and communication across address spaces is supported by a set of library routines. Remote communication must often deal with the additional problem of partial failures, and the resilience of failure can vary greatly across protocols. Also, this separation allows Cicero to adopt additional programming abstractions for protocol implementation, including the object-based abstraction provided in [20].

• ***Dataflow Execution Model***: Many languages [1, 21] and machines [22, 23] have been designed based on this model. Cicero uses this model differently, and primarily to describe event-driven execution at different granularities, and as formalism to allow Cicero to be translated to/from other protocol specification models (e.g. Petri nets [19]).

• ***Protocol Specifications***: Several Formal Description Techniques (FDT), like LOTOS [12], Estelle [13], and SDL [24], have been developed to specify protocol behavior formally. Much research has been conducted in automatically generating protocol implementations from these FDTs [8, 25]. However, the protocol implementations generated by these means are generally in the form of skeletons which must be filled in by programmer code [20]. Also, the efficiency of generated code is a concern [26]. Cicero is designed as an executable specification language to allow programmers to have direct control over generated code, and requires no additional patching to the generated code.

7 Implementation and Performance

Our Cicero implementation includes the Cicero compiler and the Cicero runtime library. Each **when** construct is compiled to a procedure and executed as a thread. The Cicero runtime library does not provide its own thread package, since thread packages are often platform-dependent. It provides interfaces to existing thread packages instead. Currently, the runtime library supports interfaces to SUN LWP [27], and UNIX Cthreads [28]. when required, the Cicero runtime can also operate with no threads by calling the **when** constructs in order.

The control mechanism and the underlying thread package are the two sources of overhead. The first overhead includes that of for emitting event instances and executing event patterns. Event instance emission requires creating an instance data structure, inserting it into the event instance queue, dispatching it to **when** constructs, and putting **when** constructs into the task queue. Pattern execution requires evaluating the pattern status, computing execution priority, and updating the status. The thread package overhead is mostly for mutual exclusion and thread management. Table 2 shows the instance emission and the pattern execution overhead with and without thread packages, on a SUN Sparc 1 workstation runing SunOS 4.1.1.

The basic runtime overhead for emitting an event instance and triggering target code is about 0.05 ms (0.03ms for instance emission and 0.02ms for pattern execution). However, with a thread package, this overhead can rise to 0.17ms–0.24ms due to the cost of mutual exclusion and thread management.

Overhead Type	NOLWP	CThread	SUNLWP
Instance Emission	0.03ms	0.13ms	0.19ms
Pattern Execution	0.02ms	0.04ms	0.05ms
Total	0.05ms	0.17ms	0.24ms

Table 2: Cicero Runtime Overhead

The overhead for pattern execution grows with size, while instance emission overhead remains constant. However, the rate of growth for pattern execution overhead is moderate, reaching 0.1ms for event pattern sizes of 10. This overhead is usually well below 5% of overall performance, especially for protocols above the transport layer. For example, a typical SUN RPC round-trip time between 2 SUN Sparc stations in a LAN environment is already 4 to 10ms. If no thread support is used, the language overhead is negligible ($< 1\%$) compared to the other delays. We expect the overhead to diminish further if running on a multiprocessor architecture, where we can parallelize pattern execution using divide-and-conquer strategies.

8 Conclusions

Cicero is a small set of language constructs extending existing programming languages to support multithread protocol implementations, and offers many advantages.

Event patterns control synchrony, asynchrony, and sequentiality in protocol execution, and provide a better implementation paradigm than thread packages alone. Active event-pattern semantics can ensure required event relationships and result in more robust protocol implementations. Cicero also provides support for multiple-thread execution so that parallelism of varying grains can be specified, and parallel architectures fully exploited. Event patterns can be translated to/from other models/languages describing protocols, making it possible to use existing tools/methods for protocol verification. Cicero can be easily learned and mastered because it is a veneer oven existing languages. Finally, its overhead is acceptable for a wide range of applications.

References

[1] C. V Ravishankar and R. Finkel. Linguistic Support

for Dataflow. Technical Report CSE-TR-14-89, Dept. of EECS, The University of Michigan, Ann Arbor, Michigan, 1989.

[2] K. S. Yap, P. Jalote, and S. Tripathi. Fault Tolerant Remote Procedure Call. In *Proc. of 8th International Conference on Distributed Computing Systems*, pages 48–54, San Jose, CA, June 1988.

[3] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[4] L. Zahn, T. H. Dineen, P. J. Leach, E. A. Martin, N. W. Mishkin, J. N. Pato, and G. L. Wyant. *Network Computing Architecture*. Prentice-Hall, Englewood Cliffs, N.J., 1990.

[5] G. v. Bochmann, G. Gerbert, and J. M. Serre. Semi-automatic Implementation of Communication Protocols. *IEEE Transactions on Software Engineering*, 13(9):989–999, September 1987.

[6] J. P. Briand, M. C. Fehri, L. Logrippo, and A. Obaid. Executing LOTOS Specifications. In B. Sarikaya and G. v. Bochmann, editors, *Protocol Specification, Testing and Verification VI (IFIP/WG 6.1)*, Amsterdam, The Netherlands, North-Holland, 1987.

[7] J. P. Ansart, P. D. Amer, V. Chari, J. F. Lenotre, L. Lumbroso, E. Mariani, and E. Mattera. Software Tools for Estelle. In B. Sarikaya and G. v. Bochmann, editors, *Protocol Specification, Testing and Verification VI (IFIP/WG 6.1)*, Amsterdam, The Netherlands, North-Holland, 1987.

[8] S. T. Vuong, A. C. Lau, and R. I. Chan. Semiautomatic Implementation of Protocols Using an Estelle-C Compiler. *IEEE Transactions on Software Engineering*, 14(3):384–393, March 1988.

[9] Y. Huang and C. V. Ravishankar. Designing An Agent Synthesis System for Cross RPC Communication. *IEEE Transactions on Software Engineering*, 20(3), March 1994.

[10] Y. Huang and C. V. Ravishankar. Cicero: A Protocol Construction Language. Technical Report CSE-TR-171-93, Dept. of EECS, The University of Michigan, Ann Arbor, Michigan, 1993.

[11] Y. Huang and C. V. Ravishankar. Accommodating RPC Heterogeneities in Large Heterogeneous Distributed Environments. In *Proc. of the 26th Hawaii International Conference on System Sciences (HICSS-26)*, January 1993.

[12] ISO. *Information Processing Systems – Open System Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, 1985.

[13] ISO. *Information Processing Systems – Open System Interconnection – Estelle (Formal Description Technique Based on an Extended State Transition Model)* , 1987.

[14] G. Berry and G. Gonthier. The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation. Technical Report 842, INRIA, 1988.

[15] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.

[16] W. Hseush and G. E. Kaiser. Modeling Concurrency in Parallel Debugging. In *Proc. of 2nd ACM SIGLPAN Symposium on Principles and Practice of Parallel Programming*, pages 11–20, March 1990.

[17] R. H. Campbell and A. N. Habermann. The Specification of Process Synchronization by Path Expression. In *Lecture Notes in Computer Science*, volume 16, pages 89–102, New York, 1974. Springer-Verlag.

[18] R. M. Karp and R. E. Miller. Properties of a Model for Parallel Computation: Determinacy, Termination, Queueing. *SIAM Journal of App. Math*, pages 1390–1411, November 1966.

[19] K. M. Kavi, B. P. Buckles, and U. N. Bhat. Isomorphism Between Petri nets and Dataflow Graphs. *IEEE Transactions on Software Engineering*, 13(10):1127–1134, October 1987.

[20] M. B. Abbott and L. L. Peterson. A Language-Based Approach to Protocol Implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, February 1993.

[21] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press U.K., 1985.

[22] W. W. Hwu and Y. Patt. HPSm, a High Performance Restricted Data Flow Architecture having Minimal Functionality. In *The 13th International Symposium on Computer Architecture Conference Proceedings*, pages 297–306, June 1986.

[23] M. Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, Englewood Cliffs, N.J., 1991.

[24] CCITT. *Specification and Description Language – Recommendation Z.100* , 1986.

[25] A. Valenzano, R. Sisto, and L. Ciminiera. Rapid Prototyping of Protocols from LOTOS Specification. *Software – Practice and Experience*, 23(1):31–54, January 1993.

[26] L. Svobodova. Implementing OSI Systems. *IEEE Journal on Selected Areas in Communications*, 7(7):1115–1130, September 1989.

[27] Sun Microsystems. *Programming Utilities and Libraries*, March 1990.

[28] K. Schwan, H. Forbes, A. Gheith, B. Mukherjee, and Y. Samiotakis. A CThread Library for Multiprocessors. Technical Report GIT-ICS-91/02, College of Computing, Georgia Institute of Technology, 1991.