Constructive Protocol Specification Using Cicero

Yen-Min Huang and Chinya V. Ravishankar, *Senior Member*, *IEEE*

Abstract—New protocols are often useful, but are hard to implement well. Protocol synthesis is a solution, but synthesized protocols can be slow. Implementing protocols will be even more challenging in the future, since we expect that more advanced communication functionality will be moved from applications into protocol implementations to reduce application development effort. This trend can be seen from the recent enhancements of RPC to include semantics for supporting group communication, transactions, fault-tolerance, etc. [1], [2], [3], [4]. Protocol developers will also be challenged to provide correct and efficient protocol implementations that manage numerous concurrent I/O channels, and to increase protocol throughput to meet real-time requirements. These requirements demand better language support to facilitate precise control of multiple-thread interactions, and aggressive exploitation of parallelism in protocol execution. Protocol synthesis is also required for dynamic creation of protocol adapters in heterogeneous environments [5], [6]. This paper describes Cicero, a set of language constructs to allow constructive protocol specifications. Unlike other protocol specification languages, Cicero gives programmers explicit control over protocol execution, and facilitates both sequential and parallel implementations, especially for protocols above the transport-layer. It is intended to be used in conjunction with domain-specific libraries, and is quite different in philosophy and mode of use from existing protocol specification languages. A feature of Cicero is the use of event patterns [7] to control synchrony, asynchrony, and concurrency in protocol execution, which helps programmers build robust protocol implementations. Event-pattern driven execution also enables implementors to exploit parallelism of varying grains in protocol execution. Event patterns can also be translated into other formal models, so that existing verification techniques may be used.

Index Terms—Protocol synthesis, protocol specification, protocol implementation, event-driven language.

1 Introduction

Implementing protocols is an important but difficult aspect of building distributed systems. New protocols are often useful because they factor out and encapsulate commonly observed interaction patterns. However, they can be hard to implement well. We envision that in the future, more of the interaction patterns of applications will be factored out into new protocols, and that the application/protocol boundary will become increasingly blurred.

This trend is already evident. For example, RPC suites now include semantics for supporting group communication, transactions, fault-tolerance, etc. [1], [2], [3]. Work has also already been done on moving much of the TCP/IP and UDP/IP protocol stacks into user space [4]. This move has allowed network protocol implementation with both high performance and flexibility, while retaining existing application programming interfaces. We expect this trend to strengthen in the future, particularly for often-used but complex protocols.

Protocol synthesis also simplifies service management issues in heterogenous environments. For example, the Cygnus system [5] generalizes the usual client-server model into a client-service model, and introduces abstract services on the network. Clients only see abstract services, and are insulated from having to deal with a heterogenous collection of service providers. Cygnus deals with heterogeneity by using agents at client sites that translate client protocols to server protocols. The work in [6], [8], [9] describes how such agents are synthesized on demand by retrieving constructive specifications in Cicero from server sites.

As distributed group/multimedia applications continue to emerge, protocol developers will also be challenged to provide correct and efficient protocol implementations that manage numerous concurrent I/O channels, and to increase the protocol throughput to meet real-time requirements. These requirements demand better language support to facilitate precise control of multiple-thread interactions, and aggressive exploitation of parallelism in protocol execution.

This paper describes Cicero, a set of language constructs designed to meet these challenges. This language differs in significant ways in its goals and approach from existing declarative protocol specification languages [10], [11], [12], [13], [14], [15]. In particular, Cicero specifications are *constructive*, so that programmers may directly control protocol execution. They are able to implement run-time event synchronization and control event sequencing and visibility through *event patterns*.

Cicero is designed to facilitate *hybrid* protocol implementation strategies. Protocols can either be implemented by hand or be synthesized from specifications. Implementing protocols by hand produces very efficient implementations, but requires thorough testing and debugging to assure correctness [16], [17]. Testing and debugging can be

- *Y.-M. Huang is with IBM, Research Triangle Park, NC.*
- *C.V. Ravishankar is with the Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109. E-mail: ravi@eecs.umich.edu.*

Manuscript received 3 Jan. 1996; revised 6 June 1997.
Recommended for acceptance by D. Wile.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 101193.

time consuming, especially for complex protocols. The alternative approach is to generate protocol implementations from protocol specifications, or to interpret specifications directly [10], [11], [12], [13], [14], [15]. This approach also offers programmers tools to construct correct protocol implementations. However, the portions of a protocol implementation that can be generated depend on the environment and the specification language used. The efficiency of generated code is also a concern [18], [19]. The performance of generated implementations may need tuning because the assumptions and the implementation decisions made by a specification compiler may not be optimal for all environments. However, tuning performance can be error-prone under these circumstances because it requires understanding of generated code.

The two approaches described above represent opposite strategies, and are effective in different situations. Cicero combines them in a protocol construction language and allows the representation of the operational (execution) aspects of protocol implementations. We have found the performance of protocols generated using Cicero [6] to be excellent. Cicero also allows explicit control of parallelism and of complex multithreaded interactions in protocol execution. Thus, protocol implementations in Cicero may exploit multiprocessor architectures.

Cicero's language constructs can be used for implementing any protocol. However, they are particularly useful for protocols above the transport layer. More potential for concurrent I/O and parallelism exists in such protocols. Such parallelism is often present at coarser granualities, and the costs of managing concurrent threads are lower. Cicero also allows programmers to turn off the multithread support when it is not needed.

1.1 Cicero Use and Status

An implementation of Cicero has existed for several years. We have successfully used the language to specify and synthesize RPC protocols in real systems [6], [8], [9]. Such synthesis supports the operation of systems like Cygnus [5], which facilitates resource sharing in heterogeneous distributed networks. Cygnus clients may retrieve RPC descriptions from server sites and locally synthesize RPC translation agents. Since servers are not required to run any code, this synthesis facility enables clients to access services offered by insular servers. This approach also scales very well.

We have found the performance of our synthesized RPCs to be as good as that of native implementations, and often better [6]. This paradox arises since we are able to tailor the semantics and operation of protocols to the immediate needs. In contrast, native implementations must strive to accommodate all cases, limiting the scope of optimizations. Another reason for the superior performance of our protocol synthesis scheme is that we allow programmers direct control over implementations.

Cicero does not make protocol verification a primary objective, though we provide translations into existing formalisms, such as Petri nets. This does leave open the possibility of using existing tools for verification.

2 Design Rationale

Cicero is designed to allow constructive specifications of protocols. Thus, it complements languages like LOTOS [20], Estelle [21], and SDL [22], designed to support declarative specifications. Cicero differs fundamentally from such languages, for it allows programmers to explicitly synchronize and order events, as well as to control the manner in which event patterns cause code to be executed.

2.1 Protocol Construction Model

Protocols describe the interactions between computational threads, specifically in terms of actions to be performed when various events occur. Thus, Cicero makes *events* the fundamental protocol structuring concept. Protocol descriptions in Cicero are operational, but still high-level. They are hence called *protocol constructions*. A construction specifies the structure of events, and relationships between events and their influences on computational threads are specified. Thus Cicero manages threads and synchronization, but the programmer may control the exact semantics.

Assume that we wish to synchronize and display a pair of incoming video and audio frames. The following Cicero fragment accomplishes this task using the synchronization operator "∧".

```
when (video_in ∧ audio_in) :
      show_frame(build_frame(video_in, audio_in));
end;
```

The parenthesized expression following **when** is an *event pattern*, a notion borrowed from the language POST [7]. The pattern is activated when instances of the **video_in** and **audio_in** events arrive. The operator "∧" in the pattern forces corresponding instances of **video_in** and **audio_in** events to be synchronized. That is, the **show_frame** routine is executed only after corresponding instances of both events have arrived. As explained in Section 3.2.4, Cicero also provides some additional operators: "," to denote parallelism, "~" to denote sequencing, and "*" to denote iteration. These operators would be needed, for example, by a code fragment designed to synchronize and display a series of video and audio frames. A detailed example appears in Appendix A.

As protocols get more complex, interactions become more complex, and understanding them becomes harder. Special difficulties arise when dealing with synchrony, asynchrony and concurrency in event occurrence and protocol execution, especially when multithreaded execution is involved. A typical challenge in protocol debugging is to detect and correct a timing-related error which occurs nondeterministically in 5 percent of all test runs. Such bugs often arise when interactions among threads are not fully understood, and are usually most difficult to correct. Dealing with such situations requires support beyond that currently available. Dealing with complex protocol interactions will become increasingly important in the future as protocol implementations use multiple-thread support to implement multicast protocols, or exploit parallelism in protocol execution to increase throughputs. Cicero is intended specifically to address these difficulties.

2.2 Language and Execution Model

Cicero is not a full language, but is a language veneer extending existing languages. It is a small set of high-level control constructs sufficient to overcome the difficulties in protocol construction mentioned in Section 2.1. Since Cicero extends an existing programming language, Cicero users need learn only the few Cicero constructs, not an entirely new language. We have chosen C as our present target language for Cicero because it is portable and is widely used.

Multithreaded execution is important to increase throughput and to exploit Multiprocessor systems. We have chosen a restricted dataflow model [23] to exploit coarse-grained parallelism in protocol execution, and to model our event-driven style of execution faithfully. An obvious analogy can be drawn between events and data tokens in a restricted dataflow model, where token arrival serves as a mechanism for triggering/firing actions. We simply associate event patterns with code segments of the proper granularity. By changing the code granularity with patterns, we can change the granularity of parallelism exploited in Cicero. This capability is useful because it allows programmers to experiment with different granularities in tuning performance.

Using the dataflow model also has other advantages. First, it is mathematically well-defined [24] and well-understood. Second, it can be translated to/from other formal models (e.g., Petri nets [24]), 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.

2.3 Basic Language Abstractions and Semantics

A natural abstraction for specifying synchrony, asynchrony, and concurrency in protocol execution is an event-driven paradigm, where a protocol is viewed as a machine reacting to internal/external events or messages [20], [21], [25], [26], [27]. To describe complex relationships between events, we have borrowed the notion of *event patterns* from POST [7]. Event patterns use *event combinators* to recursively describe relationships between events. Cicero uses the three event combinators "$\wedge$", ",", and "~", to express synchronous, asynchronous, and sequential relationships between events, respectively.

2.3.1 Active and Passive Patterns

Event-pattern semantics can be classified into two types: active pattern semantics [7], [28] and passive pattern semantics [27]. Event patterns in Cicero have active pattern semantics. They behave like safeguards, locally guaranteeing the relationships between events, rather than passively detecting these relationships. That is, our event patterns are useful for implementing control flow, and for enforcing specified relationships between events before performing actions. The passive semantics used in many other event-driven languages [25], [27] would simply detect predefined relationships between events and then perform the indicated actions. For example, in Cicero, the event pattern ($a \sim b$) delivers event a before event b to its target code. This local delivery sequence is enforced even if event b actually occurs first. In other words, the delivery of b may need to be delayed to ensure that event a is delivered first. In contrast, passive semantics would trigger actions only if it is observed that 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 passive pattern-matching semantics. For example, patterns with active semantics can be used for flagging unusual situations, and can be of direct help to programmers in identifying obscure causes for bugs. In contrast, passive pattern semantics leave it entirely up to programmers to find causes, say, when some event patterns are not observed, or expected actions not executed. Such mismatches are often caused by subtle timing problems, in which case programmers may be forced to laboriously examine all possibilities to find the problem. In contrast, with active pattern semantics, the problem can often be avoided altogether, or simply corrected by patching in a few extra patterns. We believe active pattern semantics can make implementations more robust by ensuring correct behavior, and also reduce the possibilities of inadvertently introducing timing-related bugs while tuning performance.

2.4 The Communication Primitives

Several design choices present themselves for communication primitives in Cicero: remote events, new communication constructs, or library calls. The idea of using remote events sounds attractive because it retains the elegance of event-driven abstractions. 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 [29], [6]. 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. These concepts are identical to their counterparts in POST [7], which has inspired much of Cicero design.

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.

3.1.1 Lifecycle of Event Instances

An event instance can be generated (emitted), observed, and consumed. An event instance can be explicitly emitted by programs, 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.

3.1.2 Event Instance Attributes

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 state 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 in Cicero, as in POST [7], 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 "~", 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 rules for combining event patterns are defined as follows. Let E_1 and E_2 be event patterns and c denote a combinator.

1) An atom x is an event pattern.
2) (E_1) is an event pattern.
3) $(E_1)\ c\ (E_2)$ is an event pattern.
4) $(E_1\ c)^*$ is an event pattern. $(E_1\ c)$ is equivalent to $(E_1\ c)^*$.
5) $(E_1\ c)^*k$ is an event pattern, where $k \in \{1, 2, 3 \ldots\}$.

Event patterns constructed using only rules 1) to 3) are called *simple event patterns*. Event patterns having syntax 4) and 5) are called *repeating event patterns*. Specifically, event patterns in syntax 4) are called *infinite repeating event patterns*, and the ones in syntax 5) are called *finite repeating event patterns*. Infix operators have higher precedence than suffix operators, but parentheses may be used to alter this default precedence. To simplify the syntax, the "$*$" operator may be dropped from infinite repeating event patterns. Thus, if x, y, z are atoms, $(x \wedge y) \sim z$, is equivalent to $((x \wedge y) \sim z,)^*$ which is an infinite repeating event pattern, each of whose components comprises a pair of synchronized events $(x \wedge y)$ followed by a third event z. On the other hand, $(x \wedge y) \sim (z,)^*$ matches the pair $(x \wedge y)$ followed by an infinite sequence of zs.

3.2.2 Event Pattern Instances

Actions are triggered when an instance of the associated event pattern comes into existence. An instance of an event pattern comes into existence when the specified event instance relationship is effected, and the event pattern becomes *active*. If the sequencing operator "~" is not used, the pattern becomes active as soon as the matching event instances have arrived. When the sequencing combinator "~" is used, activations are serialized; that is, the next pattern instance does not become active until after the actions triggered by the previous activation have terminated. The event instances that make an event pattern active are called the *activating instances* of the event pattern, and 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 terminate. The pattern becomes *inactive* when no active instances exist. For a finite repeating event pattern, additional overflow actions may be associated with the pattern. The overflow actions are triggered when the number of pattern instances exceeds the specified limit. These pattern instances triggering the overflow actions are called *overflow* pattern instances.

3.2.3 Actions

An action is a sequence of statements executed when an appropriate event pattern instance comes into existence. To encourage coarse-grain parallelism, an action is executed as

a thread (light-weight process), which may subsequently create more threads (actions) in executing the action. 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 4.4). 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. Actions with higher execution priorities will be scheduled and executed first. If there is a tie, actions are scheduled and executed on a first-come-first-served basis. 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 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*. Since there may be several concurrent instances of a pattern, we allow the activated code to refer to the activating pattern instance (see Section 4.2.2). The semantics of *P* are as follows:

1) (*e*): *actions.* If *e* is an atom, the associated actions are triggered when an instance of the event represented by *e* comes into existence.
2) ((E)): *actions.* This pattern is equivalent to (E): *actions.*
3) ($E_1 \wedge E_2$): *actions.* This pattern requires that the associated action be triggered only when corresponding instances of both E_1 and E_2 come into existence.
4) (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 (the activating pattern instances may be identified as in Section 4.2.2).
5) ($E_1 \sim E_2$): *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.
6) (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.
7) ($E_1 \sim$)*: *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.
8) ($E_1 \wedge$)*: *actions.* This pattern will never trigger its associated action because it must await an infinite number of instances of E_1.
9) ($E_1 \wedge$)*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 overflow actions for multiple overflows execute concurrently. The description of overflow semantics is omitted from here on because they are identical for all finite repeating event patterns.
10) (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.
11) ($E_1 \sim$)*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 exists as a veneer over existing programming languages; the extension defined by Cicero comprises five language constructs: **emit**, **when**, **cond**, **bundle**, and **escape**. The **emit** construct is used to generate new event instances. The **when** construct controls the execution of associated target code. The **emit** construct creates event instances, while the **when** construct consumes them. The **cond** construct implements conditional branches, and helps distinguish active events in an event pattern. The modularization construct (**bundle**) provides scoping for events and encourages a modular programming style by factoring out Multithreaded subproblems. The **escape** construct allows programmers to include statements in the base language.

4.1 The EMIT Construct

The **emit** construct is used to generate or signal event instances. As explained in Section 3.1, instances of the same event are ordered globally, but no ordering is defined between instances of different events. After an event instance is emitted, it is dispatched to all the **when** constructs that can observe it. Event instance fields can be set only when emitting a new instance, and become read-only after the emission of the instance. If these field values are not set, the newly-emitted instance inherits field values from the previous instance. Default values are provided for the first instance. Examples of the use of the emit construct are:

```
emit e1: (val = 1, pri = 2); /* emit e1 with field values set */
emit e1; /* emit e1 with field values inherited */
```

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

4.2 The WHEN Construct

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_statements                   emit e3;
end:  overflow_statement            end:    emit e4;
```

In the above example, when *e*1 or *e*2 instance occurs, an instance of event pattern (*e*1, *e*2) is active to trigger the action. The activating instance with the highest priority determines the priority of the outer pattern. The activating instances are accessible while the action is executing, but are consumed and become inaccessible when the action is completed. Each **when** construct consumes only its own copy of activating instances. Because priorities are associated with event instances, actions corresponding to different instantiations of the same pattern may execute at different priorities. The overflow code is executed when the number of the event-pattern instances exceeds the maximum number of repeating instances allowed in a pattern (3, in this example).

4.2.1 Parallelism in Cicero

The **when** construct defines 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. To invoke threads synchronously, a programmer must use the **bundle** construct, which will be discussed in Section 4.4.

4.2.2 Pattern Instance Variables

As we saw in Section 3.2.4, several instances of a pattern may be active concurrently. A programmer often wishes to know what the activating instances are and how many times a pattern has been active. A mechanism called the pattern instance variable provides the instance number of an event pattern. A programmer can cause the pattern instance number to be placed in a variable *i* by appending "?*i*" to the pattern. For example, in:

```
when (timeout?i, recvmsg?j):
        cond (i > 0): emit wait;
             (j > 0): emit done;
        end;
end;
```

the pattern instance variables *i* and *j* contain the instance numbers of the event patterns *timeout* and *recvmsg*, respectively. If an event pattern is not active, its pattern instance variable will contain the number *zero*. Thus, a programmer can identify active pattern instances by checking the number of pattern instance variables. In the above example, this checking is done using the **cond** construct.

4.3 The COND Construct

Our **cond** construct is similar to the LISP **cond** construct (or the switch statement in C), except that when several conditionals evaluate to **true**, the statements associated with all the true conditions will be executed in order. These statements are designed to be executed sequentially to increase the granularity of parallelism and reduce concurrency control overhead. An example of a **cond** construct can be found in Section 4.2.2.

4.4 The BUNDLE Construct

The modularization construct in Cicero is the **bundle**; it is used to group **when** constructs into a module. The **bundle** construct defines the extent of visibility for event instances, and provides an environment for sharing variables among a group of **when** constructs. By default, events and event instances generated within a **bundle** are not visible outside it, nor are event instances outside a bundle visible inside. However, event instances from outside may be passed into a bundle through explicit parameter declarations in the bundle. It is also possible to define a pool of variables inside a **bundle** that are shared among all its **when** constructs.

The **bundle** construct is similar in a general way to abstract data types in other programming languages. It is useful in factoring out the structure common to a specific class of problem. 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.

Although the **bundle** construct is similar to the traditional procedure, it differs from the traditional procedure in the modularization unit, and event instance passing. Because each **when** construct has its own thread of control, the **bundle** construct may be viewed as a container for multiple threads in Cicero. Passing event instances is different from passing ordinary parameters. For example, instance numbers are not valid across scopes. Instead, a mechanism called *event channels* is provided to pass event instances.

Event channels are one-directional. When a caller passes an event instance to a callee through an event channel, a new event instance is re-emitted in the callee's scope with the field value copied from the caller's event instance. This direction of copying is reversed when the callee completes and passes out the latest output event instance to the caller. Again, a new event instance is emitted in the caller scope to carry the output field values from the callee. The call semantics for **bundle** constructs 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.

There are two other related special events: INIT and EXIT. The INIT event is automatically generated within the scope of a **bundle** each time it is invoked. For each **bundle**, only one observer (**when** construct) is allowed to observe INIT, and the associated action will be executed before any other **when** constructs in the **bundle**. INIT is designed to perform initialization of a **bundle**. EXIT is a special event that may be emitted to exit the entire program.

4.4.1 BUNDLE Declarations

The **bundle** declaration is similar to ordinary procedure declarations except that it also defines event channels. As with ordinary procedures, the **bundle** can also have formal parameters declared in a C-like syntax. However, parameters are passed using copy-in-copy-out semantics. Copy-in-copy-out semantics are used to be consistent with the

mechanics of passing values across address spaces, and to preserve isolation. The following is an example of the **bundle** construct declaration and invocation.

```
bundle send_recv (int id, msgtype; in e1; out e2, e3)
… /* body of the bundle */
end /**** end of bundle declaration ****/

/************ the caller ************/
int id; int msgtp;
event send_e, wait_e, recv_e;
…
when send_e:
        send_recv (id, msgtp, send_e, wait_e, recv_e);
end
```

In the above example, the **bundle** *send_recv* is declared with the reserved word **bundle**, and has three event channels: one input channel *e*1 and two output channels *e*2 and *e*3, which are declared with the reserved words **in** and **out**, respectively. The **bundle** *send_recv* is invoked when an instance of event *send_e* is observed in the caller. It can return instances of two output events, *wait_e* and *recv_e*. At invocation time, an event instance *e*1 is emitted in the callee's scope with the field value copied from the activating instance of *send_e*. When the **bundle** *send_recv* completes, the latest output event instance (*e*2 or *e*3) is passed out through the output event channel. And, the corresponding event instance (*wait_e* or *recv_e*) in the caller is emitted with the values copied out from the output event instances (*e*2 or *e*3).

4.5 The Escape Construct

The escape construct allows programmers to include target language statements in a Cicero construction. This 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 Fig. 3, line 10 for an illustration of its use.

Although the escape construct is convenient for programmers, it can make the verification process difficult if abused. Cicero relies on programmers to provide the correct assertions for escape components. In particular, Cicero assumes that the code in an escape component terminates.

5 The Cicero Communication Library

Although Cicero language constructs are not tied to any specific class of protocol implementations, the Cicero communication library is designed for constructing protocols above the transport layer. This communication library provides a set of transport services to facilitate point-to-point communication. In Cicero, the point-to-point notion is captured by *communication handles* which denote associations between two end points. Multiple communication handles can be created to model group communication. In addition, each handle can bind with a different protocol to support heterogeneous communication.

The communication paradigm between two end points is the send/receive model illustrated in Fig. 1. Our send/receive model consists of four essential activities: preparing a message, sending a message, receiving a message, and processing a message. The implementations of these activities are provided by the communication library. For example, the communication library provides functions for both reliable and unreliable message delivery. Also, the communication model is based on peer-to-peer communication, and the popular client-server model is treated as a special case in our peer-to-peer communication model.

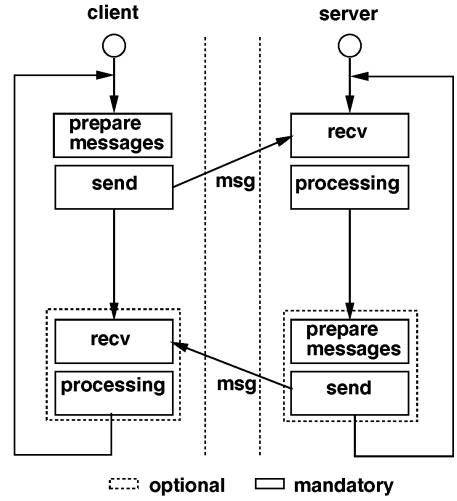


Fig. 1. The send/receive model in Cicero.

5.1 Message Types

Programmers describe a protocol by defining their own message types. They define a message type by providing a message-type ID and a function which will be invoked upon receiving a message with this type. Two types of messages may be defined: the **UDef** messages and the **UDefCtrl** messages. The **UDef** messages are used to invoke application-level functions. To perform an RPC, the client simply sends a **UDef** message to the server, and the specified server function will be invoked. When the function is executed, the results are sent back to the client using the same message type. Callbacks can also be implemented easily with **UDef** messages. For example, when a callback happens, the server agent simply sends a **UDef** message back to the client agent, and the function handling the callback will be invoked. The **UDefCtrl** messages are used to provide protocol control functions. For example, the programmers can use **UDefCtrl** messages to implement out-of-band control messages.

For the programmer's convenience, the communication library also provides five built-in control message types: **start**, **end**, **ping**, **ack**, and **error**. The **start** message is used when a client wishes to start a session. A *session* here is defined as the duration between a **start** and an **end** message, and it is identified by a session number. The **end** message is used to reset a channel when a session ends. The **ping** message is used to check whether or not the other end point (agent) is reachable and alive. The **ack** message is used to acknowledge the previous message. The **error** message is used to report an error. These five message types are summarized in Table 1.

TABLE 1
CONTROL MESSAGE TYPES

Type	Meaning
start	start a session
end	end a session
ping	are you there
ack	acknowledge a message
error	an error has occurred

6 AN EXAMPLE

In this section, we will use an example to illustrate how to use Cicero to describe an RPC protocol. The protocol described here is based on the client-server model, and only the client code segments are presented. More examples can be found in Appendix A.

6.1 An At-Least-Once RPC Protocol

In this example, we will construct an RPC protocol with at-least-once semantics by mirroring the extended finite state machine specification in Fig. 2 in Cicero code. If the server is up, but messages may be lost in transit, these RPC semantics cause the remote operation to be executed at least once.

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 FSM specification in Fig. 2 and the Cicero code segment, except that the *send_msg* event is replaced by a library call. 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 2.

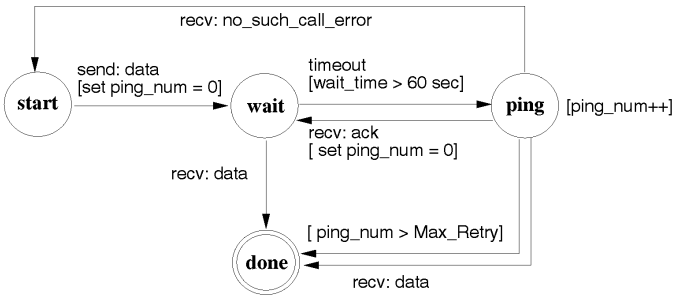


Fig. 2. An extended FSM diagram for at-least-once semantics.

TABLE 2
DESCRIPTION OF FUNCTIONS USED IN BUNDLE CLIENT_RPC()

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

The Cicero code segment is shown at Fig. 3. 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. If a reply is received, the **bundle** returns (line 26). If a timeout occurs, a *ping* event is emitted to send a ping message (line 31 and 39). A ping message is resent every 60 sec until either a reply is received, or the number of retries exceeds the limit Max_Retry (line 35 to 41). In the later case, the **bundle** returns with an error (line 36). It is possible 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).

7 IMPLEMENTATION AND PERFORMANCE

We have used Cicero to implement both customized and heterogeneous RPC protocols, and discuss these constructions and their performance in [6]. Here we focus on the performance of the language implementation. Our Cicero implementation includes the Cicero compiler and the Cicero runtime library. The Cicero compiler is implemented using Unix *lex* [30] and *yacc* [31]. Because the implementation of the compiler front-end is standard, we will focus on its runtime library.

Each **when** construct is compiled to a procedure and executed as a thread. When it observes an event instance, a **when** is put in a queue called *task queue*, which is created when its enclosing **bundle** is invoked. For each **bundle** invocation, a *dispatch thread* is created to run the **when** construct at the head of the task queue. All runnable threads are scheduled using a round-robin scheduling policy. The dispatch thread terminates when a RETURN event is observed, and the **bundle** returns.

We have isolated the implementation of event patterns into a separate package which may be used directly when extending other languages with Ciciero constructs. Details on implementing event patterns, their formal semantics, and their translations into LOTOS [20] and Petri nets appear in [32].

Since thread packages are often platform-dependent, the Cicero runtime library does not provide its own thread package. To improve portability, it provides interfaces to existing thread packages instead. The minimal functionality Cicero expects from a thread package are features for starting and terminating a thread, and for guaranteeing mutul exclusion. Currently, the runtime library supports interfaces to four thread packages, SUN LWP [33], Brown Threads [34], Unix Cthreads [35], and Mach Cthreads [36]. If no thread package is available, the Cicero runtime operates by calling the **when** constructs in the task queue in order within the dispatch routine. In the runtime library, this case is encapsulated by a special thread package interface called *NOLWP*. *NOLWP* is also used for protocol implementations that do not benefit from parallelization and wish to avoid thread management overhead.

The Cicero runtime uses locks to control access to the shared runtime data structures and the sequential execution of target instances (when sequential combinators are used).

```
 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 }
```

Fig. 3. Cicero code segment for at-least-once semantics.

For example, each event is represented by a queue of event instances, and has a read and a write lock associated with it. Depending on the operation (read/write), the proper lock must be obtained before accessing the queue. In Cicero, a multiple-reader/one-writer policy is implemented for maximizing concurrent read access to the event instance fields stored in the queue. The single writer policy serializes all updates to event queues, which guarantees the global ordering of event instances. Starvation is prevented by alternating the priority of readers and writers. Currently, locks are implemented using monitors/semaphores provided by the underlying thread package. However, in the case of *NOLWP*, locks are not necessary since there is no pre-emption.

The Cicero runtime overhead arises from two sources: the control mechanism and the underlying thread package. The control mechanism overhead consists of the overhead for emitting an event instance and executing event patterns. The overhead for event instance emission includes 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. The overhead for pattern execution includes the overhead for evaluating the status of an event pattern, computing the execution priority, and updating the status. The overhead imposed by the underlying thread package is the overhead for the locking mechanism and the thread management. Table 3 shows the instance emission and the pattern execution overhead with and without thread packages. The Cicero runtime overheads are measured on a SUN Sparc 1 workstation running SunOS 4.1.1.

TABLE 3
CICERO RUNTIME OVERHEAD

Overhead Type	NOLWP (msec)	Unix Cthread (msec)	SUNLWP (msec)
Instance Emission	0.03	0.13	0.19
Pattern Execution	0.02	0.04	0.05
Total	0.05	0.17	0.24

The basic runtime overhead for emitting an event instance and triggering target code is about 0.05 msec (0.03 msec for instance emission and 0.02 msec for pattern execution). However, with a thread package, this overhead can range from 0.17 to 0.24 msec. This increase is due to the extra overhead for locking the sharing runtime data structures and managing threads (i.e., the thread scheduling and context switch).

As the size of an event pattern gets larger, the overhead for pattern execution grows, while instance emission overhead remains constant. The growth of overheads is shown in Fig. 4. The size of an event pattern is measured by the number of nodes in an event pattern, which can be either an event or a combinator. For example, the size of the event pattern ($e1 \wedge e2$) is 3.

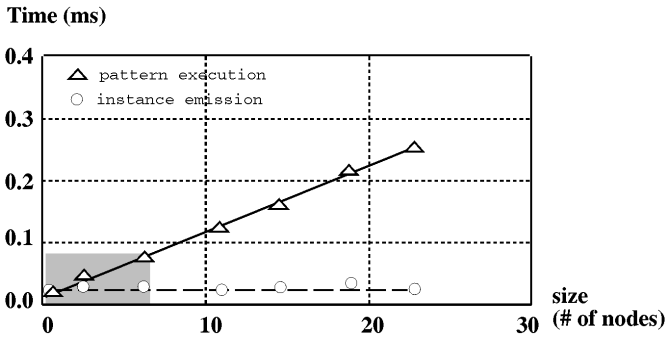


Fig. 4. The overhead growth for pattern evaluation.

As shown in Fig. 4, the rate of growth for pattern execution overhead is moderate. Since most event pattern sizes are well below 10 (the shaded area), the growth of the overhead will not be a significant factor in the overall performance. For protocols above the transport layer, the overhead is usually well below 5 percent. If the *NOLWP* option (no thread support) is specified, the language overhead is negligible (< 1 percent) compared to the other delays. We expect the overhead to diminish further if running on a Multiprocessor architecture, where we can parallelize the pattern execution using divide-and-conquer strategies.

8 RELATED WORK

Cicero is a protocol construction language, useful for either handcrafting or synthesizing protocol implementations. Cicero facilitates the handcrafting of protocol implementations by providing better control semantics through event patterns. Event patterns enable programmers to control asynchronous, synchronous, and concurrent activities as well as exploit parallelism in protocol execution. In addition, event patterns can be translated into other formal models, so that existing techniques may be used to verify the implementation. Cicero can also be used as an executable specification language for generating protocol implementations. Being an executable specification language, Cicero allows programmers to exercise direct control over the nature of synthesized implementations. Also, programmers can include customized code in the synthesized implementations directly. Otherwise, customization must be accomplished in a separate step after the code is generated, and modification to the generated code may be required.

The most novel aspect of Cicero is its approach to the integration of existing notions and abstractions to fulfill its design goals. Cicero integrates the following ideas.

Active Pattern Matching. The semantics of active pattern matching are borrowed from POST [7]. However, while POST is a general pattern-driven dataflow language, Cicero tailors its ideas to run as a language veneer for protocol construction. Although the semantics of our event patterns are superficially similar to those of *path expressions* [28] and *data path expressions* [27], the usage and semantics of our event patterns expressions are very different from theirs. 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, an event pattern can indicate that some target code be triggered only when both event $e1$ and $e2$ occur. Such constraints cannot easily be expressed by path expressions alone. Data path expressions have been used for detecting incorrect behavior in concurrent programs [27], and provide passive pattern-matching semantics. While passive pattern semantics are useful for debugging purposes, it may not be easy to locate the causes from the detected symptoms. Active pattern matching can alleviate this difficulty by fixing the symptoms directly.

Event-Driven Abstraction. Our event-driven model is similar to the notion of *event flows* in ESTEREL [25]. However, we have extended the simple event-driven model to include event patterns, allowing programmers to express complex relationships between events for controlling the execution. The Actor model [26] is another well-known model for concurrent computation in distributed systems. In Actor, as in Cicero, no ordering is defined for unrelated events. However, we globally order the instances of the same event to facilitate the coordination among multiple threads.

Separation of Local and Remote Communication Mechanisms. In the past, much work has been done on integrating local and remote communication into one abstraction. BSD *sockets* [37] and remote procedure calls (RPC) [38] are well-known examples. However, because it is designed for protocol construction, and not for general distributed computing, Cicero provides different communication mechanisms for local and remote communication. Local communication (within the same address space) is accomplished by emitting/observing events, and remote communication (across address spaces) is supported by a set of library routines based on a message passing model. This separation between the local and remote communication mechanism

allows programmers to provide different semantics for local and remote communication. This is necessary in practice because 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 [19]. Currently, Cicero libraries exist for implementing protocols above the transport-layer, such as RPC.

Restricted Dataflow Execution Model. The dataflow model [23] is well-known, and many languages [7], [39] and machines [40], [41] have been designed based on it. Instead of describing data-dependency or data-access disciplines, Cicero uses the dataflow model to describe event-driven execution, allowing programmers to explore parallelism in different granularities by changing the amount of computation in the associated actions. The dataflow model also provides the formalism to allow Cicero to be translated to/from other protocol specification models (e.g., Petri nets [24]), so that existing tools may be used to facilitate constructing protocols.

Executable Specifications. Several Formal Description Techniques (FDT), like LOTOS [20], Estelle [21], and SDL [22], have been developed to specify protocol behavior formally. Much research has been conducted in automatically generating protocol implementations from these FDTs [13], [14], [15], [41]. However, the protocol implementations generated by these means are generally in the form of skeletons which must be filled in by programmer code [19]. Also, the efficiency of generated code is a concern [18]. 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.

9 Conclusions

Using Cicero constructs for implementing protocols offers the following advantages under an acceptable overhead:

- Event patterns can control synchrony, asynchrony, and sequentiality in protocol execution, and provide a better implementation paradigm than thread packages alone.
- Event patterns can be translated to/from other models/languages describing protocols. These translations make it possible to use existing tools/methods to facilitate protocol implementation and verification.
- Cicero provides support for multiple-thread execution under a dataflow model, so that parallelism in protocol implementation can be fully exploited.
- Cicero encourages coarse-grain parallelism by combining smaller event patterns into larger ones, so that the thread management overhead can be amortized.

We have used Cicero to describe different RPC protocol implementations, so that gateway agents may be synthesized to interconnect the client and server programs using different RPC protocols [5], [29], [6].

Appendix A – Cicero Examples

Example 1. A Two-Phase Commit Protocol For N Servers.

This example describes a two-phase commit protocol involving N servers. The client starts by sending a request to all the servers with at-most-once failure semantics (as described in Section 6.1). If any server fails, the client multicasts an abort message to all servers. After ensuring that all servers have received the request, the client issues a commit request to all servers, and waits for an acknowledgment from each server. If all servers have acknowledged the commit request, the request is completed. However, if a timeout occurs before receiving a reply from a server, the client must resend the commit request to the server. This retry continues until either the client receives a reply or the number of retries exceeds the limit. In the later case, the client aborts the request by multicasting an abort message to all servers. The protocol is captured by the the Cicero code segment in Fig. 5, which consists of nine **when** constructs:

1) The first **when** construct (line 7 to 13) initializes a data structure for recording the number of retries and emits a *call_server* event to start sending requests.
2) The second **when** construct (line 15 to 21) delivers the N requests concurrently with at-most-once failure semantics (*client_rpc*()). This multicast is accomplished by emitting another *call_server* event upon the invocation of this **when** construct (line 16).
3) The third **when** construct (line 23 to 28) is responsible for collecting replies from all servers to ensure the servers have received the request. If no error has occurred, it emits a *commit* event to start sending the commit request, and a *recv* event to start threads for receiving the acknowledgment from servers.
4) The fourth **when** construct (line 30 to 34) multicasts the commit request, and starts the timer for each commit request by emitting a *wait* event. The destination of each commit request is saved in the value field of the *wait* event.
5) The fifth **when** construct (line 36 to 39) implements the timer. When a timeout occurs, it sends a *retry* event to resend the commit request. The destination of the retry is obtained from the triggering *wait* event instance.
6) The sixth **when** construct (line 41 to 47) performs the retry of the commit request. It records the number of retries for each destination. If the number of retries exceeds the limit (Max_Retry), an abort event is emitted.
7) The seventh **when** construct (line 49 to 55) will start up N threads to receive the commit acknowledgment from servers. Upon receiving an acknowledgement, it emits an *ack* event, which will be collected by the eighth **when** construct.
8) The eighth **when** construct is responsible for counting the number of acknowledgment from servers. If enough acknowledgment is collected, the **bundle** returns.
9) The ninth **when** construct is triggered when an abort event is emitted. It multicasts abort requests to all servers.

```
bundle two_phase_commit(CC_handle_t *handle, int num_sv, CC_msg_t *msg)
{
    int   err, ret, num_retry[MaxHandles];
    long  wait_time;
    event call_server, server_ok, recv, wait, retry, ack, commit, abort;

    when (INIT):
        cond (num_sv >= 1):
                { int i; for (i=0; i < num_sv; i++) num_retry[i]=0; }
                emit call_server;
             otherwise: emit Return:(val=ERR);
        end;
    end;

    when (call_server?i):       /* multicast the transaction */
        cond (i < num_sv): emit call_server; end;
        err = client_rpc(handle[i-1],msg);  /* at-most-once RPC */
        cond (err == OK) emit server_ok;
             otherwise:  emit abort;
        end;
    end;

    when (server_ok?j):
        cond (j == num_sv): /* all servers are ready to commit */
             emit commit;       /* start sending out commit messages */
             emit recv;         /* wait for reply of commit msg. */
        end;
    end;

    when (commit?m):            /* multicast commit messages */
         cond (m < num_sv): emit commit; end;
         CC_send_ctrl_msg(handle[m-1],COMMIT);
         emit wait:(val=m-1);   /* start timer for this handle */
    end:

    when (wait):
         CC_wait(wait_time);
         emit retry:(val=wait.val);
    end;

    when (retry):                /* resent commit message */
         cond (num_retry[retry.val] <= Max_Retry):
                 CC_send_ctrl_msg(handle[retry.val],COMMIT);
                 { num_retry[retry.val]++; }
                 emit wait:(val=retry.val);
              otherwise: emit abort;
    end:

    when (recv?k):
        cond (k < num_sv): emit recv; end;
        ret = CC_recv_ctrl_msg(handle[k-1]);
        cond (ret == ACK): { num_retry[k] = 0; } emit ack;
             otherwise: emit abort;
        end;
    end;

    when (ack?q):
        cond (q == num_sv): emit Return:(val=OK); end;
    end;

    when (abort?n):
        cond (n <= num_sv):
                CC_send_ctrl_msg(handle[n-1],ABORT);
                emit abort;
             otherwise: emit Return:(val=ERR);
        end;
    end;
}
```

Fig. 5. Cicero code segment for N-server two-phase commit protocol.

EXAMPLE 2. A Video/Audio Server.

In this example, we will use Cicero to describe a video/audio server, which retrieves and sends both video and audio data to its client through two stream connections. The video and audio streams are stored at separate files, and are sent through separate channels. The stored video is encoded using the MPEG standard [43], which is based on a scheme to predict motion from frame to frame in the temporal direction. Frame prediction in MPEG is based upon *intra frames* (still images) in the video. To facilitate parallel MPEG decoding, additional indices are built to locate these frames. Thus, the entire video can be partitioned into many *frame sets*, each consisting of an intra frame and a sequence of predicted frames. Audio data are indexed synchronously to the video stream. Before sending video/audio data, additional synchronization markers are introduced while decoding the video and audio data. These synchronization markers facilitate the synchronization between the corresponding video and audio frame sets and controlling the flow of outgoing video and audio data streams. Thus, they are sent to the client along with the video and audio data as a part of the communication protocol. This scheme is illustrated in Fig. 6.

Cicero code segments for our video/audio server are organized into three **bundles** (see Figs. 7 and 8).

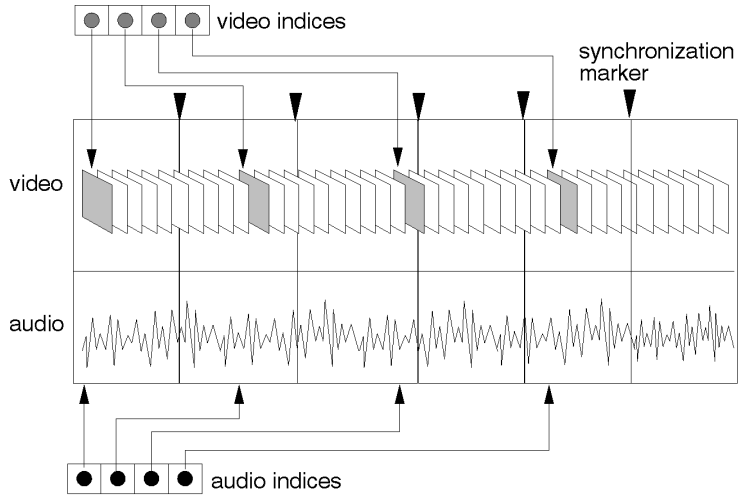


Fig. 6. Illustration of video/audio data.

```
 1 bundle proc_video_request(CC_handle_t handle[], char *name)
 2 {
 3   addr_t     vindx[MaxBlk], aindx[MaxBlk];
 4   int        num_blk, p, q, vret1, vret2, ret;
 5   event      proc_video1, proc_video2, proc_audio;
 6   event      v_rdy1, v_rdy2, a_rdy;
 7   CC_msg_t  *alist, *vlist1, *vlist2;
 8
 9   when (INIT):
10      ret = read_index(name, vindx, aindx, &num_blk);
11      cond (ret == OK): emit proc_video1; emit proc_video2;
12                        emit proc_audio;
13           otherwise: emit Return:(val=E_INDEX);
14      end;
15   end;
16
17   when (proc_video1?i):
18      { p = 2*(i-1); }
19      vret1 = proc_frame_set(VIDEO,vindx[p],vindx[p]-vindx[p-1],&vlist1);
20      cond (vret1 != OK): emit Return:(val=vret1);
21             otherwise: emit v_rdy1:(val=vlist1);
22      end;
23   end;
24
25   when (proc_video2?j):
26      {  q = 2*j-1; }
27      vret2 = proc_frame_set(VIDEO,vindx[q],vindx[q]-vindx[q-1],&vlist2);
28      cond (vret2 != OK): emit Return:(val=vret2);
29           otherwise: emit v_rdy2:(val=vlist2);
30      end;
31   end;
32
33   when (proc_audio?k):
34      ret = proc_frame_set(AUDIO,aindx[k-1],aindx[k]-aindx[k-1],&alist);
35      cond (ret != OK): emit Return:(val=ret);
36           otherwise: CC_set_udef_sendmsg(handle[1],alist);
37                      emit a_rdy;
38      end;
39   end;
40
41   when (((v_rdy1?s ~ v_rdy2?t ~)* ^ a_rdy)?m ~):
42      cond (m > num_blk): emit Return:(val=OK); end; /* done */
43      cond (s > 0): CC_set_udef_sendmsg(handle[0],v_rdy1.val);
44                    emit proc_video1;
45           (t > 0): CC_set_udef_sendmsg(handle[0],v_rdy2.val);
46                    emit proc_video2;
47      end;
48      emit proc_audio;
49      ret = send_video_audio_frame_set(handle);
50      cond (ret != OK): Return:(val=ret); end;
51   end;
52 }
```

Fig. 7. Cicero code segment for proc_video_request.

```
 1 bundle send_video_audio_msg(CC_handle_t handle[])
 2 {
 3   CC_msg_t *video_msg_head, *v_msg;
 4   CC_msg_t *audio_msg_head, *a_msg;
 5   event    send_video, send_audio;
 6   event    video_sync, audio_sync, video_end, audio_end;
 7
 8   when (INIT):
 9      emit send_video; emit send_audio;
10   end;
11
12   when (send_video~):
13      v_msg = CC_get_next_send_msg(handle[0]);
14      cond (CC_msg_type(v_msg) == DATA):
15                CC_send_udef_msg(handle[0],v_msg);
16                emit send_video;
17           (CC_msg_type(v_msg) == SYNC):
18                CC_send_ctrl_msg(handle[0],SYNC);
19                emit video_sync;
20           (CC_msg_type(v_msg) == END): emit video_end;
21           otherwise: emit Return:(val=E_MSGTP);
22      end;
23   end;
24
25   when (send_audio~):
26      a_msg = CC_get_next_send_msg(handle[1]);
27      cond (CC_msg_type(a_msg) != DATA):
28                CC_send_udef_msg(handle[1],a_msg);
29                emit send_audio;
30           (CC_msg_type(a_msg) == SYNC):
31                CC_send_ctrl_msg(handle[1],SYNC);
32                emit audio_sync;
33           (CC_msg_type(a_msg) == END): emit audio_end;
34           otherwise: emit Return:(val=E_MSGTP);
35      end;
36   end;
37
38   when (video_sync ^ audio_sync):  /* adjust for synchronization */
39      emit send_video; emit send_audio; /* continue sending data */
40   end;
41
42   when (video_end ^ audio_end): /* done */
43      CC_free_sendmsg(handle[0]); CC_free_sendmsg(handle[1]);
44      emit Return:(val=OK);
45   end;
46 }
```

Fig. 8. Cicero Code Segment for send_video_audio_msg().

```
 1 bundle proc_frame_set(int data_type, addr_t start_addr, long length
 2                       CC_msg_t **msglist)
 3 {
 4   event             proc_data;
 5   addr_t            data_addr;
 6   CC_msg_t          *msg;
 7   char              buf[MaxFrameSize];
 8   unsigned long     remain_size, size;
 9
10   when (INIT):
11      { data_addr = start_addr; remain_size = length; }
12      emit proc_data;
13   end;
14
15   when (proc_data~)*MaxNumFrames:
16      cond (data_type == VIDEO):
17               data_addr = video_frame_decode(data_addr, &size, buf);
18           (data_type == AUDIO):
19               data_addr = audio_frame_decode(data_addr, &size, buf);
20           otherwise: emit Return:(val=E_DATATYPE);
21      end;
22      remain_size -= size;
23      cond (size == 0): emit Return:(val=OK);
24           (size > 0): msg = CC_create_new_msg(size, buf);
25                       CC_append_msg(msglist, msg);
26                       emit proc_data;  /* process next block */
27           (remain_size <= 0): emit Return:(val=OK);
28      end;
29   end: Return:(val=E_OVERFLOW);
30 }
```

Fig. 9. Cicero code segment for proc_frame_set().

proc_video_request(). The **bundle** *proc_video_request*() is the top-level driver for processing a request. It reads indices and sets up threads for processing video and audio (line 9 to 15). In particular, two video threads are used to process the odd-number and the even-number frame sets concurrently to increase through-put (line 17 to 23 and line 25 to 31). Finally, it synchronizes the processed video and audio data before sending them out (line 41 to 51). To ensure both video and audio data are ready in the desired sequence, it uses the event pattern $((v\_rdy1 \sim v\_rdy2 \sim)^* \wedge a\_rdy)$ to restrict the execution of the actions. This event pattern ensures two conditions before triggering the actions: (1) the even numbered and odd numbered frame sets, which are processed concurrently, must be delivered in the frame-number order 0, 1, 2, …, and (2) the corresponding audio frame set must be ready and synchronized.

send_video_audio_frame_set(). The **bundle** *send_video_audio_msg*() is called by *proc_video_request*() to send processed video and audio data messages (see Fig. 8). It also sends the synchronization markers and performs flow control by synchronizing the video and audio sending activities periodically. Two threads (**when** constructs) are used to send video and audio data concurrently through separate channels (lines 12–23 and lines 25–36). Both **when** constructs have identical structure. They continue sending data messages until they encounter either a synchronization marker (SYNC) or the end of the message list (END). When they encounter a synchronization marker, they emits a flow control event (*video_sync* or *audio_sync*) after sending out the synchronization marker. If they reach the end of the message list, a *video_end* event is emitted. The flow control is accomplished by synchronizing both *video_sync* and *audio_sync* events before it restarts sending messages. It adjusts the flow of the outgoing video and audio streams, such that both video and audio channels will be synchronized. The last **when** construct (lines 42–45) frees up resources when no more message are to be sent.

proc_frame_set(). The **bundle** *proc_frame_set*() is called by *proc_video_request*() to decode a video or an audio frame set into a list of messages (see Fig. 9). It consists of two **when** constructs. The first **when** construct (line 10 to 13) initializes variables, and emits a *proc_data* event to start processing the data. The second **when** construct (line 15 to 29) decodes individual frame and appends the resulting message to a list, and a given flag *data_type* is used to select the proper decoding routine. A finite repeating pattern is used here to detect the overflow condition.

ACKNOWLEDGMENT

This work was partly supported by NASA's Socioeconomic Data and Applications Center operated by the Consortium for International Earth Sciences Information Networking.

REFERENCES

[1] K.S. Yap, P. Jalote, and S. Tripathi, "Fault Tolerant Remote Procedure Call," *Proc. Eighth Int'l Conf. Distributed Computing Systems*, San Jose, Calif., pp. 48–54, June 1988.

[2] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Trans. Programming Languages and Systems*, vol. 5, no. 3, pp. 381–404, July 1983.

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

[4] C. Maeda and B.N. Bershad, "Protocol Service Decomposition for High-Performance Networking," *Proc 14th ACM* Symp. Operating Systems Principles, *Comm. ACM*, Dec. 1993.

[5] R.N. Chang and C.V. Ravishankar, "A Service Acquistion Mechanism for Server-Based Heterogeneous Distributed Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 2, pp. 154–169, Feb. 1984.

[6] Y. Huang and C.V. Ravishankar, "Designing An Agent Synthesis System for Cross RPC Communication," *IEEE Trans. Software Eng.*, vol. 20, no. 3, Mar. 1994.

[7] C.V Ravishankar, "*POST: A Language for Dataflow Programming*," PhD thesis, Computer Sciences Dept., Univ. of Wisconsin, Madison, 1987.

[8] Y. Huang and C.V. Ravishankar, "URPC: A Toolkit for Prototyping Remote Procedure Calls," *The Computer J.*, vol. 39, no. 6, 1996.

[9] Y. Huang and C.V. Ravishankar, "Secure Synthesis and Activation of Protocol Translation Agents," *Distributed Systems Eng. J.*, vol. 4, 1997.

[10] T.P. Blumer and D.S. Sidhu, "Mechanical Verification and Automatic Implementation of Communication Protocol," *IEEE Trans. Software Eng.*, vol. 12, no. 8, pp. 827–843, Aug. 1986.

[11] G. v. Bochmann, G. Gerbert, and J.M. Serre, "Semiautomatic Implementation of Communication Protocols," *IEEE Trans. Software Eng.*, vol. 13, no. 9, pp. 989–999, Sept. 1987.

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

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

[14] S.T. Vuong, A.C. Lau, and R.I. Chan, "Semiautomatic Implementation of Protocols Using an Estelle-C Compiler," *IEEE Trans. Software Eng.*, vol. 14, no. 3, pp. 384–393, Mar. 1988.

[15] G. Albertengo, S. Forno, and A. Fumagalli, "TOP/PDT: A Toolkit for Development of Communication Protocols," *IEEE J. Selected Areas in Communications*,. vol. 8, no. 9, pp. 1,763–1,770, Dec. 1990.

[16] H.J. Burkhardt, H. Eckert, and A. Giessler, "Testing of Protocol Implementations—A Systematic Approach to Derivation of Test Sequences from Global Protocol Specifications," M. Diaz, ed., *Protocol Specification, Testing and Verification V (IFIP/WG 6.1)*. Amsterdam, The Netherlands: North-Holland, 1986.

[17] J. Favreau and Jr. R.J. Linn, "Automatic Generation of Test Scenario Skeletons from Protocol Specifications Written in Estelle," B. Sarikaya and G. v. Bochmann, eds., *Protocol Specification, Testing and Verification VI (IFIP/WG 6.1)*. Amsterdam, The Netherlands: North-Holland, 1987.

[18] L. Svobodova, "Implementing OSI Systems," *IEEE J. Selected Areas in Communications*, vol. 7, no. 7, pp. 1,115–1,130, Sept. 1989.

[19] M.B. Abbott and L.L. Peterson, "A Language-Based Approach to Protocol Implementation," *IEEE/ACM Trans. Networking*, vol. 1, no. 1, pp. 4–19, Feb. 1993.

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

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

[22] CCITT, *Specification and Description Language—Recommendation Z.100*. 1986.

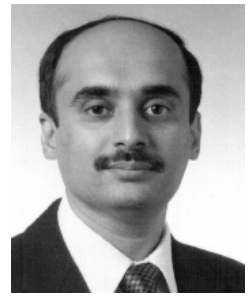
[23] R.M. Karp and R.E. Miller, "Properties of a Model for Parallel Computation: Determinacy, Termination, Queueing," *SIAM J. of Applied Math.*, pp. 1,390–1,411, Nov. 1966.

[24] K.M. Kavi, B.P. Buckles, and U.N. Bhat, "Isomorphism Between Petri Nets and Dataflow Graphs," *IEEE Trans. Software Eng.*, vol. 13, no. 10, pp. 1,127–1,134, Oct. 1987.

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

[26] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, Mass.: MIT Press, 1986.
[27] W. Hseush and G.E. Kaiser, "Modeling Concurrency in Parallel Debugging," *Proc. Second ACM SIGLPAN Symp. Principles and Practice of Parallel Programming*, pp. 11–20, Mar. 1990.
[28] R.H. Campbell and A.N. Habermann, "The Specification of Process Synchronization by Path Expression," *Lecture Notes in Computer Science 16*, pp. 89–102. New York: Springer-Verlag, 1974.
[29] Y. Huang and C.V. Ravishankar, "Accommodating RPC Heterogeneities in Large Heterogeneous Distributed Environments," *Proc. 26th Hawaii Int'l Conf. System Sciences, HICSS-26*, Jan. 1993.
[30] M. E. Lesk and E. Schmidt, "Lex—A Lexical Analyzer Generator," *Unix Programmer's Supplementary Documents, vol. 1*, 1986.
[31] S.C. Johnson, "Yacc: Yet Another Compiler-Compiler," *Unix Programmer's Supplementary Documents, vol. 1*, 1986.
[32] Y.M. Huang, "Constructive Specification and Synthesis of Agents for Custom and Cross-RPC," PhD thesis, Electrical Eng. and Computer Science Dept., Univ. of Michigan, Ann Arbor, 1994.
[33] Sun Microsystems, *Programming Utilities and Libraries*. Mar. 1990.
[34] T.W. Doeppner, "A Threads Tutorial," Technical Report CS-87-06, Dept. of Computer Science, Brown Univ., Mar. 1987.
[35] 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 Instit. of Tech., 1991.
[36] E.C. Cooper and R.P. Draves, *C Threads*. Dept. of Computer Science, Carnegie Mellon Univ., July 1987.
[37] S. Sechrest, "An Introductory 4.3BSD Interprocess Comm. Tutorial," *Unix Programmer's Manual Supplementary Documents 1*, 1:PS1:7–1—PS1:–25, 1986.
[38] A.P. Birrell and B.J. Nelson, "Implementing Remote Procedure Call," *ACM Trans. Computer Systems*, vol. 2, no. 1, pp. 39–59, Jan. 1984.
[39] W.W. Wadge and E.A. Ashcroft, *Lucid, the Dataflow Programming Language*. Academic Press, United Kingdom, 1985.
[40] W.W. Hwu and Y. Patt, "HPSm, A High Performance Restricted Data Flow Architecture having Minimal Functionality," *Proc. The 13th Int'l Symp. Computer Architecture Conf.*, pp. 297–306, June 1986.
[41] M. Johnson, *Superscalar Microprocessor Design*. Englewood Cliffs, N.J.: Prentice Hall, 1991.
[42] A. Valenzano, R. Sisto, and L. Ciminiera, "Rapid Prototyping of Protocols from LOTOS Specification," *Software—Practice and Experience*, vol. 23, no. 1, pp. 31–54, Jan. 1993.
[43] D. Le Gall, "MPEG: A Video Compression Standard for Multimedia Applications," *Comm. ACM*, vol. 34, no. 4, pp. 46–58, 1991.

Yen-Min Huang received the BS degree in chemical engineering from National Taiwan University, Taipei, in 1982, and dual MSE degrees in chemical engineering and CICE (computer information and control engineering) from the University of Michigan, Ann Arbor, in 1986. Dr. Huang received the PhD degree in computer science and engineering from the University of Michigan, Ann Arbor, at the end of 1993. He is now with IBM, Research Triangle Park, North Carolina. His current research interests include distributed systems and computer networks.



Chinya V. Ravishankar received the MS and PhD degrees in computer sciences from the University of Wisconsin at Madison in 1986 and 1987, respectively. He has been with the Electrical Engineering. and Computer Science Department at the University of Michigan, Ann Arbor, since 1986. His teaching and research at the University of Michigan have been in the area of programming languages, databases, and distributed systems. Dr. Ravishankar founded the Software Systems Research Laboratory at the University of Michigan. He is actively involved in the Engineering Research Center on Reconfigurable Machining Systems, and is a member of the Real-Time Computing Laboratory at the University of Michigan. His present research interests include large-scale distribution, heterogeneity, protocol synthesis, real-time systems, and spatial databases and data warehousing. Dr. Ravishankar is a senior member of the IEEE, a member of the Association for Computing Machinery, and a member of the IEEE Computer Society.