COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY STANFORD, CA 94305-4055

PARTIAL ORDERINGS OF EVENT SETS AND THEIR APPLICATION TO PROTOTYPING CONCURRENT TIMED SYSTEMS

David C. Luckham
James Vera
Doug Bryan
Larry Augustin
Frank Belz

Technical Report: CSL-TR-92-515
(Program Analysis and Verification Group Report No. 59)

April, 1992

This research was supported by the Defense Advanced Research Projects Agency/Information Systems Technology Office under the Office of Naval Research contract N000 14-90- J1232, and by the Air Force Office of Scientific Research under Grant AFOSR91-0354.

Partial Orderings of Event Sets And Their Application to Prototyping Concurrent Timed Systems

David C. Luckham James Vera Doug Bryan Larry Augustin Frank Belz

Technical Report: CSL-TR-92-515
Program Analysis and Verification Group Report No. 59

April 1992

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Rapide *is a concurrent object-oriented language specifically designed for prototyping large concurrent systems. One of the principle design goals has been to adopt a computation model in which the synchronization, concurrency, dataflow, and timing aspects of a prototype are explicitly represented and easily accessible both to the prototype itself and to the protofyper. This paper describes the partially ordered event set (poset) computation model, and the features of* **Rapide** *for using posets in reactive prototypes and for automatically checking posets. Some critical issues in the implementation of* **Rapide** *are described and our experience with them is summarized. An example prototyping scenario illustrates uses of the poset computation model.*

Key Words and Phrases: Rapide, partial orders, prototyping, programming languages

Contents

List of Figures

Chapter 1

Introduction

Prototypes of complex systems usually embody only certain features of the ultimate system, and abstract or omit many details in order to permit rapid construction. The objective of *prototyping* (building and analyzing prototypes) is to study various alternative decisions that must be taken in planning and developing a system. Typical examples of questions that are commonly investigated by prototyping include adequacy of system requirements, choosing between alternative system designs, developing precise specifications for components, and measuring performance as a function of design or of computing resources. Essentially, prototyping is an experimental activity in which the information provided by executing a prototype is a critical factor in making these kinds of decisions.

RAPIDE is a concurrent object-oriented language specifically designed for prototyping large concurrent systems. Since such systems consist of many components executing concurrently under various timing constraints, the *computation model* has been one of our major focuses. Our design goal has been to adopt a computation model in which the synchronization, concurrency, and timing aspects of a prototype are explicitly represented in computations and easily accessible to the prototyper. This paper describes the information provided by **RAPIDE** *poset* computations, and the features of **RAPIDE** for constructing prototypes that produce posets and react to posets. In particular, **RAPIDE** is designed to encourage (but not enforce) use of formal specifications; their application to automatically check posets for violations of requirements is illustrated. Some critical issues in the implementation of **RAPIDE** are described and our experience with them is summarized. An example prototyping scenario illustrates uses of the computation model. An outline of an early version of **RAPIDE** (then called *Reality)* is given in [BL90].

Space limitations on this version of the paper preclude our giving adequate description and references to prior related work, which will be in the final version.

Chapter 2

Computation Model

RAPIDE computations are sets of events with partial orderings on the events, called *Posets.* There may be several orderings. The principle ordering expresses *causality* between events — i.e., which events caused which others to happen, and which events happened independently. Other orderings express timing between events with respect to each clock in the prototype. There is a very simple consistency relation between causality order and time orders: an event cannot occur later (in any clock time) than an event that it causes.

Events are generated by executing calls to particular constructs in RAPIDE that model communication between modules, as explained later. Such constructs include *actions* that model asynchronous communication between modules, and *remote subprograms* that model synchronous communication between modules. **An** event contains information such as the thread of control generating or receiving the event, the name of the operation being invoked, data being passed, and time intervals (duration) for the event to happen with respect to various clocks.

By contrast, other concurrent event-based simulation languages, such as VHDL [VHD87], have computations that are linear traces of events that do not encode causality between events.

Consider a graphical picture, Fig 2.1, of a small part of a poset computation. Nodes are labelled with event names. Directed arcs depict the causality order between events; there is no time ordering for this poset.

Notice first, that all the MsgIn events form a linearly ordered chain, as do all the Send events, and all the Receive events. This would happen, for example, if the Msg-In events were generated by a single sequential thread of control (process pi), the Send events were generated by another single process (p_2) and similarly for the Receive events (process p_3).

Next, notice that each Msg-In event directly causes a unique Send event, which in turn directly causes a unique Receive event. [1] These causal relationships could be the result of communication between the three processes hypothesized above, i.e., between p_1 and p_2, and between p_2 and p_3. For example, whenever p_1 generates a MsgIn, it communicates this to p_2 which causes it to generate a Send, etc.

The partially ordered computation in Fig 2.1 shows clearly that Send events cause corresponding Receive events. It also shows there is no synchronization between any Receive event and the Send that causes the next Receive; those events happen independently of each other. This means that all Send events could happen before any Receive. Consequently, Fig 2.1 shows a possible flaw in the prototype whereby a sending process could overload the receiver by sending before it is ready to receive.

Finally we note that the partially ordered computation in Fig 2.1 is equivalent to a set of linear traces, each trace being a possible output (for the same input) by a trace-based prototyping system. The linear order in each such trace must be consistent with the partial order. Two examples are:

[1] **Since causality is transitive, we say that two events are indirectly related by causality if there is a third event on any shortest path between them.**

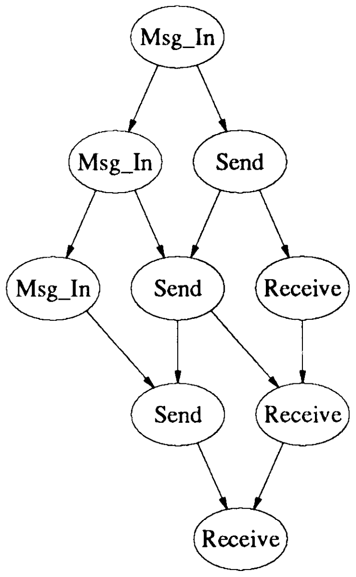

Figure 2.1: Abridged partially ordered execution.

Msg_In, Send, Receive, MsgIn, Send, Receive,
MsgIn, Send, Receive .

MsgIn, Msg_In, MsgIn, Send, Send, Send,
Receive, Receive, Receive, . . .

A single linear trace provides very little information about synchronization between the events. Indeed, the first output trace, by itself might lull us into the complacency of thinking that Send and Receive events are synchronized in a very strict way. The second trace indicates what we have already seen in the partial order. However, there is no gaurantee that a simulator that produces linear traces will produce the second trace rather than the first one.

Since prototyping is an experimental activity, it is essential that prototypes of concurrent systems yield as much information as possible. For this reason, we have adopted the poset model of computation. However, the use of posets involves dealing with two problems: since the amount of information in posets is potentially very large, *(i)* efficient implementation methods must be found, and *(ii)* useful analysis tools must be developed.

Chapter 3

Overview of Rapide

Rapide consists of three languages: the *types/module language* for structuring systems into components, the *specification language* for writing abstract specifications of behavior, and the *executable language* in which executable prototypes are written. The three languages satisfy certain compatibility requirements (e.g., they have the same visibility, scoping and naming rules, and underlying computational model). However, they can be studied and learnt separately, and each can be changed in many ways without requiring changes to the others.

We give a brief overview of the executable and specification languages to indicate in general terms how they allow modelling and specifying distributed systems by means of posets. Due to space limitations, this treatment of the language is cursory, but examples are given in Appendix A.

3.1 Types/modules language

The types/modules language provides constructs for structuring systems into components. A component consists of two separate parts: an interface defining those features through which it interacts with other components, and a module that encapsulates an executable prototype of the component.

An interface defines a type. The elements (or values) of the type are the modules that have that interface. There may be many modules with different internal implementations that have the same interface.

An interface may declare types, subprograms and actions; these are the visible constituents of modules of that type (i.e.,visible outside of the module). An interface may also contain constraints written in the specification language. Interface constraints define the visible behavior of modules, and can be viewed as a "visible contract" between a type of module and others. More important for our discussion, interface constraints can be used to automatically check the poset computations of modules.

Modules are a general construct for encapsulating the implementation of any type. Consequently, modules can be either values of a "small" type such as Integers, or values of "large" types such as a multi-threaded subsystem.

A module is created by a. module generator, which is a function that, when called with appropriate parameter values, returns modules of its type. For example,

```
X : Channels := Channels-Body( . . . );
```

results in a module, X, of type Channels with, as value, a module generated by calling Channels-Body. Thus modules can be created in a dynamic manner.

Scope rules are very simple: a module can only reference those constituents of another module, say of type T, that are declared in the interface T. A module has visibility to constituents (objects, subprograms, actions, types) declared inside itself or in its own interface. Module boundaries are also a natural way to structure a poset computation into the part that happens inside the module, and the part that is visible at its interface.

Modules may contain declarations of other types and modules. Thus a module may itself be a system of other modules, called its *components.* These components may be linked together by rules written in the executable language. As a result of such sets of rules, called *architectures,* the **out** actions invoked by one module can cause the **in** actions of another module to be invoked. In this way, architectures define dataflow between modules.

The types/modules language provides a set of predefined types such as integers, arrays, records, sets, and other common types, which can in fact be defined as interface types. So the predefined types and their special notation are actually library types defined using the features of **RAPIDE.**

The predefined types also include clock type interfaces. Clocks are modules having these predefined interfaces. A clock interface has functions for reading the clock value and the timestamps it has assigned to events. Clock modules are "active" in the sense that their internal state changes. **RAPIDE** supplies predefined module generators for clock types. Clocks can be declared and accessed just like any other type of module.

Finally, interfaces can reuse (or *inherit)* constituents of previous interfaces by a derivation construct, and similarly, modules can reuse code of other modules.

Examples of interfaces, module generators, and modules are given in Appendix A.

3.2 Executable language

The *executable language* provides simple reactive programming constructs for writing modules, modelling causality between events, and defining architectures (interconnections between sets of modules).

The essential idea is that a prototype generates a poset and also observes and reacts to it. A call to an action or remote subprogram generates an *event* which enters the computation. The resulting poset can be observed and "reacted to" by executable code. The principle programming construct to do this is the *concurrent reactive process,* which has the form,

```
when  pattern then
      executable program
end when;
```

A module may contain a set of processes, all of which compute independently. Processes within a module have visibility to that part of the poset computation that occurs within the module. This poset contains the events generated within the module, either by locally nested modules or internal action calls, or events communicated from outside modules by calls to **in** actions of the module interface.

The *pattern* is written in an event pattern language that is common to both the executable and specification languages. Patterns define subsets of poset computations. Examples of event patterns are:

```
?S : String; Msg_In(?S) => Send(?S); -- (1)

?S : String; ?R : Msg_In; ?D : Send;
?R(?S) => ?D(?S) where
   Clk. Time(?D) < Clk. Time(?R) + 1; -- (2)
```

Variables beginning with "?" are called *placeholders.* ?S has type String, and ?R, ?D are placeholders of the event types, MsgIn, Send. A pattern matches some subset, *Set,* of a poset if, when each placeholder is replaced at all of its occurrences in the pattern by the same value, the resulting instance of the pattern can be mapped (1 - 1) into *Set* so as to satisfy rules that interpret the connectives of the pattern language. E.g., E => F is interpreted as "there is a causal chain of events from E to F in *Set";* E **and** F is interpreted as "both E and F occur in Set"; (we omit discussion of other connectives).

The first pattern *(1)* above will match any subset of a poset that consists of two events, Msg_In(V), Send(V), such that there is a causal path from the Msg_In event to the Send event, and V is a string value parameter occuring in both events. For example, in Fig 2.1 there are six potential matches for this pattern (since the figure does not give the V parameter values of events, we can't tell how many of the potential matches actually satisfy the pattern). The second pattern (2) matches the subset of matches for (1) that satisfy the boolean guard where . . . . Guards are expressions. In general they may refer to functions of the computation and local state of modules. This example of a guard refers to the timestamps (discussed later) of the events.

Whenever the pattern of a process matches part of the poset computation, the process is triggered and executes. When its execution terminates, the process is ready to trigger again. In this way, a process reacts to subcomputations matching its pattern. An event may not contribute to triggering the same process more than once, but it may help to trigger different processes.

The *executable program* part of a process is written in a simple Algol-like language (roughly like Pascal). It obeys the module visibility rules, and thus can call **out** actions of its module's interface. The events triggering a process cause the events generated by that execution of the process. Also, a process is a sequential computation modelling a single thread of control: all events that it generates are linearly ordered. The causal relationships resulting from process triggering and execution are represented in the poset.

An example of a simple reactive process is,

```
?M : String;
when Take_In(?M) then
        Deliver(?M);
end when;
```

This process reacts to events generated by calling a Take-In action with some string parameter, and outputs Deliver events with the same string value. Each Take-In event will trigger the process once and cause one Deliver event. Although Take-In events may arrive independently and concurrently, the subsequent Deliver events will be linearly ordered since they are all generated by a single process.

In this simple example, the pattern triggered process is used to prototype relationships between two related computations: *(i)* receiving a message, and *(ii)* sending a message. Details of the actual computations involved in receiving and sending are omitted. The triggering pattern abstracts the receiving computation, and the call of the **out** action Deliver abstracts the sending computation. The essential features of the synchronization whereby receiving causes sending, and of sending the *same* message as was received, are captured by the process.

Timed events and statements

The executable language also has features for modelling the time taken by computations. A clock has the effect of timestamping any event within its scope. An event receives a clock's value when it is generated. A call to a clock interface function, Clock .Time(E), returns the timestamp of event E according to that clock. An event will have a timestamp from every clock in whose scope it is generated.

Execution of a statement, A, can be suspended until a clock reaches some value in the future from its current value. This is done by calling the function, Pause of a predefined system interface,

```
A Pause (Clock, T);
```

This means that when a process executes this statement, at Clock time t say, it will pause, doing nothing until Clock's value is $t + T$. At that time it will execute the call to A. A is thereby modelled as taking T ticks of Clock to happen.

We omit discussion of *time intervals* and other **RAPIDE** timing constructs.

3.3 Specification language

Constraints on poset computations are expressed in the specification language. A constraint placed in an interface constrains visible computations of modules of the type; if it is placed in a module, it constrains the internal computation.

Constraints in a type interface include constraints on parameter values of the interface subprograms and actions, algebraic constraints on the abstract state of the interface, and pattern constraints on the posets of events that can be generated from that interface. Interface constraints provide a *contract* specifying *(i)* how to use modules of that type (e.g., by constraining parameter values of **in** actions, or patterns and timing of **in** actions), and *(ii)* what those modules promise to do (e.g., by constraining parameter values of **out** actions, and patterns and timing of **out** events).

An example of an interface with a constraint is,

```
type Channels is interface
    in action Take-In(Msg : String);
    out action Deliver(Msg : String);
    < <Reliability > >
           match (?S : String; Take_In(?S) =>
                                       Deliver(?S))*~;
end Channels;
```

This constraint implies that all, and only, messages taken in are delivered. Since it contains some new pattern language connectives, we explain it as follows. The only kinds of events that can occur in a Channel's behavior are Take-In(M) and Deliver(M) events. They are constrained to occur in dependent pairs (as indicated by the "=>"), any number of these pairs may occur (as indicated by the "*"), each Deliver is caused by a Take-In with the same message (as indicated by "?S"). However, these dependent pairs of events must themselves be disjoint (as indicated by the "~") and can be independent as well. The visible poset behavior of Channels must be a union of posets matching this pattern (as indicated by **match).** A behavior satisfying the constraint could be a set of independent pairs of dependent events,

Take-In(M) $\rightarrow$ Deliver(M),
Take-In(M ') $\rightarrow$ Deliver(M'),

The behavior of the process above satisfies this constraint, but certainly does not possess as much concurrency as is allowed.

Constraints inside a module constrain the internal computation of the module. This could be a system of component modules. Thus constraints can be used to specify dataflow between systems of modules.

Formal constraints can be applied in many ways in prototyping, e.g., design capture in the process of refining system designs, formal definition of component interfaces, and automated analysis of a prototype's behavior for violations of specifications. The Appendix gives some examples of applications.

Chapter 4

Implementation Issues

Our general approach to implementing **Rapide** allows each module (and indeed each reactive process) in a system to compute independently. An underlying runtime kernel permits execution on different hardware architectures. These may vary from a single processor workstation to a loosely coupled multi-processor system with a dedicated processor for each process.

Conceptually, in this implementation model, a module is sent **in** events corresponding to its **in** actions. The processes within the module react to these events and generate **out** events which are sent to other modules. In so doing, a causal ordering relation between events must be computed along with the events, and maintained for future reference (in triggering patterns, and constructing the complete poset). The flow of events between modules is governed by sets of processes (called communication architectures) defined in the prototype itself.

Implementing **Rapide** requires dealing with several issues that are critical to its success. Due to space restrictions, we summarize three of the issues in this abstract; they will be treated in more detail in the full paper.

- how to encode and store the causal ordering in reasonable space and time.
- how to communicate the poset from one module to another in a distributed network of modules, as it is being generated, so that it can be observed and reacted to in reasonable space and time.
- how to introduce clocks into the poset computation model.

4.1 Constructing the Causality Relation

Our implementation strategy for constructing and storing the causal relation is based on an algorithm developed independently by Fidge [Fid88, Fid91] and Mattern [Mat88]. The Fidge-Mattern (FM) algorithm associates a vector with each event. The causal relation between two events can be determined by comparing their respective vectors. Unlike other proposed algorithms, FM requires no extra synchronization events, no additional communication links and no central timestamping authority. The algorithm does, however, require $O(n^2)$ space (where n is the number of processes). Charron-Bost [CB90] has shown this to be an exact bound in the general case.

In the FM algorithm, each process maintains a integer vector of length n. The ith component of process i's vector is said to be i's local counter. All vector components are initialized to 0. Each process increments its local counter before generating an event. When a process sends an event to another process, the event carries the current value of the sender's vector. The receiver then updates its vector to the component-wise maximum of its vector and the event's vector.

With this mechanism in place, the task of determining the relationship between two events can be accomplished merely by comparing their vectors. An event e_1 is earlier in the causal order then an event e_2 if and only if every component of the e_1 vector is less than or equal to the corresponding component of the e_2 vector and at least one component pair is strictly less than.

Several modifications to the FM algorithm have been proposed to improve its efficiency in specific cases or under specific requirements [SK90, MSV91]. These improvements seek to reduce the size of vectors maintained by processes and attached to events.

Rapide only requires determining the causal relationship between events arriving at a common receiver. We can take advantage of this to improve the efficiency of the FM algorithm. Further, knowledge of the communications architecture of a prototype may also be used to improve the FM algorithm.

Improvements to the FM algorithm which we have implemented to minimize the counters maitained by processes include the following:

- Sparse vectors
- If there is no communication path from process i to process j, then process j's ith vector component will always have value 0 and need not be maintained.
- In order for two events arriving at a common process to be causally ordered in the partial order, either they were sent by the same process, or their exists a communication path from one of the senders to the other. Thus if no paths exist between two senders then we know that all the events one sender sends to a particular process will be independent from all the events the other sender sends. Therefore, any vector information which was being maintained to make this determination can be discarded.
- If a communications architecture can be bifurcated such that all of the events going from one half to the other are sent to a single process in that latter half, we call that process a *gateway*. Lessening the number of vector components using gateways is shown in [MSV91].

Further, optimizations may be made on the number of components carried by events. Instead of carrying all of the vector components its sender knows, an event need only carry the components necessary for making event comparisons.

These optimizations significantly reduce the number of vector components. For example, a unoptimized **Rapide** prototype of an elevator simulation required maintaining 1,024 vector components. When the improvements outlined above were used, only 118 vector components were needed.

4.2 Orderly Observation

One of the major problems in implementing **Rapide** so that the various modules of a prototype can compute concurrently, (and be distributed over a network of processors) is the possibility that events can arrive at a module in an order that is different from their causal order. This can create problems for matching and triggering. For instance, the **Rapide** pattern language allows one to express constraints such as "when an A event happens then a B event must happen before a C event can happen." It also defines the semantics of matching so that, informally, the "earliest" event which can participate in a match is used. In order to implement these semantics, the pattern matching mechanism needs to know if there are any events which it has yet to see which are causally earlier then the events it has already seen.

There are at least three ways of dealing with this problem. One is to insure that events arrive in an order consistent with the partial order. A second is to have the ability to reverse decisions (rollback) which were made assuming no out-of-order events [Jef85]. A third approach is for an event to contain information such that a receiver may determine if there are any causally earlier events the receiver has yet to receive [RS89, ASS89].

We describe the situation in which events arrive in an order consistent with the partial order as *orderly observation.*[1] In our system we have implemented a very simple way of guaranteeing orderly observation. All events pass through a single FIFO queue. Processes synchronize with the queue when they generate and receive events. Together these two rules imply orderly observation.

To see why this is true, envision how two events could arrive at a process in an order inconsistent with orderly observation. Assume, for example, that instead of the central FIFO queue, events travel between processes by FIFO links. The following situation could arise if the links have different transmission speeds. A process i sends an event e_1 to both processes j and k. Through some causal chain the event e_1 sent to process k results in an event e_2 being sent to process j. Further e_2 arrives at process j before e_1. If, however, e_1 and e_2 must both travel through a central FIFO queue to get from one process to another, e_2 will certainly be behind e_1 in the queue. Process j will always take e_1 off the queue before e_2.

The central queue has the advantage of greatly simplifying the implementation of pattern matching used in reactive processes and constraint checking. The main disadvantage is that the central queue can become a bottleneck when the number of events being generated is high compared with the computation time of modules and processes that are generating them. The trade-off between fast pattern matching and bottlenecking in event communication may be amenable to experimental analysis by prototyping (see below).

An early implementation of an orderly observation system [BJ87] utilized piggy-backing of events. In essence, each event carries with it all events causally before it that the sender is not sure have yet arrived. While this type of implementation does not suffer from the global synchronization of our central queue, it does result in unbounded event sizes.

In rollback schemes, each process executes without regard to whether causally earlier events have yet to arrive. Whenever events arrive in an order inconsistent with the partial order, the offending process is rolled back to the time just before the inconsistency. Due to space limitations we do not discuss rollback further here. For a discussion rollback schemes see [Jef85].

In the third category is the work [RS89, ASS89]. In that scheme, each process maintains a matrix of size $O(n^2)$, making the algorithm $O(n^3)$ in space, and attaches the value of that matrix to each event it sends. A receiver can determine whether there are any causally earlier events it has yet to receive by comparing the matrix of events it receives to the matrix it maintains. We note that the improvements described in [MSV91] are applicable to this algorithm.

We are presently using RAPIDE itself to prototype the possible bottleneck effect of the central queue implementation under various timing assumptions on pattern matching and constraint checking, and also to compare it with alternative implementation strategies. Our results may well influence our implementation of the next version of RAPIDE.

4.3 Clocks

The main implementation issue dealing with clocks is in deciding when clocks increment. We view time simply as a metric associated with events. This form of time is often referred to as simulation time.

In RAPIDE, "clocks" are simply modules of a predefined type; they store and increment a single integer value. Events generated by a module are stamped with the value of all clocks visible to the module. A module with visibility to a clock is "timed" by the clock. Multiple modules may be timed by the same clock and a module may be timed by multiple clocks.

The incrementing of a clock is controlled by the RAPIDE computation, not by, e.g., the operating system or the host computer. A clock increments when all modules timed by the clock have nothing to do; that is, when the processes of the modules are not running and have no incoming events to process. Under these conditions the clock may increment an arbitrary amount, not to exceed the time of the next scheduled event.

Processes control the clock indirectly by indicating which events take time. By default an event takes zero time. RAPIDE provides three constructs for specifying the time an event takes:

[1] In the literature this has been referred to as *causal ordering* [BJ87].

```
A after (C, N);
A pause (C, N;)
A delay (C, N);
```

The first specifies that an event of action A will be generated after N time units of clock C. This statement itself takes no time to execute: **after** schedules a future event and does not block the process. The second, **pause,** blocks the process for N time units and then generates the event; **delay** behaves like **pause,** but in addition events arriving while the process is blocked are lost and cannot be used to re-activate the process.

The main issue we encountered implementing clocks was in determining when they could increment. The modules timed by a clock are identified at compile-time. The modules will be implemented by some set of independent threads of control, e.g., they could all be implemented by one thread, there could be one thread per module, or there could be one thread per **RAPIDE** process. We refer to these implementation-level threads as tasks. At run-time, these multiple tasks must communicate to agree on when a clock may increment.

Briefly, we implemented this communication as follows. One of the tasks timed by a clock was selected as the "keeper" of the clock. At run-time the keeper is supplied with a list of the other tasks using the clock. When the keeper has nothing to do at the current time, it asks the other tasks if they are ready to increment the clock. If any task is not ready, the clock is not incremented. When asked, the task also returns to the keeper the time of its next scheduled event, if any. If all tasks are ready to increment the clock, then the clock may be incremented any amount up to the next scheduled event from the tasks timed by the clock. (When there are no scheduled events and all tasks are ready, the clock is not incremented and the tasks simply wait for their next incoming event.)

Once the keeper has determined if (and how much) the clock is incremented, it communicates this to the other tasks. After a task is asked if it is ready to increment the clock, the task blocks until this second communication with the keeper. Thus, we use a two-phase commit protocol [Ull88] to determine when shared clocks are incremented.

Generally, when a clock is incremented it may be incremented more than one unit. In our implementation we supply the user with run-time options to determine how much clocks are incremented:

- Minimum: always increment one unit.
- Maximum: always increment up to the next scheduled event.
- Uniform: using a uniform distribution, increment by a random value between one and the time of the next scheduled event.
- Normal: same as uniform, but using a normal distribution.
- Nonrepeatable: when uniform or normal distributions are used, seed the random sequence generator such that clock increments are not repeatable.

In our implementation clocks are not tasks. They are simply an integer variable maintained by some task, namely, the keeper of the clock. An alternative approach would be to implement clocks as tasks, and then have each task using the clock synchronize with it. The main reason we did not do so was to minimize the number of tasks used in a **RAPIDE** computation, thus minimizing task activation and task communication overhead. Further, the protocol for deciding when to increment would not be simplified by making clocks tasks.

Another approached, based on the observation that task activation and communication overhead is high, would be to implement all the modules timed by a given clock using a single task. (Indeed, compilers for concurrent simulation languages such as VHDL [VHD87] usually use a single task.) Such an approach would not work for **RAPIDE,** short, of using one task for all modules of a program, since in **RAPIDE** a single module

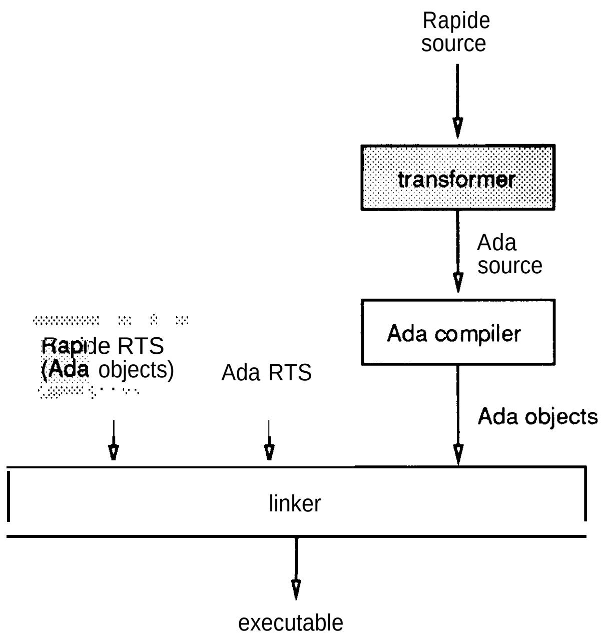

Figure 4.1: Tools used to generate an executable **Rapide** program.

may be timed by multiple clocks. Further, since modules are separately compiled units, this approach would restrict our ability to generate code for separately compiled units.

4.4 Experimental Implementation Overview

We have implemented a **Rapide** transformer and run-time system (RTS) for an early version of the language. Figure 4.1 shows the architecture of this implementation. The transformer translates **Rapide** source code into Ada source code, changing each **Rapide** module into an Ada task. The Ada source is compiled using tools developed by Software Leverage [Sof88] and Verdix [VAD91]. This Ada compiler and RTS uses multiple Unix^{TM} processes to implement tasks. The resulting executable, running on a Sequent Symmetry, uses multiple processors to excecute **Rapide** processes. The **Rapide** transformer and RTS are not machine dependent. At present, using this implementation, **Rapide** can be run on either a Sequent or a Sun workstation.

This implementation, together with a. graphical analysis toolset for postmortem analysis of posets, has been used extensively at Stanford for small prototypes. Examples include communication protocols, A telephone PBX model, small hardware devices, an IBM disk controller, distributed transactional databases, and models of parts of the **Rapide** runtime system itself. Typically, it allows prototypes of the order of two thousand lines of **Rapide** code and up to 200 processes. Optimizations are being included experimentally which may increase the complexity of prototypes by an order of magnitude.

The Sequent Symmetry used in our experiments contains 12 Intel 386 processors. Initial benchmarks running on this platform indicate that **Rapide** computations can activate 22 modules per minute, generate and send 1,600 events per minute, and match 88 patterns per minute.

Bibliography

[ASS89] Jorge Eggli Andre Schiper and Alain Sandoz. *A New Algorithm to Implement Causal Ordering,* volume 392 of *Lecture Notes in Computer Science,* pages 219-232. Springer-Verlag, 1989.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems,* 5(1):47–76, February 1987.

[BL90] Frank Belz and David C. Luckham. A new approach to prototyping Ada-based hardware/software systems. In *Proceedings of the ACM Tri-Ada Conference,* Baltimore, December 1990. ACM Press.

[CB90] Bernadette Charron-Bost. *Concerning the Size of Clocks*, volume *469* of *Lecture Notes in Computer Science,* pages 176-184. Springer-Verlag, 1990.

[Fid88] Colin J. Fidge. Timesta.mps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications,* 10(1):55–66, February 1988.

[Fid91] Colin J. Fidge. Logical time in distributed systems. *Computer,* 24(8):28–33, August 1991.

[Jef85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems,* 7(3):404–425, July 1985.

[Mat88] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proceedings of Parallel and Distributed Algorithms.* Elsevier Science Publishers, 1988. Also in: Report No. SFB124P38/88, Dept. of Computer Science, University of Kaiserslautern.

[MSV91] Sigurd Meldal, Sriram Sankar, and James Vera. Exploiting locality in maintaining potential causality. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing,* pages 231-239, Montreal, Canada, August 1991. ACM Press. Also Stanford University Computer Systems Laboratory Technical Report No. CSL-TR-91-466.

[RS89] Michel Raynal and Andre Schiper. The causal ordering abstraction and a simple way to implement it. Technical Report 501, Institut De Recherche en Informatique et Systèmes Aléatoires, November 1989.

[SK90] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. Technical report, Department of Computer and Information Science, The Ohio State University, October 1990.

[Sof88] Software Leverage, Inc., 485 Massachusetts Avenue, Arlington, MA 02174. *Symmetry Ada,* 1988.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems,* volume 1. Computer Science Press, 1988.

[VAD91] Verdix Corporation, Sullyfield Business Park, 14130-A Sullyfield Circle, Chantilly, VA, 22021. *Verdix Ada Development System*, 1991.

[VHD87] IEEE, Inc., 345 East 47th Street, *New* York, *NY,* 10017. *IEEE Standard VHDL Language Reference Manual,* March 1987. IEEE Standard 1076-1987.

Appendix A

An Example Prototyping Session

This session illustrates the stepwise refinement, of a simple prototype of a communication system based on analysis of poset computations. Formal specifications are used to automatically analyze posets, and to capture design decisions.

First prototype

We start with an interface for a communication channel that takes in and delivers messages:

```
type Channels is interface
    in action Take-In(Msg : String);
    out action Deliver(Msg : String);
    <<Reliability>>
      match (?S : String; Take-In(?S) =>
              Deliver(?S))*~;
end Channels;
```

We illustrate prototyping a Channels module that is built up from Sender and Receiver components. Interfaces for Sender and Receiver types are:

```
type Senders is interface
      in action Msg_In(Msg : String);
      out action Send(Msg : String);
      <<Reliability>>
        match (?S : String; Msg_In(?S) =>
            Send(?S))*~;
end Senders:


type Receivers is interface
      in action Receive(Msg : String);
      out action MsgOut(Msg : String);
      <<Reliability>>
        match (?S : String; Receive(?S) =>
            Msg_Out(?S))*~;
end Receivers;
```

Both interfaces can be derived from the Channels interface using **RAPIDE** derivation and renaming. They simply inherit its constituents, renaming the actions.

A module prototype for Senders can be constructed with a single reactive process. It can be reused as a module for Receivers by derivation and renaming. For Senders it is:

```
module Senders-Body return Senders is
      ?M : String;
begin
     <<Send-New-Message>>
      when Msg_In(?M) then
              Send(?M);
      end when;
end  Senders-Body;
```

Next we construct a module for Channels built out of Sender and Receiver modules. Since its components are separately compilable, its use of them is indicated by a **with** clause.

```
with Senders, Receivers;
module Channels-Body return Channels is
      -- one sender.
      Sender      : Senders := Senders-Body( );
      -- one receiver.
      Receiver   : Receivers := Receivers-Body( );
      ?Msg          : String;
begin -- architecture.
      <<Input>>
      when Take-In(?Msg) then
            Sender . Msg_In(?Msg);
      end when;
      <<Linkage>>
      when Sender. Send(?Msg) then
            Receiver . Receive(?Msg);
      end when;
      <<output>>
      when Receiver . Msg_Out(?Msg) then
            Deliver(?Msg);
      end when;
end Channels-Body;
```

The processes in this module, called *links,* connect up the actions in the channels interface and in the component interfaces, thus forming a simple architecture, depicted in Fig A.l. Take-In events trigger the process <<Input>> and cause Msg_In events at the Sender. Similarly, the <<Linkage>> link has the effect that Send events from the Sender cause Receive events at the Receiver. This models one way communication between the modules; the matching of ?Msg parameters models the dataflow.

As illustrated here, the use of pattern processes *as links* permits *communication abstraction* whereby dataflow and synchronization between modules is modelled, but there is no commitment in the prototype to any implementation details. The events caused by a link are all causally related in a linear sequence (since a single process is sequential) just like a pipeline, but all other details are hidden. Later on in the prototyping process these details may be introduced into the prototype *(refinement)* by replacing link processes by

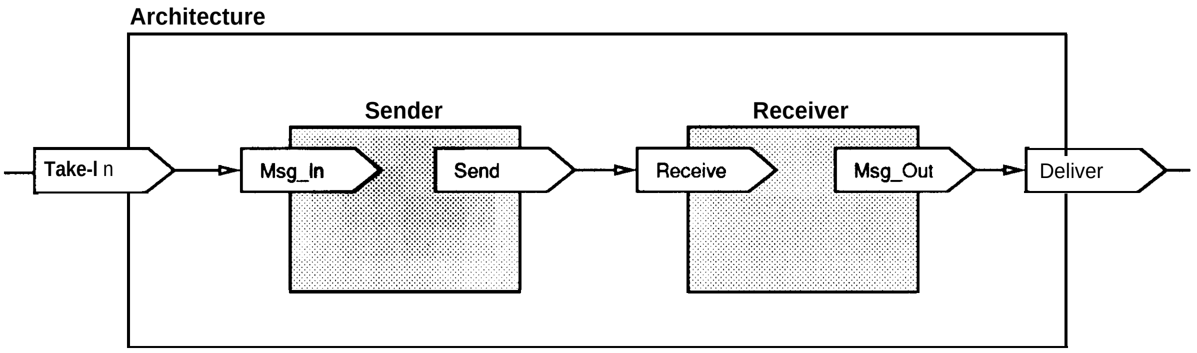

Figure A.l: Architecture of first Channels module.

modules containing other components — e. g., Linkage>> may be replaced later by a transmission module containing unreliable transmission lines.

Analyzing the first prototype

To test the Channels architecture, we construct a very simple test module that contains one instance of Channels called Channel say, and two new processes that feed data from the user console to the Channel and conversely :

```
<<Kick-Off>>
when Start then
      for i in 1 . . 3 loop
           ...  - -  get a message from Console.
           Channel . Take-In( text);
      end loop;
end when;

<<Accept-Results>>
      when Channel. Deliver(?Msg) then
       ...   – display ?Msg  on a console.
end when
```

Execution of the first Channels prototype leads to a poset containing the events shown in Fig 2.1. As discussed in Section 2 this poset shows a lack of synchronization between Receive and subsequent Send events. We might become concerned about the number of messages which can queue up if the Sender is sending faster than the Receiver can process. This might lead us to a new requirement:

No message should be sent until its predecessor has been received (except for the first message).

This new requirement can be expressed by a <<Handshake>> constraint on the computation of the modules in the Channels module as follows:

```
<<Handshake>>
  match (?M : String; Sender. Send(?M) =>
```

Receiver. Receive(?M))*=>;

This constrains Send and Receive events in the poset as follows: every Receive event must be caused by a Send event with the same argument, there can be any number of pairs of these events, and that all these pairs must be in a linear sequence.

If we add the Handshake constraint to the Channels module it will be checked automatically for consistency with the computation. If it is violated, the prototype will generate violation events, *Inconsistent,* which will be visible in the computation and whose ancestry indicates the events causing the violation. This is illustrated in Figure A.2; note that <<handshake>> is violated three times, and that Send events caused it. The reason for the violations is as follows, the second Send event happens after the first Send without an intervening Receive causing the first, Inconsistent. The third Send happens after the second Send without an intervening Receive, and it is also the case the the third Send happened after the first Send without an intervening Receive. These two situations result in the second and third violations.

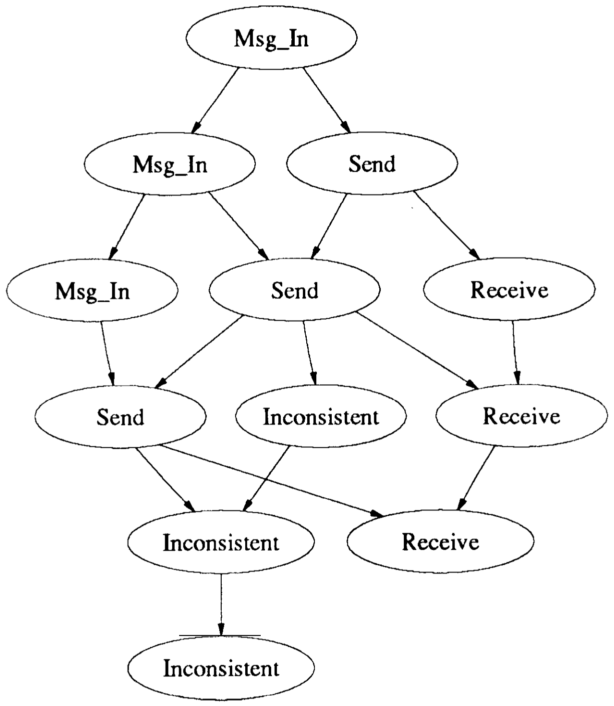

Figure A.2: Partial Order Including Violation Events.

- **Guideline: Use Specifications to Capture Design Decisions**
  Whenever analysis of a protoiype leads lo a new requirement, that requirement should be expressed formally as a constraint.

Second prototype

An *evolutionary change* is one that preserves existing specifications. A typical step in prototyping is to make evolutionary changes in the interfaces and modules to satisfy a new requirement.

A useful advantage of evolutionary prototyping in conjunction with formal specificiations is that earlier design decisions, expressed as constraints, are automatically checked in future versions of the prototype. This can largely eliminate the problem of prototypes inadvertently violating an earlier design decision.

To achieve the new <<Handshake>> requirement, we will synchronize the Sender and Receiver so that a new message is not sent until a previous message is acknowledged. To do this, we evolve both the components and their architecture in the Channels module in a series of steps which is summarized briefly as follows.

First new interfaces of both components are derived from the previous ones, inheriting the old actions and constraints. Refering to Figure A.3, the Senders type is extended with a new **in** action, Proceed, and the Receivers type is extended with a new **out** action, Ack. The new types are subtypes of the previous types. Consequently, by **RAPJDE** rules, modules of the new subtypes can be used in place of the old modules without type violations.

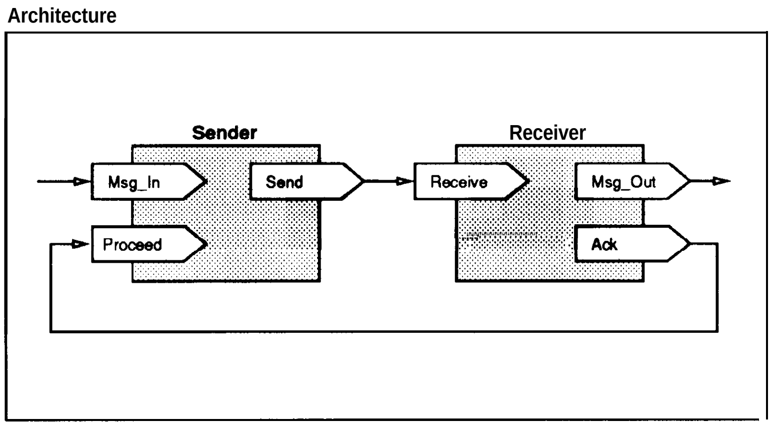

Figure A.3: Architecture of second Channels module.

Secondly, new modules for Senders and Receivers are derived from the previous modules by adding new processes and local state variables that synchronize the processes. A new Sender will wait for a Proceed before sending a new message; a new Receiver will generate an Ack upon receiving a message — the Ack event will be causally related to an incoming Receive.

Finally, the new Channels module is derived from the old one, reusing its processes. But the new components replace the previous components, and a new process connects Ack events with Proceed events. This establishes a feedback dependency between Sender and Receiver (see Figure A.3) whereby a previous Receive causes the next Send to ensure that each message has been received before its successor message is sent.

Figure A.4 shows the poset computation of the new Channels module resulting from the same test. None of the constraints are violated. One can clearly see how the Ack and Proceed events causally order the Receive and Send events to satisfy the <<Handshake>> constraint.

Modelling Timing

In order to illustrate how timing is modelled in posets, we make a short digression to introduce simplified versions of the **RAPIDE** timing constructs.

Clocks

RAPIDE provides predefined clock type interfaces. Clocks are modules having these predefined interfaces. A clock interface has functions for reading the clock value and the timestamps it has assigned to events. Clock modules are "active" in the sense that their internal state changes. **RAPIDE** supplies predefined module generators for clock types. Clocks can be declared and accessed just like any other type of module.

Timed events and statements

A clock has the effect of timestamping any event within its scope. An event receives a clock's value when it is generated. A call to a clock interface function, Clock. Time(E), returns the timestamp of event E according to that clock. An event will have a timestamp from every clock in whose scope it is generated.

Execution of a statement, A, can be suspended until a clock reaches some value in the future from its current value, This is done by calling the function, Pause of a predefined system interface,

A Pause(Clock, T);

This means that when a process executes this statement, at Clock time t say, it will pause, doing nothing until Clock's value is $t + T$. At that time it will execute the call to apA. A is thereby modelled as taking T ticks of Clock to happen.

We omit discussion of *time intervals* and other **RAPIDE** timing constructs.

end digression.

To model a requirement that Channels process messages promptly, we modify the test module, which contains Channels, so that it contains a clock, Clk. This clock is global to all modules in the system.

A new constraint, Prompt-Delivery is added to the Channels interface:

```
?T  : Take-In;
?D  : Deliver;
<<Prompt-Delivery > >
match (?M : String; ?T(?M) => ?D(?M) where
      Clk . Time(?D) < Clk . Time(?T) + 10 )*~;
```

The behavior of Channels is constrained to be a set of causally related pairs of Take-In and Deliver events in which the time at which the Deliver event was generated is less than 10 time units later than the time at which the Take-In event was generated.

The new Channels can be derived as a subtype (called Timed-Channels, say) of the previous Channels interface. The <<Prompt-Delivery>> specification actually implies the previous <<Reliability>> specification.

As a first example of a timing model for the Channels module, we model each action of its components as taking one Clk tick, and similarly for each link between their actions.

To do this, we construct new interfaces for Senders and Receivers that include timing pauses for the **out** actions. The new interfaces are derived from, and are subtypes of, the previous ones. In full, the actions in these interfaces would now be declared as follows:

```
type Senders is interface
in action Msg_In(Msg : String);
      in action Proceed;
      out action Send(Msg : String)
                                     pause (Clk, 1);

end Senders:
```

```
type Receivers is interface
      in action Receive(Msg : String);
      out action Msg_Out(Msg : String)
                                      pause (Clk, 1);
      out action Ack    pause (Clk, 1);

end Receivers;
```

Similarly, the Channels module is changed as follows:

```
with Senders, Receivers;
module Channels-Body return Channels is
      ... -- declarations of new components.
begin
            <<Linkage>>
      when Sender . Send(?Msg) then
            Receiver . Receive(?Msg) pause (Clk, 1);
      end when;
      when Receiver . Ack then
            Sender . Proceed pause (Clk, 1);
      end when;
      ...                    -- other links as before.
end Channels-Body;
```

The links between the Channels interface actions, Take-In and Deliver, and the component's actions, Msg-In and Msg_Out, are modelled as being instantaneous (no pause). Finally, the test module is changed so the test loop generates three Take-In events that each take one tick.

If we simply add up the delays between the arrival of a message (a Msg-In event) and its delivery, (see Fig. A.3) the total delay is less than 10 units, satisfying our constraint. Thus we are fairly confident that our design will meet our prompt delivery constraint.

Figure A.5 depicts the poset with timestamps produced by this timed prototype. It shows the events generated by the whole Channels model within the test module — see Fig. A.3. The second parameter value of events is the timestamp. Figure A.5 actually contains two partial orders between events, *causality,* and *timing,* but the latter is not depicted graphically. This poset shows clearly three main activities, taking in messages, delivering messages, and transferring messages from sender to receiver. These activities communicate asynchronously, but run independently (and possibly concurrently). Looking at this computation we discover a violation event was generated. Our program did not satisfy our expectations.

Why did this violation occur? Certainly the Take-In and Deliver events occur in causally related pairs, each pair occuring independently, so the poset satisfies the <<reliability>> constraint for Channels — this is checked by **RAPIDE** tools. So we know, from the logical relationship between the two constraints, that the timing guard in Prompt-Delivery>> was violated.

One hint is the causal arc between the events Sender . Msg_In("!!!" ,3) and Sender . Send("!!" ,7). This indicates that the third message was input to Sender before it had even sent the second message. We note that the time between the first Take-In and the first Deliver was three time units. The time between the second Take-In and the second Deliver was seven time units. Lastly, the time between the third Take-in and the third Deliver was eleven time units, causing the constraint violation. As we can see from Figure A.5 there is a delay of four time units between each Send and the next Proceed event. Thus we deduce that the channel can deliver a message every four time units. The problem is that our test has generated input messages at the rate of one message every tick. As a result the channel fell behind, messages queued up and eventually the Prompt-Delivery constraint was violated.

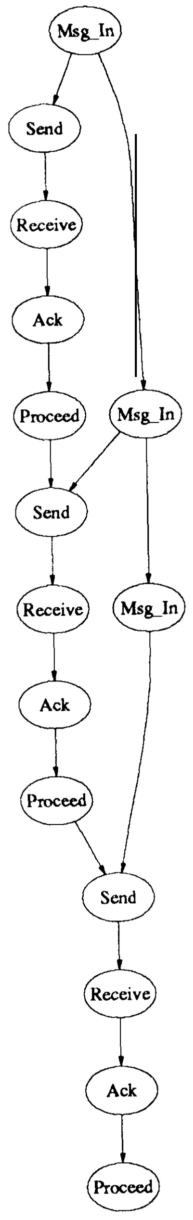

Figure A.4: Poset produced by second Channels module.

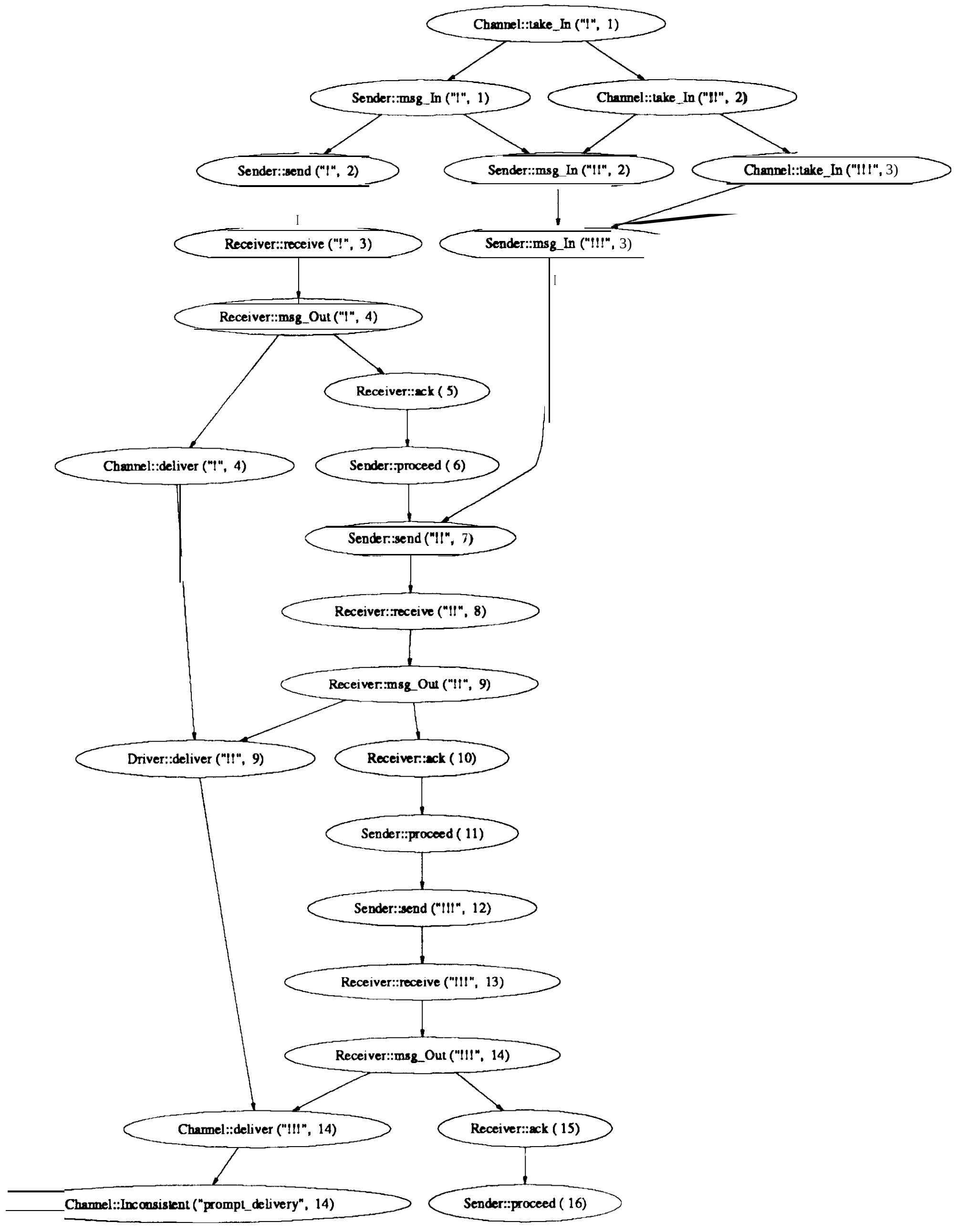
Channel::take_In ("!", 1)
Sender::msg_In ("!", 1)
Channel::take_In ("!!", 2)
Sender::send ("!", 2)
Sender::msg_In ("!!", 2)
Channel::take_In ("!!!", 3)
Receiver::receive ("!", 3)
Sender::msg_In ("!!!", 3)
Receiver::msg_Out ("!", 4)
Receiver::ack (5)
Channel::deliver ("!", 4)
Sender::proceed (6)
Sender::send ("!!", 7)
Receiver::receive ("!!", 8)
Receiver::msg_Out ("!!", 9)
Driver::deliver ("!!", 9)
Receiver::ack (10)
Sender::proceed (11)
Sender::send ("!!!", 12)
Receiver::receive ("!!!", 13)
Receiver::msg_Out ("!!!", 14)
Channel::deliver ("!!!", 14)
Receiver::ack (15)
Channel::Inconsistent ("prompt_delivery", 14)
Sender::proceed (16)

Figure A.5: Timed Partial Order with a Constraint Violation.