Partial Orderings of Event Sets and Their Application to Prototyping Concurrent, Timed Systems[1]

David C. Luckham James Vera[2] Doug Bryan Larry Augustin
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Frank Belz
TRW

December 3, 1992

submitted for publication in The Journal of Systems and Software (JSS)
special issue on applying specification, verification, and validation techniques

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 prototyper. 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. An example prototyping scenario illustrates uses of the poset computation model, with and without timing.

keywords: Rapide, partial orders, prototyping, concurrency, real-time, architecture, programming languages.

Principle contact:

Larry M. Augustin
ERL 414, M/C 4055
Computer Systems Laboratory
Stanford University
Stanford, CA 94305
Tel: (415) 723–9285
Fax: (415) 725–6949
Email: lma@dayton.Stanford.EDU

[1] This research was supported by the U.S. Air Force Office of Scientific Research under Grant AFOSR91-0354, and by the U.S. Defense Advanced Research Projects Agency/Information Systems Technology Office under the Office of Naval Research contract N00014-90-J1232.
[2] Supported by a fellowship from Bellcore.

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

1 Introduction

Prototypes of complex systems usually embody only selected 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. Thus a fundamental aspect of prototyping is the kind of information provided by a prototype's execution, and the tools to analyze it.

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. A primary 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* (partially ordered set of events) 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; the application of formal specifications to automatically check posets for violations of requirements is illustrated. An example prototyping scenario illustrates some simple uses of the poset computation model in a step by step analysis and refinement of a prototype.

Many event-based languages for describing and simulating systems are already in use, notably VHDL [VHD87] and Verilog [TM91] for hardware systems, and CSP-based languages such as LOTOS [BB89] and Esterel [BCG87] for modeling synchronous software systems. RAPIDE departs from all of these languages in adopting the poset computation model in place of linear traces of events (possibly with timestamps), and in providing asynchronous communication (as well as synchronous communication) between independent components. Many aspects of RAPIDE are not discussed in this paper due to space limits, e.g., the object-oriented type system and algebraic constraints. These aspects are dealt with in other reports and papers. An outline of an early version of RAPIDE (then called *Reality*) is given in [BL90]. The present RAPIDE tool-set is described in [Bry92]. A language reference manual for RAPIDE is in preparation.

2 Computation Model

RAPIDE computations are sets of events with partial orderings on the events (*posets*). Events are generated by processes and processes may observe events subject to visibility restrictions. There may be several orderings over the events. The principle ordering is called *dependency*[2] and is defined as follows. An event B depends on an event A (written $A \rightarrow B$) if:

1. A and B are generated by the same process and A is generated before B. (Processes are sequential.)

2. A process observes A and then generates B.

[2] The inverse of this relation is referred to in the literature as *potential causality* [Mat88] and *happened before* [Lam78, Fid88].

3. A process generates A and then writes to a variable v. Another process reads v before any intervening write to v, and then generates B.

4. $A \rightarrow C$ and $C \rightarrow B$ (i.e., transitivity).

Other orderings express timing between events with respect to each clock in the prototype. There is a very simple consistency relation between dependency order and time orders: an event cannot occur earlier (by any clock) than an event that it depends on.

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 functions that model synchronous communication between modules. An event contains information such as the process and module generating the event, the name of the operation being invoked, data being passed, the event's place in the dependency order, and time intervals (duration) of the event 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 dependency between events.

Consider a graphical picture, Figure 1, of a small part of a computation. Nodes represent events and are labeled with action names. Directed arcs depict the dependency relation between events; there is no time ordering given in this depiction.

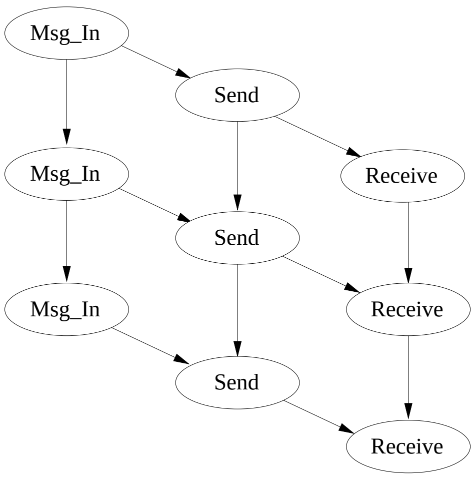

Figure 1: Abridged partially ordered execution.

Notice first that all the Msg_In events form a linearly ordered chain, as do the Send events, and the Receive events. This would happen, for example, if the Msg_In events were generated by a single process (p_1), the Send events were generated by another process (p_2), and similarly for the Receive events (p_3).

Next, notice that each Send event depends on a unique Msg_In event, and each Receive event depends on a unique Send event. These dependency relationships could be the result of communication between the three hypothesized processes, i.e., between p_1 and p_2, and between p_2 and p_3. For example, whenever p_1 generates a Msg_In, it communicates this to p_2 which then generates a Send, etc.

The computation in Figure 1 shows clearly that Receive events depend on Send events. It also shows there is no dependency between a Receive event and the next Send; those events happen independently of each other. This means that all Send events could occur before any

Receive. Consequently, Figure 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 Figure 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 example linear orders are:

1. Msg_In, Send, Receive, Msg_In, Send, Receive, Msg_In, Send, Receive

2. Msg_In, Msg_In, Msg_In, Send, Send, Send, Receive, Receive, Receive

A single linear trace provides very little information about dependency 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 guarantee 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. The first problem is beyond the scope of this paper, but is addressed in [SK90, MSV91]. A graphical, interactive partial order browser with extensive query capabilities was developed to address the second problem [Bry92]. Additionally, the linguistic tool of machine-checkable formal specifications, as discussed in Section 3.4, greatly aids analysis.

3 Overview of Rapide

Rapide consists of four sublanguages: the *types language* for describing the interfaces of components, the *specification language* for writing abstract specifications of behavior, the *executable language* in which executable prototypes are written, and the *pattern language* for describing patterns of events. The four 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 learned separately, and each can be changed in many ways without requiring changes to the others.

We give a brief overview of each of the sublanguages and indicate in general terms how they support modeling and specifying distributed systems by means of posets. Due to space limitations, we will focus on the executable and pattern languages. Readers interested in more detail are referred to the bibliography.

3.1 Types language

A system 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. Interfaces are defined in the types language; modules are defined in the executable language.

The types language supports both object-oriented and abstract data type styles of defining interfaces, and multiple inheritance methods of reusing old interfaces to define new ones. A concept of an interface being a subtype of another one allows replacement of components in a system if the new one belongs to a subtype of the type of the component being replaced. Automated checking of correct use of subtypes is based on the form of interfaces (so-called structural type-checking) rather than the use of type names (as in, say, Pascal or Ada). The overview of the types language presented here will be limited to those features necessary for understanding its limited use in the examples in this paper. The reader is referred to [MMM91] for more details.

An interface defines a type. The elements (or values) of the type are the modules that *satisfy* the interface. Informally, a module satisfies an interface if it defines all the items declared in the interface. There may be many modules with different internal implementations that satisfy the same interface. Interface types are declared using the following syntax:

type_declaration ::= **type** *identifier* **is** *interface_expression* ';'
interface_expression ::= *type_name* |
    interface
        interface_constituent_list
    [**constraint** *interface_constraint*]
    end [**interface**] [*identifier*]

Interface constituents consist of type declarations, derivation declarations, function declarations, and action declarations; these are the visible constituents of modules of that type (i.e., visible outside of the module). For example, the type of process p_2 as discussed in Section 2 could be:

```
type Transmitters is interface
    in action Msg_In(Msg : String);
    out action Send(Msg : String);
end Transmitters;
```

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 other modules. More important for our discussion, interface constraints can be used to automatically check the poset computations of modules.

Types may be parameterized, in which case they are *type constructors*. When a type constructor is applied to appropriate arguments the result is a type. Common types and constructors (e.g., integer, array, and record) typically seen in Algol-style languages can be defined in RAPIDE as interface types or type constructors. A standard preamble defines a typical set of such types and type constructors in this manner.

Finally, interfaces can reuse (or inherit) constituents of previous interfaces using a derivation declaration. In most object-oriented languages inheritance implies derivation of both interface and implementation. Inheritance in RAPIDE interfaces differs in that it does not imply code reuse—inheritance of implementation is separate from inheritance of interface.

3.2 Executable language

The *executable language* provides concurrent, reactive programming constructs for writing modules, modeling dependency between events, and defining architectures (interconnections between sets of components). Modules are a general construct for encapsulating the implementation of a component. Consequently, modules can be either values of a "small" type such as Integer, or values of a "large" type such as a multi-threaded subsystem.

A module is created by a module generator, which is a function that, when called with appropriate parameters, returns a module. The syntax for module generators is:

module_generator_declaration ::=
  module *identifier* '(' [*parameter_list*] ')' **return** *interface_expression* **is**

[*module_constituent_list*]
[**initial** *statement_list*]
[(**parallel** | **serial**) *process_list*]
[**final** *statement_list*]
[**handler** *handler_list*]
end [**module**] [*identifier*] ';'

A call to a module generator creates a new module of the specified type. For example, the declaration,

X : Channels **is** Channels_Body(...);

results in a module named X of type Channels with a value generated by calling the module generator Channels_Body. Modules can be dynamically created.

Scoping rules are very simple in RAPIDE: 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 (components, functions, 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.

A module generator is divided into five parts: constituent declarations, initialization code, finalization code, exception handler, and processes. Each part of a module may inherit executable code from the corresponding parts of other modules. Such inheritance permits code reuse independent of the type derivation hierarchy.

The constituents of a module may include other modules, called its *components*. The prototyper uses the reactive statements of the executable language to define the *communication links* among components. Components and links together form an *architecture* of the system encapsulated by the outer module.

Initialization code is invoked when the module is created. Finalization code is executed when the module becomes unnamable and its processes terminate. The handler part contains code for handling exceptions. Exceptions are not described in this paper.

The process part contains a set of processes, separated by ||:

process_list ::= *process* { '||' *process* }
process ::= *statement_list*

If the process part begins with the keyword **parallel**, each processes computes independently. If the process part begins with the keyword **serial**, only one process may execute at at time, much like coroutines [Con63]. Processes within a module have visibility to *(i)* the events generated within the module, either by its components or internal action calls, and *(ii)* to events communicated from outside modules via outer communication links that issue calls to the in-actions of the module's interface.

The essential idea is that a prototype generates a poset and reacts to it. Events are generated by calling actions. A process reacts to patterns of events using when- and await-statements:

when_statement ::= [**declare** *declaration_list*]
when *pattern* **do**
statement_list
end [**when**] ';'

```
await_statement ::= await pattern ';'
                  | [ declare declaration_list ]
                    await is
                      pattern '=>' statement_list
                      { or pattern '=>' statement_list }
                    end [ await ] ';'
```

Patterns are discussed in Section 3.3. When the pattern of a when-statement matches part of the poset computation, the process is said to observe that part of the poset, and the when-statement is "triggered" and executes. When its execution terminates, the statement is ready to trigger again. In this way, processes *react* to subcomputations. An await-statement is like a when-statement except that it only triggers once. An event may not contribute to triggering the same process more than once, but it may help to trigger different processes.

The executable language also contains control statements typical of Algol-like languages: if-then-else, loops, etc. They are not described in this paper.

An example process consisting of a single when-statement is:

```
declare
   ?M : String;
when Msg_In(?M) do
   Send(?M);
end when;
```

?M is called a *placeholder*, and is bound in a pattern to a parameter of an event. This process reacts to events that are instances of a Msg_In action with some string parameter value, and outputs Send events with the same string parameter. Each Msg_In event will trigger the process once and cause one Send event. Although Msg_In events may arrive independently and concurrently, the subsequent Send events will be linearly ordered since they are all generated by a single process (rule 1 in Section 2). Also, each Send event will depend on the Msg_In event that triggered it (rule 2 in Section 2).

In this simple example, the pattern triggered process is used to prototype relationships between two related computations: receiving a message and 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 to the out-action Send 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.

Clocks and Timed Events

In RAPIDE, *clocks* are simply modules of a predefined type. The clock type interface is part of the standard type preamble. Clocks are modules satisfying this predefined interface. A clock interface has functions for reading the clock value and the timestamps it has assigned to events. Clocks 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. Clocks can be declared and accessed just like any other type of module.

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:

A **after** N;
A **pause** N;
A **delay** N;

The first specifies that an event of action A will be generated after N time units. (The clock N refers to is determined by the type of N.) This statement itself takes no time to execute: **after** schedules a future event but 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 later be used to trigger the process.

We omit discussion of *time intervals* and other RAPIDE timing constructs. Interval timing constructs are similar to those used in timing specification in [AGH+88, ALG+90].

3.3 Pattern Language

The concept of event patterns is similar to that developed in [LHM+86, Ros91, Gen91, GL92]. Event patterns define subsets of poset computations. Patterns may be constructed using the following syntax:

```
pattern ::= basic_pattern
          | pattern binary_pattern_operator pattern
          | pattern '^' '(' iterator_operator binary_pattern_operator ')'
          | pattern where guard
          | '(' placeholder_list ':' pattern ')'
          | '(' pattern ')' | empty | any
          | pattern during '(' expression ',' expression ')'

binary_pattern_operator ::= '->' | '||' | and | or | '~' | '<=>'
iterator_operator       ::= '*' | '+' | expression
guard                   ::= boolean_expression
```

A basic pattern is simply the name of an action, with optional parameter associations. For example, the basic pattern Send("a") is only matched by Send events whose first parameter is the string "a".

The pattern operators have the following informal semantics:

- *dependent:* $P -> P'$. A match of patterns P and P' where all of the events which matched P' depend on all of the events which matched P.
- *both:* P **and** P'. A match of patterns P and P'.
- *distinct:* $P \sim P'$. A match of patterns P and P' where all of the events which matched P are distinct form the events which matched P'.
- *either:* P **or** P'. A match of pattern P or a match of pattern P'.
- *independent:* $P \,||\, P'$. A match of patterns P and P' where none of the events which matched P are dependent on any of the events which matched P' and vice versa.
- *equivalent conjunction:* $P <=> P'$. A match for P that is also a match for P'.

- *iteration:* P ^(*iterexp op*) where *iterexp* is one of $*$ (zero or more), $+$ (one or more), or n (exactly n copies), and *op* is any of the preceding binary operations. The indefinite forms ($*$ and $+$) are only used in constraints.

- *restriction:* P **where** E. A match of pattern P where the boolean expression E is true. The expression is evaluated when the last of the events matching P is received.

- *temporal restriction:* P **during**(m, n). A match for P where all the events of the match start and finish within the time interval m to n, inclusive.

Given placeholders,

?S : String; ?R : Msg_In; ?D : Send;

two examples of event patterns are:

(1) Msg_In(?S) –> Send(?S);
(2) ?R(?S) –> ?D(?S) **where** Clk.Start(?D) < Clk.Start(?S) + 2

Variables beginning with "?" are called *placeholders*. ?S has type String, ?R event type Msg_In, and ?D event type Send. A pattern matches some subset, S, of a poset if, when each placeholder is replaced at all of its occurrences in the pattern by the same value, the pattern matches S. Placeholder lists may be used within patterns to restrict this replacement to subpatterns.

The first pattern above will match any subset of a poset that consists of two events, Msg_In(v_1) and Send(v_2), such that the Send event depends on the Msg_In event and $v_1 = v_2$. For example, in Figure 1 there are six potential matches for this pattern. Since the figure does not give parameters 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. 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 of events.

3.4 Specification language

The specification language uses a combination of *algebraic constraints* and *pattern constraints*. Algebraic constraints are similar to those appearing in [KBL80, LvH85, Luc90] and so will not be discussed further here.

A constraint placed in an interface constrains visible computations of modules of the type. Constraints may also be placed in modules, where they constrain the internal computation.

Constraints in an interface include constraints on parameter values of the interface functions and actions, algebraic constraints on the abstract state of the interface, and pattern constraints on the posets of events that can be generated and observed from the 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-events), 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);
constraint
    ?S : String;
    match (?S: Take_In(?S) -> Deliver(?S))^(*~);
end Channels;
```

This constraint implies that all, and only, messages taken in are delivered. We explain it as follows. The only kinds of events visible at the Channel's interface are Take_In and Deliver. They are constrained to occur in dependent pairs (as indicated by "−>"), any number of these pairs may occur (as indicated by "∗"), and the Deliver of a pair has the same parameter as the Take_In (as indicated by "?S"). Further, these dependent pairs of events must themselves be disjoint (as indicated by "∼") and can be independent as well. The visible poset behavior of Channels must be a poset matching this pattern (as indicated by **match**). An example behavior satisfying the constraint is a set of independent pairs of dependent events:

$$\begin{array}{ccc} \text{Take\_In("a")} & \text{Take\_In("b")} & \text{Take\_In("c")} \\ \downarrow & \downarrow & \downarrow \\ \text{Deliver("a")} & \text{Deliver("b")} & \text{Deliver("c")} \end{array} \vdots$$

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

4 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);
constraint
    ?S : String;
    match (?S: Take_In(?S) -> Deliver(?S))^(*~);
end Channels;
```

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

```
type Transmitters is interface
      in action Msg_In(Msg : String);
      out action Send(Msg : String);
constraint
      ?S : String;
      match (?S: Msg_In(?S) -> Send(?S))^(*~);
end Transmitters;

type Receivers is interface
      in action Receive(Msg : String);
      out action Msg_Out(Msg : String);
constraint
      ?S : String;
      match (?S: Receive(?S) -> Msg_Out(?S))^(*~);
end Receivers;
```

Both interfaces can be derived from the Channels interface using Rapide derivation and renaming; they could simply inherit its constituents, renaming the actions.

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

```
module Transmitters_Body( ) return Transmitters is
   ?M : String;
parallel
   when Msg_In(?M) do
      Send(?M);
   end when;
end Transmitters_Body;
```

Next we construct a module for Channels built out of Transmitter and Receiver modules.

```
module Channels_Body( ) return Channels is
      Transmitter : Transmitters is Transmitters_Body( );
      Receiver    : Receivers is Receivers_Body( );
      ?Msg        : String;
parallel
      when Take_In(?Msg) do
         Transmitter . Msg_In(?Msg);
      end when;
||    Link:  -- Name the process, for later reuse purposes.
      when Transmitter . Send(?Msg) do
         Receiver . Receive(?Msg);
      end when;
||    when Receiver . Msg_Out(?Msg) do
         Deliver(?Msg);
      end when;
end Channels_Body;
```

The processes in this module connect the actions of the Channels interface and the actions of the component interfaces, forming the architecture depicted in Figure 2. This models one-way communication between the modules; the matching of ?Msg parameters models dataflow.

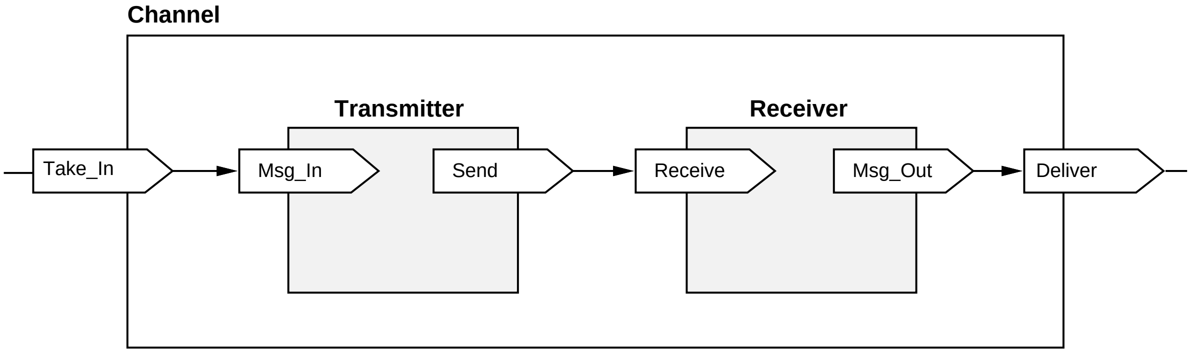

Figure 2: Architecture of first Channels module.

As illustrated here, the use of pattern-based processes permits *communication abstraction* whereby dataflow and event dependency between modules is modeled, but there is no commitment in the prototype to any implementation details. The events generated by a single process are all dependent, in a linear sequence (since a process is sequential), but all other details are hidden. Later on in the prototyping process details may be introduced into the prototype (*refinement*) by replacing these processes by modules containing other components—e.g., the second process may be replaced later by an unreliable media module that occasionally looses messages.

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 a new process that generates test data:

```
Channel.Take_In("a");
Channel.Take_In("b");
Channel.Take_In("c");
```

When considering only Msg_In, Send, and Receive events, the execution of this prototype leads to the poset in Figure 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 Transmitter 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 as follows:

```
?M : String;
match (?M: Transmitter.Send(?M) -> Receiver.Receive(?M))^(* ->);
```

This constrains Send and Receive events in the poset as follows: every Receive event must depend on a Send event with the same parameter, there can be any number of pairs of these events, and these pairs must be in a linear, dependency 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 3; note that the constraint is violated three times, and that Send events cause the violations. The second Send event happens after the first Send without an intervening Receive, causing the first violation. The third Send happens after the second Send without an intervening Receive, and it is also the case that the third Send happened after the first Send without an intervening Receive. These two situations result in the second and third violations.

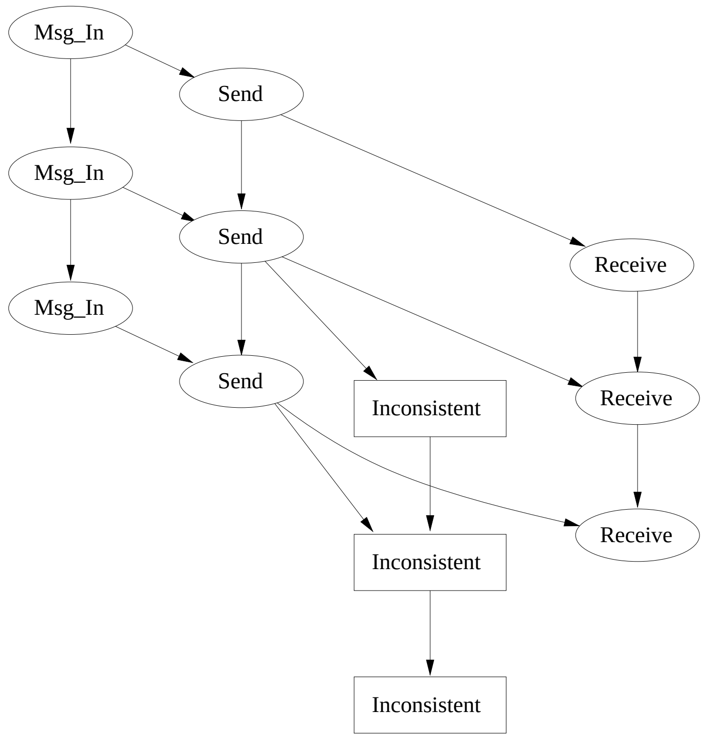

Figure 3: Partial order including violation events.

The use of the handshake constraint illustrates one of the strategies for using RAPIDE. Whenever analysis of a prototype leads to a new requirement, that requirement should be captured 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 specifications 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 handshaking requirement, we will synchronize the Transmitter and Receiver so that a new message is not sent until the 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. Referring to Figure 4, the Transmitters type is extended with a new in-action, Proceed, and the Receivers type is extended with a new out-action, Ack. The new types, New_Transmitters and New_Receivers shown below, are subtypes of the previous types. Consequently, by RAPIDE rules, modules of the new subtypes can be used in place of the old modules without type violations.

```
type New_Transmitters is interface
      include Transmitters;
      in action Proceed;
end New_Transmitters;

type New_Receivers is interface
      include Receivers;
      out action Ack;
end New_Receivers;
```

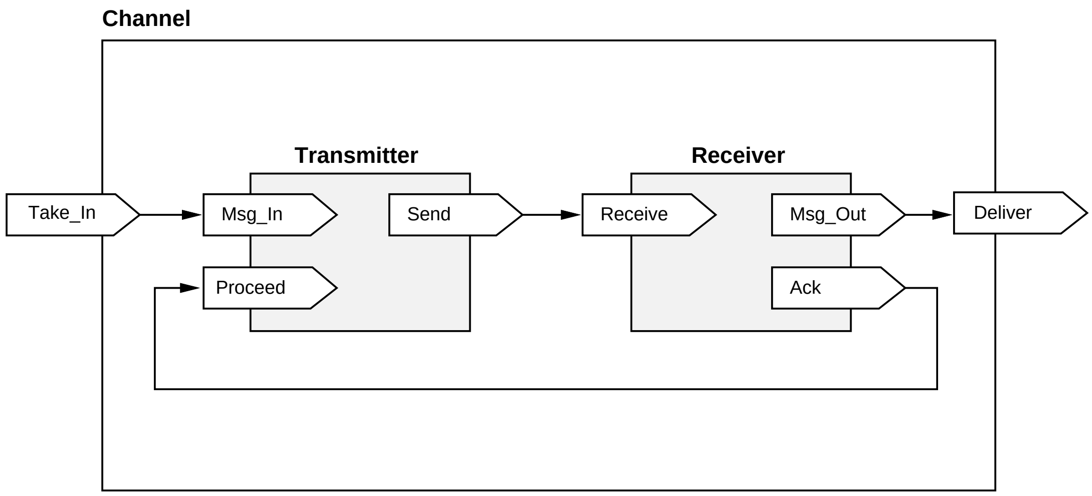

Figure 4: Architecture of second Channels module.

Secondly, modules for New_Transmitters and New_Receivers are developed. A New_Transmitter will wait for a Proceed event before sending a new message, and a New_Receiver will generate an Ack upon receiving a message.

Finally, a 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 Transmitter and Receiver (see Figure 4).

Figure 5 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 handshaking constraint.

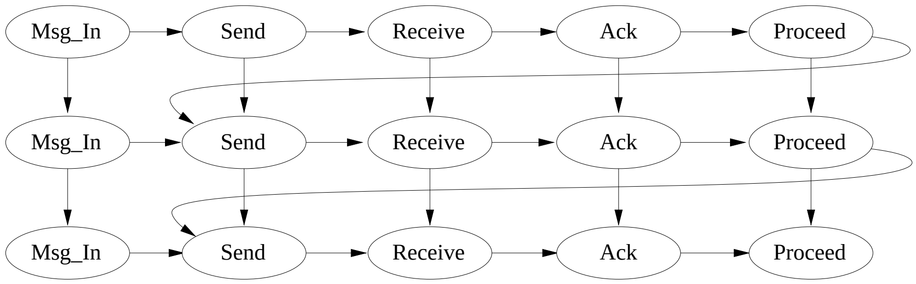

Figure 5: Poset produced by the second Channels module.

Modeling Timing

In order to illustrate the timing aspects of RAPIDE, we add a requirement that channels process messages promptly. In order for this requirement to be checked, we modify the Channels interface so that it contains a clock module, Clk, and a new constraint specifying prompt delivery:

```
Clk : Clock;
?M : String;   ?T : Take_In;   ?D : Deliver;
match (?M: ?T(?M) -> ?D(?M) during (Clk.Start(?T), Clk.Start(?T) + 9))^(*~);
```

The behavior of Channels is constrained to be a set of dependent pairs of Take_In and Deliver events in which the time that a Deliver event is generated is within 9 time units after the time that the corresponding Take_In event is generated. The new Channels can be derived as a subtype (called Timed_Channels, say) of the previous Channels interface. We note that the prompt delivery constraint actually implies the previous reliability constraint.

As a first example of a timing model for the Channels module, we model each event of its components as taking one Clk tick, and similarly for each link. To do this, we construct new modules Timed_New_Transmitter_Body and Timed_New_Receiver_Body that include timing pauses on every out-event generation. In this refinement, the Channels architectures passes the Transmitter and Receive the clock to be used. Also, we add a finite length buffer to the Transmitter to model loss of messages.

```
module Timed_New_Transmitter_Body(C : Clock) return New_Transmitters is
   Buffer : Ring(String) is Make_Ring(String, Length => 10);
   ?M     : String;
   One    : C.Ticks is 1;
   action Ready;   -- An internal action.
parallel
    when Msg_In(?M) do
       Buffer.Add(?M);
       Ready;
    end when;
```

```
||   when (Start or Proceed) and Ready do
        if not Buffer.Is_Empty( ) then
           Send(Buffer.Remove( )) pause One;
        end if;
     end when;
end Transmitters;
```

(An event of the predefined action Start is automatically generated as the first event of each module.)

```
module Timed_New_Receiver_Body(C : Clock) return New_Receivers is
   One  : C.Ticks is 1;
   ?M   : String;
parallel
   when Receive(?M) do
      Msg_Out(?M) pause One;
      Ack pause One;
   end when;
end Receivers;
```

Similarly, the Channels module is changed as follows:

```
module New_Channels_Body( ) return Channels is
      . . .  -- declarations of new components.
      One : Clk.Ticks is One;
begin
      when Transmitter . Send(?Msg) do
         Receiver . Receive(?Msg) pause One;
      end when;
||    when Receiver . Ack do
         Transmitter . Proceed pause One;
      end when;
||    include Channels_Body( ) except Link;   -- Reuse other processes.
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 modeled as being instantaneous (no pause). Finally, the test module is changed so that the three Take_In events generated each take one tick.

Figure 6 depicts the poset with timestamps produced by this timed prototype. The last parameter of events is the timestamp. Figure 6 actually contains two partial orders between events, *dependency*, and *timing*; the latter is somewhat depicted by the vertical position of events. This poset shows three main activities: taking in messages, delivering messages, and transferring messages from the transmitter to the receiver. These activities communicate asynchronously, but run independently (and possibly concurrently). Looking at this computation we discover an Inconsistent event was generated. Our program did not satisfy our expectations.

Why did this violation occur? Certainly the Take_In and Deliver events occur in dependent pairs, 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 the prompt delivery constraint was violated.

One hint is the dependency between the events Msg_In("c",3) and Send("b",7). This indicates that the third message was input to Transmitter before the Transmitter had even sent the second message. (This dependency is the result of the processes in the transmitter sharing the Buffer variable.) We note that the time between the first Take_In and the first Deliver was

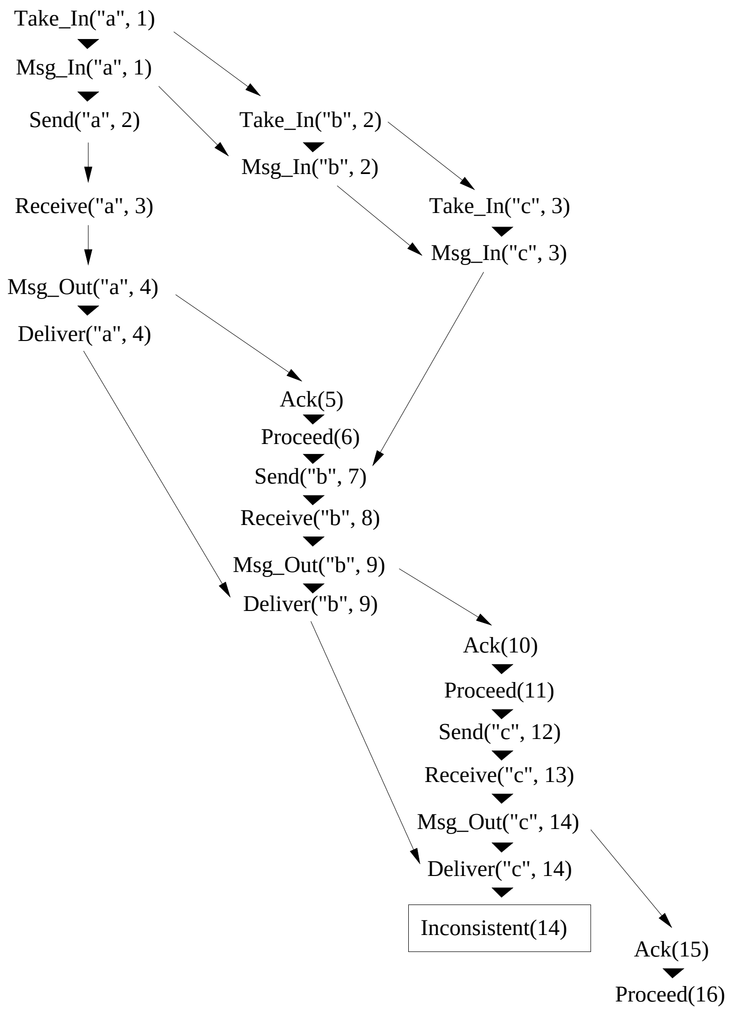

Figure 6: Timed partial order with a constraint violation.

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 a violation of the prompt delivery constraint. As we can see from Figure 6 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 every time unit. As a result the channel fell behind, messages queued up and eventually prompt delivery was violated.

5 Conclusions

We have described the information provided by Rapide *poset* computations, and the features of Rapide for constructing prototypes that produce posets and react to posets. An example prototyping scenario illustrated uses of Rapide. The scenario used subtyping in evolutionary programming, poset graphs to identify unexpected behavior, use of formal, pattern-based constraints to capture design decisions, and timed posets to isolate timing overruns.

References

[AGH+88] L. M. Augustin, B. A. Gennart, Y. Huh, D. C. Luckham, and A. G. Stanculescu. Verification of VHDL designs using VAL. In *Proceedings of the 25th Design Automation Conference (DAC)*, pages 48–53, Anaheim, CA, June 1988. IEEE Computer Society Press.

[ALG+90] Larry M. Augustin, David C. Luckham, Benoit A. Gennart, Youm Huh, and Alec G. Stanculescu. *Hardware Design and Simulation in VAL/VHDL.* Kluwer Academic Publishers, October 1990. 322 pages.

[BB89] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In van Eijk et al, editor, *The Formal description Technique LOTOS*, pages 23–73. North-Holland, 1989.

[BCG87] G. Beery, P. Couronne, and G. Gonthier. Synchronous programming of reactive systems: an introduction to Esterel. Technical Report 647, INRIA, Paris, March 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.

[Bry92] Doug Bryan. Rapide–0.2 language and tool-set overview. Technical Note CSL–TN–92–387, Computer Systems Lab, Stanford University, February 1992.

[Con63] M. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6:396–408, 1963.

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

[Gen91] B.A. Gennart. *Automated Analysis of Discrete Event Simulations Using Event Pattern Mappings.* PhD thesis, Stanford University, April 1991. Also Stanford University Computer Systems Laboratory Technical Report No. CSL–TR–91–464.

[GL92] Benoit A. Gennart and David C. Luckham. Validating discrete event simulations using event pattern mappings. In *Proceedings of the 29th Design Automation Conference (DAC)*, pages 414–419, Anaheim, CA, June 1992. IEEE Computer Society Press.

[KBL80] B. Krieg-Brückner and D. C. Luckham. Anna: Towards a language for annotating Ada programs. *ACM SIGPLAN Notices*, 15(11):128–138, 1980.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[LHM+86] D. C. Luckham, D. P. Helmbold, S. Meldal, D. L. Bryan, and M. A. Haberler. Task sequencing language for specifying distributed Ada systems: TSL–1. In Habermann and Montanari, editors, *System Development and Ada, proceedings of the CRAI workshop on Software Factories and Ada. Lecture Notes in Computer Science. Number 275*, pages 249–305. Springer-Verlag, May 1986.

[Luc90] David C. Luckham. *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs.* Texts and Monographs in Computer Science. Springer-Verlag, October, 1990.

[LvH85] D. C. Luckham and F. W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–23, March 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.

[MMM91] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of standard ML modules with subtyping and inheritance. In *Proceedings of the ACM conference on POPL 1991*. ACM Press, January 1991. Also Stanford University Computer Systems Laboratory Technical Report No. CSL–TR–91–472.

[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.

[Ros91] David S. Rosenblum. Specifying concurrent systems with TSL. *IEEE Software*, 8(3):52–61, May 1991.

[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.

[TM91] D. E. Thomas and P. R. Moorby. *The Verilog hardware description language*. Kluwer Academic Publishers, 1991.

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