Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events *

David C. Luckham †

August 29, 1996

Abstract

This paper describes the RAPIDE concepts of system architecture, causal event simulation, and some of the tools for viewing and analysis of causal event simulations. Illustration of the language and tools is given by a detailed small example.

1 Introduction

RAPIDE-1.0 [LKA+95],[LV95] is a computer language for defining and executing models of system architectures. The result of executing a RAPIDE model is a set of events that occurred during the execution together with *causal* and *timing* relationships between events. The production of *causal history* as a simulation result is, at present, unique to RAPIDE among event-based languages. Sets of events with causal histories are called *posets* (partially ordered event sets). [1] Simulators that produce posets provide many new opportunities for analysis of models of distributed and concurrent systems.

RAPIDE-1.0 is structured as a set of languages consisting of the Types, Patterns, Architecture, Constraint, and Executable Module languages. This set of languages is called the RAPIDE *language framework*.

The purpose of the framework is twofold: *(i)* to encourage multi-language systems, *(ii)* to define language components that may be applied to, or migrated into, other event generating systems. Towards *(i)*, we anticipate that the Executable Module, Constraint or Architecture sublanguages may be changed in fairly substantial ways, and that the Executable Module and Constraint sublanguages may be interchanged with other languages provided certain compatibility requirements are met. Towards *(ii)*, for example, the use of constraints expressed in terms of event patterns will have many applications to systems that generate events, not just the RAPIDE simulator. Such applications could include monitoring distributed object systems based on CORBA (or other commercial middleware) for security, for conformance to standards, and for many other properties.

The Types language provides the basic features for defining interface types and function types, and for deriving new interface type definitions by inheritance from previous ones. Its semantics

*This project is funded by DARPA under ONR contract N00014-92-J-1928 and AFOSR under Grant AFOSR91-0354

†Present members of the RAPIDE project team are: David Luckham, Walter Mann, John Kenny, James Vera, Sigurd Meldal, Neel Madhav, Alex Santoro, Francois Guimbretierre, Woosang Park, Marc Abramowitz, Alvin Cham, Claudio Garcia.

[1] Posets differ from Pomsets [Pra86] in that every event in a poset has a unique identifier; but posets produced by RAPIDE can contain equivalence classes of events, as we discuss later.

consists of the general rules defining the subtype (and supertype) relationship between types so as to allow dynamic substitution of modules of a subtype for modules of a supertype. The other sublanguages of the framework are extensions of the Types language. They assume the basic type definition features, and add new features in a way compatible with strong typing (i.e., every expression has a type). The architecture language extends the types language with constructs for building interface connection architectures. The Executable Module language adds modules, control structures, and standard types and functions. Standard types (i.e., data types available in many languages) are specified in a separate document as interface types. The Constraint language provides features for expressing constraints on the poset behaviors of modules and functions. The Event Pattern Language is a fundamental part of all of the executable constructs (reactive processes, behavior rules and connection rules) in the executable module and architecture languages, and also of the constraint language.

This paper is a short introduction to some of the topics and issues surrounding RAPIDE. Specifically, we discuss :

1. interface connection architectures.
2. the RAPIDE concepts of *event*, *cause*, and *causal event history*.
3. tools for depicting and analyzing causal event histories.
4. constructing a small example – a model of the Dining Philosophers – and viewing its behavior.
5. some research and development issues.

We do not have the space to deal with different concepts of *system architecture*, or the design of *constraint languages* that are specially suited to causal event behaviors. More information on RAPIDE can be found in the Internet Web page:

http://anna.stanford.edu/rapide/rapide.html

The treatment of concepts of *architecture* given here is an extremely cursory and incomplete excerpt from one of our publications. It is included here because there is currently so much vagueness about "architecture", and so much use of the term without any attempt to define what it means. People mean many different things by "architecture". It is important that the reader has some understanding of the concepts of architecture that have motivated the design of RAPIDE.

RAPIDE is an event-based language and simulation toolset. We give a short but quite detailed overview of *events*, causal histories of events, and RAPIDE computations. This is followed by a short description of the present tools to support architecture modelling, i.e., building models of system architectures, simulating architectures and analyzing simulation results.

Finally, there is an example illustrating some features of the RAPIDE POV (Point of View Viewer) for browsing causal event histories and constraint violation detection.

2 Interface Connection Architectures

The concept of architecture we shall illustrate here is called *interface connection architecture* [LVM95], so called because all communication between modules is explicitly defined by connections between interfaces — no longer are connections buried in the modules, but instead they are defined between the features in interfaces. Figure 1 depicts this kind of architecture.

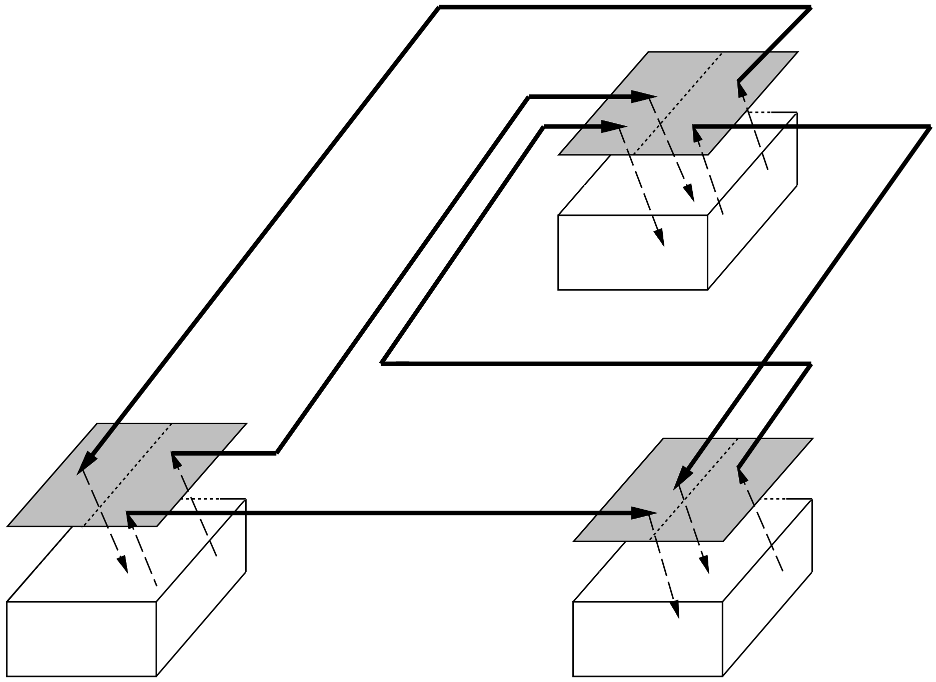

Figure 1: An interface connection architecture and conforming system

An interface connection architecture can be defined *before* the modules of the system are built. It can be used as a plan or early prototype of the system. To define interface connection architectures requires more sophisticated interfaces than are found in programming languages, and a completely new concept to define connections between interfaces. RAPIDE provides new features for representing *interface connection architecture*, not normally found in programming languages or middleware IDLs. In summary, the architecture features are:

- *interfaces* that specify both the features a module provides and, in addition, the features it requires from other modules. Moreover, there are two kinds of features, those implying synchronization (i.e., functions) and those implying asynchronous communication (i.e., actions). RAPIDE interfaces are more complex than, say, package specifications in Ada or classes in C++ which do not specify features required from other objects.

- *behaviors* in interfaces: Behaviors are sets of reactive rules that define abstract, executable specifications of the behavior that is required of modules in order to conform to that interface.

- *connections* between interfaces define relationships between the required features of interfaces and the provided features of the interfaces. The simplest kind of connection is *identification* between a required feature and a provided feature. [2] Identification connections have the effect that whenever a required feature is used then the connection invokes the provided feature in its place. More general kinds of connections allow sets of required features to be connected to sets of provided features.

  Connections are dynamic. A connection can depend upon runtime parameters, or the sets of features that are connected can vary at runtime, or the interfaces that are connected can also vary dynamically.

[2] Also called a *basic* connection.

- *constraints* are declarative statements that restrict the behavior of the interfaces and connections in an architecture. They can be used to explicitly specify requirements on the behavior of an architecture as a whole, or of its individual components. Conformance to constraints can be checked at runtime, or, in some cases, decided by proof methods.

RAPIDE interfaces can contain executable behaviors as illustrated in the example ahead. Connections are also executable in the sense that whenever their required features are invoked, they result in execution of the provided features that they connect to. Consequently, in RAPIDE an interface connection architecture can be executed (or simulated) before modules are programmed for its interfaces.

Example: *An Interface Connection Architecture with asynchronous connections*

```
type Producer(Max : Positive) is interface
    action out Send(N : Integer);
    action in  Reply(N  : Integer);
behavior
     Start => Send (0);
    (?X in Integer) Reply(?X) where ?X < Max => Send(?X + 1);
end Producer;

type Consumer is interface
    action in Receive(N : Integer);
    action out Ack(N : Integer);
behavior
    (?X in Integer) Receive(?X) => Ack(?X);
end Consumer;


architecture ProdCon() return SomeType is
    Prod : Producer(100);
    Cons : Consumer;
connect
    (?n in Integer)
    Prod.Send(?n) => Cons.Receive(?n);
    Cons.Ack(?n) => Prod.Reply(?n);
end architecture ProdCon;
```

Commentary:

Here is a simple example of an interface connection architecture. There are two interface types, Producer [3] and Consumer. They define the interfaces of producer and consumer objects. These interfaces define asynchronous communication features called *actions*. The Producer interface contains two actions, an **out** action Send and an **in** action, Reply. Objects of this type can generate Send events and can receive Reply events. The Producer interface also contains reactive behavior transition rules. The first one triggers on a Start event (which all objects receive when they are elaborated) and reacts by generating an **out** event, Send(0). The second behavior rule triggers on Reply events containing integer data provided the data is less than Max, and generates a Send event containing the next integer.

[3] More accurately, Producer is a *type constructor*; when it is applied to a value for its parameter, Max, the result is a type.

The Consumer interface type is similar. It can receive Receive events, and its behavior is to react by generating Ack events containing the same data.

The ProdCon architecture contains two components, Prod and Con, each of an interace type. Connections between these two components are defined by two reactive connection rules. The first triggers whenever Prod generates a Send and reacts by generating a Receive event of Cons with the same data. So it connects the Prod component's **out** action Send and the Cons component's **in** action Receive. The second rule connects the Cons component's **out** action Ack with the Prod component's **in** action Reply.

The architecture defines the communication between the two components in terms of the actions in their interfaces. The architecture can be refined by introducing detailed modules for Prod and Cons. The refined system is called an *instance* of the interface connection architecture. As long as the new modules communicate only by calling the actions in their own interfaces, they will communicate in the instance only as defined by the connections in the architecture. The instance conforms to the architecture if its modules behave consistently with the behavior rules defined in their interfaces, and preserve its communication. Conformance implies that many properties of the system are defined by its architecture. For example, conformance implies that the instance will have the property that an interval of integers, $0, 1, 2, \ldots, Max - 1$ is communicated between the producer and consumer, according to a protocol whereby there is an acknowledgement from the consumer before the next integer is sent by the producer.

□

An interface connection architecture can be used to analyze properties of a system that conforms [4] to that architecture, but we have to omit discussion of this topic here.

Although Figure 1 depicts connections between the interfaces as static wires, interface connection architectures in RAPIDE can be *dynamic* architectures. Typically, a *static* architecture has a fixed number of components and a fixed number of connections, and the properties of the connections do not vary at runtime — hardware architectures that can be modelled in languages like VHDL [VHD87] are a typical example. In a *dynamic* architecture the numbers of (interfaces of) components can vary at runtime, and connections between the interfaces can exist or not, depending upon runtime conditions. An air traffic control system is an example of a dynamic architecture with varying numbers of aircraft and connections that depend upon distance, radio frequency, and other factors.

2.1 Conformance of a System to an Architecture

An interface connection architecture can be built *before* any system of modules that, in some sense, *has* that architecture. How can it be decided if a system has that architecture?

An architecture defines a constraint on all of its instances. That is, an architecture is a formal constraint on the system's behavior. Conversely, a system *has* an architecture if it *conforms* to it. There are three basic conformance criteria:

1. **decomposition** : for each interface in the architecture there should be a unique module corresponding to it in the system (i.e., the component implementing that interface). [5]

[4] see Section 2.1.

[5] More sophisticated decomposition criteria, such as requiring an abstraction mapping from sets of system components to interfaces in the architecture, are allowed by RAPIDE mapping constructs, but are beyond the scope of this presentation.

2. **interface conformance**: each component in the system must conform (as described below) to its interface. Since RAPIDE behaviors and constraints can be part of interfaces, this conformance criterion is, in general, stronger than the syntactic interface conformance usually required by programming languages.

3. **communication integrity**: All communication between components is constrained by the architecture. Two components cannot communicate with each other directly unless a connection (possibly conditional on runtime parameters) between their interfaces is defined by the architecture.

Conformance of modules to interfaces in programming languages usually only requires a module to contain features that match names and parameter signatures of features in their interfaces — a simple compiletime check. In RAPIDE conformance of modules to interfaces requires satisfying both behavior and semantic constraints. In general, determining if a module conforms to an interface with semantic constraints is an undecidable problem, although there are many useful cases where it can be decided by practical methods. Developing tools for testing architecture simulations for conformance to constraints, and proving correctness of architectures, is a challenging activity at present. The RAPIDE toolset supports testing for interface conformance by both compiletime and runtime checking (see Section 6). In the future, tools for applying proof techniques to interface conformance may be added.

Although determining interface conformance in the presence of behaviors and constraints is a tougher problem than the usual syntactic signature requirements between interfaces and modules, it allows us to conclude much more about the modules of a system. Interface constraints allow us to specify modules sufficiently to ensure that any two modules conforming to an interface can be interchanged without changing the behavior of the system. Also, if the connections in an architecture are correct — i.e., the constraints on a provided feature logically imply the constraints on a required feature connected to it — then instances of the architecture where interfaces have been assigned modules conforming to those interfaces will also have correct connections.

There are many strategies one may adopt to try to ensure that a system satisfies communication integrity. For example, one may adopt restrictions on the coding of modules (called a *style guides*). A possible style guide could be that a module should be constrained to only communicate with other modules of the system, or its parent architecture (i.e., the architecture of which it is a component), through its own interface, as shown in (Figure 1). This style helps to ensure that only the communication defined in the architecture takes place between modules. [6] Sufficient conditions for communication integrity involve *(i)* restrictions of the RAPIDE visibility rules, and *(ii)* restrictions of the types of parameters of actions and functions so that particular types of objects cannot be passed between components.

2.2 The Role of Constraints in defining architectures

Constraints in RAPIDE are event pattern constraints. That is, they define patterns of events which must, or must not, occur during the execution. Constraints can be part of a RAPIDE interface or an architecture. Generally, constraints in an interface are used to specify restrictions on the behavior of modules with that interface. Constraints in architectures are used to restrict the activity in the architecture. For example, constraints can specify certain sequencing of communication between components, such as might be required by a particular protocol. When a module or architecture executes, its behavior is checked for conformance to the constraints.

[6] This style, although encouraged, is not enforced by RAPIDE for reasons discussed later.

3 Event-based Computation

A *computation* is a set of events together with partial orderings that relate events in the set. The partial orderings in RAPIDE 1.0 computations represent *dependence* between events and the *time* at which events happen with respect to various clocks. The dependence relation is also called the *causal* relation since it models which events caused an event to happen. Computations are also called *executions*.

3.1 Events

An *event* is an object generated by a call to an action. An action declaration defines an associated *event type*; this event type is the type of events generated by calls to that action. Every action call generates new event which is distinct from all previous events.

The constituents of an event are the name of the generating action, parameters, information [7] defining which events caused the event, and timestamps. Thus an event, may be defined as a tuple consisting of these data. Not all of these constituents are visible to the user. Some constituents (such as the dependency information) are useable only through predefined operations.

3.2 Operations on events

An event is *generated* by an action call . The predefined **event** type provides the following operations on events:

1. E . Action_Name — a string naming the action used to generate E,
2. E . From_Module — the object that generated E,
3. E . parameter_name — the value of a component of E corresponding to the parameter_name of the action E . Name,
4. C . Start (E) — the start time of E according to clock C, undefined if E was not generated in the scope of C,
5. C . Finish (E) — the finish time of E according to clock C, undefined if E was not generated in the scope of C.

Timing orderings are imposed by the clocks (if any) in a RAPIDE 1.0 program. Each clock partially orders the events that are generated within its scope.

3.3 Relationships between events

RAPIDE provides facilities for defining and referencing two kinds of relationships between events:

- *dependence*,
- *time* with respect to a clock.

Both relationships are partial orderings of events.

[7] Dependency is captured by means of so-called Fidge-Mattern vectors of counters [Fid91],[Mat88]; we omit discussion of details here.

4 How events are generated in Rapide

Roughly speaking, all active modules [8] in a RAPIDE model execute independently. They observe and generate events, and they do this independently of each other unless their activity depends upon the events they observe. Modules can themselves be multi-threaded, so dependency is not defined at the object level, but rather at the level of *process*, *behavior rule*, *connection rule* and reading/writing *operations* on particular types of object.

Models are *hierarchical*, that is modules can themselves be architectures of modules [9]. Whether or not two modules can communicate at any given point in an execution depends upon a concept of the *context* of an module. Roughly speaking, a module can observe events resulting from a module invoking its interface actions and functions, can invoke interface features of a module in its context, and can braodcast events to those modules in whose context it is. Context starts out initially corresponding to well known scope rules of algol-like languages, but varies at runtime because modules can be passed as event or function parameters to other modules.

4.1 Generating dependent events

There are three kinds of language features that define dependence between events:

- *reactive* rules and processes. Reactive rules are transition rules in interface behaviors, connection rules in architectures, and mapping rules in maps. Reactive processes are **when** statements in modules. Whenever a set of events with the required relationships is observed (see later) by a reactive rule or process, and *matches* its trigger, then that rule or process executes (i.e., the rule or process *triggers*). The events then generated by the rule or process on that particular execution depend upon those events that triggered it.
- *sequential* code — events generated by sequential executions have a strict linear dependence represented by their order of generation,
- *ref objects* — a **ref** type object defines dependence between the events generated by the processes that share the object. If an event results from a computation that dereferenced a **ref** object, then the event depends on the (unique) event that last changed the contents of that **ref** object — i.e., operations on **ref** objects are linearly ordered.

4.2 Generating timed events

A type or module can be *timed* by having a clock associated with it, or by being placed in the scope of a clock. The type Clock and several subtypes of it are predefined. Events receive start and finish time values for each clock within whose scope they are generated.

Action calls may specify the time taken for actions to be performed with respect to a clock. If action A is defined to take duration d with respect to clock C, then for an event E generated by A, $\mathsf{C.Start(E)} + d = \mathsf{C.Finish(E)}$. If no duration is defined, the events are generated infinitely fast with respect to any clock.

[8] We use "Module" and "object" synonomously here.
[9] called their components

4.3 Observation of Events

The events generated by a module of a RAPIDE program are *visible* to, and can be *observed* by, other modules if it is in their context. Conversely, events are made *available* to a particular module for observation when other modules call its interface **provides** functions and **in** actions.

The ability to call the interface functions and actions of a module depends upon visibility rules of the language. For example, a typical way (but not the only way) that a module can communicate with another module is by generating events which trigger connection rules in the architecture in which both modules are components. Both components are visible to the architecture. The connection rules are then executed by the architecture and result in calls to the other module's interface functions or actions.

4.4 Observing events by pattern matching

Features for observing events, and the dependency ordering between events, are provided by the *pattern language*. Briefly, patterns are templates that allow the definition of posets of events. Patterns can specify relationships between events by means of dependence operators. For example, "A depends on B" is written as $\mathsf{A} \rightarrow \mathsf{B}$, and "A is independent of B" is written as $\mathsf{A} \mid\mid \mathsf{B}$.

The timing parameters of events can be accessed by clock operations as described in Section 3.2. The pattern language also defines abstract pattern operations dealing with time and events.

Patterns can be used as the triggers of: *(i)* processes, *(ii)* transition rules in interface behaviors, *(iii)* connections in architectures, and *(iv)* map rules. Events that are made available to a module are observed by *matching* the pattern triggers of the reactive rules or processes in the module. [10] If and when an event contributes to matching the pattern trigger of a rule or process, it can no longer be observed by that rule or process.

Finally, patterns can be used in formal constraints. Events that are available to a module are also observed by matching the constraints of that module. Violations of constraints are reported as they happen.

4.5 Orderly observation

Events are *observed* (that is, considered for matching in patterns) in an order consistent with the causal ordering of the events. That is, an event may only be observed after all events it depends on, and independent events may be observed in any possible order. This principle is known as *orderly observation*.

Orderly observation is important in efficiently matching patterns that refer to dependencies between events.

5 Computations

The events generated by all the modules of a RAPIDE 1.0 program comprise the computation [11] generated by the program. A computation consists of a set of events, S, a dependence partial ordering, $\leq_d$, and timing partial orderings, $\leq_C$, for objects C of type Clock.

$A <_C B$ is defined as $C.Finish(A) < C.Start(B)$.

[10] Pattern matching is explained in the RAPIDE LRMs (see http://anna.stanford.edu/rapide/rapide.html).

[11] Also called an *execution* or *simulation history*.

Posets generated by RAPIDE programs satisfy two invariants.

The dependence *and* time orderings satisfy a consistency invariant:

- **Consistency invariant between dependence and time**
  For all events A, B, and each clock, C,

$$\mathrm{A} \leq_d \mathrm{B} \rightarrow \mathrm{A} \leq_C \mathrm{B}.$$

This means that, with respect to each clock, an event in the past may not depend on an event in the future.

The consistency invariant may be expressed within RAPIDE 1.0 as a constraint:

```
never
    ( ?A in event(), ?B in event(), ?C in Clock )
    ?A -> ?B where not ?C.Finish(?A) ≤ ?C.Start(?B);
```

Similarly, the various time orderings obey a consistency invariant:

- **Consistency invariant between time orderings**
  For all events A, B, and clocks, C, C',

$$\mathrm{A} <_C \mathrm{B} \rightarrow \textbf{not } \mathrm{B} <_{C'} \mathrm{A}.$$

This means that, with respect to any two clocks, an event that precedes any event temporally with respect to one clock may not follow that event temporally with respect to the other clock.

This invariant may be expressed within RAPIDE 1.0 as a constraint:

```
never
    ( ?A in event(), ?B in event(),
      ?C1 in Clock, ?C2 in Clock )
    ?A ~ ?B where ( ?C1.Finish(?A) < ?C1.Start(?B) and
                    not ?C2.Finish(?B) < ?C1.Start(?A) );
```

6 Architecture Simulation and Analysis

When a RAPIDE model is executed a causal event history of its behavior is generated. Each object receives a Start event when it is elaborated. It may react by generating events. Connections between components of the model (i.e., objects it contains) then trigger and generate events at the interfaces of components. Components react to these events, and generate further events, which are communicated by connections to other components, and so on. Models can be organized hierarchically, and connections can also connect actions in the interface of an object with actions in interfaces of its components. A model can also interact with its environment through a predefined I/O module.

The simulation result is a poset showing the causal history of events in the execution, independent activity, timing, dataflow and other properties. This gives us the capability to simulate

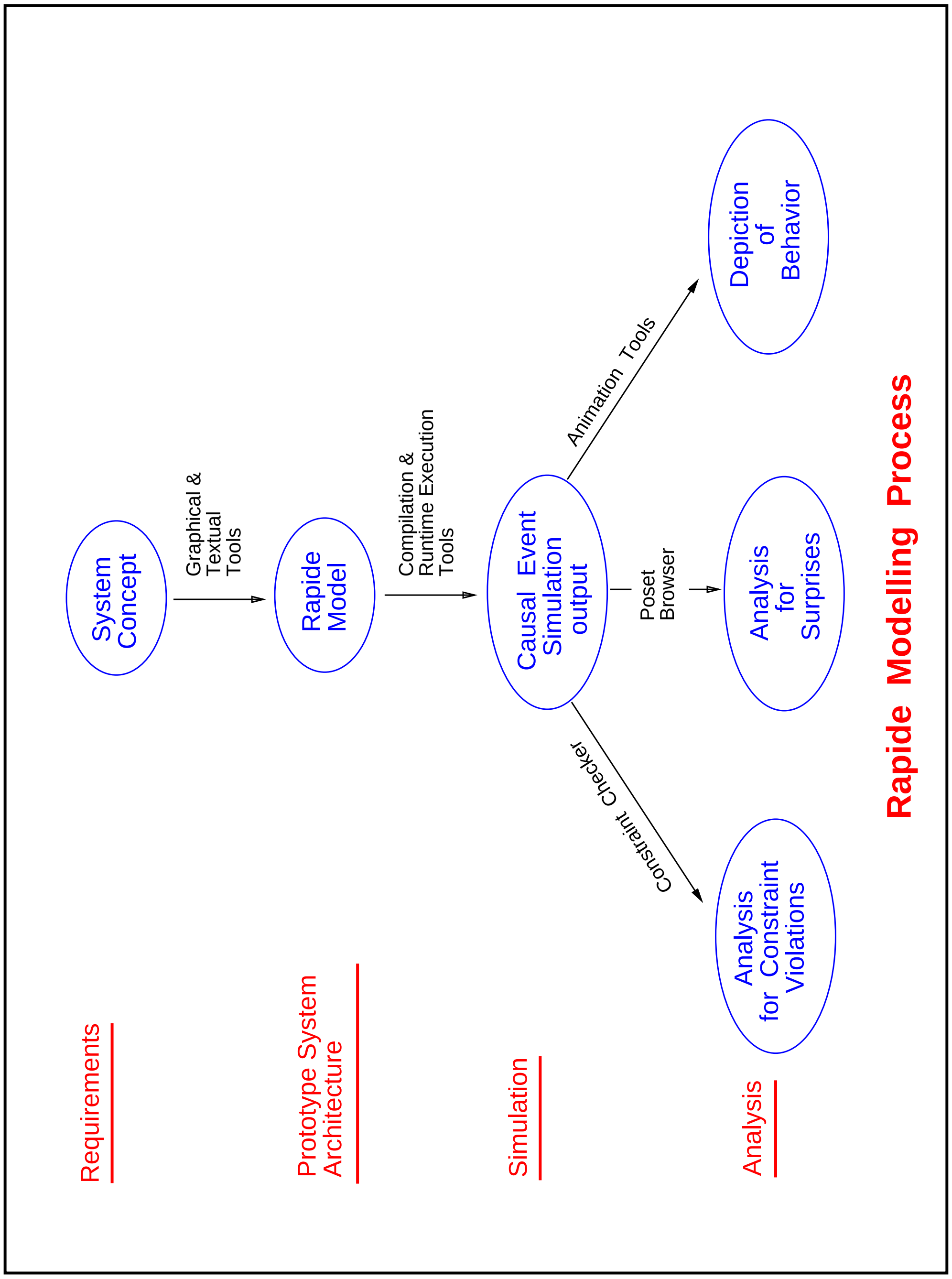

Figure 2: The Rapide simulation and analysis toolset

the behavior of a system architecture very early during the requirements analysis and design phases of the system. [12]

[12] The RAPIDE tools shown in Figure 2 have been implemented by Walter Mann, James Vera, John Kenney, Alex Santoro, Francois Guimbretierre, Earnest Lam, Alvin Cham, Marc Abramowitz, Wolf Polak, and Doug Bryan.

To make maximal use of causal event simulations requires new kinds of analysis tools. RAPIDE is presently supported by three kinds of tools for analyzing simulations:

- *Constraint Checkers.* Constraint checkers automatically detect violations of constraints in the simulations. Whenever a violation is detected, the checker enters an event in the simulation at the causal position where it happened. The causal history of a violation event indicates why the violation happened.
- *Poset Browsers.* It is often necessary to browse a simulation simply to see how a given architectural design behaves. Indeed, one of the most important uses of early lifecycle simulation is experimental. Poset bowsers usually represent causal event simulations in a DAG form, nodes representing events and directed arcs representing causality. They supply a user interface with pattern directed operations to display pieces of large simulations, and to organize a display into a hierachy of sub-simulations.
- *Animation Tools.* The wealth of information in posets together with the somewhat abstract display formats provided by browsers, often results in the human viewer missing important properties. [13] Animation tools give the user a capability to animate in visual forms the event activity in any poset. usually, a picture of the architecture is used as the basis for animation, and both the poset and the animation can be viewed simultaneously. The user is thereby given a human understandable intepretation of a poset simulation — in fact more than one animation view of the same poset can be provided.

6.1 Architecture-Driven System Development

RAPIDE is also intended to allow exploration of new methods of using architectures to build and test systems. In one approach to *architecture driven* system development, a RAPIDE interface connection architecture is used as a framework for building a system. One starts with an architecture consisting of interfaces and connections. The architecture is executed under different input scenarios to simulate the behavior of a system with that architecture. Assuming simulations of the architecture show behavior that meets the requirements for the system, modules are then assigned to interfaces one at a time. Each module must conform to the interface it is assigned to (RAPIDE type rules help towards this, but as mentioned in Section 2.1, testing satisfaction of semantic constraints is a difficult problem.) When a module is assigned to an interface, the module is executed and the role of the interface behavior is to act as a constraint to which the module's behavior must conform (in addition to the semantic constraints). The result of assigning a module to an interface is called an *instance* of the architecture. Each instance is tested for comformance to the architecture's interface constraints, and also to the constraints on the architecture's connections. The final result should be a system of modules satisfying the architecture's interface and constraints.

This style of architecture-driven system development has an analogy with hardware. Interface connection architectures can be viewed as "architecture boards" in which interfaces play the role of "plugs" and "sockets" into which component modules can be plugged, and connections play the role of "wires" between the sockets.

Methodology surrounding the use of architectures, to prototype behavior and predict system performance early in the life cycle, and to develop finished systems by instantiation (i.e., replacing interfaces by modules, probably in languages other than RAPIDE) is beyond the scope of this overview. There are many outstanding research questions surrounding RAPIDE, both practical (e.g., developing good simulation and analysis tools), and theoretical (e.g., determining if a module conforms to its interface, and if the connections in an architecture satisfy the architecture constraints).

[13] This happens in all manner of simulations nowadays, whether or not causal information is avialable.

7 Dining Philosophers

The familiar classic example of the dining philosophers due to Dijkstra is a simple distributed system with a resource contention problem. Here is a version of it that serves to illustrate some of the RAPIDE architecture concepts and issues related to designing "user friendly" tools for analyzing posets.

You can see the code for this architecture in the figures ahead. There are two types of component, a type of round table at which a number of philosophers can sit, and a type of philosopher.

To understand the architecture, first scan the interface types, then see how the objects of those types are connected in the architecture.

The Table interface is parameterized by the number of philosophers — it is a type constructor, each instance of which is a type of table object, e.g., the type of tables for one philosopher, two philosophers, etc. A table controls and dispenses the chopsticks by means of its interface actions. Actions specify the events that objects of the interface type can receive or generate. Tables can receive StickRequested and StickRecovered events (mode **in**), and can generate ReleaseStick events (mode **out**).

The behavior of tables is specified by the reactive rules in the behavior part. Essentially, a table will release a chopstick to a requestor when the chopstick becomes free (i.e., is initially free, or is recovered).

The rule,

```
(?a in Chopstick, ?i in PhilosopherId)
StickRequested(?a, ?i) and FreeStick(?a) ||> ReleaseStick(?a, ?i);;
```

will trigger whenever a chopstick ?a is requested by something denoted by ?i (the architecture connections will show that ?a, ?i have to be bound to ids for a Chopstick and a Philosopher) *and* ?a is free. "?a is free" if an event FreeStick(?a) has been generated and not yet used to trigger a rule (the RAPIDE semantics of behavior rules allows an event to be used only once to trigger any given rule, but may contribute to triggering different rules by helping to match their pattern triggers.) When the rule triggers it generates an **out** event, ReleaseStick with the bindings of ?a, ?i.

The ||> operator indicates an *agent* rule. The semantics is that each triggering of an agent rule is executed by a new thread of control. As a result, the event generated by the rule will be causally dependent upon the two triggering events, but will be independent of any events generated by previous triggerings of the rule.

A Chopstick gets to be free again according to this behavior rule:

```
(?a in Chopstick)
StickRecovered(?a)    ||> FreeStick(?a);;
```

Note that FreeStick is an internal event of the Table types (i.e., is declared in the behavior part), which means that it is not visible at the architecture level.

The event behavior of a Table (i.e., the causally related events it will generate in response to the events it receives) can be predicted from the semantics of behavior rules. The three behavior rules of Tables execute independently, so the events they generate are independent – unless the rules are triggered by events that other rules generated, like FreeStick. So a ReleaseStick event will depend upon a FreeStick and a RequestStick event – and by transitivity, the events that caused them. So it will also have a StickRecovered event (with the chopstick in question as argument) in its history.

Each table is a multi-threaded object, executing independently of other objects, unless it communicates with them by means of the architecture's connection rules.

```
----------------- Types ----------------------------

    type PhilosopherId is integer;
    type Chopstick is integer;

----------------- Table type. ----------------------

type Table(numPhils : integer) is interface
    action in  StickRequested(n : Chopstick; id: PhilosopherId),
                 StickRecovered(n : Chopstick);
    action out ReleaseStick(n : Chopstick; id: PhilosopherId);
behavior
    action  FreeStick(n : Chopstick);
begin
    start =>  for i: integer in 0..(numPhils-1) do          -- table rule 1.
                   FreeStick(i);
               end for;;

    (?a in Chopstick, ?i in PhilosopherId)                  -- table rule 2.
     StickRequested(?a, ?i) and FreeStick(?a) ||> ReleaseStick(?a, ?i);;

    (?a in ChopStick)                                       -- table rule 3.
     StickRecovered(?a) ||>  FreeStick(?a);;
end Table;
```

The philosopher interface type specifies philosopher objects. They can perform two activities: they can generate RequestStick and PutDownStick events (**out** actions), and they can receive StickReceived events (an **in** action). These events carry Ids for a chopstick and a Philosopher. Philosophers are generated by a simple module generator, newPhilosopher, which takes the Id and the number of philosophers as parameters, and generates a module containing instances of the concurrent processes. These processes define the behavior of philosophers: i.e., when a philosopher generates RequestStick and PutDownStick events, and when it generates certain internal events such as thinking and eating.

The semantics of RAPIDE processes allows us to reason about the event behavior of Philosophers. A philosopher must acquire two chopsticks in order to eat rice. The two processes that trigger on a Hungry event will request the two chopsticks independently. Another process triggers when a pattern of two StickReceived events match (each having a correct chopstick for the philosopher), where the events may be either causally dependent or not (the $\sim$ relation). This process then generates an Eat event, which in turn will trigger a process that generates PutDownStick events. Consequently, the StickReceived events will be in the causal history of each of the PutDownStick events.

```
-------------------Philosopher type --------------------
type Philosopher is interface

   action out RequestStick(n: Chopstick; id: PhilosopherId),
                  PutDownStick(n : Chopstick);
              in   StickReceived(n : Chopstick; id: PhilosopherId);

end;


--------------------module generator for philosophers--------

module   newPhilosopher(id: PhilosopherId; numPhils:integer) return Philosopher is
    action Hungry(),
           Think(),
           Eat();

parallel
    when Start
    do
        Hungry();
    end;


    ||
    when Hungry
    do
        RequestStick(id, id);                        -- request left chopstick
    end;


    ||
    when Hungry
    do
        RequestStick((id+1) mod numPhils, id);     -- request right chopstick
    end;


    ||
    when
        StickReceived(id, id) ~                      -- received left chopstick
        StickReceived((id+1) mod numPhils, id)     -- received right chopstick
    do
        Eat();
    end;
```

```
    ||
    when
        Eat
    do
        PutDownStick(id);
        PutDownStick((id+1) mod numPhils);
        Think();
    end;
end;
```

NOTE: we could have used behavior rules in the philosopher interface, as we did for Tables, but we would have to introduce the number of philosophers as a parameter, and this is an implementation detail which should not appear in the interface. So we illustrate the use of module generators (which, generally, can contain more complex programs than interface behaviors).

The architecture, diners1(), contains one table and an array of 5 philosophers. Philosophers (i.e., the objects generated by the newPhilosopher module generator) have an Id number which differentiates them from their colleagues. The table associates two chopsticks with each philosopher by means of chopstick Ids (actually all Ids are integers anyway!). The Ids encode the model of a chopstick being placed to the left and to the right of each philosopher, In diners1, philosophers and the table interact through actions defined in their interfaces. These actions are wired together by connection rules in the **connect** section of the architecture.

A connection rule triggers on a pattern of **out** events generated by objects in the arhcitecture (and also **in** events of the architecture's interface received by the architecture itself) and in turn generates a pattern of **in** events which are received by objects (and also **out** events of the architecture's interface). [14] An example connection rule is:

```
    (?a in Chopstick, ?i in PhilosopherId) Ta.ReleaseStick(?a, ?i)
     to
     Ph[?i].StickReceived(?a, ?i);
```

which is a point-to-point communication between the Table and each of the requesting Philosophers. It connects the Table's **out** action, ReleaseStick to a unique Pilosopher's **in** action, StickReceived, depending upon the Id of the Philosopher. Here "connect" means that whenever the Table generates a ReleaseStick event the rule will trigger and generate a StickReceived event of the Philosopher whose Id is the binding of ?i. The connection operator, **to** means that the two events are causally equivalent (i.e., indestinguishable by either the cause or time relations).

We use a generate statement to define the connections between the **out** actions of each Philosopher in the array and the **in** actions of the Table — a "fan-in" of 5 rules; the generate is an iterative statement that defines the set of connections that are instances of its connection rules for each value of the iteration parameter, j. [15] There are two sets of 5 rules in the **generate**. The first set of rules defines connections so that whenever a Philosopher generates RequestStick events, the Table will receive StickRequested events with the same parameter bindings. The second set of connections trigger whenever a Philosopher generates a PutDownStick event for a chopstick ?a and then generate a StickRecovered for the same ?a which is received by the Table.

When diners1 is executed, the Philosophers activity is triggered by their Start events, which makes them hungry, then they request chopsticks, etc. Philosophers are active for only one

[14] Architectures can be nested as components of other architectures through their interface events.

[15] Generate connection rules can be found in hardware description languages like VHDL. In combination with the RAPIDE pattern-triggered connections, they are a powerful notation for defining large numbers of connections.

cycle of eating and thinking; it is easy to change this so they continue to cycle, but we don't do it to keep our posets small.

NOTE that different executions of diners1 may produce posets with different causal relationships between events because of the non-deterministic behavior rules (if two rules trigger on the same events, RAPIDE does not define an order in which they execute). Indeed, some posets will show that all philosophers ate while other posets will show the usual deadlock with nobody eating.

```
--------------- Dining Room architecture. --------------------

architecture diners1() is

   num_philosophers : integer is 5;

   Ph : array(PhilosopherId, Philosopher) is (
                                   0 is newPhilosopher(0, num_philosophers),
                                   1 is newPhilosopher(1, num_philosophers),
                                   2 is newPhilosopher(2, num_philosophers),
                                   3 is newPhilosopher(3, num_philosophers),
                                   4 is newPhilosopher(4, num_philosophers)
                                       );
   Ta: Table(num_philosophers);

connect
   (?a in Chopstick, ?i in PhilosopherId) Ta.ReleaseStick(?a, ?i)
    to
    Ph[?i].StickReceived(?a, ?i);

    for j : PhilosopherId in 0..num_philosophers-1   generate

       (?a in Chopstick, ?i in PhilosopherId) Ph[j].RequestStick(?a, ?i)
        to
        Ta.StickRequested(?a, ?i);

       (?a in Chopstick )Ph[j].PutDownStick(?a)
        to
        Ta.StickRecovered(?a);
    end;
end architecture diners1;
```

8 Viewing and Analyzing Posets

Figure 3 is a DAG representation of a poset generated by this model. The POV (Point Of View) viewer displays a poset in this kind of representation, and provides capabilities to explore the poset. [16]

[16] The POV is being implemented at Stanford by Francois Guimbretierre, Earnest Lam, Alvin Cham and Marc Abramovitz. An earlier prototype viewer was implemented by Doug Bryan.

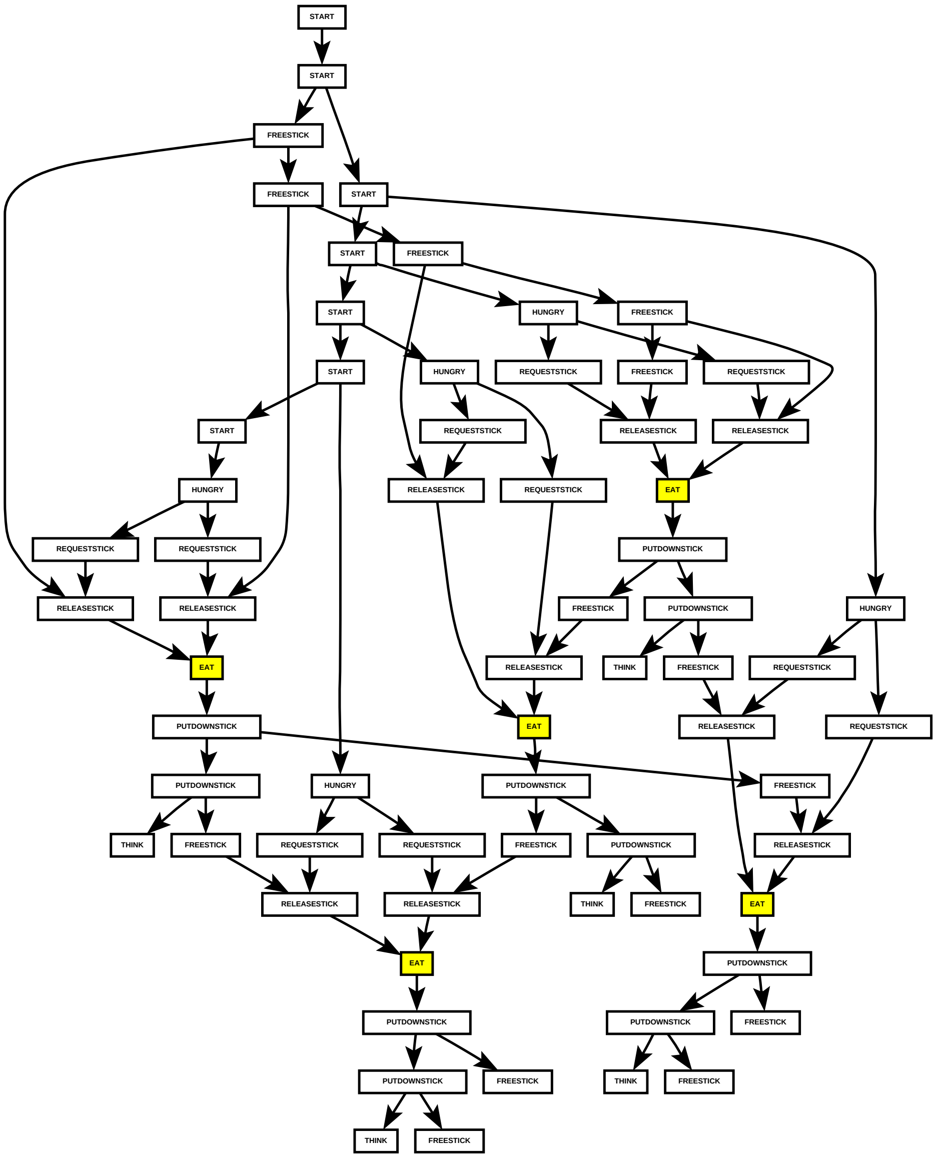

Figure 3: A poset generated by the diners1 system

Figure 3 is a very simple example of a poset. The DAG is oriented vertically with the causally earlier events at the top. Here are a few things it tells us at first glance about the execution of our dining philosophers. There are 7 Start events, one for the Diners1 architecture, the Table and the 5 Philosophers. They are all in a causal chain corresponding to the execution of a single thread that elaborated Diners1. The second start is the Table's because it immediately causes a FreeStick. Each start causes a sub-poset of events, the initial events being all independent of one another — so there are initially 7 threads of control after starting. We can see the Table's first sequential behavior rule results in a single causal chain of FreeStick events. (Using the POV we could trace this chain by placing the cursor on its arcs. An arc "lights up" when the cursor is on it and the thread id and causally related events coresponding to the arc are displayed in the window. So a chain corresponding to a single thread can be easily traced.)

A Philosopher, after starting, immediately generates a Hungry event which then triggers two parallel processes (see the newPhilosopher module). This causes two independent RequestStick events. These are shown as causing ReleaseStick events generated by the Table. How did that happen? Well, a philosopher's RequestStick action is connected to the Table's StickRequested action by a **to** connection rule. This kind of rule implies that the two events, a triggering RequestStick and the event it generates, StickRequested, are *equivalent*. So the node showing "RequestStick" is an equivalence class of two events, but only one of them is displayed. (By clicking with cursor and mouse on an event node, POV will display all events in the equivalence class.)

Tracing a little further, a StickRequested event received by the Table, and a FreeStick event with the same Chopstick Id, will trigger the Table's second behavior rule and generate a ReleaseStick event. So, a ReleaseStick event is caused by two events. Again, due to a **to** connection rule (the first one), a node labelled with "ReleaseStick" represents an equivalence class containing also a StickReceived event received by a Philosopher. When a Philosopher receives two StickReceived events, it eats and then generates PutDownStick events for the two sticks. Notice that the PutDownStick events are causally ordered because they are generated by a single sequential process in a Philosopher, in constrast to the RequestStick events. There is no good reason for this — they could be put down independently.

More importantly, notice that since an Eat event must have two StickReceived events causing it, and it causes the PutdownStick events, then the StickReceived events are in the causal history of both PutDownStick events for the same pair of chopsticks that were used in eating.

The PutdownStick events are connected (and equivalent by the third **to** connection rule) to the Table's StickRecovered events, which cause the Table's behavior to generate FreeStick events. A Philosopher's second PutDownStick event is followed by a Think event, which is the end of its activity.

Figure 4 shows some features of POV. First, the poset in Figure 3 has been filtered down to the poset in Figure 4. This filtering was done by using the POV to select the causal history of the third StickReceived event. Events that are not selected are deleted from the poset. In general one can select a sub-poset matching a given pattern. This feature allows the user to reduce the poset to events "of interest". So, what we are looking at is the causal history of a particular StickReceived event. All StickReceived events in this poset have been "selected" – a facility of the POV which results in selected events being colored. The tearoff windows show the result of querying the selected (colored) events with the POV by clicking on them. Each window shows the data of the event: *(i)* the name of the corresponding action, *(ii)* the Id of the object generating the event (by calling the action), *(iii)* the Id of the object receiving the event, and *(iv)* parameters of the events. Parameter #1 of these events is the chopstick Id. So we see that Philosopher #31 received sticks #1 and #2, which allowed it to eat, and causally afterwards to put down stick #1, which was then freed. Stick #1 was then received by Philosopher #30. [17] It is not hard to see using the POV on the full poset that all StickReceived events with the same chopstick as parameter are causally related. This gives us a hypothesis (**H**) that StickReceived events with the same chopstick parameter are always causally related in any poset behavior of this model, which is easily shown to be true.

The poset in Figure 3 has varied during the writing of this paper. Sometimes it shows that all Eat events are causally related. But, the model can also generate posets in which some Eat events are independent because the Philosophers may be scheduled differently on different executions. The DAG layout of the poset makes it hard to see which of these two posets we have in the Figure. This illustrates the need for ways to select events of interest — e.g., we might like to be able to say, "show all the Eat events" and get displayed the subposets containing just those events. More generally, we may want to organize a complex poset into subcomputations

[17] Stick #2 was also put down, but does not appear in the causal history of the event whereby Philosopher #30 receives stick #1.

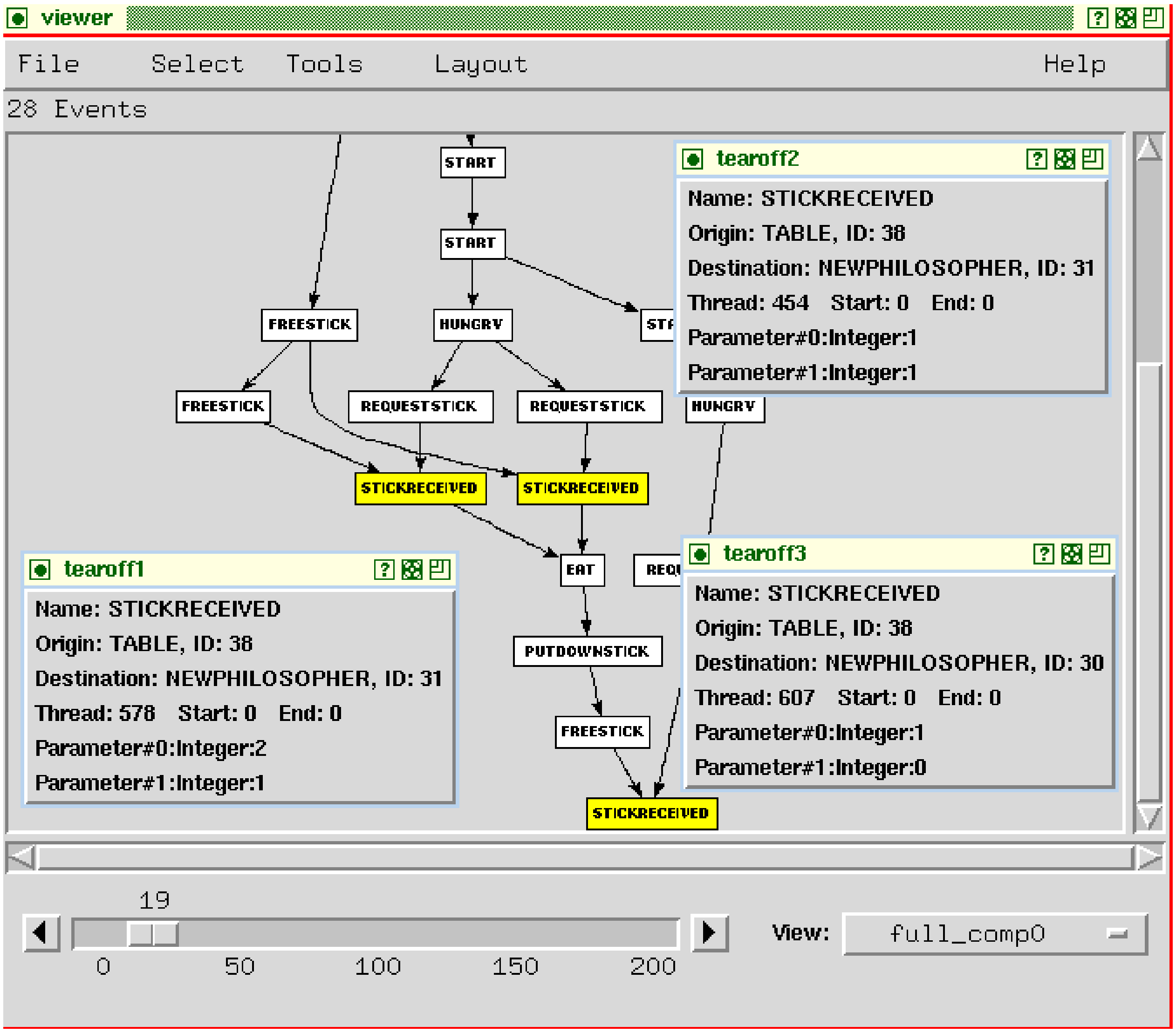

Figure 4: Exploring a poset with the POV

corresponding to abstraction levels in the model's architecture and components — e.g., just the top level events in Diners1. POV provides such capabilities, but we cannot describe them here.

Figure 5 shows another behavior of Diners1 — one in which deadlock occurs.

9 Constraining poset behaviors

Figure 5 brings up another question. Remembering that Rapide is intended as a prototyping language, which means essentially that it can be used to explore properties of a proposed system architecture *before* it is really well understood or "completely decided upon". If the posets produced by a model are large or complex, one may well miss seeing some important aspects of the model's behaviors with the POV. So, Rapide provides a formal constraint language which

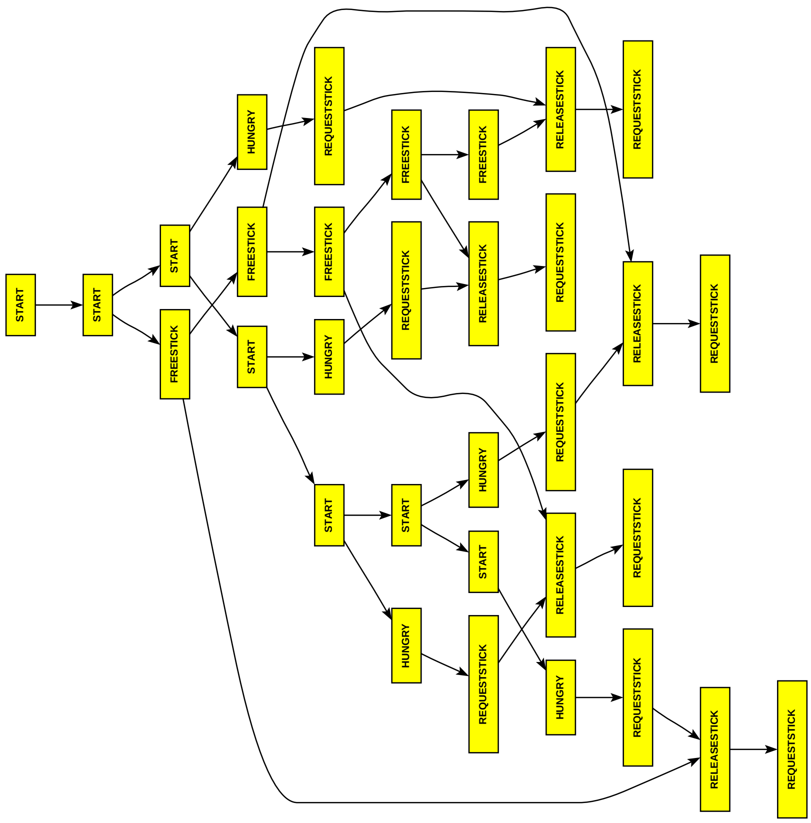

Figure 5: Deadlock in the dining room

can be used to specify constraints on behaviors. There is a constraint checker which will detect if a poset contains violations of constraints. [18]

Constraints can be used in two ways. First, to automatically analyze posets for particular properties. Secondly, to build a purely constraint-based model of a system architecture — one that has no executable behavior at all, but simply bounds the allowable behaviors by constraints. In the latter case, a constraint-based architecture is one approach to defining a

[18] The present constraint checker detects violations of only a subset of the constraint language. It is experimental, and a new constraint checker is planned.

standard for systems to conform to.

We do not have space to go into the details of the RAPIDE constraint language, [19] so we give just a couple of examples of constraints on the dining philosophers.

An example of a constraint in the interface of the Philosopher type, specifying part of the behavior of philosopher objects is:

```
-- Philosopher_Protocol:
   match [* rel --->]
    (?id : Integer)
        (RequestStick(?id, ?id) ~ RequestStick(?id, (?id+1) mod numSeats)) --->
        (StickReceived(?id, ?id) ~ StickReceived(?id, (?id+1) mod numSeats)) --->
        (PutDownStick(?id, ?id) ~ PutDownStick(?id, (?id+1) mod numSeats))   ;
```

The poset occuring at the interface of a Philosopher (i.e., those events corresponding to the actions declared in the interface) must match the constraint. The pattern of the constraint is a pair of RequestStick events in any causal relation to one another (i.e., dependent or independent, denoted by the $\sim$ relation), which must both cause two StickReceived events (also this pair may be in any causal relationship to one another), which must cause a pair of PutDownStick events. The iterator preface of the constraint ([* **rel** $\longrightarrow$]) specifies arbitrarily many (*) matches of the pattern, and each match must be causally related to the next one (i.e., in the relation, $\longrightarrow$). So the interface behavior must be a causal chain of matches of the pattern. Many different modules can satisfy this constraint. Similar constraints can be used to specify essential properties of Tables. Our present constraint checker does not check for violations of constraints like this example.

Note, that since the constraint specifies relationships between both **out** and **in** events of philosophers, it specifies a property of the environment that Philosophers interact with. Namely, whenever a Philosopher generates RequestStick events, the environment shall causally respond with StickReceived events — and only then. Certainly, the environment cannot simply send a Philosopher unsolicited chopsticks, independently of a request for them. This point, that the interfaces of types of objects need to specify something about the environment in which the objects are intended to operate, is an on-going research area in Architecture Definition Languages at present. Causal relationships between the environment and the objects are a powerful constraint mechanism.

Now, what about deadlock? This is a constraint on the architecture, but because of the causal relations in our model, it can be expressed as a constraint on the interface events of Tables:

```
never [i:positive 0..4 rel || ] Ta.ReleaseStick(i, i);

never [i:positive 0..4 rel || ]
        Ta.ReleaseStick(i, (i+1) mod 5);
```

These two constraints are placed in the architecture, Diners1. They are violated if a Table ever generates five causally independent ReleaseStick events, one to each Philospher. Because of hypothesis (**H**) about our Diners1 model, one of these two constraints will be violated if and only if deadlock happens. [20] Our present constraint checker detects violations of this kind of "never" constraint — see Figure 6.

[19] The constraint language is being redesigned by Walter Mann, John Kenny, Sigurd Meldal, Woosang Park and David Luckham.

[20] We leave finding an elegant proof of this to the reader.

10 Research and development Issues

In this final section we mention a few of the research and development issues surrounding RAPIDE and its present toolset. Many open topics, particularly to do with the design of Architecture Definition Languages, and with animation tools, are omitted altogether.

10.1 Tracking Causality

The causal history of an event is a data component of the event itself as described in Section 3. This causal history is computed when an event is generated by the well-known method of vectors of counters associated with threads of control — a method attributed independently to Fidge [Fid91] and Mattern (F-M) [Mat88]. One problem affecting the ability of the RAPIDE simulator to handle large models is the space consumed by the F-M vectors — the worst case upperbound is n^2 where n is the number of threads of control. [21] Making the F-M vector method more efficient, e.g., by taking advantage of the communication structure defined in the model's architecture to reduce the size of the vectors, and garbage collect them at appropriate points in a computation. This is a very important issue for us. [22]

There may be methods of encoding and tracking causal history that are entirely different from F-M vectors. If there are, they need to be investigated and compared for efficiency with F-M.

10.2 Patterns and Constraints

Design of pattern and constraint languages for specifying patterns of posets with timing is very much in its infancy. The RAPIDE constraint language is currently undergoing a major revision. Although there are close similarities with languages such as regular expressions and various temporal logics, the specification of causal dependence and independence introduces a new dimension.

Perhaps an even more pressing issue is the development of efficient algorithms for matching patterns on posets. [23] Efficiency of pattern matching affects both the execution of reactive rules, and the checking of posets for constraint violations.

10.3 Poset Viewers

Design of poset viewers and their implementation is entirely new ground. I mention two areas: User Interfaces and Graph layout algorithms.

For user interface design there is very little relevent experience with previous tools to draw upon. For example, Netscape Viewers had the previous Macintosh UI as well as various office software UIs to draw ideas and idioms from. But there is not much prior art on which to draw for poset viewer interfaces. We must pretty much experiment on our own with such questions as: [24]

- what common processes of poset viewing and manipulation will users find most natural and useful — those are the ones the UI should make easy to carry out,

[21] A performance anlaysis of F-M vectors is being undertaken by Park and Vera, using the RAPIDE model of the Sparc V9 instruction set.

[22] James Vera is currently working on this issue, and some previous results are given in [MSV91].

[23] John Kenney is working on a new RAPIDE pattern matcher.

[24] The new POV is being developed by Francois Guimbretierre and Marc Abramowitz.

- what tools for poset analysis should be provided to support the processes,
- how to organize access to the tools into menus in the Viewer UI.

Our basic premiss that posets are best represented by some kind of DAG format appears quite natural. The visualization tool (called Raptor) which is not discussed here, provides the ability to construct other visualizations of posets. But, in order to persist with DAG formats as our primary visual presentation, we need layout algorithms that are both *(i)* fast, and *(ii)* stable — i.e., the layout does not change radically under small operations on the poset, such as deleting one or two events. Both of two these issues with DAG layout algorithms appear to be interesting research areas at present.

References

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

[LKA+95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[LV95] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[LVM95] David C. Luckham, James Vera, and Sigurd Meldal. Three concepts of system architecture. *submitted to the Communications of the ACM*, July 1995.

[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, New York, NY, August 1991. ACM Press. Also Stanford University Computer Systems Laboratory Technical Report No. CSL–TR–91–466.

[Pra86] V.R. Pratt. Modeling concurrency with partial orders. *Int. J. of Parallel Programming*, 15(1):33–71, February 1986.

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

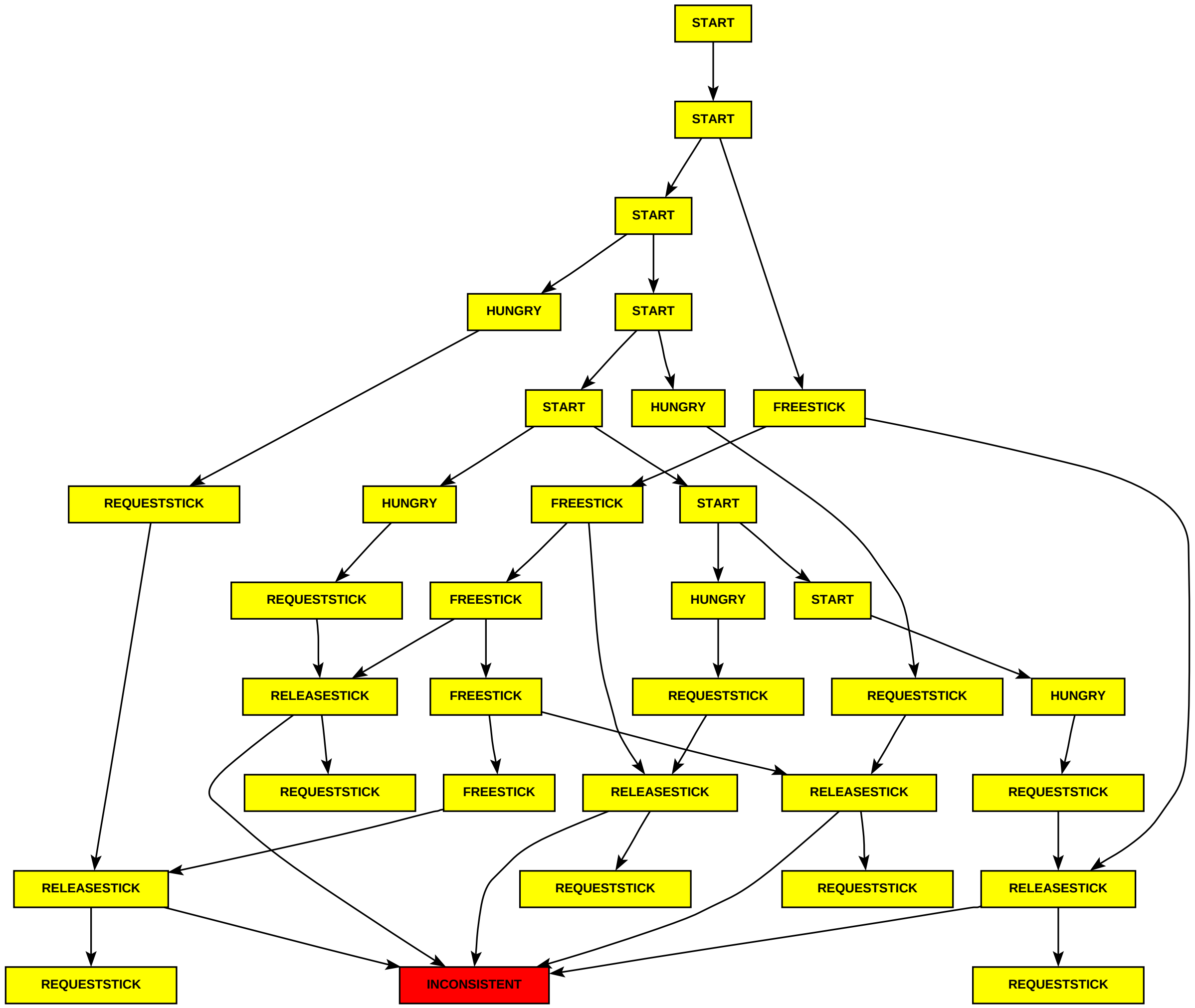

Figure 6: Deadlock with constraint violation history