COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY STANFORD, CA 94305-4055

RAPIDE-0.2 Examples

Alexander Hsieh

Technical Report: **CSL-TR-92-510**
(Program Analysis and Verification Group Report No. 57)

February 1992

This work was supported by the US Defense Advanced Research Projects Agency, under contract N00014-90-J-1232-P00003.

RAPIDE-0.2 Examples

Alexander Hsieh

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

February 1992

Technical Report: **CSL-TR-92-510**
(Program Analysis and Verification Group Report No. 57)

Abstract

RAPIDE-0.2 is an executable language for prototyping distributed, time sensitive systems. We present in this report a series of simple, working example programs in the language.

In each example we present one or more new concepts or constructs of the RAPIDE-0.2 language with later examples drawing on previously presented material.

The examples are written for both those who wish to use the RAPIDE-0.2 language to do serious prototyping and for those who just wish to be familiar with it. The examples were not written for someone who wishes to learn prototyping in general.

Keywords—RAPIDE-0.2, *prototyping, tutorial.*

Contents

Chapter 1

Introduction

RAPIDE-0.2 [BL90] is an executable language for prototyping distributed, time sensitive systems. We present a series of simple example programs in RAPIDE-0.2 to be read as a tutorial. These examples will hopefully instruct the reader on the RAPIDE-0.2 language as well as give some intuition on how use the language to build his own prototypes.

Each chapter presents a simple RAPIDE-0.2 example. In each example one or more new language concepts or constructs are used and discussed. Later examples draw on material given in previous examples. The examples are not intended to teach the user about prototyping but instead to cover the basic ideas and principles behind the RAPIDE-0.2 language.

In each example there is an introduction to briefly describe what we are trying to prototype as well as new RAPIDE-0.2 concepts or constructs that are used. Following this is the main discussion about the new concepts or constructs and then a more technical section on implementation. Finally a full program listing is given.

We encourage the reader to play with the examples and Appendix B gives details on how to start experimenting with them. All the programs are available if you have access to Anna.stanford.edu. Appendix C gives the location of the programs. Also in the appendices are a keyword index and two special sections introducing the partial order browser (POB) and the Illustrated Run-time System (IRS).

The partial order browser is a tool used to view the a partially ordered set that is produced by the program. This is important for analysis of the prototype. The IRS is a run-time tool used primarily for examining the run-time behavior of the program in detail. The IRS is only described in the appendices.

Here is a summary of the concepts and constructs discussed in each chapter.

Light Switch: *design units, actions, events, when-processes, triggers.*

Satellite Communication Link: *the main design unit, placeholders, prototype development.*

Snooze Alarm: *clocks and time, guards.*

Dish-Washer: *constraints.*

Satellite Communication Link II: *connections.*

Baking a Cake *more on constraints.*

Library: *properties.*

Acknowledgments

We thank David Luckham, Frank Belz, Doug Bryan and the rest of the PAVG crew at Stanford for their comments, suggestions, discussions and general helpfulness.

Chapter 2

Light Switch

2.1 Introduction

Welcome to the first RAPIDE-0.2 example! Here we will model a simple switch which can have two states – on or off. Being the first example it will inevitably be necessary to introduce several concepts of the language. Our example introduces *design units, actions, events, when-processes* and *triggers.*

2.2 Discussion

A prototype consists of one or more components (subcomponents). In RAPIDE-0.2 each component is modeled by a *design unit.* You might imagine a design unit as a physical object whose surface is the part we interact with whilst the inside appears to be a black box. These ideas are encapsulated in the *specification* and the *body* of the design unit respectively. A design unit is also a *type* so we can create *instances* of them by declarations. For example if we had a design unit type Human then the following declarations: *design units*

```
George  : Human;
Vivien  : Human;
```

would give us two instances of the design unit Human which are identified by the names George and Vivien. In RAPIDE-0.2 design units are our building blocks. We instantiate as many as necessary to construct our prototype.

In our example we want to model a switch and what better to do than to use a design unit for it. We would like to be able to turn the switch on and off and have the switch recognize that it is being turned on and off.

Here is the specification of the Switch design unit:

```
design Switch is

   in   action Turn-On;
   in   act ion Turn-Off;
   out action Switch-Is-On;
   out  action  Switch-Is-Off;

end Switch;
```

actions

events

What does this say? The specification contains four declarations of *actions*. **Two** are *in-actions* actions and the other two are *out-actions.* These actions indicate what sort of activities we want the Switch to be capable of doing. It can generate an *event* of the Switch-Is-On action or of the Switch-Is-Off action, or it can observe either a Turn_On or Turn-Off event sent from another design unit instance. An event is simply an instance of an action and we shall use the name of the action to refer to one of its events. We can tell by the context whether we are referring to an action or to its event.

When a RAPIDE-0.2 program is run the events generated constitute a partially ordered set, the ordering representing a dependency between events. A RAPIDE-0.2 tool, the *partial order browser,* exists for the viewing of the partially ordered events. The *partial order* is used for analyzing the behavior of the prototype (see section 5.2 and Appendix D for more information about the partial order and the partial order browser). This advanced topic is beyond the scope of this document and will not be discussed in detail.

How exactly do you generate these events then? To see this let's look at the body of the Switch design unit.

```
design body Switch is

begin

   << On >>
   when Turn_On then
     Switch-Is-On;
   end when;

   << Off >>
   when Turn-Off then
     Switch-Is-Off;
   end when;

end Switch;
```

when-processes

The body will usually contain a set of declarations before the **begin** (there happens to be none in this example). Sandwiched between the **begin** and end are *when-processes.* In our case there are two of them. < < On > > and << Off > > are *labels* for the processes.

When a design unit is instantiated the when-processes start to concurrently monitor events generated by other design unit instances. Turn_On is the *trigger* for the process *triggers* << On >> . The << On > > process waits for a Turn_On event and, if and when it observes it (we say the when-process is *triggered* at this point), the sequence of statements in the when-process is executed sequentially. In this case the event Switch-Is-On will be generated. Similarly if and when the << Off >> when-process observes a Turn-off event the event Switch-Is-Off is generated.

Perhaps now we can understand how our design unit models a switch. When it detects that it is being turned on, via a Turn_On event sent from some other design unit instance, it responds by issuing a Switch-Is-On event. This could be used to tell the other design unit instance that the switch is now on. Similarly for turning the switch off.

2.3 Implementation

We have described how the Switch design unit works. We have some code for it but what on earth do we do with it? How shall we test and experiment with it's behavior? This becomes a more general question when we have larger systems with many components interacting together. RAPIDE-0.2 does give us the ability to build an architecture. However since this example has a rather trivial architecture we shall leave the discussion of architecture for chapter 3 and instead discuss how to test our switch.

As we shall describe n chapter 3 we have a main design unit which handles the architecture of the prototype. In this example it is named Switch-Handler:

```
with Switch;

design Switch-Handler is end;

design body Switch-Handler is

   Light-Switch : Switch;

begin

   when start then
      Light_Switch::Turn_On;
      Light-Switch::Tum-Off;
   end when;

end Switch-Handler;
```

The statement with Switch indicates that the Switch-Handler design unit has a dependency on the Switch design unit. The declaration in the body instantiates a design unit named Light-Switch of the design unit type Switch.

There are two actions that are implicitly defined for all design unit instances. The events corresponding to these actions are technically generated by the Run Time System (RTS), but can be thought of as being generated by the design unit instance at the appropriate time. The events can be used in the trigger of a when-process.

One of these implicit actions is the Start action. A single Start event is generated by each design unit instance when the prototype is run. For each individual design unit instance it is the first event that can be observed. It is useful for initializing design unit instances.

The single when-process in Switch-Handler is triggered by the observation of the Start event that the design unit instance generates at the beginning of execution. When it is triggered, two events are generated sequentially – Light_Switch::Turn_On and Light_Switch::Turn_Off. The syntax

Light-Switch: :

makes the events specific to the design unit instance named Light-Switch.

Let us now summarize what will happen when we execute the compiled program. The compiler will have to be informed that Switch-Handler is the main design unit (see section 3.2.2) and it will automatically instantiate it once. Switch-Handler itself instantiates a Switch named Light-Switch so we now have two design unit instances. Both of these will observe a Start event but only Switch-Handler will react to its own Start event, generating the two events Light_Switch::Turn_On and Light-Switch::Tum-Off. These are observed by Light-Switch which reacts to them by generating the events Switch-Is-On and Switch-Is-Off respectively. Since there are no when-processes to observe these there is nothing further to do and the program ends.

You can refer to Appendix B for information on how to compile and run the program. Also in the appendix is the partial order graph for the execution of the program along with an explanation.

2.4 Program Listing

2.4.1 Design Unit Switch

```
design Switch is

   in   action Turn-On;
   in   act ion Turn-Off;
   out  action Switch-Is-On;
   out  action Switch-Is-Off;

end Switch;

design body Switch is

begin

   << O n  >>
   when Turn_On then
      Switch-Is-On ;
   end when;

   << Off >>
   when Turn-Off then
      Switch-Is-Off;
   end when;

end Switch;
```

2.4.2 Design Unit Switch-Handler

```
with Switch;

design Switch-Handler is end;

design body Switch-Handler is

   Light-Switch : Switch;
```

```
begin

   when start then
      Light_Switch::Turn_On;
      Light_Switch::Turn_Off;
   end when;

end Switch-Handler;
```

Chapter 3

Satellite Communication Link

3.1 Introduction

This example describes a simple system of three cities which have communication links via a satellite. Cities communicate with each other by transmitting messages to the satellite which in turn transmits the message to the destination city.

We start with an overview of prototype development in RAPIDE-0.2. We then go on to describe design unit instance architectures and the communications between the design unit instances in the architecture. In the course of this we introduce the *main* design unit instance and we also meet *placeholders.*

3.2 Discussion

3.2.1 An overview of Prototyping in RAPIDE-0.2

What is a prototype? A prototype is a construct that attempts to model the behavior of a system. In RAPIDE-0.2 our construction happens to be a program which is run on a computer. The following discussion focuses on the paradigm for prototyping in RAPIDE-0.2. Section 3.2.2 then discusses the exact details of how RAPIDE-0.2 is used to prototype systems.

In building a prototype in RAPIDE-0.2 we start simple. This means abstracting out as much detail as possible whilst keeping the essential features of the system being prototyped. This allows for early design analysis and faster recovery from mistakes or decision changes. In RAPIDE-0.2 components of the system are modeled as separate entities. As the prototype evolves components are broken into subcomponents revealing more detail and leading to a more precise model of the system.

Figure 3.1 shows a prototype that has evolved. The figure shows a prototype with two components, A and D, with A being broken up into two subcomponents B and C. The system being prototyped is represented by the largest rectangle labeled Prototype. The arrows between A and D indicate that they communicate with each other as we expect components

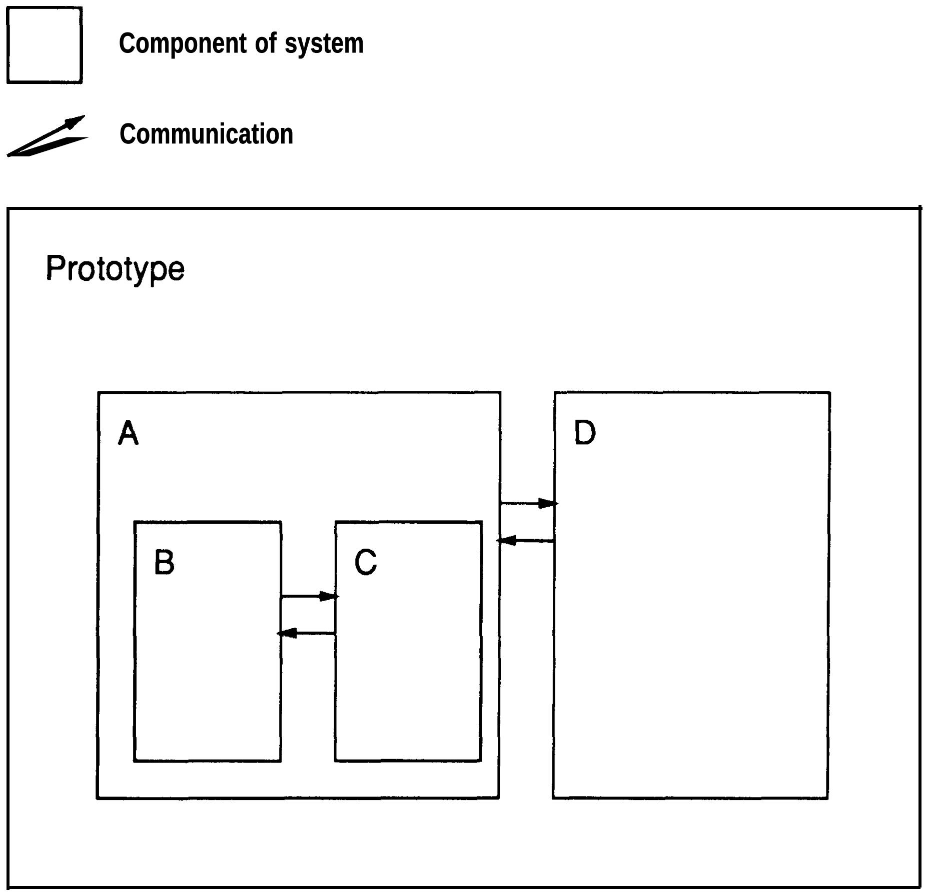

Figure 3.1: **A prototype with two components A and D, each showing their subcomponents**

to do. We say that the prdtotype has evolved because A has two subcomponents, B and C. B and C communicate as indicated by the arrows between them. In an earlier version of the prototype the design unit instance A by itself would have described the properties of the component it was modeling. B and C are introduced during evolution to increase the detail of description of the component.

Figure 3.1 raises the question of how D can communicate with C if it wanted to (we don't see any communication lines between D and C in the figure). The way this would happen is that D communicates with A which then passes the message to C. This makes sense since D is really communicating with A, not C. D doesn't know about the subcomponents of A, it only knows about the interface that A presents to it. A is a wrapper for the details within itself (this is very similar to information-hiding in some languages) and so any other component wishing to communicate with it, or some subcomponent of it, should communicate with it at the highest level possible.

In the same way that A is a wrapper for B and C, Prototype is a wrapper for A and D. It hides the details inside it from the outside world. If we wanted to use Prototype in an even larger system we could just, "insert" it as one piece into the larger system.

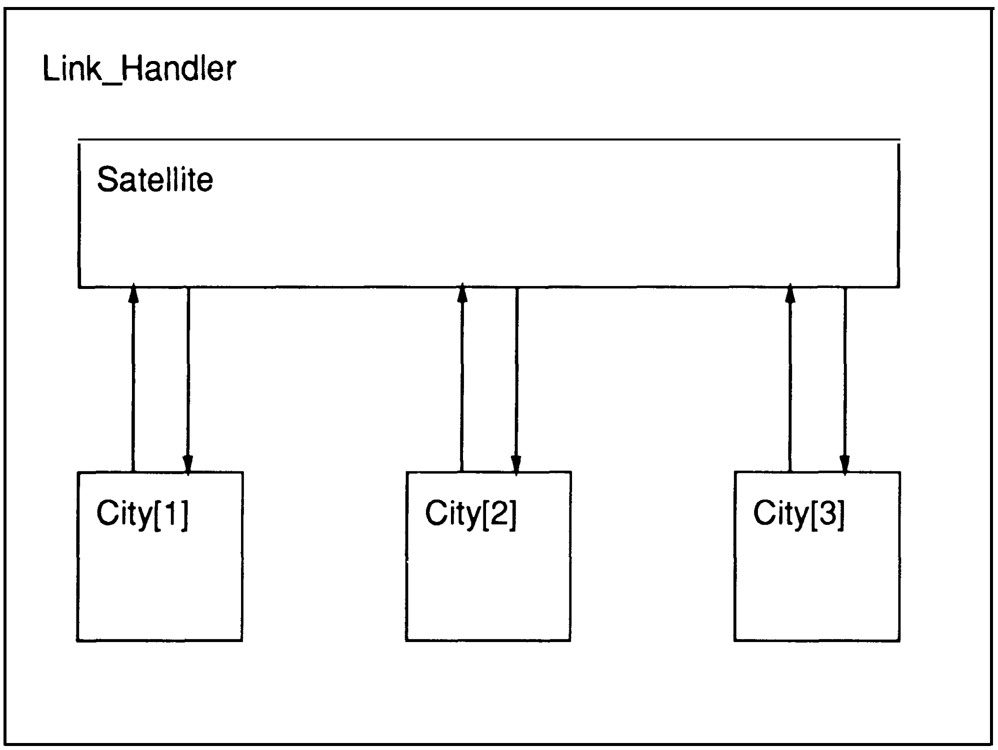

Figure 3.2: **Architecture for Satellite Communication Link**

For a more concrete example of a prototype let's look at the current Satellite Communication Link example. Figure 3.2 shows the prototype wrapped by something called Link-Handler containing four components - the Satellite and three Citys. The communication architecture shows that cities can only communicate with the satellite which itself can communicate with all three cities. This is what we need to build our intended system where cities communicate with each other via a satellite.

Figure 3.3 shows the Satellite Communication Link after some evolution. The satellite is broken into three components: a Receiver, a CPU and a Transmitter. The intended behavior of the three subcomponents is as follows. A City sends a message to the Satellite at the **uplink** frequency. The **receiver** (some kind of antenna) receives the message and passes it to the CPU. The CPU processes the message, possibly with some error-correction and flow-control, and passes it to the transmitter which transmits the message at the **downlink** frequency.

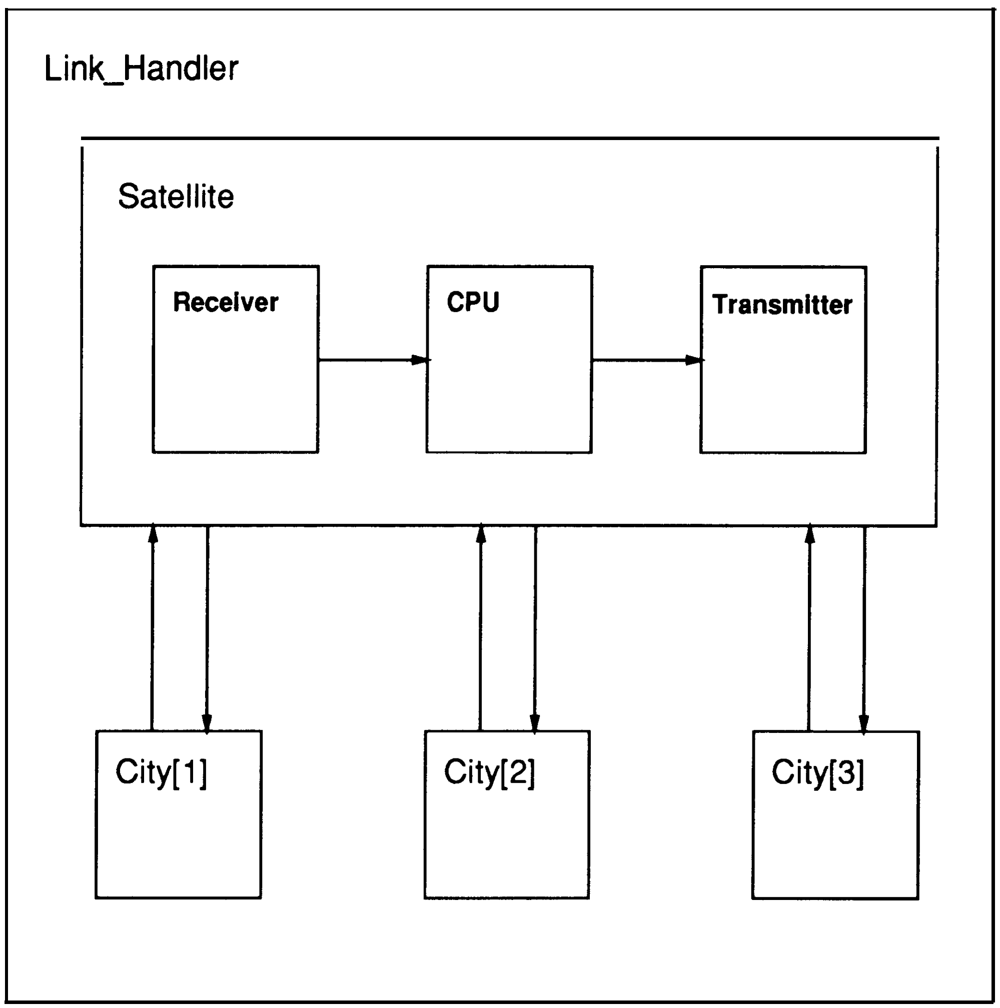

Figure **3.3: Possible evolution of the Satellite Communication Link prototype in which the satellite is broken into three components.**

3.2.2 **Architecture and Communication in** RAPIDE-0.2

In this section we describe how an architecture of design unit instances is built in RAPIDE-0.2 and the communications between them.

Figure 3.1 has a direct translation into a RAPIDE-0.2 architecture. Each component of the prototype is a design unit instance. Each subcomponent of a component is a design unit instance. In fact every rectangle is a design unit instance. A subcomponent design unit instance of a component design unit instance is created when the component design unit instance *instantiates* it. Giving some more terminology, a design unit instance is instantiated by its *parent* and is therefore the instantiators *child.* Two design unit instances sharing the same parent are *siblings.*

In RAPIDE-0.2 there is a special design unit instance called the ***main*** design unit instance. Since all design unit instances must be instantiated by another one we would be in a chicken and egg situation if there were not some special design unit instance. This is the main design unit instance. In figure 3.1 this is just the Prototype wrapper that encloses A and D.

The translation of figure 3.1 is now clear. Every rectangle is a design unit instance. The Prototype rectangle is the main design unit instance. If a design unit instance is inside another design unit instance it is instantiated by the closest design unit instance wrapping it. For example, B is instantiated by A and D is instantiated by Prototype.

We can now see that the main design unit instance has a special role because it is the base of the prototype architecture and will also provide the communication between the main components of the system being prototyped. We shall call the main design unit instance the "handler" for the prototype as it handles the highest level of the architecture.

Recall that design unit instances communicate by passing events around. Design unit instances have rules governing which events are visible to them and to whom events are sent when the design unit instances generate them. These are the scoping rules. Let's suppose that D_1 is the parent of D_2 and D_2 performs an *out-action*. Then the event generated can only be observed by D_1 and no other design unit instance. Note that only D_2 can generate events corresponding to out-actions declared in it's specification. Events corresponding to *in-actions* of D_2 can only be generated by D_1 and can only be observed by D_2.

internal actions

There is one more kind of action, the *internal-action.* Intuitively they are used to pass messages from one when-process to another within the same design unit instance. They are part of the black box of the design unit, invisible to the outside world. The design unit instance that declares the internal-action is the only design unit instance that can perform the action and observe the event generated. We will meet internal-actions in a later example.

We can now see that events flow between design unit instances along branches of a tree, and this tree is precisely the "instantiation tree". An example of such a tree is shown in figure 3.4. In the figure boxes represent design unit instances, undirected edges represent a parent-child relationship with the parent above the child, and directed edges indicate a flow of events between design unit instances. For example, design unit instance A is a child of the main design unit instance. A can observe events generated by the main design unit instance which match any of A's own in-actions and any events generated by the out-actions of the design unit instances B and C, it's children. Events generated by out-actions of A can only be observed by the main design unit instance.

In the section 3.2.1 we said that communications between components were between

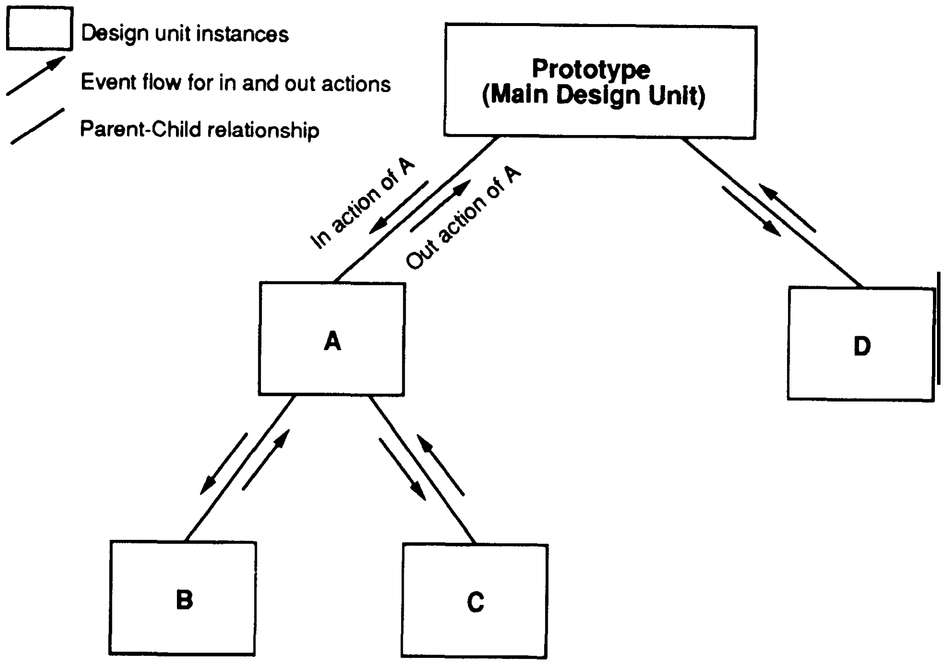

Figure 3.4: Example of an "instantiation tree" and the flow of events within it. Parents are drawn above the child.

what we now call siblings and that if a component has subcomponents then it would be able to direct communication to them. This has a translation into RAPIDE-0.2 but is not as precise because in RAPIDE-0.2 there is no direct communication between siblings. Instead we implement communication between siblings by using the common parent as an intermediary. How this is precisely done is shown later.

3.3 Implementation

There are three design units. One for the satellite, one for cities and one for the handler.

Let's look at the specifications of the two design units City and Satellite and see what they do.

```
design City is

   in   action Send_Trigger(to : integer; message : string);
   in   action Receive_Message(from : integer; message : string);
   out  action Send-Message(to : integer; message : string);

end City;

design Satellite is

   in   action Receive-From-City(from, to : integer; message : string);
   out  action Relay-Message-To-City(from, to : integer; message : string);

end Satellite;
```

The Satellite design unit should receive a message from a city and then relay it on to the destination city. Thus it has an in-action Receive-From-City and an out-action Relay-Message-To-City. Actions can have parameters, just like these two do. In each case we need to know at least the destination and the content of the message. Here, just for completeness, the sender of the message is included. The parameters from and to encode as integers the sender and recipient of the message.

Similarly in the design unit City we have two actions proclaiming the sending and receiving of messages. The third action Send-Trigger is an artifice used by the handler to induce the city to send a message (described further in section 3.3.2).

Looking at the respective bodies of the two design units we see how the journey of the messages are realized.

```
design body City is

   ?message : string;
   ?to      : integer;

begin

   << Send-Messages >>
   when Send_Trigger(?to, ?message) then
      Send-Message(?to, ?message);
   end when;

end city;

design body Satellite is

   ?from, ?to: integer;
   ?message : string;

begin

   << Relay-Messages >>
   when Receive_From_City(?from, ?to, ?message) then
      Relay-Message-To-City(?from, ?to, ?message);
   end when;

end Satellite;
```

The City design unit has a single when-process that says "when I am told to send a message (by the handler) I will send it". The satellite has a when-process that says "when I receive a message I should relay it onto the correct destination".

placeholders

The curious looking objects ?from, ?to and ?message are *placeholders*. A placeholder lexically has a "?" as the first character. Placeholders are best explained by example. In the Satellite design unit the when-process labeled << Relay-Messages >> waits until it observes a Receive-From-City event at which point the placeholders *bind* to the three parameters in the event observed. From the specification of the Satellite design unit we see the parameters are two integers followed by a string. So if the event was Receive_From_City(3,2,"hello") then we would get the following bindings:

?from	$\leftarrow$	**3**
?to	$\leftarrow$	**2**
?message	$\leftarrow$	"hello"

These bindings stay in force until execution reaches end when.

So much for the building blocks of our prototype. We now want to glue them together using the handler to produce the representation in figure 3.2. There are two issues to be discussed about the handler. In section 3.3.1 we discuss how we use the handler as a router to bind the architecture together and in section 3.3.2 we discuss an I/O interface with the user.

3.3.1 The handler as a router

One key responsibility of the handler is to direct the flow of information from one design unit instance to another. To do this the handler instantiates the necessary design units itself:

```
Cities : array[ 1..3] of City;
sat    : Satellite;
```

These two statements instantiate one satellite called Sat and an array of three cities named Cities. In RAPIDE-0.2 the elements of the array are denoted Cities[1], Cities[2] and Cities[3].

The architecture is built from two simple when-processes.

```
< < connect_city_to_satellite > >
when Cities[?city] : :Send_Message(?to, ?message) then
  Sat::Receive_From_City(?city, ?to, ?message);
end when;

< < connect-satellite-to-city > >
when Sat::Relay_Message_To_City(?from, ?to, ?message) then
  Cities[?to] ::Receive_Message(?from, ?message);
end when;
```

These two processes represent the directed edges in figure 3.2. The first says "when I observe a Send-Message event from a city I will hand that information over to the satellite by generating the Receive-From-City event for the satellite". Notice that for these events we tag the name of the event with the design unit instance name. This makes sense since we need to specify which design unit instance we are interested in in each case. In the case of Send-Message the placeholder ?city is used to capture a Send-Message event issued by *any* city.

The second when-process says "when I observe a Relay-Message-To-City from the satellite I will generate the Receive-Message event for the appropriate city".

Our method of building the architecture is very general. Whenever two design unit instances need to communicate with each other via an out-action and an in-action we can write a corresponding when-process to handle it. This is also the general method used when a component is broken into subcomponents. For example, in figure 3.3, the Satellite design unit instance needs to route events from the Receiver to the CPU to the Transmitter. The Satellite design unit instance will contain when-processes to do this whilst the main design unit instance named link-handler will contain when-processes to allow communication between cities and the satellite.

3.3.2 The handler as an user-interface

The handler can also be used as a front-end to the user. We can give the prototype instructions and it can respond by printing messages onto the console to show it's status. However it must be emphasized that the primary tool for studying the behavior of the prototype is through the *partial order,* the partially ordered set of events generated during execution of the program. The interaction with the user should be viewed as a method for controlling part of the behavior of the prototype.

In this example we need messages for the cities to send to each other. We do this by prompting the user for a sender, a recipient and the content for each message. Once we have this we can generate a Send_Trigger event for the appropriate city to get it to send the message. We can write all of this in one when-process as follows.

```
<< Input-Output >>
when start then
   put_line(" Welcome!   Please input originator, destination,");
   put_line( "and content of your messages.");
   new-line;
   loop
      put("Send message from which city (0 to quit) -> ");
      get_line( send-from) ;
      if (send-from /= 0) then
         put( "Send message to which city -> ");
         get_line( send-to) ;
         put( "Enter the message - > ");
         get_line(msg);
         Cities[ send-from] : :Send_Trigger( send-to, msg) ;
      else
         exit;
      end if;
   end loop;
end when;
```

Upon execution this when-process will observe the Start event and execute the statements in sequential order. On each iteration through the loop the user is prompted for information and then the Send_Trigger event is generated. When the user enters a "0" for the sender of the message the loop is exited and the when-process terminates. Any message still in the system will continue it's journey, generating events until the destination city observes the Receive-Message event for that message. When all messages have been received the program terminates as no more events can be generated.

3.4 Program Listing

3.4.1 Design Unit City

```
design City is

   in  action Send_Trigger(to : integer; message : string);
   in  action Receive_Message(from : integer; message : string);
   out act ion Send-Message( to : integer; message : string);

end City;

design body City is

   ?message : string;
   ?to      : integer;

begin

   << Send-Messages >>
   when Send_Trigger(?to, ?message) then
     Send-Message(?to, ?message) ;
   end when;

end city;
```

3.4.2 Design Unit Satellite

```
design Satellite is

   in  action Receive-From-City(from, to : integer; message : string);
   out action Relay_Message_To_City(from, to : integer; message : string);

end Satellite;
```

```
design body Satellite is

   ?from, ?to : integer;
   ?message   : string;

begin

   < < Relay-Messages > >
   when Receive_From_City(?from, ?to, ?message) then
      Relay-Message-To-City(?fmm, ?to, ?message);
   end when;

end Satellite;
```

3.4.3 Design Unit link-handler

```
with City, Satellite;

design link-handler is
end link-handler;

design body link-handler is

   Cities : array[ 1..3] of City;
   sat    : Satellite ;

   ?from, ?to            : integer;
   ?message              : string;
   ?city                 : integer;
   send_from, send-to    : integer;
   msg                   : string;

begin

   < < connect_city_to_satellite > >
   when Cities[?city]::Send_Message(?to, ?message) then
      Sat::Receive_From_City(?city, ?to, ?message);
   end when;

   < < connect_satellite_to_city > >
   when Sat::Relay_Message_To_City(?from, ?to, ?message) then
      Cities[?to]::Receive_Message(?from, ?message);
   end when;
```

```
    << Input-&t& >>
  when start then
    put_line( "Welcome!  Please input originator, destination,");
    put_line( "and content of your messages.");
    new-line;
    loop
      put("Send message from which city (0 to quit) -> ");
      get_line( send-from) ;
      if (send-from /= 0) then
        put( "Send message to which city - > ");
        get_line( send-to) ;
        put( "Enter the message -> ");
        get_line(msg);
        Cities[send_from]::Send_Trigger(send_to, msg);
      else
        exit;
      end if;
    end loop;
  end when;

end link-handler;
```

-

Chapter 4

Snooze Alarm

4.1 Introduction

A Snooze *Alarm* is a device used by students to give them some extra sleep each morning. It consists of a regular alarm clock with a *sleep* feature. When the alarm rings the student can hit the *sleep* button which turns off the ringing for a prescribed time (usually of the order of ten minutes for real alarm clocks). After this time has elapsed, if the alarm has not been switched off completely, the alarm will ring again. This process can be repeated many times until the student decides that it is finally time to get up for lunch.

This example introduces the idea of a clocked design unit and, in general, the idea of *clocks* in RAPIDE-0.2. We also meet guards.

4.2 Discussion

In RAPIDE-0.2 there is a predefined type named *time*. In RAPIDE-0.2 you can consider the value of the type time to be the natural numbers. A *clock* is an object which returns values of type time. If a design unit is *clocked* it carries with it a clock which can be read by the design unit instance. The clock is monotonically increasing and, without going into the details, whenever a design unit instance has "nothing to do" it increments its clock. Moreover, the clock is incremented as much as possible until there is something to do. Each time the clock is advanced as far as possible a Tick event is generated. Tick, like Start, is a predefined action in RAPIDE. *clocks*

The two design units in this example are named Alarm-Clock and Alarm-Handler. Both of these are clocked:

> **design** Alarm-Clock **is clocked**
>
> **design** Alarm-Handler **is global clocked**

If a design unit is clocked it has it's own clock, independent of any other clocks that may be around. However, if a design- unit is global *clocked* it, along with all its descendants (in the "instantiation tree"), share the *same* clock whether or not the descendants are explicitly clocked or not. In the example, Alarm-Handler instantiates the design unit Alarm-Clock so they will share the same clock. If a design unit uses any of the timing constructs it must explicitly be declared to be clocked even if one of its ancestors is declared to be global clocked.

delays

Once we have a clocked design unit we can simulate delays in processes. For example in the design unit Alarm-Clock the when-process labeled << Ringing >> contains the line:

```
ring_clock pause 1;
```

The pause causes the when-process to suspend itself for one clock unit before generating **a** ring-clock event and then proceeding.

One use of pause is to simulate the interval of time it takes to complete some kind of action. For example, if boiling an egg takes five minutes we could write:

```
boil-egg pause 5;
```

Another use of pause is to create your own user-clock for a design unit instance.

```
act ion clock-tick;

when start then
   clock-tick;
end when;

when clock-tick then
   clock-tick pause 1;
end when;
```

The idea of internal-actions was introduced earlier in section 3.2.2. Here we see our first example clock-tick. This user-clock code segment generates a clock-tick event once every clock interval, and has the side-effect of forcing the clock to tick once every clock interval too. If this were not present it might have been possible for the clock to never tick at all.

We conclude this section by describing departure and arrival times.

departure time
arrival time

Each event has associated with it a *departure* time and an arrival time. The departure time is that time which was on the clock of the design unit instance that generated the event when the event was generated. The arrival time is that time which was on the clock of the design unit instance receiving the event at the time it arrived at the design unit instance. Notice that since design unit instances can have different clocks the arrival time of an event can be less than, equal to, or even greater than it's departure time. If the event is generated and observed by two design unit instances which share the same clock then the arrival time and departure time will always be equal. The departure time and arrival time of an event can be found in the second to last and last parameters of the event respectively. The declaration of an action:

```
in action C(x : integer; y : boolean);
```

hides the implicit departure and arrival time parameters of the action. When writing the action in a trigger, placeholders can be used to bind to the departure and arrival times (these placeholders are optional if you do not wish to obtain these two times):

```
when C(?int, ?bool, ?departure, ?arrival) then . . .
```

The values ?departure and ?arrival can then be used in the when-process.

4.3 Implementation

4.3.1 Alarm-Clock

The alarm clock we have is based on the automaton shown in figure 4.1. In our implementation of the automaton, states will correspond to variables and transitions will correspond to events. A variable named state will be used to remember which of the three states *Idle, Set to ring,* and *Ringing* the automaton is in. The events Set-Alarm, Tick, Stop-Ringing and Sleep move the machine from state to state as shown in the diagram. These events are generated by actions in our Alarm-Clock design unit:

```
design Alarm-Clock is clocked

   in  action Set-Alarm (set-time : TIME) ;
   in  action Sleep;
   in  action Stop-Ringing;
   out action Ring;

end Alarm-Clock;
```

Notice that the parameter set-time is of type time which was introduced earlier. Set-Alarm tells the alarm clock at what time to start ringing. Sleep tells the alarm clock that we want no more ringing for a little while (our sleep feature). Stop-Ringing tells the alarm clock to stop ringing if it is ringing.

The automaton is very simple. We allow the user to set the alarm in states IDLE and SET-TO-RING only, and only use the Sleep feature or kill the alarm with the Stop-Ringing in state RINGING. Any other combination of user interactions and state have no effect on the automaton (Tick is not user controlled). The absence of these other combinations correspond to, for example, the user hitting the Sleep button in the idle state.

We use the time of the design unit instance clock to be the time of the alarm clock.

The astute reader will have noticed that there is a potential bug in the program as described so far. Since the design unit instance clock is being used as the clock for the alarm and we are triggering off the internal action Tick we can get in trouble if the design

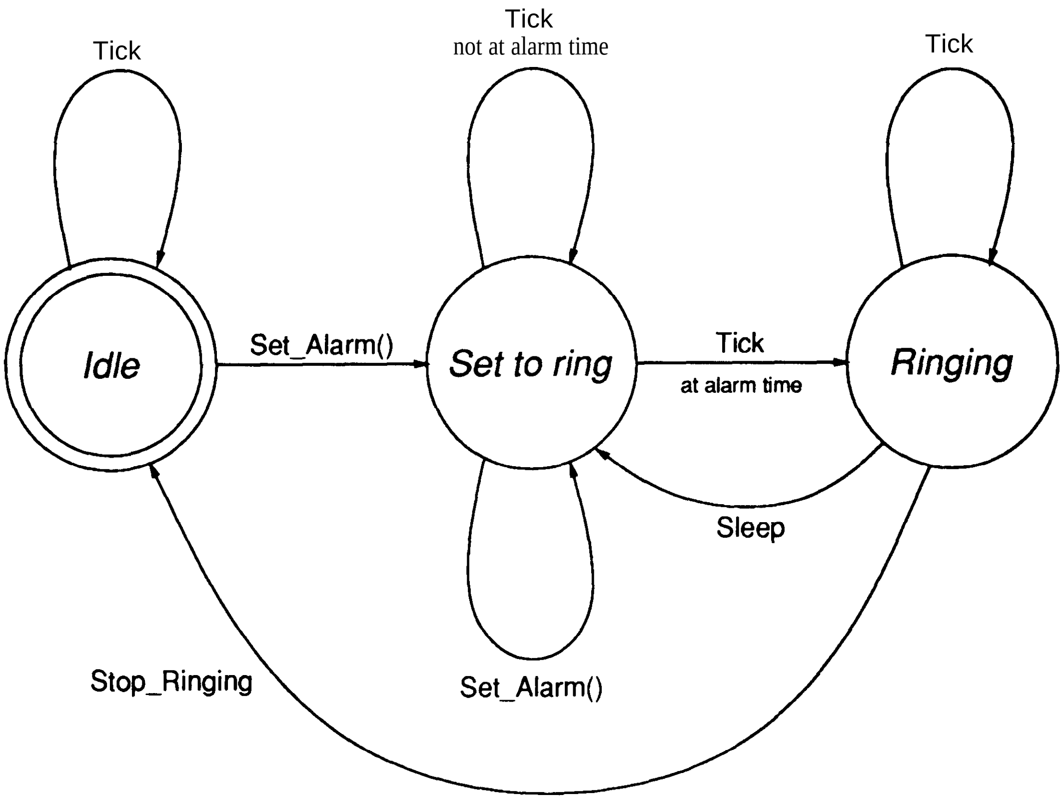

Figure 4.1: **State transition diagram for the alarm clock**

unit instance clock does not issue a Tick for every clock value. Recall that this could happen should there be nothing to do for at least two clock intervals. We are saved in this case because the handler happens to have something to do every clock interval. However in general we should construct our own user clock as described earlier.

The automaton shown in figure 4.1 is a little misleading as it doesn't show that the automaton needs a memory to keep track of what time to ring the alarm. We use the variable alarm-set-time to remember the time for the alarm to ring. This variable is one of several declarations in the body of the design unit:

```
action ring-clock;
alarm-set-time     : TIME := 0;
sleep-interval     : TIME := 4;
?time_to_wake_up   : TIME;
IDLE               : constant integer := 1;
SET-TO-RING        : constant integer := 2;
RINGING            : constant integer := 3;
state              : integer : = IDLE;
```

The internal-action ring-clock has to do with ringing the alarm (described later). The variable sleep-interval is the number of clock units that the alarm clock will stay silent when the sleep button is pressed. alarm-set-time is the time we request the alarm to ring. The placeholder ?time_to_wake_up is used with the action Set-Alarm The three states of the automaton are described by the constants IDLE, SET-TO-RING and RINGING. The variable state records in it which state the machine is in. It is initially set to IDLE, the idle state.

The implementation is based on the automaton. The state of the automaton is recorded in the variable state. The transitions are simulated by when-processes that trigger when a valid event has occurred. Let's examine one of the when-processes to get a feel for this:

```
<< Setting_the_Alarm >>
when Set_Alarm(?time_to_wake_up)
   where (state = IDLE or state = SET-TO-RING) then
      state : = SET-TO-RING ;
      alarm-set-time := ?time_to_wake_up;
end when;
```

There are several things going on here. Here we see the first example of a *guard:* *guards*

```
(state = IDLE or state = SET-TO-RING)
```

The body of the when-process will only be executed if the event Set-Alarm is observed *and* the guard (a boolean expression) is satisfied. If the guard is not satisfied the event is not reused for this when-process'.

'This is true if there is only one action in the trigger. If there are two or more, an event can be used more than once to trigger the when-process as long as the same combination of events is only used once.

The above process deals with the two transitions that can set the alarm time. This is seen in the guard where we test if the current state is IDLE or SET-TO-RING. If the guard is true we set the state to SET-TO-RING and the variable alarm-set-time is set to the time we wish the alarm to ring.

The when-processes << Stop_The_Ringing > > and << Sleep-Feature >> work in a similar way. The when-process << Ring-Alarm >> works as follows.

```
<< Ring-Alarm >>
when Tick where
  (state = SET-TO-RING and clock = alarm-set-time) then
    state := RINGING;
    ring-clock;
end when;
```

The when-process waits for a lick event where the current state is SET-TO-RING and the current value of the design unit's clock is the same as the value stored in alarm-set-time. When this situation occurs the automaton moves to state RINGING and the alarm is activated by issuing a ring-clock event.

The final when-process << Ringing >> does the ringing for us:

```
<< Ringing >>
when ring-clock where (state = RINGING) then
  Ring;
  ring_clock pause 1;
end when;
```

When a ring-clock event is observed and the automaton is in state RINGING the event Ring is generated. This is the out-action of the design unit instance and corresponds to the ringing of the clock. The internal-action ring-clock is just a helper here. Another ring_clock event is then set to be generated in one clock unit's time. This when-process iteratively triggers itself. It stops when state is no longer RINGING.

There is one final important point to make. It has to do with critical sections. The when-processes in the body of the design unit instance execute in parallel and so there can be interleaving between statements in two or more when-processes. In our prototype there might have been a problem with the following two when-processes:

```
<< Setting-the-Alarm > >
when Set_Alarm(?time_to_wake_up)
  where (state = IDLE or state = SET-TO-RING) then
    state := SET-TO-RING;
    alarm-set-time := ?time_to_wake_up;
end when;
```

```
<< Ring_Alarm >>
when Tick
   where (state = SET-TO-RING and clock = alarm-set-time) then
      state := RINGING;
      ring-clock;
end when;
```

Suppose the automaton is currently in state SET-TO-RING and that alarm-set-time is equal to the clock value (so that the alarm is about to go off). Suppose that a Set-Alarm event has been generated at this time too (the user wants to reset the time at which the alarm should ring). We also know that there is a Tick event at this time. Then both of the when-processes above will be triggered. The interleaving,

```
<< Ring-Alarm >>              state := RINGING;
<< Setting-the-Alarm >>       state := SET-TO-RING;
<< Setting-the-Alarm >>       alarm-set-time := ?time_to_wake_up;
<< Ring-Alarm >>              ring-clock;
```

will leave the automaton in state SET-TO-RING with an event, ring_clock having been generated. In this prototype this is not a problem since the when-process << Ringing >> has a guard that will prevent the alarm clock from producing Ring events. However, **in** another prototype an interleaving like this might cause an implementation of it to violate it's specifications. In RAPIDE-0.2 there is a solution to this. You can declare a design unit body to be ***sequential***: *sequential*

```
design body Alarm-Clock is sequential


end Alarm_Clock;
```

This declares the body of the design unit to be a protected region in which at most one of its when-processes can be executing at any time. This prevents interleaving between when-processes.

4.3.2 Alarm-Handler

In this example we use the handler only for I/O. The I/O corresponds to the user pressing buttons on the alarm clock and for the ringing of the alarm.

The Alarm-Handler has two major when-processes (along with a couple of minor ones). The first is:

```
<< ring_to_console >>
when my-alarm : :Ring then
   put_line( "RRRRRRIIIIIIIIIINNNNNNNNNNNNNGGGGGGGGG");
end when;
```

This outputs a string to the console to inform the user that the alarm is ringing.

The other major process is:

```
< < get_command_from_console > >
when query-user then


end when;
```

Please refer to the listing for details. This is a when-process that takes commands from the console and translates them into something the Alarm-Clock instantiation (called my-alarm here) can understand. Each branch in the if statement, except the one that exits the if statement, ends with the line:

```
query-user pause 1;
```

You might ask why we need this and not just have a loop that keeps reading commands from the console instead. The reason is that whilst we are in the loop the value of the clock will always remain the same (assuming there are no pause or other delay-type constructs inside the if statement). This is certainly no good if we want to request the alarm to ring at some time in the future and actually observe it go off!

4.4 Program Listing

4.4.1 Design Unit Alarm-Clock

```
design Alarm-Clock is clocked

   in   action Set_Alarm(set_time : TIME);
   in   act ion Sleep;
   in   act ion Stop-Ringing;
   out  action Ring;

end Alarm-Clock;
```

```
design body Alarm_Clock is clocked sequential

   action ring-clock;
   alarm-set-time     : TIME := 0;
   sleep-interval     : TIME := 4;
   ?time_to_wake_up   : TIME;
   IDLE               : constant integer := 1;
   SET-TO-RING        : constant integer := 2;
   RINGING            : constant integer := 3;
   state              : integer := IDLE;

begin

   << Setting-the-Alarm >>
   when Set_Alarm(?time_to_wake_up)
      where (state = IDLE or state = SET-TO-RING) then
         state := SET-TO-RING;
         alarm-set-time := ?time_to_wake_up;
   end when;

   << Ring-Alarm >>
   when Tick
      where (state = SET-TO-RING and clock = alarm-set-time) then
         state := RINGING;
         ring-clock;
   end when;

   << Stop-The-Ringing >>
   when Stop-Ringing where (state = RINGING) then
      state := IDLE;
   end when;

   << Sleep-Feature >>
   when Sleep where (state = RINGING) then
      state := SET-TO-RING;
      alarm-set-time := clock + sleep-interval;
   end when;

   << Ringing >>
   when ring-clock where (state = RINGING) then
      Ring;
      ring_clock pause 1;
   end when;

end Alarm-Clock;
```

4.4.2 Design Unit Alarm_Handler

```
with Alarm-Clock;

design Alarm-Handler is global clocked
end Alarm-Handler;

design body Alarm-Handler is

   my-alarm  : Alarm-Clock;

   at-time : TIME;
   command  integer;

   act ion query-user;

begin

   << boot_up >>
   when start then
      query-user;
   end when;

   << ring_to_console >>
   when my_alarm::Ring then
      put_line( "RRRRRRIIIIIIIIIINNNNNNNNNNNNNGGGGGGGGG");
   end when;
```

```
  << get-command-from-console >>
  when query-user then
  put("The current time is "); put(clock); new-line;
  put_line( "Your wish is my command... zzz...");
  put("(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?");
    get-line(command);
    if (command = 1) then
      put("At what time do you wish to set the alarm :- ");
      get_line(at_time);
      my_alarm::Set_Alarm(at_time);
      query-user pause 1;
    elsif (command = 2) then
      my-alarm : :Stop_Ringing ;
      query-user pause 1;
    elsif (command = 3) then
      my_alarm::Sleep;
      query-user pause 1;
    elsif (command = 4) then
      query-user pause 1;
    elsif (command = 5) then
      my-alarm : :Stop_Ringing;
    else
      put_line("Sorry, no such button, please try again.");
      query-user pause 1;
    end if;
  end when;

end Alarm-Handler;
```

Chapter 5

Dish-Washer

5.1 Introduction

Imagine you are the manager of a high-class restaurant and you are confronted with a man who cannot pay his bill because he has left his wallet at home. As the title of this example suggests you are going to make him do the dishes! You throw him into the kitchen and start piling up the plates for him to wash. Every so often you bring him more plates, adding to the ones he already has. After every plate the man washes he complains that he has so many to wash and repeatedly asks when you are going to release him. Eventually you feel compassion for him and you let him go.

In this example *we* meet *constraints.* *constraints*

5.2 Discussion

What do we mean by a constraint? A constraint is a method by which we can ask the language to watch for certain patterns of events and, if they break the constraint, a special event called *Inconsistent* is generated by the design unit instance containing that constraint. Note that when we use a constraint in our design unit it doesn't mean that we are *forcing* the design unit to conform to our "rules", rather we are letting it do it's thing and checking to see if it is conforming to our "rules" as we go merrily along. In this manner we can write our prototypes and check that it is behaving as we want it to behave. We can use constraints to debug our own code or to debug our prototype.

Here are some examples of constraints (A, B, C... are actions and ?I, ?J... are placeholders):

Constraint 1.

```
when A(?I) where (?I > 0) then
   B(?J) where (?J < 0)
before C;
```

Constraint 2.

```
when (D and E) then
  F
before (G or H);
```

Constraint 3.

```
not K before L;
```

Constraint 1 and Constraint 2 look like when-processes. In order to understand these constraints we first have to understand how to read partial order graphs which result from running a RAPIDE-0.2 program.

A partial order graph is a directed acyclic graph. Each node in the graph represents an event from the computation. A directed edge represents an ordering between two events. In RAPIDE-0.2 the ordering is with respect *to potential causality* [Fid88, Mat88]. **A** directed edge from an event A to event B indicates that A causally preceded B. A can causally precede B in two situations:

1. when A triggers a when-process and B is generated in that when-process.
2. when A and B are in the same when-process and A precedes B in the linear order in the when-process (remember that the statements in the when-process are executed sequentially),

Let's return to our constraints. To interpret the constraints imagine that you have the partial order graph[1] for the computation in front of you. The semantics of Constraint 1 say that wherever you see an event A(?I), with ?I greater than 0, and an event C causally after A then there had better be an event B(?J), with ?J less than 0, that causally follows **A** and is also causally before C. Some possibilities are illustrated in figure 5.1.

The semantics of Constraint 2 say that wherever you see a pair of events D and E which may be causally related but not necessarily, and there is an event G or an event H (or both) which is causally after both D and E then there had better be an event F that is causally after both D and E and is also causally before the G or H event (or both). This constraint is equivalent to the following pair of constraints:

```
when (D and E) then
  F
before G;

when (D and E) then
  F
before H;
```

Some possibilities (for the G constraint only) are illustrated in figure 5.2.

[1] **see Appendix D for how you can use the partial order browser to display a partial order**

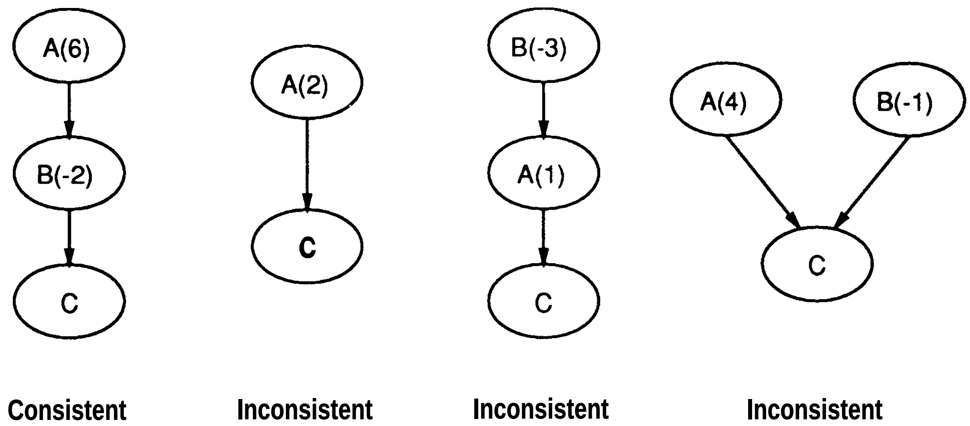

Figure 5.1: Subgraphs of full partial order giving examples of consistent and inconsistent patterns of events for Constraint

The semantics of example Constraint 3 says that wherever you see an event L there had better not be an event K that causally precedes it. This example could be rewritten:

```
when start then
   not K
before L;
```

A constraint that might appear in the Light Switch example of chapter 2 is

```
when Turn-On then
   Turn-Off
before Turn-On;
```

which expresses the intuition that if the light switch is on we cannot turn it on before we have first turned it off.

The constraint that appears in the Dish-Washer design unit is:

```
<< Negative-plates >>
not Complain(?plates_left) where (?plates_left < 0);
```

<< Negative-plates >> is a label for the constraint and acts in the same way as labels for when-processes. The semantics are that we had better not observe a Complain(?plates_left) event where ?plates_left is bound to a value less than zero. Physically we are trying to ensure that the number of unwashed plates is a non-negative quantity. This should be ensured in the code but we might make a mistake. If so the constraint would tell us if an event Complain was generated with a negative parameter.

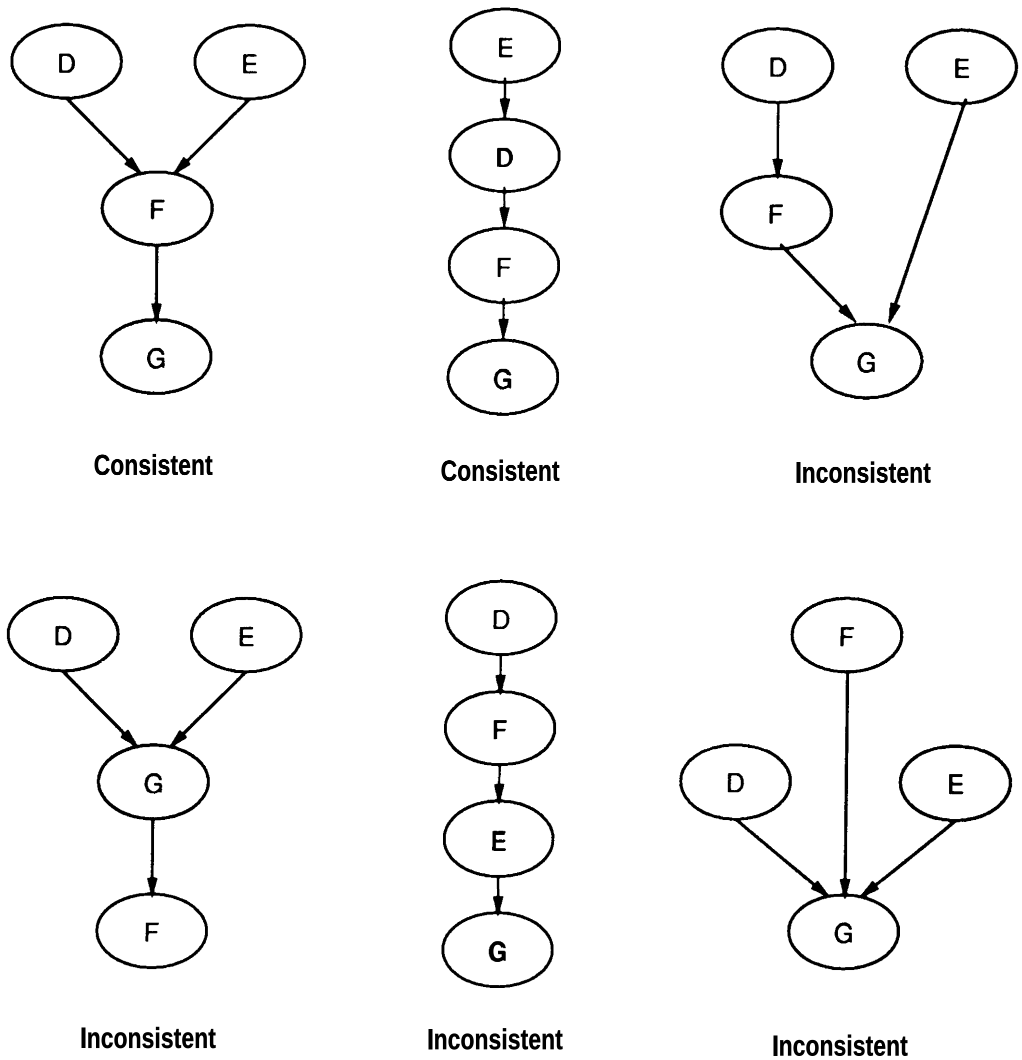

Figure 5.2: **Subgraphs of full partial order giving examples of consistent and inconsistent patterns of events for Constraint 2**

WARNING: The user should be aware that there are some subtleties involved with constraints in RAPIDE-0.2 that can cause problems. The examples above happened to be chosen quite carefully.

5.3 Implementation

We irnplement the dish washer as a design unit. Internally we want it to know about the plates it has to wash and that it should wash one per clock unit. Externally it should be able to accept plates from the manager, complain about all the work it has to do and also to be able to realize when it has been released from service.

The washer design unit is named Dish-Washer. The program is self-documenting and you should simply refer to the listing to see what is going on.

The design unit Dish-Washer is handled by the design unit Washer-Handler. It has one when-process to echo the complaints of the washer to the console. **A** second when-process handles input from the console. The user may either give the washer more plates or may release him from service.

Both design units are clocked with the handler being globally clocked. We need global clocks so that we can coordinate the interaction between the washer and the manager. Having it globally clocked means that the design units have to move in step. We want the washer to wash at some fixed rate. If there were local clocks the washer could run away from us by washing thousands of dishes before it ever accepted any from the manager. Notice that we could not put a loop in the << get-command >> when-process in the handler for otherwise whilst we are in this loop we would never see the complaints from the washer.

5.4 Program Listing

5.4.1 Design Unit Dish-Washer

```
design Dish-Washer is clocked

   in   action More_Plates(how_many_more : integer);
   out  action Complain(too_many_dishes : integer);
   in   action Release-Man;

end Dish-Washer;
```

```
design body Dish-Washer is

   action wash-some-plates;

   prisoner         : boolean := true;
   plates-to-wash : integer := 5;

   ?how_many-more : integer;
   ?plates_left        : integer;

   < < Negative-plates > >
   not Complain(?plates_left) where (?plates_left < 0);

begin

   << More_to_wash >>
   when More_Plates(?how_many_more) then
      plates-to-wash := plates-to-wash + ?how_many_more;
   end when;

   << boot-up >>
   when start then
      wash-some-plates;
   end when;

   < < do-some-washing > >
   when wash-some-plates where (prisoner) then
      plates-to-wash := plates-to-wash - 1;
      Complain(plates_to_wash);
      wash-some-plates pause 1;
   end when;

   << Free-Man >>
   when Release-Man then
      prisoner := false;
   end when;

end Dish-Washer;
```

5.4.2 Design Unit Washer-Handler

```
with Dish-Washer;

design Washer-Handler is global clocked
end Washer-Handler;

design body Washer-Handler is

   restaurant-slave : Dish-Washer;
   how-many-more: integer;
   ?too_many-dishes integer;
   act ion query_user;

begin

   << complain_to_console >>
   when restaurant_slave::Complain(?too_many_dishes) then
      new-line;
      put_line("How much longer do I have to stay here? " &
               "I still have");
      put(?too_many_dishes);
      put_line(" dishes to wash!! ");
      new-line;
   end when;

   << boot-up >>
   when start then
      query-user;
   end when;
```

```
   << get-command >>
   when query-user then
      put_line("How many more dishes would you want " &
               "the man to wash?");
      put("Enter number (negative number to release man) :- ");
      get_line( how-many-more) ;
      if (how-many-more >= 0) then
         restaurant_slave::More_Plates(how_many_more);
         query-user pause 1;
      else
         restaurant_slave::Release_Man;
      end if;
   end when;

end Washer-Handler;
```

Chapter 6

Satellite Communication Link II

6.1 Introduction

This example is another version of the Satellite Communication Link with only one small change. The purpose is to introduce the idea of *connections.*

6.2 Discussion

connections

Connect statements provide a method to build architectures. We have seen how in chapter 3 the handler was used to glue together the architecture of the prototype. In particular we used a when-process to create a one-way communication channel from the out action of a City design unit instance to an in-action of the Satellite design unit instance:

```
<< connect-city-to-satellite >>
when Cities[?city]::Send_Message(?to, ?message) then
  Sat::Receive_From_City(?city, ?to, ?message);
end when;
```

A connect statement is also used in building the architecture in this fashion:

```
<< connect-city-to-satellite >>
connect Cities[?city]::Send_Message(?to, ?message) with
  Sat::Receive_From_City(?city, ?to, ?message);
end connect;
```

There are two related differences between these two approaches. The first is in the semantics. The second is in the partial order. Consider the following example.

```
with human;
with button;

design body Press-Button is

   George  : human;
   on-off  : button;

   connect George::push_button
      with on_off::button_being_pushed;
   end connect;


end Press_Button;
```

Press-Button contains two design unit instances representing a human named George, and a button named on-off. We can view the action of George pushing a button in two ways. From George's point of view he is pushing a button. From the buttons point of view it is being pushed. These two actions are really one and the same even though the two parties involved have their own perspectives. We express this idea in the connect statement above. In the partial order George::push_button and on_off::button_being_pushed are the same event and we have two ways of naming the event.

Going back to our Satellite Communication Link example we can replace the two **when**-processes with two connect statements. How we now view the sending and receiving of messages is different. Before, when a city sends a message, this *causes* the satellite to receive it. Now, with a connect statement, the sending of a message and the reception by the satellite are the *same* event. Appendix B.6 shows how the partial order graph is changed.

The syntax of connect statements allows the connection of any observable pattern of events (observed by some design unit instance) to be connected to any set of actions that can be performed (by that design unit instance). A slightly more complicated example is:

```
connect A or B with
   C;
   D;
end connect;
```

6.3 Implementation

The program is the same as for Satellite Communication Link except for the replacements described above

6.4 Program- Listing

6.4.1 Design Unit City

```
design City is

  in  action Send-Trigger(to : integer; message : string);
  in  action Receive-Message&m : integer; message : string);
  out action Send-Message(to : integer; message : string);

end City;

design body City is

  ?message : string;
  ?to        integer;

begin

  << Send-Messages >>
  when Send_Trigger(?to, ?message) then
    Send-Message(?to, ?message);
  end when;

end city;
```

6.4.2 Design Unit Satellite

```
design Satellite is

  in  action Receive-From-City(from, to : integer; message : string);
  out action Relay-Message-To-City(from, to : integer; message : string);

end Satellite;
```

```
design body Satellite- is

  ?from, ?to : integer;
  ?message   : string;

begin

  << Relay-Messages >>
  when Receive-From-City(?fm, ?to, ?message) then
    Relay-Message-To-City(?fmm, ?to, ?message);
  end when;

end Satellite;
```

6.4.3 Design Unit link-handler

```
with City, Satellite;

design link-handler is
end link-handler;

design body link-handler is

  Cities : array[1..3] of City;
  Sat    : Satellite;

  ?from, ?to          : integer;
  ?message            : string;
  ?city               : integer;
  send-from, send-to  : integer;
  msg                 : string;

  << connect-city-to-satellite >>
  connect Cities[?city]::Send_Message(?to, ?message) with
    Sat::Receive_From_City(?city, ?to, ?message);
  end connect;

  < < connect-satellite-to-city > >
  connect Sat::Relay_Message_To_City(?from, ?to, ?message) with
    Cities[?to]::Receive_Message(?from, ?message);
  end connect;
```

```
begin

   << Input-Output >>
   when start then
      put_line("Welcome!  Please input originator, destination,");
      put_line( "and content of your messages.");
      new-line;
      loop
         put("Send message from which city (0 to quit) -> ");
         get_line( send-from) ;
         if (send-from /= 0) then
            put("Send message to which city -> ");
            get_line(send_to);
            put("Enter the message -> ");
            get_line(msg);
            Cities[send_from ]::Send_Trigger(send_to, msg);
         else
            exit;
         end if;
      end loop;
   end when;

end link-handler;
```

Chapter 7

Baking a Cake

7.1 Introduction

This example illustrates the constraint language of RAPIDE-0.2.

In baking a cake (and indeed in cooking in general) it is important to throw in your ingredients and mix them in the right order. This example shows how we can constrain the order we do things when we bake a cake.

7.2 Discussion

The process that we are going to bake the cake by is the following:

(a) Buy the ingredients for the cake (the dry ingredients are flour, sugar and baking soda, the wet ones are water, milk and eggs).

(b) Put the dry ingredients into the "dry" bowl.

(c) Put the wet ingredients into the "wet" bowl.

(d) Mix the dry ingredients.

(e) Mix the wet ingredients.

(f) Mix the wet and dry ingredients together.

(g) Put the mix into the baking tray and stick it in the oven.

(h) Remove the cake from the oven.

However we would all agree that (b) through (e) could be done in various orders. Indeed we don't have to put all three dry ingredients into a bowl before mixing them, we could put two in and mix those before mixing in the third.

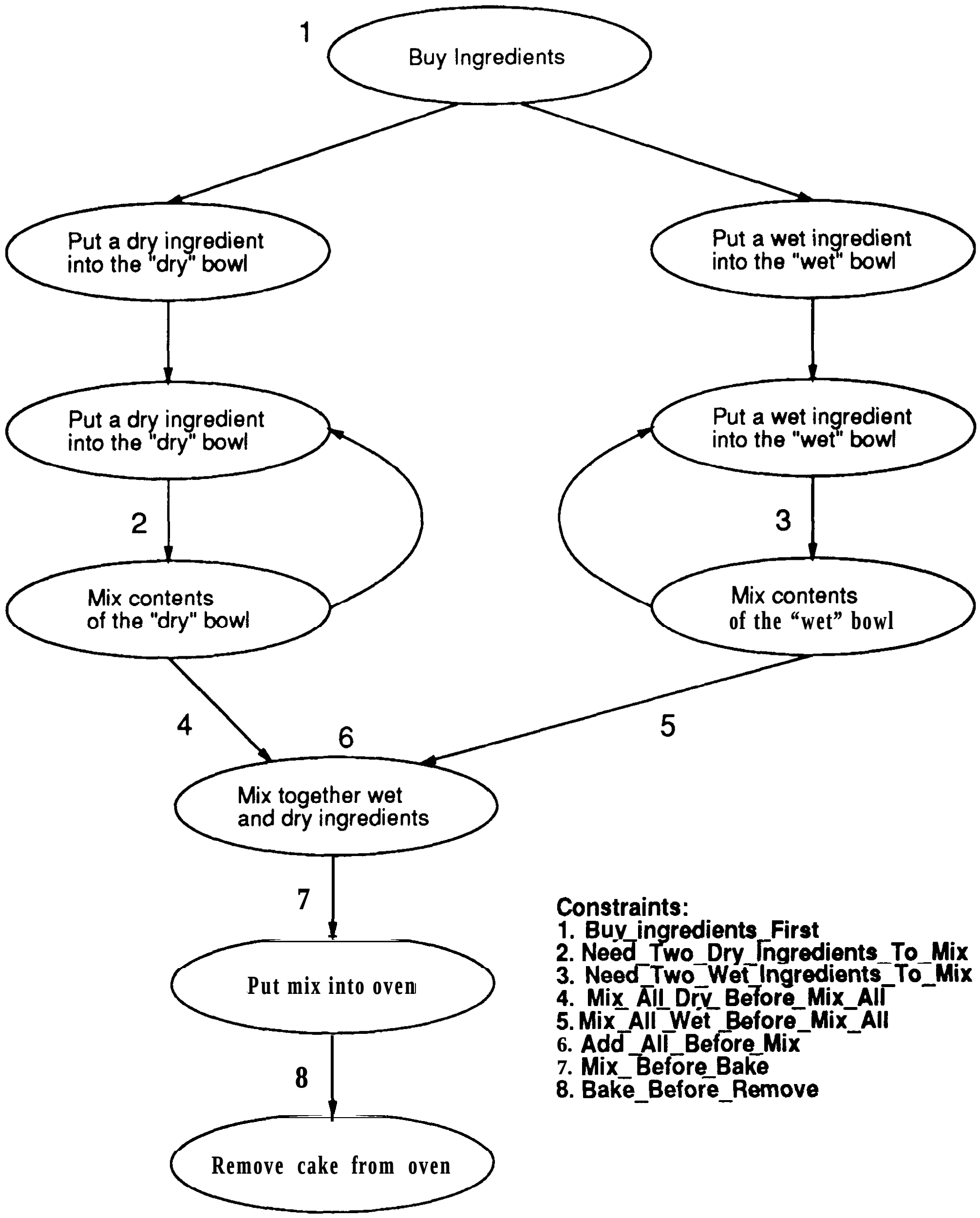

Figure 7.1: Schedule for baking the cake

Somehow we have-to specify exactly what we are going to allow and translate it into constraints.

The constraints we shall impose are as follows. The order in which things must be done are as in the list above except for the following:

1. You can put dry and wet ingredients into their respective bowls in any order you like (one dry, two wet, one dry etc.).
2. There must be two items in the dry bowl before you can mix them. Similarly for the wet bowl.
3. When all three dry ingredients are in the dry bowl they must be mixed before they can be mixed with the wet ingredients. Similarly with the wet ingredients.

These ideas are shown in figure 7.1. The figure is a pictorial representation of what needs to be done in order to get the cake baked. Note that the addition and mixing of the two types of ingredients can be done in parallel. The figure also shows an intuitive idea of where the constraints we will use apply in the schedule of baking the cake.

It is fairly easy to implement a straight ordering of events. We will only have to think a little harder when it comes to the addition and mixing of the dry and wet ingredients.

7.3 Implementat ion

The only interesting part of the implementation is the construction of the constraints. There are two major pieces to it. The first is getting the strict ordering of (a), (f), (g) and (h). The second is to deal with the variety of different ways of mixing the ingredients ((b) through (e)).

Here is a solution to the first part:

```
action  Buy-Ingredients;
act ion Add-Dry-Ingredient (I : integer) ;
action Add_Wet_Ingredient(I : integer);
action Mix_Dry_Ingredients;
action  Mix-Wet-Ingredients;
action  Mix-Dry-And-Wet-Ingredients;
action  Bake-Cake-In-Oven;
action  Remove-Cake-From-Oven;

?I, ?J : integer;
?Act   : action-type;

   < < Buy-Ingredients-First > >
when start then
   Buy-Ingredients
until ?Act where (?Act /= Inconsistent);
```

```
< < Mix-Before-Bake >>
not (Bake-Cake-In-Oven or Remove-Cake-From-Oven)
before Mix-Dry-And-Wet-Ingredients;

< < Bake-Before-Remove > >
not Remove-Cake-From-Oven
before Bake-Cake-In-Oven;
```

?Act is declared as an action type. It matches an event of any action. It is used in the constraint labeled Buy-Ingredients-First. What does this constraint say? It says look at the partial order graph for the computation; find the event Start (for this design unit instance) and find any event that matches ?Act, except for Inconsistent events, where ?Act is causally after the Start event. Then either ?Act must be a Buy-Ingredients event or a Buy-Ingredients event should have occurred causally after Start and before ?Act. If not then the constraint is violated. In short we had better have a Buy-Ingredients event immediately after the Start event. The until allows ?Act to match with Buy-Ingredients. The guard is there to prevent an infinite number of Inconsistent events being generated by this constraint (if the guard wasn't there an Inconsistent event can violate this constraint thus generating another Inconsistent which can in turn violate the constraint etc. etc. etc.).

The constraint << Mix_Before_Bake >> says that we must mix the dry and wet ingredients before putting the mix into the oven or removing the cooked cake from the oven. This constraint is necessary because of the following sequence of events:

Start ⇒ Buy_Ingredients ⇒ Bake-Cake-In-Oven

This sequence would not violate any of the other constraints presented above and below.

The constraint << Bake_Before_Remove >> ensures we bake the cake before we remove it from the oven.

The second set of constraints are:

```
< < Add_All_Before_Mix > >
Add-Wet-Ingredient (1) and Add_Wet_Ingredient(2)
   and Add_Wet_Ingredient(3) and Add_Dry_Ingredient(1)
   and Add_Dry_Ingredient(2) and Add_Dry_Ingredient(3)
before Mix-Dry-And-Wet-Ingredients;

< < Mix-All-Dry-Before-Mix-All > >
when Add-Dry-Ingredient (1)
   and Add_Dry_Ingredient(2) and Add_Dry_Ingredient(3) then
   Mix-Dry-Ingredients
before Mix_Dry_And_Wet_Ingredients;
```

```
<< Mix_All_Wet_Before_Mix_All >>
when Add_Wet_Ingredient(1)
  and Add_Wet_Ingredient(2) and Add_Wet_Ingredient(3) then
  Mix_Wet_Ingredients
before Mix-Dry-And-Wet-Ingredients;

<< Need-Two-Dry-Ingredients-To-Mix >>
(Add_Dry-Ingredient (?I) and Add_Dry_Ingredient(?J))
  where ?I /= ?J
before Mix-Dry-Ingredients;

<< Need_Two_Wet_Ingredients_To_Mix >>
(Add_Wet_Ingredient(?I) and Add_Wet_Ingredient(?J))
  where ?I /= ?J
before Mix-Wet-Ingredients;
```

There are three dry and three wet ingredients. These are described with integers. For example the event Add_Dry_Ingredien t (1) would correspond to adding the dry ingredient number 1 to the dry mixing bowl.

The constraint << Add_All_Before_Mix >> says that we must have added all six ingredients before we mix them together.

<< Mix_All_Dry_Before_Mix_All >> says that we should mix all the dry ingredients together before mixing them in with the wet ones.

Similarly << Mix_All_Wet_Before_Mix_All >> says we should mix all the wet ingredients together before mixing them in with the dry ones.

<< Need_Two_Dry_Ingredients_To_Mix >> says that we are not allowed to mix the dry ingredients before we have added at least two of them to the bowl. Similarly for << Need_Two_Wet_Ingredients_To_Mix >> except for wet ingredients.

The rest of the design unit is for getting commands from the console. From the console you can perform one action at a time by choosing an option at the prompt. The when-process contains a loop that keeps reading in your commands until you quit.

7.4 Program Listing

7.4.1 Design Unit BakeCake

```
design BakeCake is

end BakeCake;
```

```
design body BakeCake is

   command : integer;

   action  Buy-Ingredients;
   action Add_Dry_Ingredient (I : integer) ;
   action Add-Wet-Ingredient (I : integer) ;
   action  Mix_Dry_Ingredients;
   action Mix_Wet_Ingredients;
   action  Mix_Dry_And_Wet_Ingredients;
   action  Bake-Cake-In-Oven;
   action  Remove-Cake-From-Oven;

   ?I, ?J : integer;
   ?Act   : action-type ;

   < < Buy-Ingredients-First > >
   when start then
      Buy-Ingredients
   until ?Act where (?Act /= Inconsistent);

   < < Mix-Before-Bake > >
   not (Bake-Cake-In-Oven or Remove-Cake-From-Oven)
   before  Mix-Dry-And-Wet-Ingredients;

   < < Bake-Before-Remove > >
   not  Remove-Cake-From-Oven
   before  Bake-Cake-In-Oven;

   < < Add-All-Before-Mix > >
   Add_Wet_Ingredient(1)  and Add_Wet_Ingredient(2)
      and Add_Wet_Ingredient(3)  and Add_Dry_Ingredient( 1)
      and Add_Dry_Ingredient(2)  and Add_Dry_Ingredient(3)
   before  Mix-Dry-And-Wet-Ingredients;

   < < Mix_All_Dry_Before_Mix_All > >
   when  Add-Dry-Ingredient  (1)
      and Add_Dry_Ingredient(2)  and Add_Dry_Ingredient(3) then
      Mix-Dry-Ingredients
   before  Mix-Dry-And-Wet-Ingredients;

   < < Mix_All_Wet_Before_Mix_All > >
   when Add_Wet_Ingredient( 1)
      and  Add-Wet-Ingredient(2)  and  Add-Wet-Ingredient(3)  then
      Mix-  Wet-Ingmdients
   before  Mix-Dry-And-Wet-Ingredients;
```

```
  < < Need_Two_Dry_Ingredients_To_Mix > >
  (Add-Dry-Ingredient(?I)  and Add_Dry_Ingredient(?J))
     where ?I /= ?J
  before Mix_Dry_Ingredients;

  < < Need-Two-Wet-Ingredients-To-Mix > >
  (Add_Wet_Ingredient(?I)  and Add_Wet_Ingredient(?J))
     where ?I /= ?J
  before  Mix-Wet-Ingredients;

begin

  << get-commands >>
  when start then
    loop
      put_line("What would you like to do next?“);
      put_line(" 1) buy the ingredients.“);
      put_line(" 2) put flour into ‘dry’ bowl.“);
      put_line(" 3) put sugar into ‘dry’ bowl.“);
      put_line(" 4) put baking soda into ‘dry’ bowl.“);
      put_line(" 5) put water into ‘wet’ bowl.“);
      put_line(" 6) put milk into ‘wet’ bowl.“);
      put_line(" 7) put eggs into ‘wet’ bowl.“);
      put_line(" 8) mix contents of ‘dry’ bowl.“);
      put_line(" 9) mix contents of ‘wet’ bowl.“);
      put_line(" 10) mix contents of ‘dry’ " &
               “and ‘wet’ bowl together.“);
      put_line(" 11) put mix into baking tray " &
               “and put cake into oven.“);
      put_line(" 12) remove cake from oven and eat it!“);
      put_line(" 13) Give up on cake.“);
      put(" Enter option -> ");
      get-line(command);
```

```
      if (command = 1) then
         Buy-Ingredients;
      elsif (command = 2) then
         Add_Dry_Ingredient(1);
      elsif (command = 3) then
         Add_Dry_Ingredient(2);
      elsif (command = 4) then
         Add_Dry_Ingredient(3);
      elsif (command = 5) then
         Add- Wet-Ingredient (1) ;
      elsif (command = 6) then
         Add_Wet_Ingredient(2);
      elsif (command = 7) then
         Add_Wet_Ingredient(3);
      elsif (command = 8) then
         Mix-Dry-Ingredients;
      elsif (command = 9) then
         Mix-Wet-Ingredients;
      elsif (command = 10) then
         Mix_Dry_And_ Wet-Ingredients ;
      elsif (command = 11) then
         Bake_Cake_In_Oven;
      elsif (command = 12) then
         Remove-Cake-From-Oven;
      elsif (command = 13) then
         exit;
      else
         put_line("Sorry, no such button, please try again.“);
      end if;
   end loop;
end when;

end BakeCake;
```

Chapter 8

Library

8.1 Introduction

In this example we will consider a library as a place which contains books and is willing to lend them out to people.

This example introduces the idea of *properties.* *properties*

8.2 Discussion

You can think of properties as describing the internal state of a design unit. Or you can think of them as unconstrained arrays. We will use properties to store information about the books in the library.

A simple example of a use of properties might be to record a series of integers. Suppose you were playing blackjack with an infinitely large deck of cards and you wanted to remember which cards had been used. You assign an integer to each card and each time a card is used you set the boolean-valued property element corresponding to the integer to true. In RAPIDE-0.2 you would write:

```
?card : integer;
property card_used(integer) : boolean := false;

when play_card(?card) then
  card_used(?card) := true;
end when;
```

The property declaration says that card-used is boolean-valued, that it takes an integer as an argument, and that the default value of each property element is false. Examples are card_used(10) or card-used(23). Each of these corresponds to one of your cards. You are required to give a default value for the property elements when you declare the property. The when-process sets the value of the property element corresponding to a card to true when that card is played. If you wanted to be slightly more descriptive you could write:

```
?suit : string;
?value : integer;
property card_used(integer, string) : boolean := false;

when play_card(?value, ?suit) then
  card_used(?value, ?suit) := true;
end when;
```

In this example we declare two properties:

```
property book-owned(string)     : boolean := false;
property book-on-loan(string) : boolean := false;
```

The first records titles of books which are in the library. For example:

```
book-owned( “Twenty Thousand Leagues under the Sea”) := true;
```

would indicate that the book titled “Twenty Thousand Leagues under the Sea” was a book that was owned by the library. Note that the default value for this property is false.

The second property describes whether the book is currently out on loan, true if it is and false if it is not.

As we mentioned before you can think of properties as describing the internal state of a design unit or you can think of them as unconstrained arrays. This works as follows in our example. We can view part of internal state as the status of the books in the library, whether they are owned and whether they are on loan. Alternatively the two properties book-owned and book-on-loan could be viewed as unconstrained arrays since both can take arbitrary strings as arguments.

The specifications for our library will be

1. The library contains books that are keyed by title.
2. The library lends out books to borrowers.
3. The library must be correct.

By “correct” we mean that the contents of the library are consistent with the transactions that have occurred between the library and borrowers, and that all transactions are valid. For example a borrower cannot return to the library a book which he never borrowed. These specifications are very simple and obviously inadequate to describe a real library. We might want to include data about the books such as the author, the publisher and so on. However it does have the strong requirement that it must be correct. An interesting extension might be to allow the books to be on loan only for a fixed period of time. If the borrower does not return it we send him a nasty message. Alternatively we could fine him when he returns the book.

8.3 Implementation

The Library design unit is centered around the properties described above. There are three in-actions and one out:

```
in    act ion Book-Donation (title : string) ;
in    act ion Borrow_Book( title : string) ;
in    act ion Return_Book( title : string) ;
out  action Librarian-Message-Out(msg : string);
```

Each in-action has a corresponding when-processes. The when-processes set the values of the properties appropriately and test for invalid transactions. The librarian responds to actions of the user by sending out appropriate messages.

Initially the library starts out empty (the property book-owned is defaulted to false). The library accumulates books by donations from borrowers. Once a book has been donated it can be borrowed.

8.4 Program Listing

8.4.1 Design Unit Library

```
design Library is clocked

   in    act ion Book_Donation( title : string) ;
   in    act ion Borrow_Book( title : string) ;
   in    act ion Return_Book( title : string) ;
   out  action Librarian-Message-Out(msg : string);

end Library;

design body Library is

   property book-owned(string)     : boolean := false;
   property book-on-loan( string) : boolean := false;

   ?title : string;
```

```
begin

  << Introduce-New-Book >>
  when Book-Donation(?title) then
    book-owned(?title) := true;
    Librarian-Message-Out(“Thank you for your contribution!“);
  end when;

  << Lend-Out-Book >>
  when Borrow_Book(?title) then
    if (book-owned(?title) = false) then
      Librarian-Message-Out( “Sorry, we don’t have that book.“);
    elsif (book-on-loan(?title) = true) then
       Librarian-Message-Out (“Sorry, someone else " &
                              “has that book right now”);
    else
      book-on-loan(?title) := true;
      Librarian-Message-Out(“You can have the book for " &
                            “five days! ");
    end if;
  end when;

  << Book-Brought-Back >>
  when Return_Book(?title) then
    if (book-owned(?title) = false) then
      Librarian-Message-Out( “That book doesn’t exist! ");
    elsif (book-on-loan(?title) = false) then
      Librarian_Message_Out("You can’t bring a book back " &
                            “that we already have!“);
    else
      book-on-loan(?title) := false;
      Librarian-Message-Out( “Thank you for returning the book.“);
    end if;
  end when;

end Library;
```

8.4.2 Design Unit Library-Handler

```
with Library;

design Library-Handler is global clocked
end Library-Handler;
```

```
design body Library_Handler is

   My-Local-Library  :  Library;

   command : integer;
   ?msg      : string;
   title       : string;

   act ion query-user;

begin

   < <  boot_up > >
   when start then
      query-user;
   end when;

   < < Message-From-Librarian > >
   when My_Local_Library::Librarian_Message_Out(?msg) then
      put_line( ?msg);
   end when;
```

```
<< get-command-from-console >>
when query-user then
  put_line(" (1) Deposit new book");
  put_line(" (2) Take out a book");
  put_line(" (3) Return a book");
  put_line(" (4) Stop going to this library");
  put("What would you like to do next -> ");
  get_line(command);
  if (command = 1) then
    put(" Enter title of book -> ");
    get_line( title);
    My_Local_Library::Book_Donation(title);
    query-user pause 1;
  elsif (command = 2) then
    put(" Enter title of book -> ");
    get_line(title);
    My_Local_Library::Borrow_Book(title);
    query_user pause 1;
  elsif (command = 3) then
    put(" Enter title of book -> ");
    get_line( title);
    My_Local_Library::Return_Book( title);
    query-user pause 1;
  elsif (command = 4) then
    null;
  else
    put_line("Sorry, no such command, please try again.");
    query-user pause 1;
  end if;
end when;

end Library-Handler;
```

Appendix A

Keyword Index

Appendix B

Compiling and running the examples

You are encouraged to experiment with the examples and play with the partial order browser. What follows is a brief description of how to set yourself up and get the programs running. Appendix C gives the locations of the programs if you have access to Anna.stanford.edu.

B.l Getting going

This documentation assumes you are on a Unix platform and have obtained all the files you are interested in. The relevant files for each example are listed in the individual sections below.

Each example should have it's own directory with all the files for that example in that directory. The first thing to do is to create a CPL library for your directory (we will denote the Unix shell prompt by ">"):

```
> cpl.mklib
```

This creates some directories and files which are needed for compilation.

All of the examples have a make file named `Makefile`. You can do all the compilations for an example by typing `make` at the Unix prompt.

The Light Switch section below describes in more detail the sequence of compilations that are needed.

Two environment variables need to be mentioned. The environment variable `TMPPATH` should be set to wherever you want temporary files to be held during compilation. e.g.

```
setenv TMPPATH /usr/tmp
```

The `CPL_RTS` environment variable (RTS stands for Run-Time System) gives the user a choice over run-time options. Full details are in the CPL man-page. Two of the more important options are binary-log and echo-events. binary-log is discussed in the partial order browser appendix. It tells the RTS to generate a file from which the partial order browser can read information about the computation. The echo-events option can be set:

```
setenv CPL_RTS echo-events
```

This tells the RTS to write to the console all the events that are generated. This is equivalent to a linear trace of the events in the computation.

B.1.1 Anna.stanford.edu

This is a special section for those working on Anna.stanford.edu. The development of the tools for RAPIDE-0.2 are being done on Anna.stanford.edu and consequently there are two versions of the tools residing on the machine.

The stable version of the transformer (the program which compiles your RAPIDE-0.2 programs into Ada) is called the stage version and the development version is called the *dev* version. Anna.stanford.edu is also a parallel machine and the Ada compiler is capable of compiling the transformed RAPIDE-0.2 code into a *serial* executable or into a ***parallel*** executable.

The possible choices of version and environment for the RAPIDE-0.2 programmer are given when the CPL library is created as shown by the following real example:

```
anna > cpl .mklib
Which version and environment?
   1) dev   serial
   2) dev   parallel
   3) stage parallel
>
```

The library contains code that is linked into the final executable. These libraries are assumed by the stage and dev transformers. It is required that you match the transformer to the library. Thus, if you chose option 1 or 2 you must use the *dev* transformer and if you chose option 3 you must use the stage transformer. The locations of the transformers are:

```
/cpl/executable/drivers/dev/cpl
/cpl/executable/drivers/stage/cpl
```

The transformer referred to in the CPL man-page is the stage version.

The provided **Makef** ile for each example has as the first line

```
CPL = cpl
```

If the **Makefile** is used as it stands the transformer used will be the one that is in your (Unix) environment variable PATH unless you had given the full pathname for the transformer. In this case you must correctly match the transformer to the library chosen. As a way of controlling which transformer is used you can explicitly change the line in the **Makef** ile e.g.

```
CPL = /cpl/executable/drivers/dev/cpl
```

to use the dev transformer.

All the examples worked on the stage version of the tools at the time of writing. As development continues on the toolset it is possible that the examples may need updating or the tools may be enhanced beyond what is described in this document. A Technical Note (CSL-TN-92-387, Program Analysis and Verification Group Report No. 58) is available which describes the current status of the tools and gives manual pages for the tools.

B.2 Light Switch

You should have the following files:

```
switch.cpl
switch_handler.cpl
Makefile
```

You can do all the compilations in one fell swoop by typing make at the Unix prompt. Or you can do it by hand:

```
> sem switch.cpl
> cpl switch.cpl
> sem switch_handler.cpl
> cpl -M switch-handler switch_handler.cpl
> ada switch.a
> ada -M main-switch-handler -o light-switch switch-handler.a
```

The first line applies the CPL *Flexible Semanticizer* [MKS92] to **switch.cpl**. The Flexible Semanticizer is a stand-alone parser and semanticizer for RAPIDE-0.2. The idea behind the Flexible Semanticizer is that should the language that it is being applied to change it can be upgraded in a much faster time than the code that does the semanticizing in the compiler. It will soon be the case that the Flexible Semanticizer will hook into the transformer and become the default semanticizer.

The second line applies the RAPIDE-0.2 transformer to **switch.cpl**. The transformer transforms RAPIDE-0.2 source code to Ada [Ada83] source code. The fourth line does the same for the CPL file switch-handler. cpl. The **-M** option indicates that the design unit switch-handler is the name of the main design unit. In the Ada source code the main program will be correspondingly named Main-switch-handler. switch. cpl should be transformed before switch-handler. cpl because the design unit switch-handler has a dependency on the switch design unit.

We now have the Ada source code which is compiled in the regular (Ada) fashion. First switch.a is compiled, then switch-handler.a. The **-M** option indicates the main Ada procedure is main-switch-handler and the **-o** option indicates that we wish to name the executable light-switch. Again we compile switch. a first because switch-handler. a has a dependency on it.

The executable, `light-switch,` can be run by simply typing it's name. Since there is no interaction with the user it will run without interruption until it is finished:

```
> light-switch
>
```

The program generates the partially ordered events which the partial order browser can display. Figure B. 1 shows the graph of the partial order of events generated by `light-switch` (Appendix D describes how to do this).

We observe from figure B.l that the two events Turn-on and Turn-Off caused the events Switch-Is-On and Switch-Is-Off respectively. Two more observations about the semantics of RAPIDE-0.2 can be seen from the graph. Firstly, the Start events causally precede all events in their respective design units. Secondly, Turn-On causally precedes Turn-Off. This comes from the fact that Turn_On is sequentially before Turn-Off in the same when-process.

B.3 Satellite Communication Link

You should have the following files:

```
city. cpl
satellite.cpl
link-handler.cpl
Makefile
```

Compile the files by running make and then run the executable by running link. You will be given several prompts to which you should enter responses. Here is a sample run:

```
> link
Welcome!  Please input originator, destination,
and content of your messages.

Send message from which city (0 to quit) -> 1
Send message to which city -> 2
Enter the message -> Hello!
Send message from which city (0 to quit) -> 3
Send message to which city -> 1
Enter the message -> How's the weather?
Send message from which city (0 to quit) -> 0
>
```

B.4 Snooze Alarm

You should have the following files:

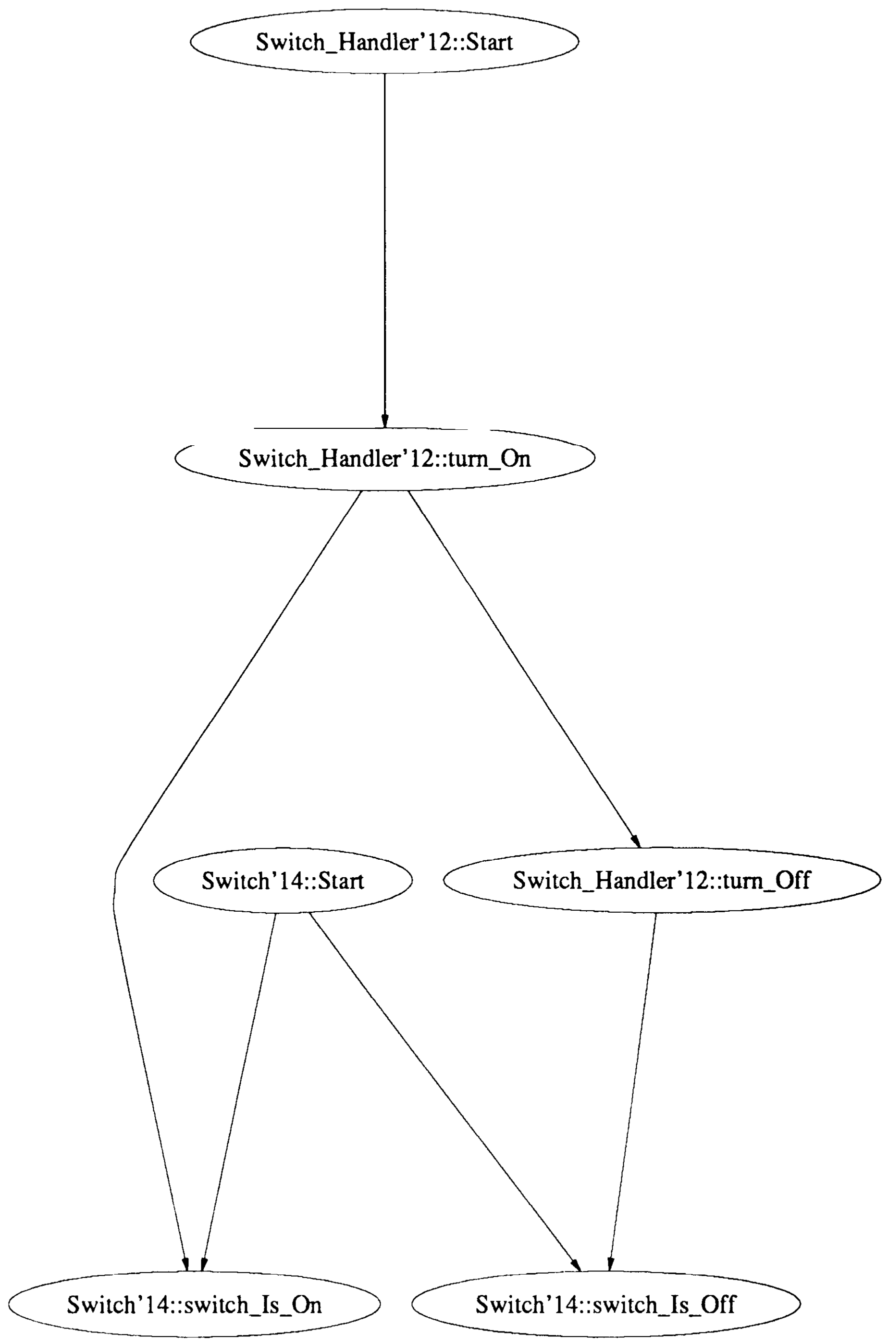

Figure B.l: partial order of events

```
alarm.cpl
alarm-handler.cpl
Makefile
```

Compile the files by running make and then run the executable by running alarm. You will be confronted with a list of possible actions you can take. You can interact with the alarm clock by entering appropriate integers and following any instructions that come up. Each time you enter an instruction the clock will advance by one unit. Here is a sample run:

```
> alarm
The current time is          0
Your wish is my command... zzz...
(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?1
At what time do you wish to set the alarm :- 6
The current time is          1
Your wish is my command... zzz...
(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?1
At what time do you wish to set the alarm :- 3
The current time is          2
Your wish is my command... zzz...
(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?4
RRRRRRIIIIIIIIIIINNNNNNNNNNNNGGGGGGGGG
The current time is          3
Your wish is my command... zzz...
(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?4
RRRRRRIIIIIIIIIIINNNNNNNNNNNNGGGGGGGGG
The current time is          4
Your wish is my command... zzz...
(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?3
The current time is          5
Your wish is my command... zzz...
(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?4
The current time is          6
Your wish is my command... zzz...
(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?4
The current time is          7
Your wish is my command... zzz...
(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?4
RRRRRRIIIIIIIIIIINNNNNNNNNNNNGGGGGGGGG
The current time is          8
Your wish is my command... zzz...
(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?4
RRRRRRIIIIIIIIIIINNNNNNNNNNNNGGGGGGGGG
The current time is          9
Your wish is my command... zzz...
(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?1
At what time do you wish to set the alarm :- 15
```

```
The current tiine is 10
Your wish is my command... zzz...
(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?4
RRRRRRIIIIIIIIIINNNNNNNNNNNNGGGGGGGGG
RRRRRRIIIIIIIIIINNNNNNNNNNNNGGGGGGGGG
The current time is 11
Your wish is my command... zzz...
(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?2
The current time is 12
Your wish is my command... zzz...
(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?4
The current time is 13
Your wish is my command... zzz...
(1) set alarm (2) stop ringing (3) sleep (4) iterate (5) quit ?5
```

The iterate option does nothing apart from advancing the clock by one clock unit.

The partial order browser was used to generate figure B.2 which shows the partial order for the sample run above. The graph has been manipulated so that only the main features are showing and in a more intuitive form. The actions Start, next and ring_clock were switched off using the toggles in the options window. In the styles menu "time-line" and "rank by process" were chosen. In the labels menu "action", "parameters" and "departure time" were chosen. The events in the partial order have either one or two parameters showing. For Set-Alarm the first parameter is the time at which the alarm is set to ring and the second is the departure time of the event. For Sleep, Stop-Ringing and Ring the single parameter is the departure time. For Tick the first parameter and second parameter are both equal to the departure time (the second is really the departure time, the first is a redundant parameter called "now").

B.5 Dish-Washer

You should have the following files:

```
washer.cpl
washer_handler.cpl
Makefile
```

Compile the files by running make and then run the executable by running washer. During the execution of the program you will play the part of the manager who gives the washer plates to wash. Every so often the handler will print a message on the console echoing the complaints of the washer about all those plates he has to wash. You can enter in a non-negative integer to give the man plates to wash or you can enter a negative integer to release the man and end the program.

By playing with the number of dishes you give the man you can control when the constraint will be violated. Here is a sample run:

```
> washer
```

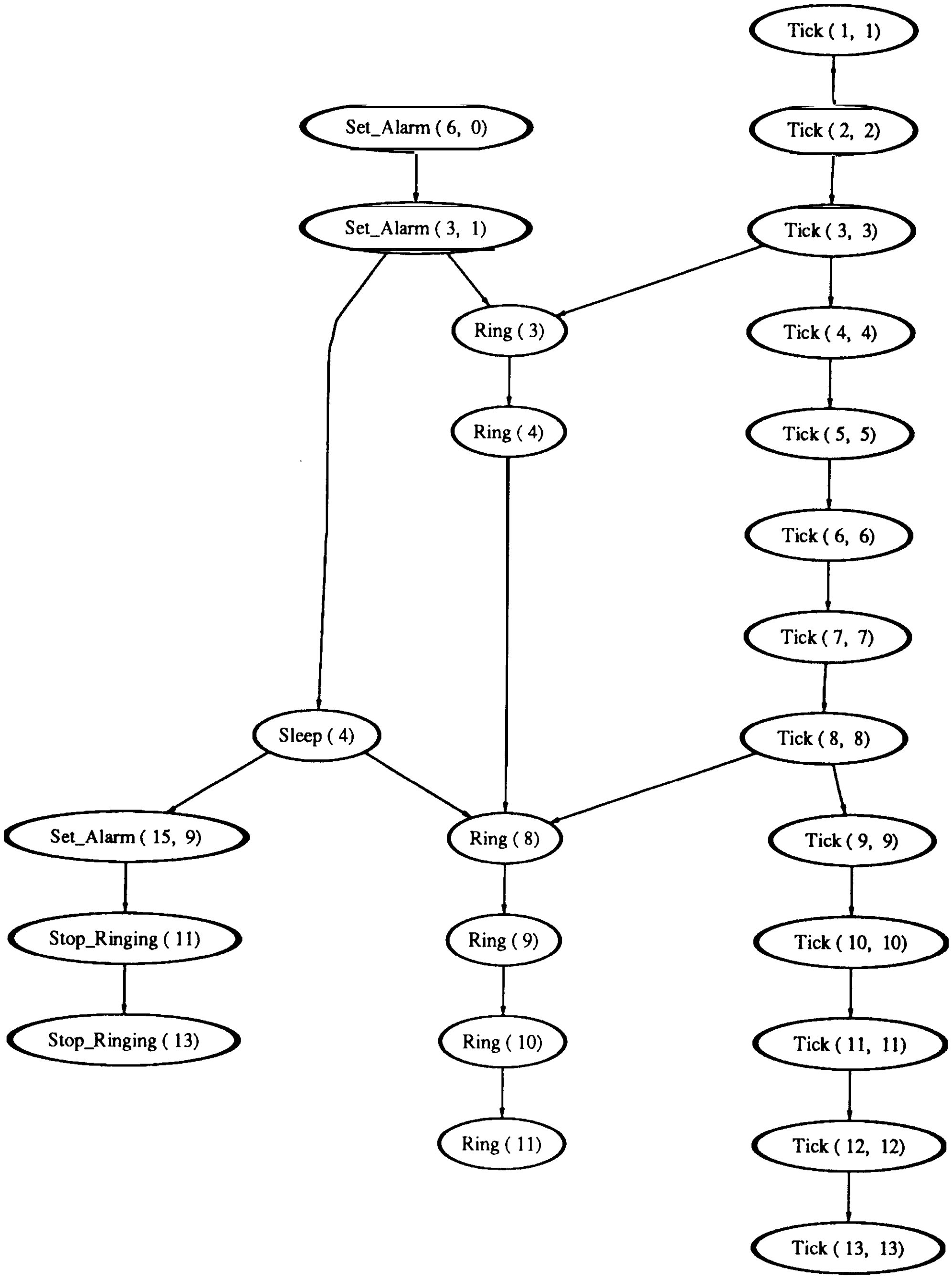

Figure B.2: partial **order for a sample run of the alarm clock example**

```
How much longer do I have to stay here? I still have
  4 dishes to wash!!

How many more dishes would you want the man to wash?
Enter number (negative number to release man) :- 2
How many more dishes would you want the man to wash?
Enter number (negative number to release man) :- 0

How much longer do I have to stay here? I still have
  5 dishes to wash!!

How many more dishes would you want the man to wash?
Enter number  negative number to release man)  - 0

How much longer do I have to stay here? I still have
  4 dishes to wash!!

How many more dishes would you want the man to wash?
Enter number (negative number to release man) :- 3

How much longer do I have to stay here? I still have
  3 dishes to wash!!


How much longer do I have to stay here? I still have
  5 dishes to wash!!

How many more dishes would you want the man to wash?
Enter number (negative number to release man) :- -1
>
```

B.6 Satellite Communication Link II

You should have the following files:

```
city.cpl
satellite.cpl
link-handler.cpl
Makefile
```

Compile the files by running make and then run the executable by running `link`. Use the inputs as in the example in Appendix D:

```
> link
Welcome! Please input originator, destination,
and content of your messages.
```

```
Send message from which city (0 to quit) -> 1
Send message to which city -> 3
Enter the message -> hi there
Send message from which city (0 to quit) -> 2
Send message to which city -> 1
Enter the message -> be there on time!
Send message from which city (0 to quit) -> 0
>
```

When the partial order graph is viewed you should get a graph not visibly different from figure B.3.

Compare this with the example in Appendix D which does not use connect statements. Notice in the first version the Receive-Message events are causally related to the Send-Message events whereas in the second version they are not but instead have a doubly-directly edge joining them.

B.7 Baking a Cake

You should have the following files:

```
cake. cpl
Makef ile
```

Compile the files by running make and then run the executable by running link. When run you will see something like:

```
> cake
What would you like to do next?
 1) buy the ingredients.
 2) put flour into 'dry' bowl.
 3) put sugar into 'dry' bowl.
 4) put baking soda into 'dry' bowl.
 5) put water into 'wet' bowl.
 6) put milk into 'wet' bowl.
 7) put eggs into 'wet' bowl.
 8) mix contents of 'dry' bowl.
 9) mix contents of 'wet' bowl.
 10) mix contents of 'dry' and 'wet' bowl together.
 11) put mix into baking tray and put cake into oven.
 12) remove cake from oven and eat it!
 13) Give up on cake.
 Enter option ->
```

You can now enter in the action you would like to perform. For example you could enter in "8" to mix the contents of the dry bowl. Of course this would generate an Inconsistent

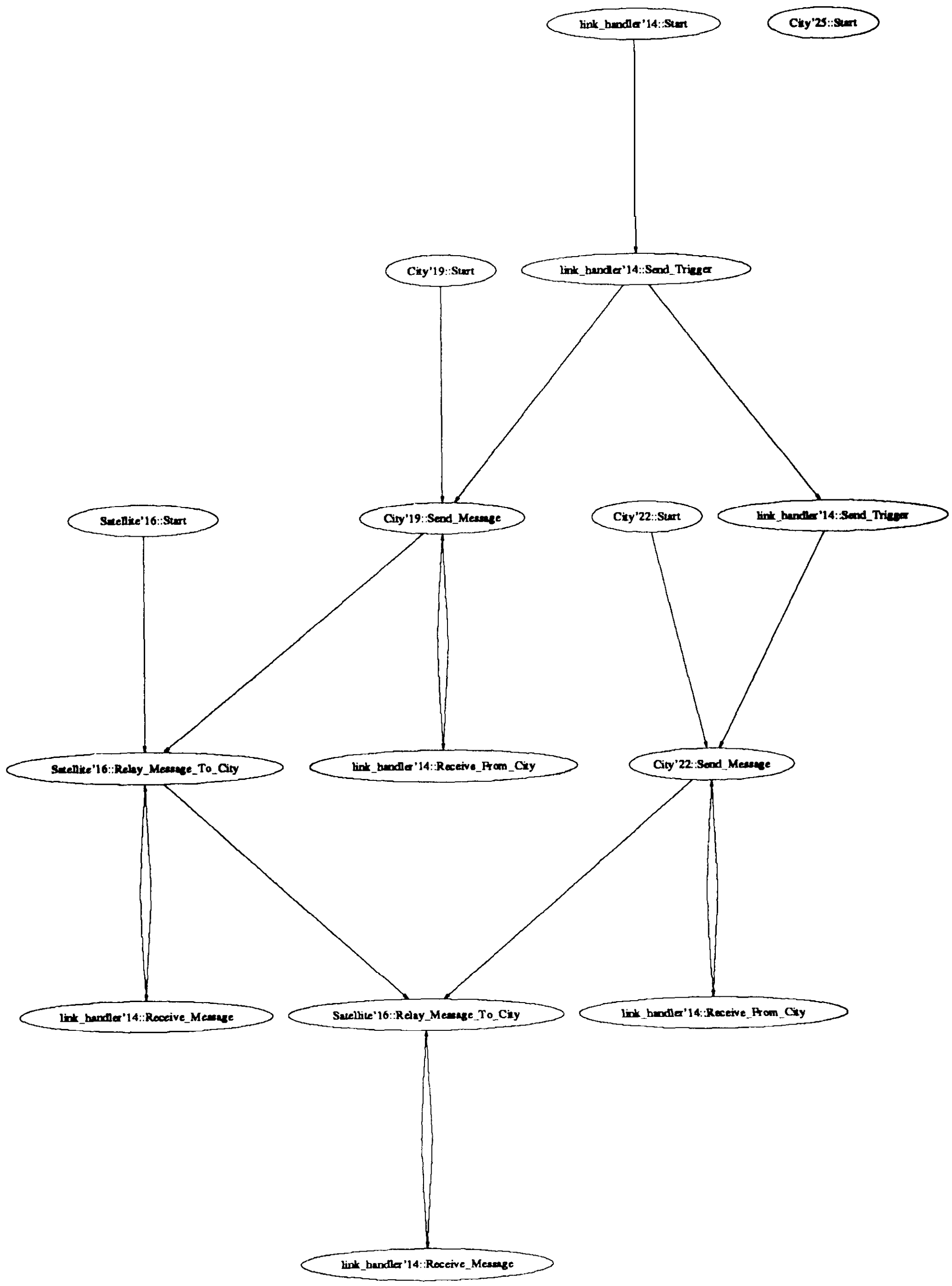

Figure B.3: **An example partial order for Satellite Communication Link II**

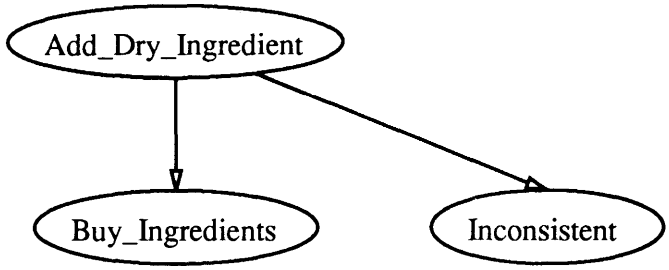

Figure B.4: partial order for adding flour before buying ingredients

event since you haven't even bought the ingredients yet. You can enter in as many actions as you like until you are finished (in which case enter in "13").

This example brings out an advantage of having a partial order over linear trace for a computation. Consider the following input to the program: 2, 1, 13. Here we are adding flour before buying the ingredients and we expect to violate a constraint. With the environment variable CPL-RTS set as follows

```
setenv CPL-RTS "binary-log echo-events"
```

we can get both a linear trace of the computation and also a partial order when the program is run. The partial order is shown in figure B.4. The (edited) linear trace is

```
Add-Dry-Ingredient -> Buy-Ingredients -> Inconsistent
```

Even though the computation was very small the partial order has given us the valuable information that the Inconsistent event was causally related to the Add_Dry_Ingredient event and not the Buy-ingredients event. The linear trace does not give us this information. It only tells us that the Inconsistent event came after both the Add_Dry_Ingredient event and the Buy-Ingredients event. For larger computations information like this from the partial order that cannot be gleaned immediately from a linear trace is extremely valuable for debugging and analyzing the prototype.

B.8 Library

You should have the following files:

```
library.cpl
library-handler.cpl
Makefile
```

Compile the files by running make and then run the executable by running library. When run you will see a list of options:

```
> library
 (1) Deposit new book
 (2) Take out a book
 (3) Return a book
 (4) Stop going to this library
What would you like to do next ->
```

The first option is for building up the resources of the library. The second and third are for borrowing and returning books. The fourth is quit the program. On all options except (4) you are prompted for the title of a book. If you wish to refer to a book that already exists you have to get the title exactly right (including capital letters).

Here is a sample run:

```
> library
 (1) Deposit new book
 (2) Take out a book
 (3) Return a book
 (4) Stop going to this library
What would you like to do next -> 1
 Enter title of book -> Gone with the Wind
Thank you for your contribution!
 (1) Deposit new book
 (2) Take out a book
 (3) Return a book
 (4) Stop going to this library
What would you like to do next -> 2
 Enter title of book -> War and Peace
Sorry, we don't have that book.
 (1) Deposit new book
 (2) Take out a book
 (3) Return a book
 (4) Stop going to this library
What would you like to do next -> 1
 Enter title of book -> Tom Sawyer
Thank you for your contribution!
 (1) Deposit new book
 (2) Take out a book
 (3) Return a book
 (4) Stop going to this library
What would you like to do next -> 2
 Enter title of book -> Gone with the Wind
You can have the book for five days!
 (1) Deposit new book
 (2) Take out a book
 (3) Return a book
 (4) Stop going to this library
What would you like to do next -> 3
 Enter title of book -> Gone with the Wind
```

```
Thanks you for returning the book.
 (1) Deposit new book
 (2) Take out a book
 (3) Return a book
 (4) Stop going to this library
What would you like to do next -> 4
```

Appendix C

Where to find the files

For those with access to Anna.stanford.edu here are where you can find the relevant files. All files are in subdirectories of /u10/alexh/cpl/cookbook.

Example	Subdirectory
Light Switch	switch
Satellite Communication Link	link/no-connect
Alarm Clock	alarm
Dish Washer	dishwash
Satellite Communication Link II	link/connect
Baking a Cake	cake
Library	library

This document is available as a postscript file and can be found on Anna.stanford.edu under:

/u10/alexh/cpl/cookbook/examples-doc/cookbook.ps

Appendix D

Partial Order Browser

D.1 Introduction

The partial order browser is a graphical RAPIDE-0.2 tool for examining the *partial order,* the partially ordered set of events generated in a RAPIDE-0.2 computations.

There is a Unix environment variable CPL-RTS (CPL Run Time System) which should be set correctly if you want to use the partial order browser. It should be set sometime before the executable file for the example is run (so you can set it after compilation if you need to):

```
setenv CPL-RTS "binary-log"
```

This tells the RTS to generate a binary log file a run time from which the partial order browser can read information about the computation. In any event, when the executable is run, the RTS will generate a text log of the events which contains a semi-readable form of the events generated.

Once CPL-RTS is set correctly and the executable has been run you can run the partial order browser with the pob command with the name of the executable as the argument. For example, in the Satellite Communication Link example you would type:

```
pob link
```

Using the Satellite Communication Link example we will give a hands-on tutorial for using the partial order browser.

D.2 Example from the Satellite Communication Link

This section will explain some of the features of the partial order browser using output from the Satellite Communication Link example. It will be nowhere near exhaustive and will leave much of the task of discovery to the user. This section will *not* discuss the meaning of the partial order diagrams but only the environment of the partial order browser.

D.2.1 Getting going

To coordinate ourselves you should run the Satellite Communication Link example with the following inputs:

```
> link
Welcome ! Please input originator, destination,
and content of your messages.

Send message from which city (0 to quit) -> 1
Send message to which city -> 3
Enter the message -> hi there
Send message from which city (0 to quit) -> 2
Send message to which city -> 1
Enter the message -> be there on time!
Send message from which city (0 to quit) -> 0
>
```

When the partial order browser is started up using pob link two windows will pop up on your screen. The large one has title "pob" and is the window in which the partial order diagram will be displayed. The smaller one has title "DAGd" and contains a number of toggle options. In accordance with the pob man-page we will call these windows the *graph* and *options* windows respectively. When the graph window has finished drawing you should see in it something remarkably similar to figure D.l.

In the diagram the nodes represent events and directed edges represent orderings. You can find the complete description of each node by clicking and holding any mouse button on the node you are interested in. Find one of the (two) nodes labeled **link_handler'14::Receive_Message** and click and hold a mouse key on it. You should see near the top left corner of the diagram box the following:

From	⇒	**link_handler'14**
To	⇒	City '19
Of-Action	⇒	Receive-Message
By	⇒	connect-satellite-to-city
Parameters	⇒	(2, "be there on time!", 0, 0)

This says that the event is Receive-Message, was generated by the design unit instance link-handler'14, observed by the design unit instance City'l9, inside the when-process labeled connect-satellite-to-city and had the parameters as above. A couple of things need a quick explanation. The numbers (e.g. "'14") tagged onto the end of the design unit instance names (e.g. link-handler'14) are associated with the numbering of Ada tasks when the RAPIDE-0.2 program is transformed to Ada. All design unit instances receive a unique number. If you have labeled everything you shouldn't have to worry about the numbers. The parameters listed above are, in respective order, the number of the city that sent the message (in this case number 2), the message that was sent, the departure time of the event from the issuing design unit instance and the arrival time of the event at the observer design unit instance.

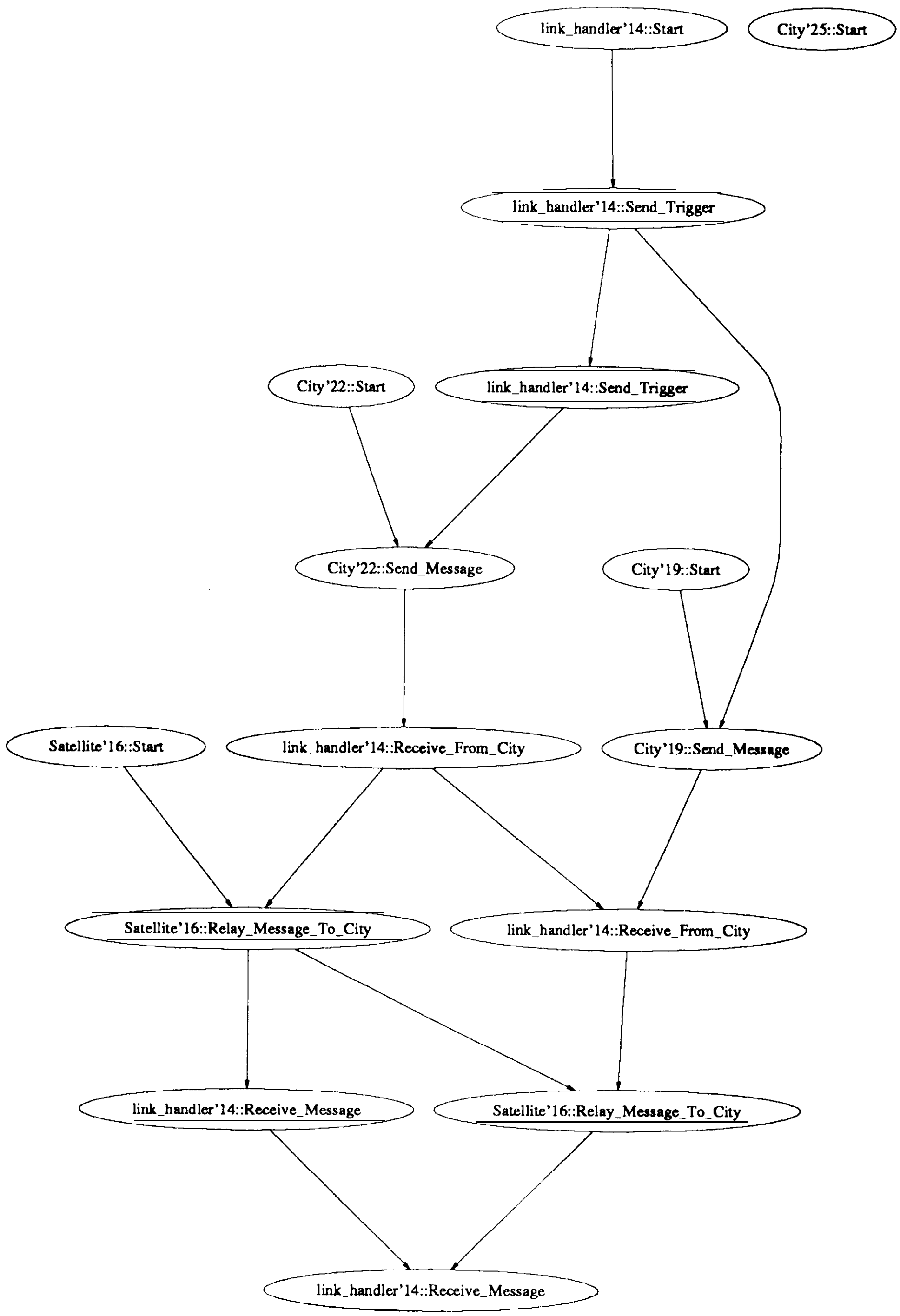

Figure D.l: Initial display

D.2.2 Menus

The description of events in the nodes is a subset of the ones listed above and you can control which one(s) you would like displayed using the "labels" menu at the top of the graph window. Click on the title "labels" and toggle which ones you wish to be displayed. After you have chosen your toggles you will need to select `redraw` from the control menu.

There are several other pull-down menus and here is a brief description of what they contain (see the partial order browser man-page for more details):

orderings: These toggle switches define which relations are to be used when determining orderings between events.

labels: These toggle switches determine what information is to be displayed in each event node.

style: These toggle switches determine how the graph is to be laid out and drawn in the graph window.

operations: These are a miscellaneous set of commands which allow you to highlight portions of the graph, make a rearrangement of it or ask about relations between two events.

subcomputation: These commands ask the partial order browser to display specific subgraphs of the full graph using the settings in the options window.

zoom: These commands allow you to zoom in and out on the graph.

print: Various commands for printing or saving a copy of the graph.

control: Commands for updating the graph window after toggles in the options window have been set (the graph is not automatically redrawn when a toggle is used), reading in binary log files and for quitting the partial order browser.

Try playing with the menus. If you get stuck just Quit the partial order browser and restart it. There is no change to the original binary log file when you are using the partial order browser so you can feel free to experiment.

One other feature that is available but not in the menus is for dragging the whole graph around the window. To do this click on the background of the graph window (i.e, not on a node) and drag the mouse button in the direction you want to move the graph.

D.2.3 The options window

Figure D.2 shows the options window for this example with some of the toggles off (empty circle) and some on (circle with dot). There are four toggles along the top row "action", "from dui" , "trigger" and "to dui" (dui is the acronym for design unit instance). These are used for "selecting" events for highlighting (operations menu) or displaying subgraphs of the full graph.

Then, standing alone, is the "all" toggle which is a quick way of toggling all the "computation elements" (see below).

Figure D.2: Options Window

The computation elements make up the remainder of the toggles. It consists of three sections (which are not clearly marked in the window).

The first section lists the events that were generated. In our example these are Start, Send_Trigger, Send-Message, Receive-From-City, Relay-Message-To-City and Receive-Message.

The second lists the design unit instances and those when-processes within the duis that were actually triggered in the computation. In our example these happen to be the rest of the toggles. Each sub-list for a design unit instance starts with a toggle for the design unit instance (e.g City'19), followed by a toggle for the (implicit) when-process that generates special events for that design unit instance like Start (e.g 20), followed by toggles for each when-process in the design unit instance that was triggered in the computation. Notice that City'25 did not send any message and so does not have it's when-process Send-Messages listed.

The third lists those duis which generate events but did not do so in the computation (none in this example).

D.2.4 Time for some action

Let's do an example (you may want to quit and restart the partial order browser here). In more complicated examples we would be interested in only seeing part of the computation. Let's say that we want to see all the events that the design unit link-handler generated. The strategy is to use appropriate toggles in the options window and then use the subcomputation menu to display our desired graph.

Which toggles should be on and which off in the options window? First turn off all the toggles by clicking each of the top row and then clicking on the "all" toggle. Now we turn only the ones we want back on. In the top row we turn on "action" and "from dui" because we are interested in events and those that come *from* a particular design unit instance. The possible actions that link-handler can generate are Start, Receive-From-City and Receive-Message so click on these. Then to select link-handler as the design unit instance we want to examine just click on link-handler'14. This toggle represents the whole design unit instance so all the when-process toggles belonging to link-handler'14 will go on too. Now go to the subcomputation menu and select "from options, full". You should see something resembling figure D.3.

Suppose now we wanted to check that all the Receive-Message events are there as we expect (we are expecting two of them). There are various ways we can do this but let's choose to do this by highlighting the Receive-Message events observed by the City design unit instances. First clear all the options. Then select "action" and "to dui". Then select Receive-Message. Also select City'19, City'22, and City'25. Now go to the operations menu and select "highlight options". This will highlight the selection you have made from the options window. Two events should be highlighted.

For further inspiration it is recommended that you consult the partial order browser man-page.

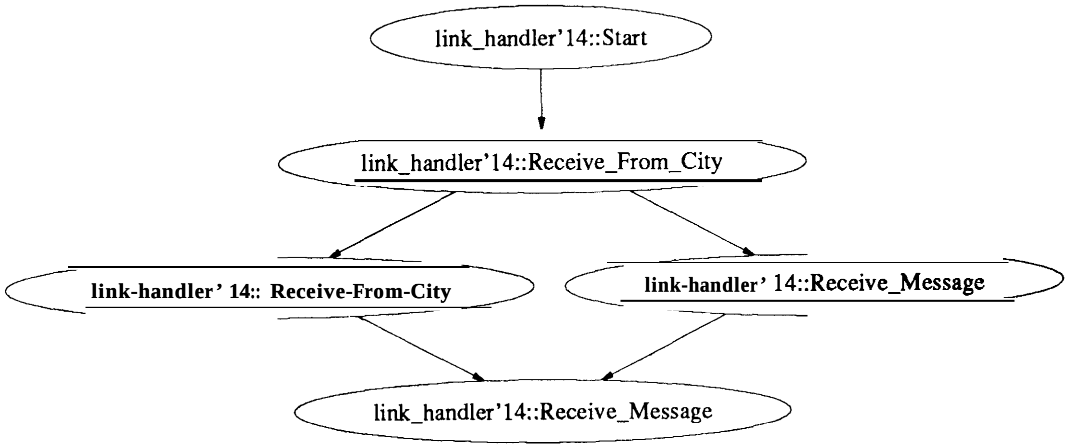

Figure D.3: **Subcomputation**

Appendix E

Illustrated Run-time System (IRS)

The IRS is a run-time tool built for examining and testing the run-time behavior of the RAPIDE-0.2 prototypes.

Figure E.l gives an example of how the IRS might look on the the screen. Boxes represent design unit instances and edges represent parent-child relationships between design unit instances. Small dark squares either along an edge or inside a design unit instance are CPL events.

Every design unit instance can receive events from itself and any parent or child that it has. The incoming events are queued in *dui (design unit instance) queues*. When an event is generated it is enqueued into the appropriate dui queue. The destination design unit instance dequeues it later. The basic operation in the IRS is to *step* through the computation. The basic unit of a step is the enqueuing or dequeuing of an event. By stepping through a computation the user can watch it develop at his own pace. Running the partial order browser concurrently with the IRS is a powerful way of analyzing the computation.

The IRS also allows scheduling decisions to be made. For example events can be re-ordered within a dui queue (if the reordering does not violate the orderly observation principle[1]). New events can be inserted into dui queues. This is a flexible method for testing the behavior of a single component of the prototype.

More detailed information about the IRS can be found in the IRS man-page.

[1] The orderly observation principle says that events that are ordered must be observed in that order

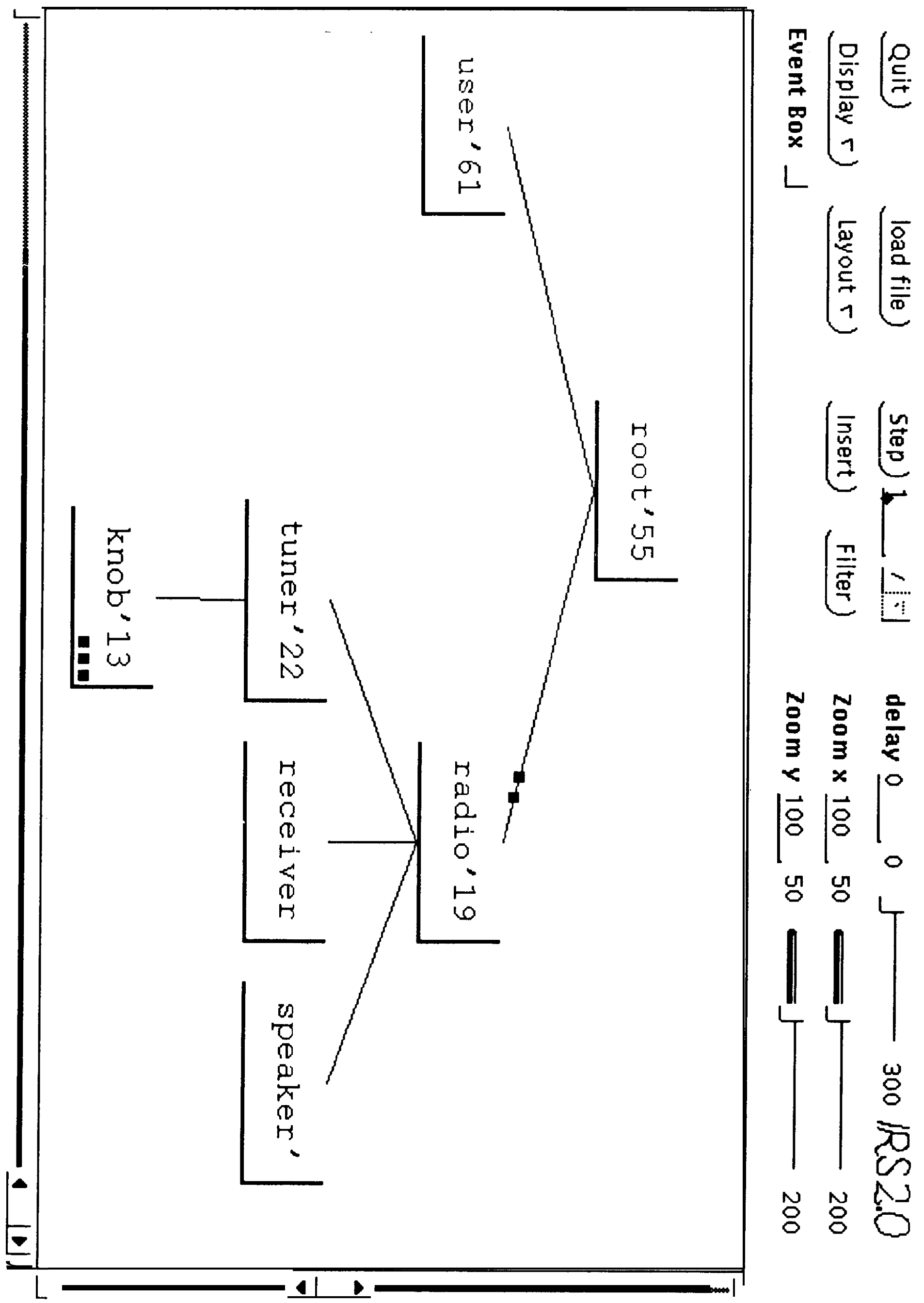

Figure E.1: Sample display of IRS

Bibliography

[Ada83] US Department of Defense, US Government Printing Office. The *Ada Programming Language Reference Manual,* February 1983. ANSI/MIL-STD-1815A-1983.

[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.

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

[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.

[MKS92] W. Mann, J. Kenney, and S. Sankar. Prototype semantic analyzers for rapidly changing languages. To appear as a Stanford University Technical Report, 1992.