DESIGNING RELIABLE PROGRAMS WITH RAPIDE

Neel Madhav*
Research Associate
Computer Systems Lab
Gates Computer Science Bldg., 4A
Stanford University
Stanford, CA 94305-9040
madhav@cs.stanford.edu

David C. Luckham*
Professor
Computer Systems Lab
Gates Computer Science Bldg., 4A
Stanford University
Stanford, CA 94305-9040
dcl@anna.stanford.edu

Key words: Rapide, prototyping, software architecture, formal constraints, partial orders of events, analysis tools.

Abstract

Rapide is a language for prototyping large, distributed systems. Rapide allows the scale design of a system to be constructed and analyzed before resources are applied to the construction of the actual system.

Two important facets of designing reliable systems are (1) system architecture — the components in the system and the communication paths between the components, and (2) system behavior — the requirements on the components and the communication. Rapide facilitates the design of system architecture and behavior by (1) providing language features to realize system designs, (2) providing an expressive model for capturing the execution behavior of systems, and (3) providing techniques and tools for analyzing system execution behavior.

This paper introduces the essential concepts of Rapide and gives an example of system design using Rapide.

Rapide has 4 sublanguages — (1) a type language, (2) an architecture definition language, (3) a constraint language and (4) an executable language. The paper introduces the Rapide architecture sublanguage and the Rapide constraint sublanguage.

The Rapide model of system execution is a set of significant events partially ordered by causality (also called posets). This paper discusses Rapide execution models and compares them with totally ordered event based models.

Rapide provides tools to check constraints on posets to browse posets and to animate events on a system architecture. This paper briefly discusses the Rapide analysis tools.

1 Introduction

This paper introduces the essential features of Rapide [4, 6, 3] for designing system architecture and behavior. The paper presents language features of Rapide, the Rapide execution model and briefly introduces tools for analyzing system execution behaviors.

The architecture of a system expresses the structural aspects of the system — the modules, the communication patterns, the sharing of data and the composition structure of the modules. The design of the architecture of a complex system is crucial for the reliability of the system and the possibility of its reuse [1, 10, 4, 6, 7, 5].

The constraints on the behavior of the components of a system determine the design of that component. Formal constraints thus play an important role in system design.

In general, a system architecture is hierarchical, each component has an architecture of subcomponents. However, the hierarchical nature of architectures is not discussed further in this conference paper.

Rapide facilitates the design of system architecture and behavior by (1) providing language features to realize system designs, (2) defining an expressive model for capturing system architecture and behavior and (3) providing techniques and tools for analyzing system models.

Rapide reference architectures have modules and connections between those modules. Each module in Rapide is described through an interface. Interfaces are types of modules. The interface of a module lists ways in which the module may interact with other modules. In addition, interfaces and architectures may have formal constraints on the behavior of modules and architectures.

Rapide architectures and models are event-based. System behavior and architectural connections are modeled through partial orders of events, also called *posets*. The partial ordering represents causal relations between events. If an event B is partially ordered with respect to an event A, event A is said to have *caused* event B. The causal relationships between events are inferred from the structure of Rapide programs and are not discussed further in this paper.

The difference between execution models that are partial orders of events compared to total orders of events is the following — if an event A precedes an event B in a total order, there is no guarantee that A preceded B (or in our terminology, caused B[1]). However, if A caused B, then A is guaranteed to have occurred before B. A single partial order model of system execution thus provided strictly more information about the relationship between events than does a total order model.

Once a system is designed, it is executed to produce a poset of events. This poset is a model of the execution behavior of the actual system. Rapide provides tools to analyze poset models of system execution to get early feedback on the behavior of the actual system.

The Rapide constraint language provides constructs for high-level specifications of modules. The language allows or forbids the occurrence of *patterns* of events in an execution model. If a constraint is violated by the execution

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

[1] Intuitively, an event A caused event B if B could not have occurred if A did not occur. The actual definition of causality may vary with programming environments and languages

of a Rapide system, the design of the Rapide system is changed to adhere to the constraint. Thus, early feedback and redesign of systems is facilitated by Rapide. Unreliable features in the design and behavior of the system are thus detected and removed at an early stage of the design of programs.

There are a number of analysis tools available to examine posets (and more analysis tools are planned). One of the analysis tools is a constraint checker that ensures that posets adhere to given constraints. Another tool is a poset browser called the Partial Order Viewer and yet another tool is an animator that shows the movement of events between different modules called Raptor (Rapide Animator).

Section 2 presents language features in Rapide that support design of system architectures and formal constraints. Section 3 presents the Rapide poset execution model for analyzing system architecture and behavior. Section 4 presents Rapide constraints. Section 5 briefly introduces analysis tools.

2 Rapide Architectures

Rapide architectures are made up of components and Rapide has constructs for specifying connections between components.

2.1 Modules and Interfaces

The components of a Rapide architecture are called *modules*. Every *module* has an interface that lists the ways in which the module *may* interact with other modules. The *constituents* of an interface are names and types of *objects* and *actions* that the module may use to interact with other modules.

The example below is the interface of a Pump module :

```
type Pump is interface
      in action Set_Limit(Amt:Integer),
      Turn_On_Pump(), Handle_Up(),
      Handle_Down(), Turn_Off_Pump();
      out action Report_Charge(Amt:Integer);
constraint -- Constraints on the pump actions.
end interface;
```

We declare the type Pump to be an interface. The interface lists the objects provided and required by modules of type Pump and the **in** and **out** actions of modules of type Pump. The *actions* are *kinds* of activity of interest that may occur in a module of type Pump. The occurrence of an action is called an *event*. The **in** actions of a module are those that are generated by the environment[2] of a module that the module may *observe*. The **out** actions of a module are those generated by the module that the environment of the module may observe. A Pump module (module of type Pump) may observe and react to events Set_Limit, Turn_On_Pump and Turn_Off_Pump. A Pump module may generate Report_Charge events. The specification of how the events generated by a module may be observed by other modules is given through an architecture.

The following is the syntax of interfaces.

```
type_declaration ::=
  type identifier is interface_expression ';'
interface_expression ::=
  interface { interface_constituent }
  end [ interface ]
interface_constituent ::=
  provides { object_name_declaration }
| requires { object_name_declaration }
| in action { action_declaration }
| out action { action_declaration }
| constraint constraint_list
```

[2]The environment of a module M are all other modules in the system except M.

An interface is the *type* of modules. An interface I may be viewed as putting constraints on the structure and behavior of modules of type I. The *provides* section lists the names and types of objects a module of type I makes available to its environment. The *requires* section lists the names and types of objects a module of type I expects from the environment. *Events* are defined as the occurrence of interesting activity in a system. *In actions* define the kinds of events a module of type I may observe from its environment. *Out actions* define the kinds of events a module of type I may generate (and other modules may observe). *Constraints* specify the arrangements of events that may occur in a module of type I. Constraints are discussed in Section 4.

The interfaces of customer modules and the operator module are the following :

```
type Customer;
type Operator is interface
      in action Get_Money(Amt:Integer;P:Pump;
                          C:Customer),
                Pump_Result(P:Pump;Amt:Integer),
                Change_Request(P:Pump;C:Customer);
      out action Activate_Pump(P:Pump;Amt:Integer);
                 Give_Change(C:Customer;P:Pump;
                             Amt:Integer);
constraint
      -- Constraints on the operator actions.
end interface;
type Customer is interface
      in action Get_Change(P:Pump;Amt:Integer);
      out action Prepay(Amt:Integer;P:Pump),
      Start_Pump(P:Pump), Turn_Gas_On(P:Pump),
      Turn_Gas_Off(P:Pump), Stop_Pump(P:Pump),
      Ask_For_Change(P:Pump);
constraint -- Constraints on the customer actions.
end interface;
```

2.2 Architectures

An *architecture* is a list of declarations of modules, a list of connections between modules and a list of constraints.

The following example of an architecture is an Automated Gas Station.

```
architecture Automated_Gas_Station is
   N_Pumps : Integer is 2;
   N_Customers : Integer is 5;
   P : array [Integer] of Pump is (1 .. N_Pumps);
   C : array [Integer] of Customer is
         (1 .. N_Customers);
   O : Operator;
   ?C : Customer; -- Placeholders used to connect
   ?P : Pump;     -- actions.
```

```
   ?Amt : Integer;
connect
   ?C.Turn_Gas_On(?P) to ?P.Handle_Up();;
   ?C.Turn_Gas_Off(?P) to ?P.Handle_Down();;
   ?C.Prepay(?Amt,?P) to O.Get_Money(?Amt,?P,?C);;
   O.Activate_Pump(?P,?Amt) to ?P.Set_Limit(?Amt);;
   ?C.Start_Pump() to ?P.Turn_On_Pump();;
   ?C.Stop_Pump()to ?P.Turn_Off_Pump();;
   ?P.Report_Charge(?Amt) to O.Pump_Result(?P,?Amt);;
   ?C.Ask_For_Change(?P) to O.Change_Request(?P,?C);;
   O.Give_Change(?C,?P,?Amt) to
      ?C.Get_Change(?P,?Amt);;
constraints
   -- Constraints on connections
   --and events in the automated gas station.
end Automated_Gas_Station;
```

The architecture definition has 3 sections — declarations, connections and constraints. The architecture declares components O, P and C. O is a component of type Operator, P is an array of components of type Pump and C is an array of components of type Customer. The declaration section also has declarations of some variables and some *placeholders*. Placeholders are special variables that may be rebound each time a construct they are part of is executed. For example, each connection has its own copy of the placeholders ?P, ?C and ?Amt. However, within each connection, different occurrences of the same placeholder must have the same value.

The connections express the communication between modules. Placeholders are used to express multiple connections through a single connection statement. For example, the third connection statement connects up the Prepay action of Pump modules to the Get_Money action of O.

Figure 1 shows the gas station architecture.

The following is the syntax of architectures.

```
architecture ::=
  architecture identifier is
    declarations
    connections { connection }
    [ constraints constraints ]
  end identifier;
connection ::=
  pattern to pattern ';;'
| other pattern connections
```

Patterns are CSP [2] like expressions and are described in Section 2.3. Connections connect up patterns of actions to other patterns of actions.

2.3 Pattern Language

Rapide connections connect patterns to patterns. Event patterns define subsets of event computations.

The syntax of patterns is :

```
pattern ::=
  basic_pattern
| pattern binop pattern
| { placeholder_declaration } pattern
| '(' pattern ')' |
| empty
| any
|   other patterns
binop ::=
  '->'
| and
| or
| '~'
| other binary operators
basic pattern ::=
  module_name '.' action_name
    '(' parameters ')'
```

Basic patterns *match* any event in the event computation that is an instantiation of the basic pattern. P1 -> P2 is the sequence pattern (match of P1 must precede match for P2), P1 **and** P2 is a match for both P1 and P2, P1 **or** P2 is a match for P1 or P2 and P1 **or** P2 is a match for P1 and along with a match for P2 such that they match distinct sets of events. Rapide has a rich pattern language, with other operators and patterns. We do not describe the full pattern language in this paper.

3 Rapide Execution Model

The execution behavior of a Rapide system is modeled as a partial order of events. The partial ordering represents causal ordering. By contrast other event based simulation systems such as VHDL have executions that have total orders of events that do not encode causal relationships between events.

The causal ordering between events is inferred from the structure of the Rapide program. There are 4 simple rules for inferring causality. These rules are not given in this paper in the interests of brevity.

An example of a poset is shown in Figure 2. The event O.Get_Money(5,1,C1) denotes the operator O receiving 5 units of money from customer C1 for pump 1. The event O.Get_Money(10,1,C2) denotes the operator O receiving 10 units of money from customer C2 for pump 1. The event C2.Start_Pump(1) denotes customer C2 starting pump 1. The event C1.Start_Pump(1) denotes customer C1 starting pump 1. The arrows denote causal relationships. Event O.Get_Money(5,1,C1) precedes O.Get_Money(10,1,C2) in the causal ordering. Event O.Get_Money(5,1,C1) precedes event C2.Start_Pump(1), O.Get_Money(10,1,C2) precedes C1.Start_Pump(1) and C2.Start_Pump(1) precedes C1.Start_Pump(1). Since partial orders are transitive, O.Get_Money(5,1,C1) also precedes C1.Start_Pump(1). However C2.Start_Pump(1) is independent with respect to O.Get_Money(10,1,C2).

The causal relationships and independence between events provide more information than a total ordering.

4 Rapide Constraints

Rapide constraints constrain the behavior of modules and communication between modules through constraining posets. This section introduces Rapide constraints through an example constraint on the gas station.

The Rapide constraint language constrains posets of events by specifying patterns of events that must not occur in a poset (or must occur in a poset).

The following is an example of a constraint on the gas station architecture :

```
never (O.Get_Money(?A,?P,?C1) ->
```

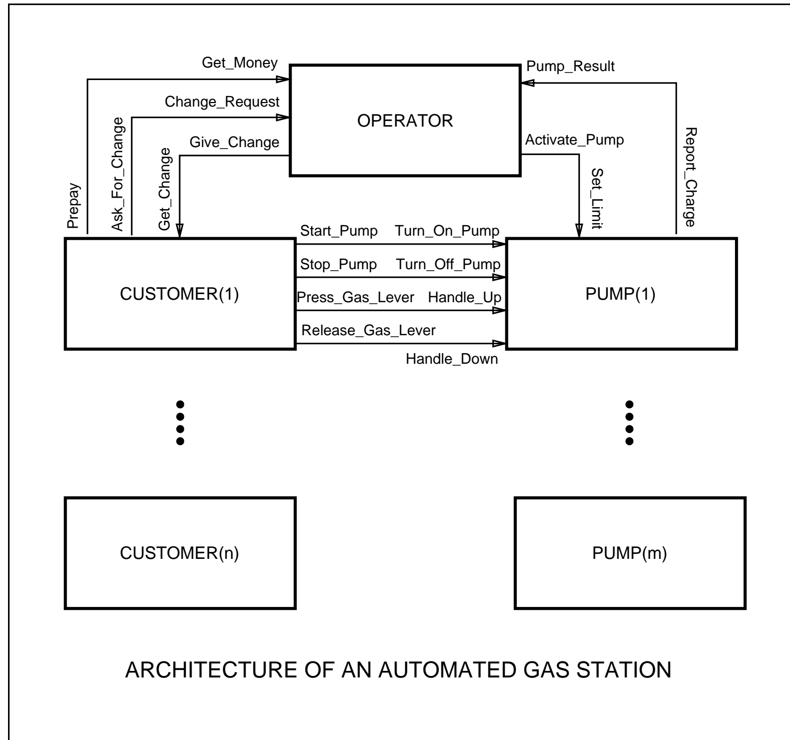

Figure 1: Pictorial Representation of a Gas Station Architecture

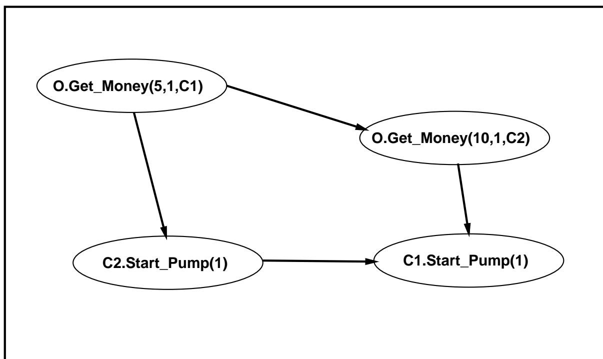

Figure 2: A Poset Representing Part of a Gas Station Execution

O.Get_Money(?A,?P,?C2)) ~
(?C2.Start_Pump(?P) –>
?C1.Start_Pump(?P));

The constraint says that it should never happen that the operator accepts money from a customer, ?C1, for a pump and then accept money from another (different) customer, ?C2, for the same pump and then the customers pump gas in an order opposite of the order in which they paid.

The ~ operator says that its two pattern arguments must match distinct sections of the poset. The ~ operator puts no further restrictions on the relation between its arguments.

If the constraint is applied to Figure 2, we see that the pattern
(O.Get_Money(?A,?P,?C1) –>
O.Get_Money(?A,?P,?C2))
matches the two Get_Money events in the poset and the pattern
(?C2.Start_Pump(?P) –>
?C1.Start_Pump(?P))
matches the two Start_Pump events in the poset. The two matches are distinct, therefore the pattern
O.Get_Money(?A,?P,?C1) –>
O.Get_Money(?A,?P,?C2)) ~
(?C2.Start_Pump(?P) –>
?C1.Start_Pump(?P))
matches the complete poset. The constraint is violated since it is a never constraint.

The constraint is a classic race condition constraint and is fairly simple to express in Rapide. The constraint makes

a number of assumptions including the assumption that the same customer does not appear twice to pump gas. We do not discuss these assumptions in the interests of brevity.

The syntax of a subset Rapide constraint language is:

constraint ::=
 observe *filter* *constraint_body* **end**
| *constraint_body*
filter ::=
 pattern
constraint_body ::=
 [**not**] **match** *pattern* ';'
| **never** *pattern* ';'

A constraint has a filter (the filter may be omitted) and a constraint body.

The filter is a pattern used to restrict the poset that the constraint applies to, to smaller size. Any poset is first filtered by finding all possible matches for the filter pattern in the computation. The result is called the filtered computation.

If the constraint body is **match** pattern, the constraint body pattern must match the filtered computation exactly (no left over events). If the constraint body is **not match** pattern, the constraint body pattern must not match the filtered computation exactly. If the constraint body is **never** pattern, the constraint body pattern must never match the filtered computation — there must be no subcomputation of the filtered computation that matches.

Rapide constraints and filters are more expressive and complex than presented in this paper. We present a simple version of Rapide constraints in the interests of brevity.

5 Rapide Analysis Tools

There are a number of Rapide analysis tools to examine and analyze posets and more tools are planned.

The three tools we briefly discuss are a poset browser called the *Partial Order Viewer* (POV), the *Rapide Animator* (Raptor) and the *Rapide Constraint Checker*.

The Partial Order Viewer allows posets to be viewed browsed and reduced in size. It allows users to check for surprises in their programs. One example of the use of the Partial Order Viewer on a small poset is a picture very close to the one in Figure 2. Posets may be reduced in size through pattern matching or choosing the kinds of events that should be displayed through an event picker.

The Rapide Animator [11] allows posets to be animated on an architecture picture. It allows users to visualize the interactions in their programs. One example of the use of the Rapide Animator on an architecture and a small poset is a picture very close to Figure 1 with events in Figure 2 animated on the architecture picture. When events are animated they travel from the "module of origin" to "destination modules".

The Rapide Constraint Checker checks posets for given constraints and reports violations. These violations may be reported when the program has finished running or the user may get reports of violations as the Rapide program executes.

6 Conclusion

We have dealt with two important facets of designing reliable systems (1) system architecture — the components in the system and the communication paths between the components, and (2) system behavior — the requirements on the components and the communication.

We have presented Rapide facilities for designing reliable programs: (1) language features to realize system designs, (2) an expressive model for capturing the execution behavior of systems, and (3) tools for analyzing system execution behavior.

Rapide is used in a number of other ways including architecture conformance checking — whether a system conforms to a Rapide reference architecture [8] and runtime monitoring of systems.

Rapide has been used to design a number of complex systems and to tackle issues of reliability in those systems [9].

Projects are underway to instrument Java programs to produce events (instead of Rapide) and then to use Rapide techniques and tools to analyze and constraint check Java programs.

References

[1] D. Garlan and M. Shaw. *An Introduction to Software Architecture*, volume I. World Scientific Publishing Company, 1993.

[2] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[3] David C. Luckham. A language and toolset for simulation of distributed systems by partial orderings of events. In *DIMACS Partial Order Method Workshop IV, Princeton University*, 1996.

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

[5] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, September 1995.

[6] David C. Luckham, James Vera, Doug Bryan, Larry Augustin, and Frank Belz. Partial orderings of event sets and their application to prototyping concurrent, timed systems. *Journal of Systems and Software*, 21(3):253–265, June 1993.

[7] David C. Luckham, James Vera, and Sigurd Meldal. Three concepts of system architecture. To be published, 1996.

[8] Neel Madhav. Testing Ada 95 Programs for Conformance to Rapide Architectures. In *Ada-Europe, Montreux, Switzerland*, 1996.

[9] Alexandre Santoro and Woosang Park. SPARC-V9 architecture specification with Rapide. to appear, Stanford CSL Technical Report, 1995.

[10] Mary Shaw and David Garlan. Characteristics of higher-level languages for software architecture. Technical Report CMU-CS-94-210, CMU, December 1994.

[11] James Vera, Alex Santoro, and Moataz Mohamed. *Raptor - The Rapide Animator*. Stanford University, Computer Systems Lab ERL 456, Stanford, CA 94305-4055, 1994.