μRapide: An Executable Architecture Definition Language

David C. Luckham   James Vera [*]

April 7, 1993

Abstract

μRapide *is an executable architecture definition language intended primarily for defining architectures of time-sensitive concurrent and distributed systems, both hardware and software. Its features and semantics are based on distributed event processing. It is a small language providing only four major features: event patterns, interfaces, architectures, and mappings. It is a subset of the* RAPIDE-*1 prototyping language which contains additional object-oriented features and reactive programming constructs. μRapide can be used by itself to define architectures, or in conjunction with* RAPIDE-*1 to build detailed prototypes, or as an event processing ether for interconnecting application programs written in other programming languages.*

1 μRapide

μRapide is an executable architecture definition language (ADL for short). It is equivalent to a subset of the RAPIDE-1 language features for defining system architecture. It is intended primarily for defining architectures of concurrent and distributed systems.

Because of the emphasis on distributed systems and the need to express asynchronous communication, μRapide is an event processing language. Events are simply tuples of information such as who generated the event, what activity was done, data values, the time and duration, etc. The semantics and implementations of μRapide are defined in terms of event processing — generating events, sending events from one place to another, receiving events and deciphering events. This is the most general paradigm for modelling communication between components that are computing independently. Synchronous communication models such as function call or Ada task rendezvous can be modelled in terms of event processing.

μRapide event processing includes capabilities to detect and generate sets of events with causal dependencies and timing between events. μRapide executions (or computations) are made up of events that are related by *dependency* (also called *causality*) and *time.* [1] These relations between events are partial orderings and the executions are called *posets* — partially ordered sets of events.

There are four main features in μRapide for defining architectures: *event patterns, interfaces, architectures* and *mappings.*

- *Event patterns are the basic means of describing posets.*
  They are used in the other constructs to define behaviors, constraints, and mappings.

- *Interfaces are types whose members are objects called components.*
  An interface gives an external view of the behavior of a type of component. It defines how components of its type react to events by changing state and generating new events. An interface can define small static objects like integers or big dynamic objects like airplanes.

- *Architectures define the flow of events between interfaces.*
  An architecture defines how components communicate by means of events.

[*]This project is funded by DARPA under ONR contract N00014-90-J-1232 and AFOSR under Grant AFOSR91-0354

[1]In RAPIDE-1 a system may contain several clocks, each defining a timing relation on some of the events, but in μRapide there is only a single global clock.

- *Mappings define how architectures are related.*

Two of these features, interfaces and architectures, define dependencies between events. An interface, for example, can define how events received by a type of component cause other events to be generated. An architecture can define how events generated by some components cause other events to be received by (the same or other) components.

For expository purposes μRapide presented here is a subset of the RAPIDE-1 language framework [Tea93]. We omit discussion of the full type system with object-oriented derivation of interface types, the reactive programming language constructs, and the constraint language. Instead, we assume without description a small Pascal-ish basis of types, variables, functions, and expressions.

Discussion: Previous concurrent event-based simulation languages, such as VHDL [VHD87], LOTOS [BB89] and Esterel [BCG87] have execution models that are linear traces of events, arranged in increasing time order. Traces do not encode causality between events. Practical algorithms for generating poset models of distributed computations have been developed only recently [MSV91] even though the idea of using posets is quite old. In the new RAPIDE-1 prototyping language [BL90, LVB+93] event-based models of distributed systems, when executed, generate posets. This provides maximal information about the behavior of prototypes, which is the main purpose of prototyping. Various previous event-based languages are equivalent to subsets of RAPIDE-1. Also, special subsets of RAPIDE-1 have applications beyond prototyping. For example a subset that processes events without dependencies is appropriate for composing application programs written in different languages by implementing event communication between them.

2 Event Patterns

Events are the members of action types, and, conversely, an *action* is a type of events. An action is defined by an action name, a, and a finite list of types, $< t_1, t_2, \ldots t_n >$ called the *signature* of a. An *event* of action a is a tuple of information with a unique event identifier. The event contains the action name, a, data objects of the signature types, and certain other information such as the component that generated it, or the component that is the destination of the event, the event's timestamps and dependency history. We use a tuple notation, $a(v_1, \ldots, v_n)$, for events. An event's identifier is not part of the tuple notation. Distinct events may have the same tuple.

Event patterns (or simply, *patterns*) are expressions that define sets of events and their dependency and timing relationships (i.e., posets).

Syntax

```
pattern ::= basic_pattern
     | pattern binary_pattern_operator pattern
     | pattern '^' '(' iterator_operator
          binary_pattern_operator ')'
     -- which we print as
     -- pattern^{iterator_operator binary_pattern_operator}
     | pattern where guard
     | '(' placeholder_list ':'  pattern ')'
     | '(' pattern ')'
     pattern during '(' expression ','
          expression ')'

binary_pattern_operator ::= '->' -- printed as →
     | '||' | and
     | or | '~'

iterator_operator ::= '*' | '+'
     | integer_expression

guard ::= boolean_expression

basic_pattern ::= [performer_part '.']
     action_part ['(' parameter_association ')']

performer_part ::= component_expression

action_part ::= action_name
     | type_expression '::'action_name
```

Semantics

μRapide uses two special kinds of variable. The first is called a *placeholder*. Placeholder names always begin with a "?" to distinguish them from variable names which always begin with an alphabetic character. Placeholders are typed and declared the same way as ordinary variables, and they can be used similarly to build expressions. However, they differ from ordinary variables in that they can only be bound to an object as a result of pattern matching (below).

The other kind of special variable is an *iterator*. An iterator is a universal quantifier over a type. Iterators are typed and declared the same way as ordinary variables. They differ from ordinary variables in that they may not be assigned values. Iterator names always begin with a "!" which distinguishes them from ordinary variables and placeholders. A *statement* containing an iterator is equivalent to a list of statements separated by semi-colons, where each member of the list is an instance of the statement with the iterator replaced by an object of its type. A *pattern* containing an iterator is equivalent to either the conjunction (**and**) of, or the list (separated by ",") of, instances of the pattern with the iterator replaced by each object of its type. The conjunction or list meaning applies when the pattern containing "!" is the left or right side (respectively) of a basic connection rule (see Section 4).

The process of deciding if a given poset matches a pattern is called *pattern matching*. This can be defined formally as a recursive function taking as arguments a pattern and a finite poset. That is, given a pattern $PatExp$ and a finite poset Pos, there is a recursively defined algorithm for finding a type-correct binding $\mathcal{B}$ of each placeholder to an object such that $PatExp \mid \mathcal{B}$ matches Pos, according to the clauses below; $PatExp \mid \mathcal{B}$ is the instance of $PatExp$ that results when each placeholder is replaced at all of its occurrences by its binding in $\mathcal{B}$.

If $PatExp$ is a basic pattern, then $Pos = PatExp \mid \mathcal{B}$.

If $PatExp$ is a composite event pattern, built up from event patterns, P, P', and a pattern operation, then Pos matches it if Pos is the union of the matches for P and P' under the same binding of placeholders, $\mathcal{B}$, as follows:

1. *dependent:* $P \rightarrow P'$. Pos is a match if all of the events in the match for P' depend on all of the events in the match for P.

2. *both:* P **and** P'. Pos is a match if there are matches for patterns P and P'.

3. *distinct:* $P \sim P'$. Pos is a match if all of the events which matched P are distinct from the events which matched P'.

4. *either:* P **or** P'. Pos is a match if there is a match of pattern P or a match of pattern P' (in this case a match for both patterns is not needed).

5. *independent:* $P \mid\mid P'$. Pos is a match if none of the events which matched P are dependent on any of the events which matched P' and vice-versa.

6. *after:* $P < P'$. Pos is a match if all of the events which matched P have timestamps (according to the μRapide clock) less than those of all of the events which matched P'.

7. *iteration:* $P^{exp\ op}$ is shorthand for the pattern built up from exp copies of P and the operation *op* where exp is one of $*$ (zero or more), $+$ (one or more), or n (exactly integer n copies), and *op* is any of the preceding binary operations.

8. *restriction:* P **where** E. Pos is a match if it is a match of P and the boolean expression E is true when Pos is *observed* by the component or architecture containing the pattern (see Section 3).

Discussion: An event is observed by a component C when it is generated if C generates the event, else when C selects it (see Section 3). A poset is observed when all of its events have been observed.

Examples

Examples of patterns and their matches are shown in Figure 1.

Example: An iterator

```
    with Sender, Receiver;
    ...
    ?s : Sender;
    !r : Receiver;
    ?d : Data;
    ...
-- A connection rule.
    ?s.Send(?d) => !r.Receive(?d);;
```

Pattern	Matching posets
a	An a event, i.e., any event whose tuple is in E_a.
$a(5)$	Any a event whose second tuple element equals 5.
$a(?\mathrm{i})$ **and** $b(?\mathrm{i})$	An a event and a b event whose tuples have the same second value.
a **or** b	An a event or a b event.
$a \rightarrow b$	An a event and a b event where the b event depends on the a event.
$a < b$	An a event and a b event where the timestamps of a are less than those of b.
$a \,\|\|\, b$	An a event and a b event where neither event depends on the other.
$a^{5\rightarrow}$	Five a events all of them ordered by the dependency relation.
$a^{*\|\|}$	Zero or more a events, all of which are independent.
$a^{+\rightarrow}$	One or more a events in a linear dependency sequence.
$a(?\mathrm{i})$ **where** $?\mathrm{i}<4$	An a event who second tuple element is less than 4
$?o\,.\,Do\_It$	Any *Do_It* event generated by any object of the type of $?o$.

Figure 1: Examples of patterns and matches.

Comments: The connection rule is triggered by any poset that matches the left side pattern — an event in which a sender generates a send of some data. Both `?s` and `?d` are bound by the match. The right side restricted pattern is then executed. This results in events in which every receiver object in the system (i.e., every match for `!r`) receives the same data, `?d`.

Discussion: Patterns are used in the other three major features. For example, a pattern may be used in an interface to define concurrent behavior consisting of several events being generated simultaneously.

Patterns are used in three ways.

1. to recognize (or *trigger* on) posets with particular features.
2. to specify constraints on posets, thus constraining behaviors.
3. to generate posets in response to a trigger, thus generating behaviors.

In the first two applications the most general patterns may be used. but in the third use, the pattern must fully specify a poset. This means, for example, that it can't be an "or" of two posets. So only *restricted patterns* can be used to generate behaviors.

3 Components

An interface defines a type of components (e.g., integers, or airplanes). It provides an abstract definition of the externally visible behavior of the components — i.e., the behavior that is visible to, and may be observed by, other components. Each component of the type has a copy of the interface through which it interacts with other components. An interface may define the kinds of events that a component can receive or generate, the functions it provides to other components or requires from other components, its states and state transitions, and constraints on its external behavior.

Syntax of interface types

The syntax of interfaces is shown in Figure 2. Standard Pascal-like types (Boolean, string, integer, array, record etc., but not pointers) and object declarations are assumed here without definition. These are called the Procrustean types and objects. [2] In μRapide *name types* (below) are used instead of pointers.

interface ::=
  [*with_clause*]
  interface *name* [*interface_params*] **is**
    interface_constituents
  [**behavior**
    declarations
    {*state_transition_rule*}]
  [**constraints**

[2] After Procrustes, a villainous son of Poseidon in Greek myth who altered the size of visitors to fit his facilities.

```
        {constraint}]
    end name ';'

with_clause ::= with name{ ',' name}';'

interface_params ::=
    '('name-list ':'name
        {','name_list ':'name}')'

interface_constituents ::= {interface_constituent}

interface_constituent ::= action_declaration
    | subprogram_declaration
    | service_declaration

action_declaration ::=
    mode action signature {',' signature} ';'
mode ::=
    out
    | [in]

subprogram_declaration ::=
    mode function subprogram_signature
        {',' subprogram_signature} return name';'
    mode procedure subprogram_signature
        {',' subprogram_signature} ';'

service_declaration ::=
    mode service service_signature
        {',' service_signature} ';'

service_signature ::=
    name ['[' expr '..'  expr ']']
        ':'  interface_name

action_signature ::= name
    ['('name-list ':'name
        {','name_list ':'name}')']

subprogram_signature ::= name
    ['('name-list ':'[parameter_mode] name
        {','name_list ':'[parameter_mode] name}')']

parameter_mode ::= [in]
    | out
    | in out

constraint ::= pattern ';'
declarations ::=
    Pascal-like variable, type and function declarations
    and placeholder, iterator and name declarations

state_transition_rule ::=
    trigger '=>' transition_statement ';;'

trigger ::= pattern -- see Section 2
    | boolean_expression

transition_statement ::= {state assignment}
    [restricted_pattern]

restricted_pattern ::= a pattern not containing
    or, and, where, ~, * or +.
```

Figure 2: Definition of the *interface* syntax.

Semantics

An interface specifies the types of events and function calls its objects can receive and generate by declaring, respectively, *in* and *out* actions and *in* and *out* functions.

An interface optionally contains a behavior which consists of a set of procrustean types and objects, and transition rules. Only procrustean objects are allowed. Objects declared in an interface model *state*. Transition rules model how objects of the intefacetype react to patterns of received events by changing their states and generating events. An interface also optionally contains a *constraint* part which is a set of constraints over the behavior of all objects of the interface type.

An object of an interface type receives *in* events from the system it is in (this is defined by the architecture containing that object as a *component*). As a consequence, it executes its transition rules and generates *out* events which are sent to other objects. An object can also generate its own *in* events and receive its own *out* events. Similarly, an object receives *in* function calls; the object must declare functions or state transition rules that are executed in response to these calls. When an object calls its *out* functions it depends upon the architecture containing it to connect those calls to *in* functions in other components. In the execution model, a full function invocation (i.e., a function call and the corresponding function return) are represented by two events, the function call and the function return.

A state transition rule has two parts, a trigger and a body. A trigger is either an event pattern or a boolean-valued expression. A body is an optional set of state assignments followed by a restricted pattern which describes a poset.

The execution semantics of transition rules are as follows. First, all of the boolean triggers are evaluated. If any of them are true, the process of arbitrarily choosing one of the true triggers, executing its rule body and then revaluating all the boolean triggers, is repeated until none of the boolean triggers are true.

Next, the component selects one of the events it has received. (The received events consist of *in* events generated by the architecture as the result of connection rules (Section 4), and the events generated by the component itself.) An event is selected by a particular component at most once. If no events are available to be selected, the component waits for one.

Events are selected in an order that is consistent with their dependency and time orders. [3]

Selecting an event may *trigger* one or more transition rules. A rule R triggers if some subposet of the events thus far selected by the component but not yet used in triggering R match its pattern. If more than one poset of selected events could trigger a rule, an earliest (in time and dependency) and largest poset is chosen. Those events are then no longer in the selected set that may be used to trigger R again.

If R triggers, the binding of placeholders used to match its pattern is applied to its body, and that instance of R's body is then ready for execution. The set of rule body instances that are ready for execution are then executed in some order.

Executing a rule body consists of making the state assignments, generating a poset of new events that matches the instance of the restricted pattern, and adding the poset to the execution of the overall system. When the new events are added they depend on all of the events in the match of the trigger; they also depend on events that triggered the last rule to assign to any state veriable that is read by the rule (e.g., reading can occur in a boolean restriction in matching the trigger or in right side of a state assignment).

Execution of the component then continues with evaluating boolean triggers first, and then selecting events, as above.

Below is a schema for the above ordering of activities:

```
loop
   evaluate all boolean triggers;
   while (one or more boolean triggers
            evaluated true) loop
      choose one of the true triggers and
         execute its body;
      re-evaluate all boolean triggers;
   end while;
   select an event E;
   for all pattern triggers triggered by E loop
     execute the trigger's body
   end for;
end loop;
```

Note that any event may take part in triggering a given rule at most once, although it may take part in matching several rules. Also, a single μRapide transition rule can express the reaction of a concurrent/distributed system since its output pattern can define a complex poset of events with various dependencies and timing.

Constraints specify restrictions on the behavior of an interface. That is, the behavior that results from executing transition rules must match the pattern of a constraint. For example, a constraint can specify that rules trigger only in certain orders, in certain states, etc.

Discussion: Constraints may be supported in different ways by various tools. For example, executions can be checked and violations detected (runtime checking tools), or transitions may be allowed to execute only if a constraint won't be violated (safe execution), or transition rules can be proved consistent with constraints (verification).

Services

A *service* of type T in interface I declares that type I objects either provide (*in* service) or require (*out* service) the *actions* and *functions* of T. This allows I to specify that it provides or requires particular types of connections to or from its architectural environment. Since a service is a type, it is a concise notation for replication (e.g., one can declare arrays of a service) of large numbers of connections.

Declaring an *in* service S of type T is the same as including all of the actions, functions and service

[3] This is called the **orderly observation principle.**

declarations of T (i.e., the *constituents* of interface T) with their modes. To name them the name "S" must be appended with the usual "." notation before the name of the constituent. For example, if interface I declares a service S of a type T which has an out action A, then "S.A" is a name of the out action A of I. An *out* service is similiar except that the *modes* of the service's constituents are reversed; *in* (*out*) functions, actions and services become *out* (*in*) functions, actions and services. Additionally the mode of function parameters is reversed (*in out* parameters are unaffected). An *in* service and *out* service of the same type in objects (usually of different types) may be connected together easily since they have complimentary in and out constituents.

Functions

A function body is semantically equivalent to a single rule with a potentially more complicated state assignment part. A function *foo* is equivalent to a rule which is triggered by the event *foo_call* and which performs the computation of the function and then generates a *foo_return* event.

A rule with a function call treated as sugar for several rules without function calls. For example, the transition rule

```
A => a := 5; x := sin(3); y := 8;;
```

is equivalent to the rules

```
A => a := 5; sin_call(3,...);;

sin_return(?p,...) => x := ?p; y := 8;;
```

The elided information represents the data needed to associate calls and returns and to store intermediate values.

A function may be implemented in the standard Pascal manner or with a pattern triggered rule. When a rule is used, the pattern is the function name with the "_call" suffix. Its parameters consist of all of the *in* parameters of the function call. The rule must generate an event whose name is the function name with the "_return" suffix and whose parameters are the *out* parameters of the function call.

Name types

Instead of pointers, μRapide has *name* types. For every interface type T there is a corresponding type $\&T$ called the *name type of* T. T is said to be the *base type* of $\&T$. For each object e of the interface type T there is a corresponding object in the name type $\&T$ called the name of e. Each interface type T has an operator "&" defined on it which converts an object of T into the corresponding name in the name type of $\&T$. The only operators defined on name types are equality ("="), assignment (":="), and dereferencing ("*"). Equality and assignment are defined as usual. The dereference operator of a name type $\&T$ is defined to convert a name in $\&T$ to the corresponding object of the interface type T. An example of name types is given in Section 4.

Examples

Example: Automobile controls.

```
interface AutoControls is
  function Speedometer return MPH;
  function Gas return Gallons;
  in Action Steering_Wheel(A : Angle);
  in action Accelerator(P : Position);
  in action Brake(P : Pressure);
  out action Warning(S : Status);
  ...  -- other constituents.
end AutoControls;
```

Comments: Instruments are specified as *in* functions or *out* actions. An AutoControls component (i.e., an object having this type of interface) must supply bodies or reactive rules for computing the functions. A user can call these functions, if its *out* functions are connected to them by the architecture, and obtain their return results. So, in effect, they *output* readings when asked. Similarly, a user can receive *out* events if the architecture connects them to *in* events of the user.

Controls are specified as *in* actions. An AutoControls object can observe, select and react to *in* events corresponding to these actions. A user must output events containing position, pressure or angle data; this output is connected by the architecture to these *in* actions and thereby observed by the Autocontrols as *in* events.

Example: Specification of concurrent behavior.

```
...
behavior
   Speedometer > 55  =>
      Accelerator(0)  ||  Brake(High)  ||  Warning(On);;
   ...
```

Comments: This transition rule could be part of the behavior of the AutoControl interface. It is triggered by a Boolean condition and specifies a reaction consisting of the generation of three independent events. Two of these are *in* events which will be received by the AutoControl object itself. The other is an *out* event which will cause other events according to the architecture's rules.

Example: Automobile interface.

```
with AutoControls;
interface Automobile is
   in service Controls : AutoControls;
-- ... other features
behavior
   CurrentSpeed  : MPH;
   CurrentGas    : Gallons;
   ?p            : Gallons

   function Controls.Speedometer return MPH is
   begin
      return CurrentSpeed;
   end Speedometer;

   Controls.Gas_call()  =>
       Controls.Gas_return(CurrentGas);;
   ...
end interface;
```

Comments: An automobile's interface is defined to provide the AutoControls interface as an *in* service. This allows cars (objects of type automobile) to be connected to other objects that have *out* control services – i.e., have been specified to use controls. The Automobile interface will contain lots of other constituents as well, probably other services. The behavior part illustrates two ways that the autombile interface can specify its services. There are two different ways of handling function calls, a *function body* and a *pattern-triggered transition rule.*

Example: A driver interface.

```
with AutoControls;
interface Driver is
   out service ControlPanel : AutoControls;
-- ... other features.
behavior
   ...
   ControlPanel.Speedometer > 55  =>
            ControlPanel.Accelerator(Slow);
   ...
end Driver;
```

Comments: The driver interface also contains a copy of the AutoControls interface called ControlPanel, but as an *out* service. This means that drivers can specify interaction with ControlPanel by calling the *in* functions to get instrument readings (i.e., treating them as *out* functions), and by generating events of the control *in* actions (i.e., treating them as *out actions*). The modes of the *out* service interface are reversed. As shown here, the transition rule calls the Speedometer function and outputs Accelerator events.

As we shall see in Section 4, drivers and cars can be connected together by connecting their AutoControl type services with a single connection rule as indicated in Figure 3.

Example: A pipeline.

```
interface Pipeline is
   in action Input(X : Msg);
   out action Output(X : Msg);
behavior
     First : Boolean := True;
     ?M : Msg;
     Input(?M) where First    => First := False;
                                   Output(?M);;
     (Input(?M) where not First) and Output =>
                                   Output(?M);;
end Pipeline;
```

Comments: This is an interface for components that output a linear sequence of dependent events containing the messages in any poset of input events. Any poset of input events may be sent to a pipeline, including concurrent and independent events. The first one selected will be one of those that is not dependent of any other input event be-

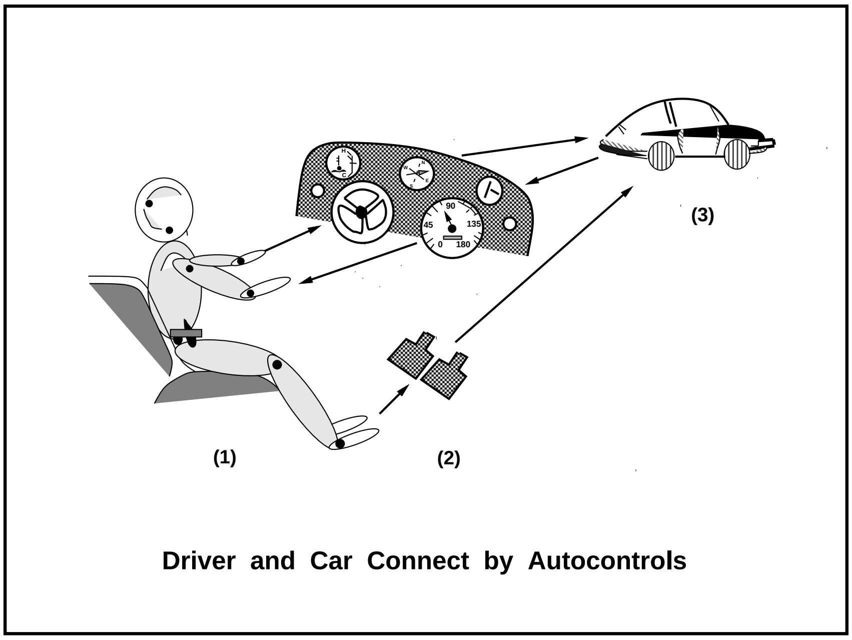

Figure 3: Driver (1) and Car (3) connected via the Autocontrol service (2).

cause events are selected in an order consistent with their dependency. It will be selected in the state in which First is true. Therefore it can only trigger the first rule because of the guard conditions. When this rule is executed, before any other event is selected, First is assigned false and an output is generated. Any other input or output event is observed (i.e., is selected or generated) in this state, and so helps to trigger only the second rule.

It is possible that several input events may have been selected before the first output event is selected. If so, the first output event can be used only once with one of these inputs to trigger the second rule. The poset consisting of the output and input events must be an earliest one in the input poset order. As a result a second output event is generated. It will depend on both the input event used in this triggering, and the first output event. Similarly, any subsequent output will depend on the previous outputs, and so a linear sequence of dependent outputs is generated as shown in Figure 8 (in this figure, Radio events are equivalent to pipeline Input events, and Receive events are equivalent to pipeline Output events). The messages in the linear sequence will be in an order consistent with the poset order of the input messages. That is, messages will be output in the causal order that they were input; independent inputs can be output in any order.

4 Architectures

An architecture is a set of components (objects of some interface types) and a set of rules defining how the components communicate by sending each other events or calling each others functions. Communication rules are defined using event patterns. Event patterns provide the expressive power to define both static and dynamic architectures. A static architecture has a fixed number of components and the conditions under which they communicate does not vary — hardware architectures are a typical example. A dynamic architecture may contain varying numbers of components (of some given set of interface types) and the conditions under which they communicate can vary — an air traffic control system is an example of a dynamic architecture with varying numbers of aircraft and communication conditions that can vary with distance and other factors.

Syntax

An architecture contains declarations of types, system components and other objects, a set of connection rules, and a set of constraints.

architecture ::=
  [*with_clause*]
  architecture *name* [**for** *name*] **is**
    declarations
  connections
    {*connection*}
  [**constraints**
    constraints]
  end *name* ';'

declarations ::=
  component, placeholder and
  Pascal-like type declarations

connection ::=
  basic_pattern_list '=' *basic_pattern_list* ';'
  | *basic_function_pattern* '='
    basic_function_pattern ';'
  | *pattern* '=>' *restricted_pattern* ';'
  | *pattern* '=>' *component_generation* ';'

basic_pattern_list ::= *basic_pattern*
  | *basic_pattern_list* ',' *basic_pattern*

component_generation ::=
  new *name*

constraints ::= {*pattern* ';'}

Figure 4: Definition of the *architecture* syntax.

Semantics

The components and interface types of components in the system are declared in the declarations section of an architecture.

Static architectures may simply declare all components by naming them in object declarations. On the other hand, dynamic architectures may declare the interfaces of components and rely on creation rules (below) to define when, during execution, components of those types are created or destroyed.

The connection part contains connection rules and creation rules. Connections define communication between components by events, and creation rules define event conditions that lead to creation of new components.

Initially some events are input to components. Events are generated by the components in the architecture. These events are selected one-by-one (in any order that is consistent with their dependency and temporal orders) and matched with the pattern triggers of the connections. The semantics of selecting events and matching connection rules is similar to the semantics for transition rules in components (Section 3). As before, the essential points are: *(i)* any event may contribute once to triggering a particular connection rule but may trigger many different rules, and *(ii)* if more than one poset of the selected events can trigger a rule then an earliest (in the dependency order) maximal poset is used.

A basic connection is an "=" between two basic patterns, or more generally, between two lists of basic patterns. Consider a connection between two basic patterns. Whenever an event matches (triggers) the left pattern, Pat, in a basic connection rule, $Pat = Pat'$, then *(i)* all placeholders in the right pattern, Pat', must be bound by the match, *(ii)* the rule results in generating a new *in* event whose tuple is the instance of Pat', and this event is received by the component named in that tuple, and *(iii)* the two events are equivalent with respect to dependency and time.

Equivalence of the two events means that all other events have the same dependency relationship to both of the events, and also the two events have the same timestamps. This does not mean that the events are equal, but simply that they cannot be distinguished by dependency or time.

A basic connection between two lists of basic patterns is a shorthand for several basic connections. That is, a match of any one of the left patterns causes (or triggers) the generation of the events, one for each pattern in the right list, and the events generated are all equivalent to the triggering event with respect to dependency and time.

The
semantics of general connections, $Pat => Pat'$, are as follows. The values associated with the last event to be selected in matching Pat are used to evaluate

any **where** condition guarding *Pat*. When *Pat* is matched, all placeholders in the right pattern, Pat', must be bound by the match. Then the rule is executed, resulting in *(i)* generating a poset of new *in* events that matches the instance of Pat', and the events are received by the objects named in those events, and *(ii)* the resulting poset is added to the execution of the architecture so that each new event depends on all events in the triggering poset.

Out functions are bound to *in* functions with connections. The *out* function is the left pattern and the *in* function is the right pattern. The *in* function must return the type of the return value of the *out* function (or a subtype of that type). Placeholders are used to copy parameters of the *out* function call to the call of the *in* function.

A pattern may also be used as a constraint. This means that poset behaviors generated by the architecture must match the constraint pattern. Constraints may, for example, require connections to trigger in certain orders, thus defining a communication protocol. As discussed in Section 3, constraints may be supported in different ways by various tools.

Examples

Example: A basic connection.

```
  ...
  ?P : Person; ?L : Lift; ?B : Button;
connections
  ?P.Push(?L,?B) = ?L.Button_Light_On(?B);;
```

Comments: This basic connection links pairs of events. An event of any person pushing a button of a lift triggers the rule and produces a new event of the lift registering that the button's light is on. The two events are identical with respect to dependency and time – i.e., they have the same dependencies with all other events and the same timing. It is not possible to distinguish the two events either by a clock or by looking at their causal history. So the connection links persons and lift buttons in a very strong way; the actions of pushing buttons and lighting buttons appear identical according to time and causal history.

Example: A dynamic architecture

```
with Airplane, Control_Center;
architecture Air_Control_Sector is
    ?A : Airplane; ?M : Msg;
    SFO : Control_Center;
    ...
connections
    ?A . Radio(?M) where ?A . InRange(SFO)
         => SFO . Receive(?M);;
    ...
end Air_Control_Sector;
```

Comments: Assume the interfaces of Airplane and Control_Center are already defined. The connection defines event communication between any airplane object and a particular control center as depicted in Figure 5. Whenever any airplane (a match for ?A) generates a Radio event containing a message and the InRange predicate of that airplane is true in the state when the radio event is selected, then SFO will receive a Receive event with the same message. The connection triggers only when an airplane is in range.

This connection rule is a conditional broadcast between all airplanes and the control tower. It defines communication in a system that may have varying numbers of airplane components. It imposes dependencies between pairs of Radio and Receive events as shown in the poset Figure 6.

Discussion: The relationship between generating an event and the event being selected is nondeterministic. If an airplane can move independently of generating radio messages, it could be in range when it generates a radio event and out of range when that event is selected to trigger a receive event. If timing is present, events take no time between generation and selection unless specified to do so. In a timed model, if movement takes time, and radio events do not, InRange will not change between generation and selection. On the other hand, if the airplane is very fast ... Discussion of time and clocks is omitted due to space.

Example: Connecting drivers and cars

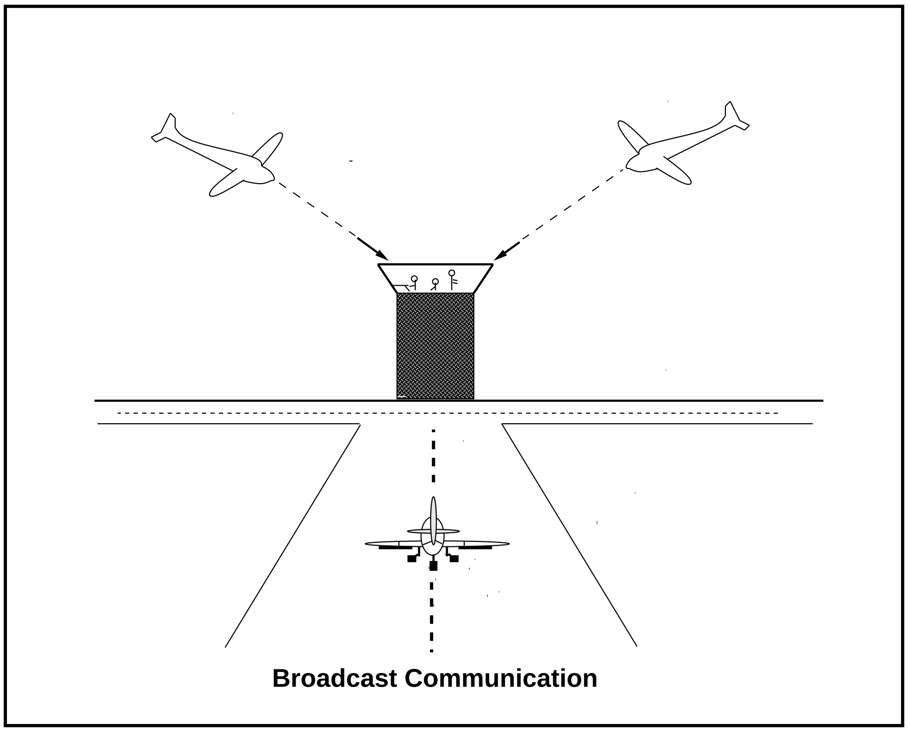

Figure 5: Broadcast architecture.

This example illustrates the use of services.

```
with Driver, Automobile;
architecture Vehicle_Test is

   Dummy  : Driver;
   TestCar : Automobile;

connections

   Dummy.ControlPanel = TestCar.Controls;
   ...   -- other connections.
end Vehicle_Test;
```

Comments: In Section 3 automobiles were defined to have an *in* service of type AutoControls, and drivers were defined to have the corresponding *out* service. The connection in Vehicle_Test is shorthand for a set of basic connections between each pair of constituents, *in f* and *out f*, or *out f* and *in f* in the two services. See Figure 3. So, for example, whenever Dummy calls its Speedometer function, the call is connected to the Speedometer *in* function in TestCar. Similarly, whenever the Dummy generates an *out* SteeringWheel(angle) event, TestCar will receive an *in* event with an identical tuple.

Example: Intelligent network architecture
The use of Name types is illustrated. Our first interface is the Broker. The Broker stores the names of Providers and the jobs they can perform. Clients use the Broker to find Providers for particular jobs. We do not include the interface definitions of the Client and Provider types as they are non-instructive.

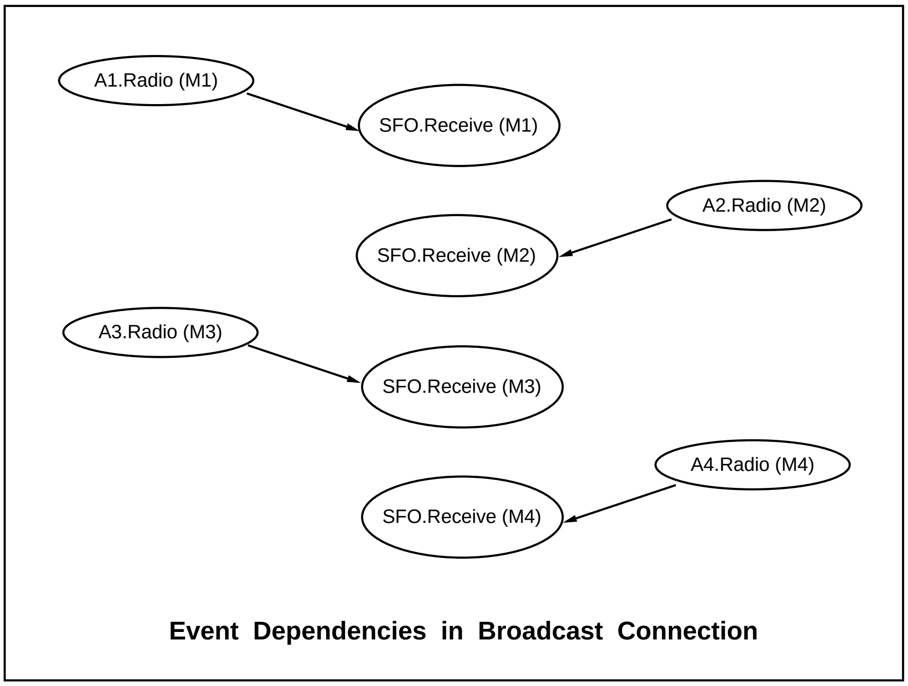

Figure 6: Broadcast events.

```
interface Broker is
  in  function Provider_Lookup(J : Job)
        return &Provider;
  in action Register_Provider(J : Job; P : &Provider);
behavior
  -- only store one Provider per Job type
  Jobs : array [Job] of &Provider;
  ?J   : Job;
  ?N   :  &Provider;

  function Provider_Lookup(J : Job)
  return &Provider is
  begin
     return Jobs[J];
  end;

  Register_Provider(?J, ?N) => Jobs[?J] := ?N;;
end;
```

```
with Broker, Client, Provider;
architecture Network is
  NTT       : Broker;
  clients   : array [1..NUM_CLIENTS] of Client;
  providers : array [1..NUM_PROVIDERS] of Provider;
  ?P        : Provider;
  ?J        : Job;
  ?C        : Client;
  ?N        : &Provider;
  ?param    : parameters;
connections
  ?P.Register(?J) =
        NTT.Register_Provider(?J, &?P);;
  Client::Request_Provider(?J) =
        NTT.Provider_Lookup(?J);;
  ?C.Request_Service(?J, ?N, ?param) =
     *?N.Service_Request(?J, ?param, &?C);;
end Network;
```

The broker, clients and servers are instantiated in the system architecture. All of the communication among the broker, clients and servers is defined by three connection rules. The last rule bears some explanation.

It states that when a client generates a Request_Service event one of whose parameters is the name of a Provider, then a Service_Request event is sent to the Provider which corresponds to that name and one of the parameters of that event is the name of client.

4.1 Connections as Components

Any μRapide architecture can be built from components (of various interface types) and only basic connection rules. Some of the components play the role of "high level" connections between other components — we call these components *connectors*.

To see this we show that connection rules, $Pat => Pat'$, can be expressed as components of interface types together with basic connection rules. In the connection, $Pat => Pat'$, the basic patterns in Pat are instances of *out* actions of components and interface types, C_1, while the basic patterns in Pat' are instances of *in* actions of components and interface types, C_2. So define an interface type of the form,

```
interface Connector is
  in action Ai(...); -- actions in C_1.
  ...
  out action Bi(...); -- actions in C_2.
  ...
behavior
   Pat => Pat';
end Connector;
```

Then replace the connection rule in an architecture S by an object Link and basic connection rules as shown below:

```
architecture S is
   Link : Connector;
   ...  -- as before.
connections
   ?Oi : Ui;  -- types in C_1.
   ?Ii : Vi;  -- types in C_2.
   ?Oi.Ai = Link.Ai;
   Link.Bi = ?Ii.Bi;
   Obj_i.Ai = Link.Ai;  -- objects in C_1.
   Link.Bi = Obj_i.Bi;  -- objects in C_2.
   ...
end S;
```

Example: Pipelining air traffic control.

```
architecture PipeLine_Control_Sector is
     ?A : Airplane; ?M : Msg;
     SFO : Control_Center;
     Pipe : Pipeline;
connections
     ?A.Radio(?M) where InRange(?A, SFO)
          = Pipe.Input(?M);
     Pipe.Output(?M) = SFO.Receive(?M);;
     ...
end PipeLine_Control_Sector;
```

Comments: We have changed the air control sector architecture so that all radio events are sent to a pipeline (see the example in Section 3). The pipeline's output events are sent to SFO. Essentially, the pipeline is used as a connector between airplanes and the control as shown in Figure 7. Now all airplanes communicate with SFO by a single pipeline instead of by broadcast.

As we have seen, a Pipeline orders its outputs linearly so that the outputs depend on an input and all the previous outputs. The Receive events will all be in a linear dependence chain, as shown in Figure 8. In fact, the nodes in this figure are equivalence classes of pairs of events — each radio event has an equivalent input, and each output has an equivalent receive.

The semantic differences between the broadcast and pipeline architectures are shown clearly by the posets in the two figures, 6, 8. Note that trace-based languages do not make any distinction, and trace-based simulators could produce the same traces for these two architectures.

5 Mappings

Mappings can be used to define how one architecture is related to another one, or how an architecture is related to an interface (called a *binding*). The idea is to define how events in one system correspond to events in another. In many cases, there is quite a wide difference between systems. For example, when two systems are at different levels of abstraction many events in one may correspond to just one event in the other (as is often the case in hierarchical design). Patterns provide the necessary expressive power to define these kinds of mappings and bindings.

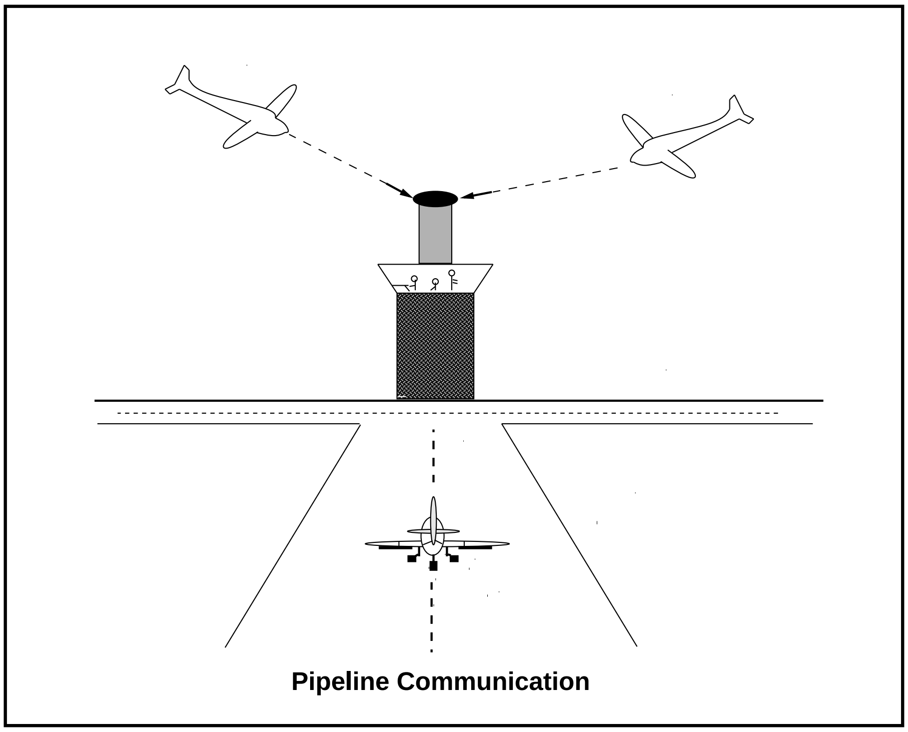

Figure 7: Pipeline architecture.

Bindings.

The μRapide *connections* (defined in architectures) are used to express bindings between an architecture and an interface. A binding is essentially an encapsulation/abstraction mechanism which declares, "treat this architecture as a detailed implementation of that interface".

To bind an architecture to an interface, the **architecture** construct is used. The actions of the interface are made visible in the architecture by naming the interface (**for** *name*, see Figure 4). The binding rules are expressed using the syntax of connections and are included in the connections of the architecture.

Maps.

A map may be defined between any pair of interfaces or architectures. Syntax of the *map* construct is given in Figure 9. The **from** and **to** names are names of architectures or interfaces. Maps may declare local objects .

map ::=
[*with_clause*]
map *name* **from** *name* **to** *name* **is**
declarations
rules
{ *rule*}
end map *name* ';'

rule ::=
trigger '=>' { *map_statement* ';'}';'
trigger ::= *pattern*
map_statement ::= [{ *state assignment*}]
[*restricted_pattern*]

Figure 9: *map* syntax.

Semantics. Rules in a mapping have the same semantics as transition rules in components (or non basic connections in architectures) *except* they do not define dependencies between the triggering events and generated events.

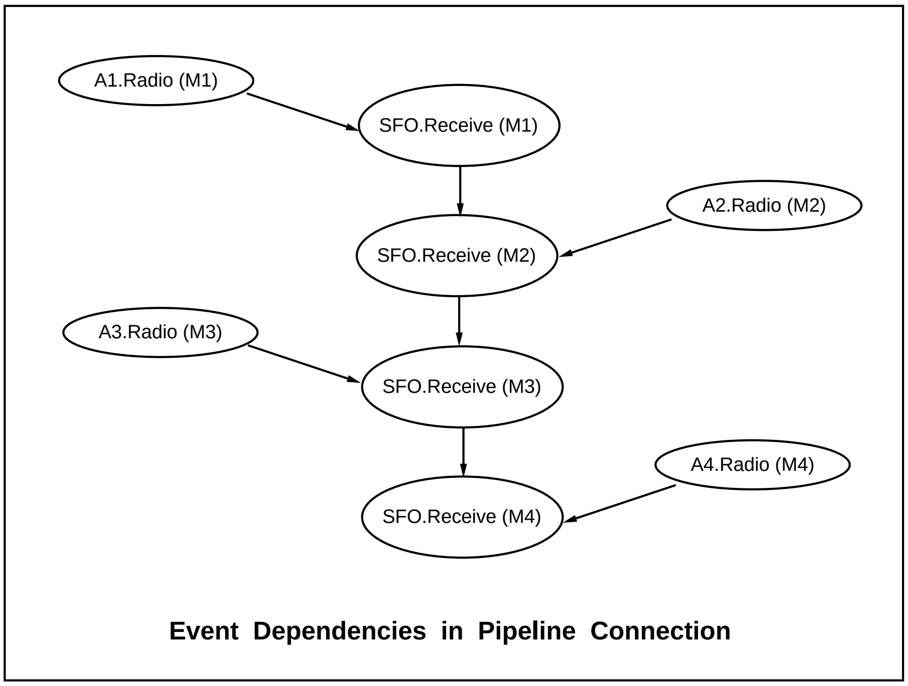

Figure 8: Pipeline events.

5.1 Example: A simple microprocessor

This is an example of a μRapide architecture and refinement map showing some of the complexities of "real life" applications. The power of the event pattern language is illustrated and event-based concepts are used to explain the example.

The original version of this example in [GL92] consisted of three architectures in VHDL for a simple 16-bit microprocessor at three commonly used design levels of abstraction: instruction level, register transfer level (RTL) and gate level. This work reported the results of using refinement mappings written in VAL (VHDL annotation language) to control the complexity of the VHDL simulation. The gate level simulation for a very small input data sample produced 8073 events. Clearly, manual inspection of this amount of output is difficult and error prone, even though it is very small in comparison with industrial simulations. [4] By using VAL refinement mappings to map the gate level architecture to the (RTL) architecture, and then map the RTL architecture to the instruction level architecture, the number of events in the mapped simulation was reduced to 5. Design errors at the gate level and RTL (typically incorrect architecture connections) which are difficult to detect at that level, were made manifest at the instruction level in the form of missing events.

Thus a powerful application of refinement maps to design hierarchies lies in mapping complex low level simulations into behaviors of a higher level, more abstract architectures – called the *mapped behavior*. The mapped behavior is much smaller and simpler. This has the following benefits:

- manual inspection of the mapped behavior is feasible.
- formal constraints are generally part of high level architectures since they embody design requirements; low level simulations can be "mapped up" and automatically checked

[4] Indeed, industry experience has related instances where large scale gate-level simulations have indicated design errors in microprocessors which go undetected in analysis of simulator output, and are only uncovered after manufacture (at far greater cost!).

against high level constraints.

- errors in the mapped behavior can be traced back to the low level architecture by analyzing where the trigger patterns of the map matched in producing the high level error.

Below we give the μRapide RTL architecture of the microprocessor and the mapping to the instruction level. Figure 10 is a picture of the RTL architecture, and Figure 11 is a picture of the map from patterns of RTL events to instructions; it shows the timing and dependencies of a set of RTL events that would trigger the map, resulting in a load event.

The μRapide global clock values are the same at all levels in the design hierarchy. In Figure 11 these readings are shown horizontally at both levels. The RTL pattern involves 12 events that trigger the refinement map for a load instruction. The arcs show the timing relations between the events that are required by the trigger — i.e., some events are required to occur after others, whereas some may occur in any time order. The CL events, for example, are device clock events. There are 4 CL events, defining three device clock cycles. The trigger requires particular events to occur on each cycle. The shadow of the pattern depicts the simulation time duration of the load instruction.

First, the interfaces of RTL component types (registers, buffers, controllers, and logic unit) are given. State transitions and constraints are omitted from these interfaces for brevity. One may assume either that there are state transitions or else an executable gate level architecture for each of these interfaces. Compilation dependencies between the interfaces are expressed by **with** clauses. Some of these interfaces could be derived from others by object-oriented features of RAPIDE [Tea93].

```
interface TypePreamble is
   type bit;
   type bit2 is array [1..2] of bit;
   type bit12 is array [1..12] of bit;
   type bit16 is array [1..16] of bit;
   type operations is (land, lor, lnot, lxor);
   type states is (if1, if2, if3, if4, ex1, ex2, ld, st);
end TypePreamble;

-- interfaces of RTL components.
```

```
with TypePreamble, Register_Logic;
interface Register is
   in action   Din(val : bit16),
               Clk(val : bit);
               Ce(val :bit),
               Oe(val : bit),
               Rst(val : bit);
   out action  Dout(val : bit16);

   out action  Load(r : bit16);
   out action  OE(r : bit16);
end Register;

interface Two_Output_Register_Logic is
   in action   D(val : bit16),
               Ce(val : bit),
               Rst(val : bit),
               Oe1(val : bit),
               Oe2(val : bit);
   out action  D1(val : bit16),
               D2(val : bit16);
end Two_Output_Register_Logic;


with TypePreamble, Two_Output_Register_Logic;
interface Two_Output_Register is
   in service  X : Two_Output_Register_Logic;

   in action   Clk(val : bit);

   out action  Load(r : bit2);
   out action  OE1(r : bit2);
   out action  OE2(r : bit2);
end Two_Output_Register;

with TypePreamble;
interface Buffer is
   in action   Din(val : bit16),
               Oe(val : bit);
   out action  Dout(val : bit2);

   out action  output(b : bit16);
end Buffer;

with TypePreamble;
interface Logic_Unit is
   in action   A(val : bit16),
               B(val : bit16),
               Op(val : bit2);
   out action  C(val : bit16);

   out action  alu(a,b,c : bit2; op : operation);
end Logic_Unit;
```

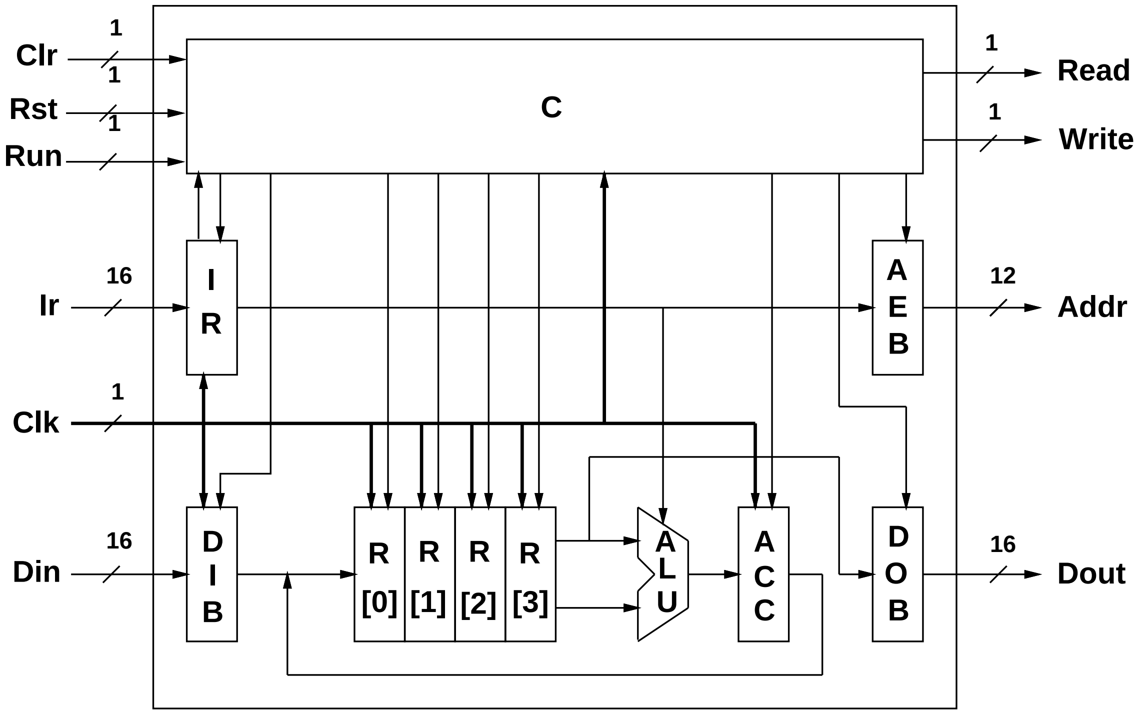

Figure 10: CPU register level architecture.

```
with TypePreamble, Two_Output_Register_Logic;
interface Controller is
   in action    op(val : bit2),
                r1(val : bit2),
                r2(val : bit2),
                clr(val : bit),
                run(val : bit),
                rst(val : bit),
                clk(val : bit);
   out action   irCE(val : bit),
                accCE(val : bit),
                accOE(val : bit),
                dinOE(val : bit),
                irRST(val : bit),
                readOE(val : bit),
                writeOE(val : bit);
   out service  reg [0..3] : Two_Output_Register_Logic;

   -- used to report state changes
   out action state(s : states);
behavior
   Current_State : States;
   ...
end Controller;
```

Next an RTL interface and architecture are given. The architecture, RTL_CPU_Arch, corresponds to Figure 10. So, for example, A in the figure is a logic_unit, C is a Controller, and R is a bank of four Two_Output_Registers. The μRapide connections consist of bindings between the RTL interface and architecture, and internal connections between components – the architecture proper.

```
-- RTL interface of the microprocessor – see Fig 10.
with TypePreamble;
interface RTL_CPU_Interface is
   in action    Clk(val : bit),
                Rst(val : bit),
                Run(val : bit),
                Clr(val : bit),
                Din(val : bit2),
                Ir(val : bit16);
   out action   Dout(val : bit2),
                Addr(val : bit12),
                Read(val : bit),
                Write(val : bit);
```

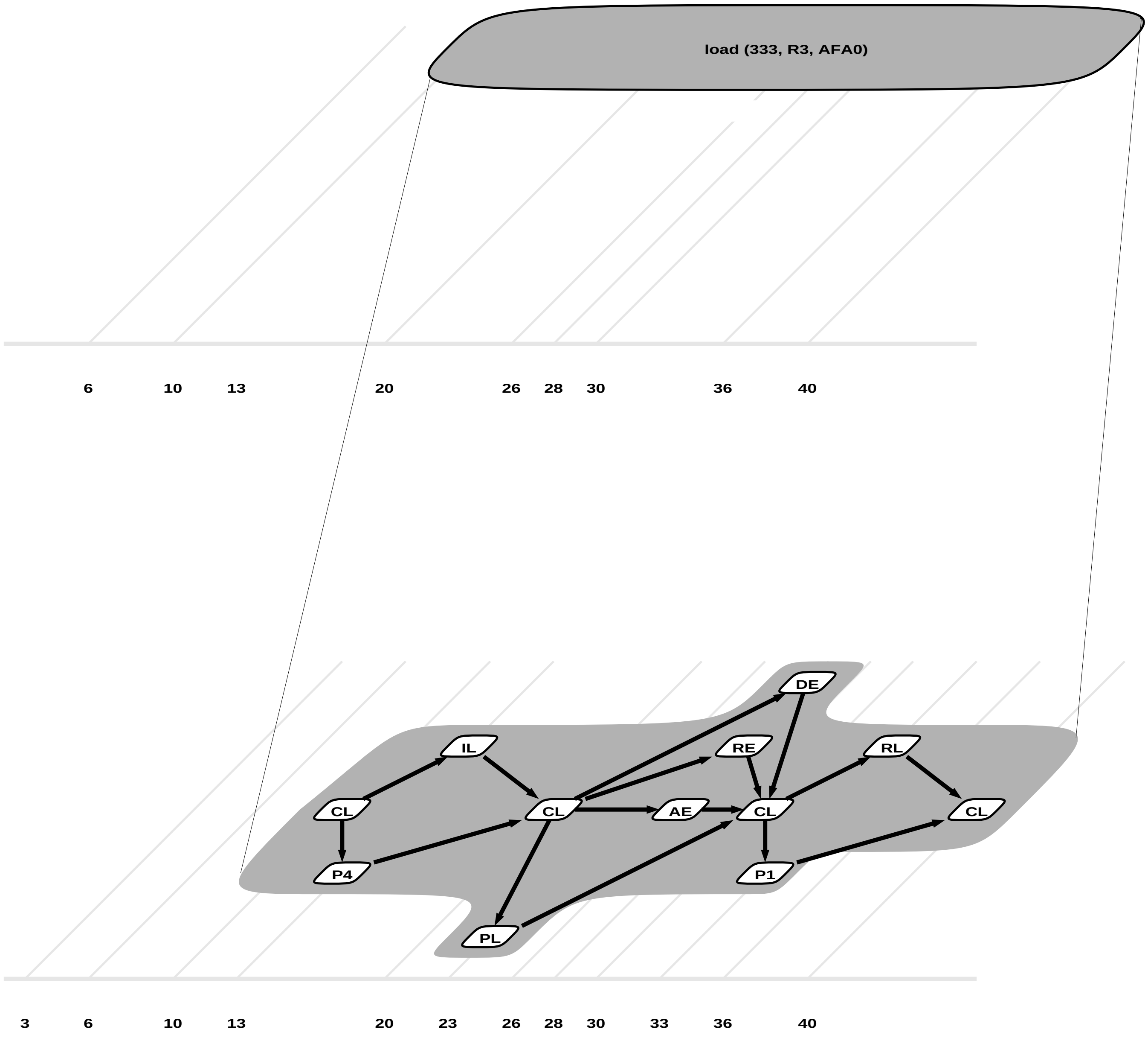

Figure 11: Map from register events to Load instruction.

```
end RTL_CPU_Interface;
```

For simplicity, if the parameter profiles of the patterns on both sides of the "=" are equivalent and what is wanted is that the parameter values of the left pattern be copied into the corresponding positions of the right patterns, we allow the parameter list to be omitted from both sides. This convention is used in the following architecture.

```
-- RTL architecture corresponding to the figure.
with TypePreamble, Register, Two_Output_Register,
     Buffer, Logic_Unit, Controller;
architecture RTL_CPU_arch for
     RTL_CPU_Interface is
  DIB, DOB,
       AEB  : Buffer;
  IR, ACC   : Register;
  R         : array [0. .3] of Two_Output_Register;
  ALU       : Logic_Unit;
  C         : Controller;
  ?b1       : bit;
  ?i        : integer;

connections
  -- input bindings
  Din = DIB.din;
  Ir = IR.din;
  Clk = R[0].clk, R[1].clk, R[2].clk, R[3].clk,
        ACC.clk, C.clk, IR.clk;
  Clr = C.clr;
  Run = C.run;
  Rst = C.rst;

  -- architecture connections.
  C.dinOE = DIB.OE;

  DIB.dout,ACC.Dout = R[0].X.din, R[1].X.din,
                      R[2].X.din, R[3].X.din;

  R[?i].D1 = ALU.A, DOB.Din;
  R[?i].D2 = ALU.B;
-- The following connection connects each out service of
-- the controller to the in service of the corresponding
-- register in the register bank. This is equivalent to 28
-- basic connections between pairs of actions.
  C.reg[?i] = R[?i].X;

  ALU.D = ACC.Din;
  C.accCE = ACC.CE;
  C.accOE = ACC.OE;

  IR.dout(?b16) = ALU.op(?b16[7. .8]),
        C.op(?b16[1. .2]), C.r1(?b16[3. .4]),
        C.r2(?b16[5. .6]), AEB.din(?b16[5. .16]);

  C.writeOE = DOB.OE, AEB.OE;
  C.irCE = IR.CE;
  C.irRST = IR.Rst;

-- output bindings.
  C.readOE = Read;
  C.writeOE = Write;
  AEB.dout = Addr;
  DOB.dout = Dout;

end RTL_CPU_arch;
```

These connection rules have a particularly simple event-based semantics since they are all basic connections defining equivalences between single events. There are no restricting conditions, so the architecture is static. Each connection means that whenever a given component generates an output event then the corresponding component will receive an (equivalent in dependency and time) input event. They define equivalences between pairs of single events.

The component interfaces also define dependencies between input and output events by state transition rules (which we have omitted) or by their underlying gate-level architectures. So, when this μRapide architecture is executed in response to some input, it will generate a poset of events that gives the dependencies between events generated by the components as well as their timestamps according to the μRapide global clock (see Figure 11).

In VHDL one must declare the connecting wires (called *signals*) and bind each component's ports (ports in VHDL correspond to actions in μRapide) to the wires. Here, μRapide basic connections allow us to define the "connecting wires" directly. Also, μRapide interface services are used to define 20 connections between the controller and the register bank in one connection rule. The same architecture in VHDL given in [Gen91] took approximately 170 lines of VHDL to specify whereas it took less than 30 lines here. Since errors in architecture connections are common, it is important to develop succinct notation for them.

Below, we give the microprocessor instruction level interface showing some of the instructions and the refinement mapping SimRef. The SimRef maps finite RTL posets (those matching its triggers) to single events at the instruction level. So we expect an RTL behavior to be mapped to a much simpler instruction level behavior.

```
-- CPU instruction set.
with TypePreamble;
interface Instruction_Level_CPU is
   out action Load(a : bit12; r : integer; d : bit2);
   out action Store(a: bit12; r : integer; d : bit2);
   out action Exec(d1, d2 : bit2; s1, s2 : integer;
                    d3 : bit2; op : operations);
end Instruction_Level_CPU;

-- Refinement mapping.
with TypePreamble;
map SimRef from RTL_cpu_arch
           to Instruction_Level_CPU is
   ?reg,?reg1,?reg2  : register;
   ?a                : bit12;
   ?r,?s1,?s2,?s3    : integer;
   ?d,?d1,?d2,?d3    : bit2;
   ?i                : bit16;
   ?op               : operation;

   function instrF(val : bits2) return states is
     ...
   function regname(val : Register) return integer is
     ...
rules
-- mapping rule defining Load instruction.
Clk(1)
 < (Ir.Load(?i) where instrF(?i)=ld and C.State(if4))
 < Clk(1)
 < (AEB.Oe(?a) where (?a=?i[5. .16]) and
    Read(1) and  DIB.Output(?d) and C.State(ld))
 < Clk(1)
 < (?reg.load(?d) where ?r = regname(?reg) and
     C.state(if1))
 < Clk(1)
    => Load(?a, ?r, ?d);

-- mapping rule defining Store instruction.
clk(1)
 < (IR.load(?i) where instrF(?i) = st and C.state(if4))
 < clk(1)
 < ((?reg.OE1(?d) where ?r = regname(?reg)
       < DOB.OE(?d)) and
     AEB.OE(?a) where ?a = bits(5,16,?i) and write(1)
     and C.state(st))
 < clk(1)
 < C.state(if1)
 < clk(1)
    => store(?a, ?r, ?d);

-- mapping rule defining Exec instruction
clk(1)
 < (IR.load(?i) where instrF(?i) = ex and C.state(if4))
 < clk(1)
 < (?reg1.OE1(?d1) where ?s1=regname(?reg1)
     and ?reg2.OE2(?d2) where ?s2=regname
     and C.state(ex1) and ALU.ALU(?op))
 < clk(1)
 < (Acc.load(?d3) and C.state(ex2))
 < clk(1)
 < (?reg.load(?d3) where ?s3=regname(?reg)
     and C.state(if1))
 < clk(1)
    => exec(?d1,?d2,?s1,?s2,?d3,?s3,?op);
end SimRef;
```

In defining a refinement map, it is important to specify the trigger of each rule so that it triggers on posets of low level events that are sufficient to signify the corresponding high level event. If the triggers can match some subset such as the upper or lower bound events of the appropriate posets, the mapped behavior will not be correct, but may contain extraneous events. Patterns provide a powerful and succinct notation for specifying sufficient posets.

The Load rule, for example, specifies the RTL events which correspond to an instruction level Load event. Any clock event with a parameter of 1 (indicating a rising edge of the device clock) may initiate a Load behavior. On the first clcok cycle the instruction register Ir must load a instruction whose instruction field indicates a Load and the controller C must transition to state if4. On the second clock cycle the address enable buffer AEB must output a particular address value ?a which is extracted from the instruction loaded in the previous cycle, the processor must output a Read, the Data In Buffer DIB must output data ?d, and the Controller must transition to a ld state. On the third clock cycle a register (any one in the register bank) must execute a Load of the data ?d output by the DIB in the previous cycle and the Controller must transition to state if1. When that clock cycle completes, this poset maps to a Load instruction which indicates the data ?d was loaded from address ?a into register ?r.

The above mapping rules define completely the set of RTL events together with their timing which correspond to the instruction level events. Since causality is not expressed in the triggers (because this example is taken from VHDL) the mapping rules are *not sufficient.* For example, it would be possible, for some set of RTL events with the correct timing to trigger a load mapping rule when a load instruction is not executed.

Acknowledgements

μRapide is an experimental ADL which is being tested and debugged by several patient people who are using it to express non-trivial architectures. This activity has resulted in many helpful questions and suggestions. Especially, our thanks are due to John McHugh and John Munson of University of North Carolina and TRW, Frank Belz of TRW, and members of the Stanford Program Analysis and Verification Group including John Kenney, Walter Mann, Alexandre Santoro, Larry Augustin, and Sriram Sankar.

References

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

[BCG87] G. Berry, P. Couronne, and G. Gonthier. Synchronous programming of reactive systems: an introduction to Esterel. Technical Report 647, INRIA, Paris, March 1987.

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

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

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

[LVB+93] 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*, to appear, 1993.

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

[Tea93] Rapide Design Team. *The Rapide-1 Executable Language Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, March 1993.

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