An Event-Based Architecture Definition Language

David C. Luckham, James Vera

Abstract— **This paper discusses general requirements for architecture definition languages, and describes the syntax and semantics of the subset of the RAPIDE language that is designed to satisfy these requirements. RAPIDE is a concurrent event-based simulation language for defining and simulating the behavior of system architectures. RAPIDE is intended for modelling the architectures of concurrent and distributed systems, both hardware and software. In order to represent the behavior of distributed systems in as much detail as possible, RAPIDE is designed to make the greatest posible use of event-based modelling by producing causal event simulations. When a RAPIDE model is executed it produces a simulation that shows not only the events that make up the model's behavior, and their timestamps, but also which events caused other events, and which events happened independently.**

The architecture definition features of RAPIDE are described here: event patterns, interfaces, architectures and event pattern mappings. The use of these features to build causal event models of both static and dynamic architectures is illustrated by a series of simple examples from both software and hardware. Also we give a detailed example of the use of event pattern mappings to define the relationship between two architectures at different levels of abstraction. Finally, we discuss briefly how RAPIDE is related to other event-based languages.

Keywords— **Rapide, architecture definition languages, partially ordered event sets, architecture, prototyping, concurrency, simulation, formal constraints, event patterns, causality.**

I. INTRODUCTION

RAPIDE [LVB+93], [LKA+95] is an *executable architecture definition language* (EADL). Although it has many of the features of present-day event-based simulation languages, it also provides new features to represent system architecture (see, e.g., [LVM]).

In this paper we first describe design requirements for EADLs. The architecture definition features of RAPIDE are then presented. Their semantics are described in terms of causal event executions and illustrated by a series of simple examples. We show how these features can be used to model the behavior of both static and dynamic architectures. We also show how mappings can relate widely different architectures, one at a high level of abstraction and another at a much more detailed level. Finally, we give a brief history of RAPIDE and the current status of its supporting toolset.

Other features of RAPIDE are described in other papers: these are object-oriented features for deriving new interface types and modules from previous ones [KLM94], [Tea94c], concurrent reactive programming constructs [Tea94a], and formal constraints [Tea94b]. An earlier version of RAPIDE was outlined in [LKA+95], and its use to model one large-scale example, the X/Open DTP standard architecture [XoD92] standard was described. In that paper we discussed how mappings, in conjunction with formal constraints, can be used to test conformance of systems to architectural standards.

The current simulation toolset for RAPIDE has been built for modelling and simulation of architectural designs during the early phases of system development, and also for testing conformance of systems to architectural designs.

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

II. FEATURES FOR DEFINING ARCHITECTURE

There is now a widespread belief that software engineering must go beyond object oriented methods to a new technology based upon "architecture". Technologies such as CORBA [Gro91], for example, allow distributed systems of interacting modules to be wired together. However, an architectural plan of the system is needed to both guide the "wiring-up" and to prototype the behavior of the system before effort is put into building the modules (i.e., the system's *components*).

An *architecture* in RAPIDE is an executable specification of a class of systems. It can be at any level of abstraction. An architecture consists of *interfaces*, *connections*, and *constraints*. The interfaces specify the behavior of components of the system, the connections define the communication between components using only the features specified in their interfaces, and the constraints restrict the behavior of the interfaces and connections. This is called an *interface connection architecture* [LVM] since the communication between system components is defined by connections between their interfaces. When a RAPIDE architecture is executed it produces a causal event history which is automatically checked for conformace to constraints.

Interface connection architectures can be built quickly in RAPIDE. They have two main purposes. First, to build with relatively little effort, a prototype that enables one to study and predict behavior before effort is put into building a full system. Second, to define a "plan" or "framework" to guide construction of a system, possibly by automated synthesis methods. To achieve these goals,

- RAPIDE must be powerful enough to define interface connection architectures that can be executed and their properties measured,
- the system, when it is built, must conform to the architecture.

A. Requirements for ADLs

The requirement of "sufficient power" leads us to the following general principles to be satisfied by the RAPIDE

design.

◦ **Component abstraction:** *Interfaces, which are the feature in* RAPIDE *for component abstraction, should define : (i) the facilities* **provided** *and the facilities* **required** *by a component, (ii) the component's behavior in a form allowing execution and analysis.*

Interfaces in languages such as Ada [Ada94] (e.g., *task types* and *package specifications*), the public parts of C++ classes, or entity interfaces in VHDL [VHD87], specify only the signatures of their provided facilities (e.g., functions, task entries or ports). None define features they *require* from other components, nor do they define behavior independently of a module instance.

◦ **Communication abstraction:** *Connections, which are the feature in* RAPIDE *for defining the communication between components, should (i) use only the interfaces of components, and (ii) define communication in a form allowing execution and analysis.*

In programming languages, although interfaces restrict the visibility into modules, communication between modules is implemented in the modules. For example, communication is represented by function calls buried in classes (C++) or package bodies (Ada), or task entry calls buried in task bodies (Ada). Communication is implemented but there is no communication abstraction.

On the other hand, hardware simulation languages do provide communication abstraction, but only for static architectures. In VHDL, for example, communication between entity interfaces is defined in structural architectures by port maps that wire interfaces together. Separate configurations associate entity bodies (modules) with interfaces in an architecture; different configurations define different implementations of an architecture, but the communication is defined once in the architecture for all possible configurations of it. Similarly, static connections between a fixed number of component interfaces are expressed in Verilog [TM91] by parameter bindings.

◦ **Communication integrity:** *Interfaces may communicate directly only if there is an architecture connection between the interfaces.*

This requires the architecture's connections to define all the direct communication between pairs of interfaces. It is possible for two unconnected interfaces to communicate through a third interface – this is called indirect communication.

◦ **Dynamicism:** RAPIDE *should be capable of modelling architectures of dynamic systems in which the number of components and connections may vary when the system is executed.*

◦ **Causality and Time:** RAPIDE *should be capable of expressing casual dependency and independency between behaviors of interfaces and connections, and their timing.*

These two requirements are forced by the wealth of dynamic and distributed systems where architecture definition and modelling has become a primary issue. Many event-based modelling languages provide simulation timestamps and determinisitic event interleaving. Concurrency is thereby expressed in simulation results, but dependence and independence of behaviors is not. Moreover, current ADL's cannot model dynamic systems.

◦ **Hierarchical Refinement:** RAPIDE *should allow both components and connections in an architecture to be replaced by (sub)architectures to form new architectures.*

This is a requirement that is needed to allow hierarchical design methodologies as currently practiced in many areas (e.g., hardware or protocols), to be applied to architecture modelling. Most programming and simulation languages support hierachical refinement of components, generally by allowing an interface to be implemented by a module containing a set of submodules. We require connections also to be refinable into architectures.

◦ **Relativity:** RAPIDE *should provide features for interpreting the behavior of one architecture as behavior of another architecture or an interface.*

This requirement is more general than, and complementary with, hierarchical refinement. When architectures, often for the same system, are defined at different levels of abstraction, they tend to differ widely in the kinds and numbers of components and the types of data being communicated. Current technology does not provide any means to explicitly define the relationships between such architectures. Here, we require RAPIDE to provide features for relating architectures. Such features have several applications – e.g., by relating architectures at different levels of abstraction, one architecture may serve as a constraint on the other. We illustrate this in Section VII.

B. Conformance to Architecture

A system "has" an architecture if it *conforms* to it; conversely a RAPIDE architecture is a *constraint* on systems. There are three basic conformance criteria:

1. **decomposition** : for each interface in the architecture there should be a unique module corresponding to it in the system (i.e., the component implementing that interface). [1]
2. **interface conformance**: each component in the system must conform to its interface. Since behavioral constraints can be part of RAPIDE interfaces, this conformance requirement is stronger than the syntactic interface conformance usually required by programming languages.
3. **communication integrity**: the system's components communicate directly only as specified by the interface connections of the architecture.

A system must conform to an architecture in order that the architecture can be used to predict the system's behavior or to decide various issues about maintaining and modifying the system (see [LVM]).

How to test or prove that a system conforms to an architecture is beyond the scope of this paper. There are style guidelines for using RAPIDE to help ensure communication integrity. One guideline requires components to access only their own interface constituents to communicate with other

[1] Mappings discussed later allow a more sophisticated decomposition between a single interface and a subsystem of components.

components. This helps to ensure that *all* direct communication between components is by interface connections. Also, the types of objects that can be communicated must be restricted – e.g., only immutable objects – to prevent communication trojan horses.

III. Causal Event Simulation

RAPIDE is an event processing language. Events are simply tuples of information containing, e.g., who generated the event, what activity was done, data values, the time and duration, etc. The semantics of RAPIDE are defined in terms of event processing — generating events, sending events from one component to another, and observing events. Components have the ability to generate events independently of one another. Asynchronous communication is modelled by connections that react to events generated by components and then generate events at other components; the events reacted to *cause* the events generated by a connection. Causality between events is also modelled by reactive behaviors of components. In addition, synchronous communication can be modelled by connections between function calls. The result of executing a RAPIDE architecture (a set of interfaces and connections) is a *poset* [2] showing the dependencies and independencies between events.

Let us start with some intuitive examples of how posets capture the semantics of communication architectures.

The components in figure 1 are airplanes and a control tower. Connections between them are depicted by arrows. The intuitive semantics of this picture are that airplanes can generate Radio events containing data which cause Receive events containing the same data that can be observed by the control tower. When this architecture is defined in RAPIDE and executed, a typical resulting poset is shown in figure 2. Here, events are depicted as nodes and dependencies between events as directed arcs. The poset shows that airplanes, A1, A2, A3, A4 have generated Radio events independently, and that each Radio *caused* a Receive event at the control tower, SFO; also the Receive events are independent, so SFO may observe them in any order, possibly concurrently. Timestamps and other information are also contained in events.

A variant of this architecture may require all communication to the control tower to be pipelined as shown in figure 3. Here, airplanes are connected to the control tower by a pipeline connector component which orders events it receives into a sequence which is then received by the tower.

When this architecture in figure 3 is executed, resulting posets show additional dependencies due to the pipeline connector; all Receive events are ordered into a linear dependency sequence. This implies that the Receive events are observed one-by-one at SFO in their dependency order.

The semantic differences between the broadcast and pipeline architectures are shown clearly by the posets in the two figures, 2, 4. Trace-based simulation languages do not capture event dependencies and would produce the same event trace for these two architectures.

[2] Partially Ordered Set of Events.

IV. Event Patterns

Event patterns are expressions used in defining behaviors of components, connections, and mappings between architectures. They are fundamental in representing dynamic architectures.

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

RAPIDE 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).

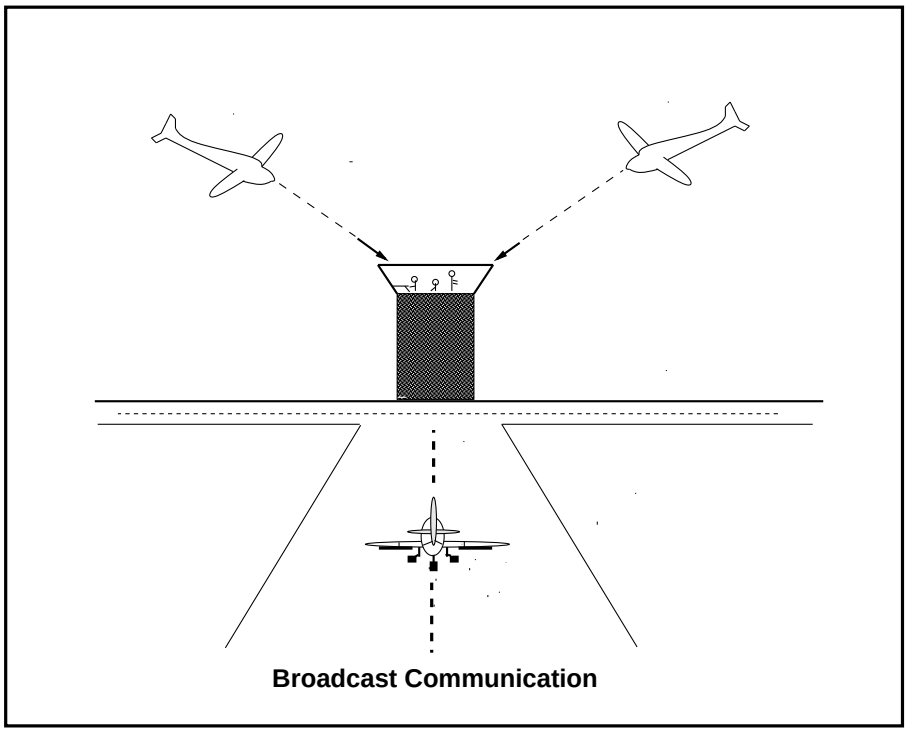

Fig. 1. A broadcast architecture.

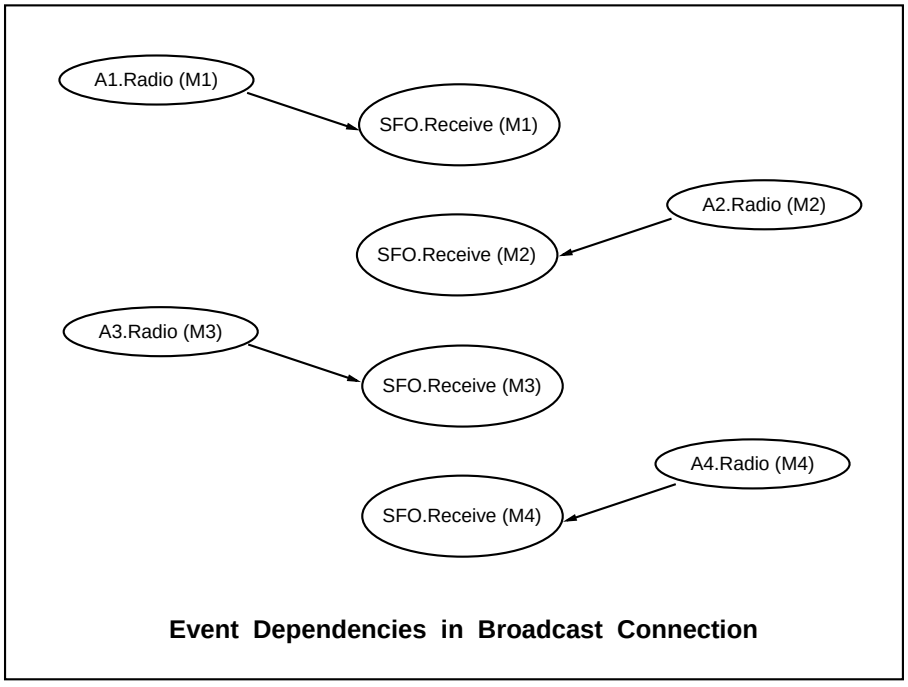

Fig. 2. Broadcast events.

The other kind of special variable is an *iterator*. An iterator is a universal quantifier over a type. Iterator names always begin with a "!" to distinguish them from ordinary variables and placeholders. Iterators are typed and declared the same way as ordinary variables. They differ from ordinary variables in that they may not be assigned values. A *pattern* containing an iterator is equivalent to the conjunction (**and**) of the instances of the pattern with the iterator replaced by each object of its type. Iterators are important in defining "fan out" connection rules (see Section VI).

The process of deciding if a given poset matches a pattern

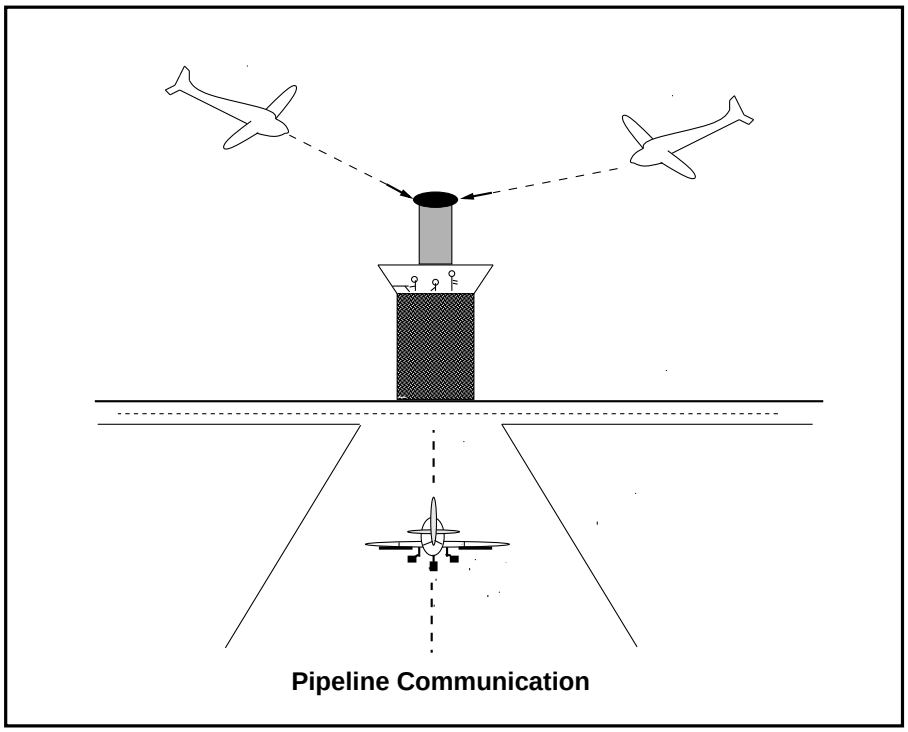

Fig. 3. A pipeline architecture.

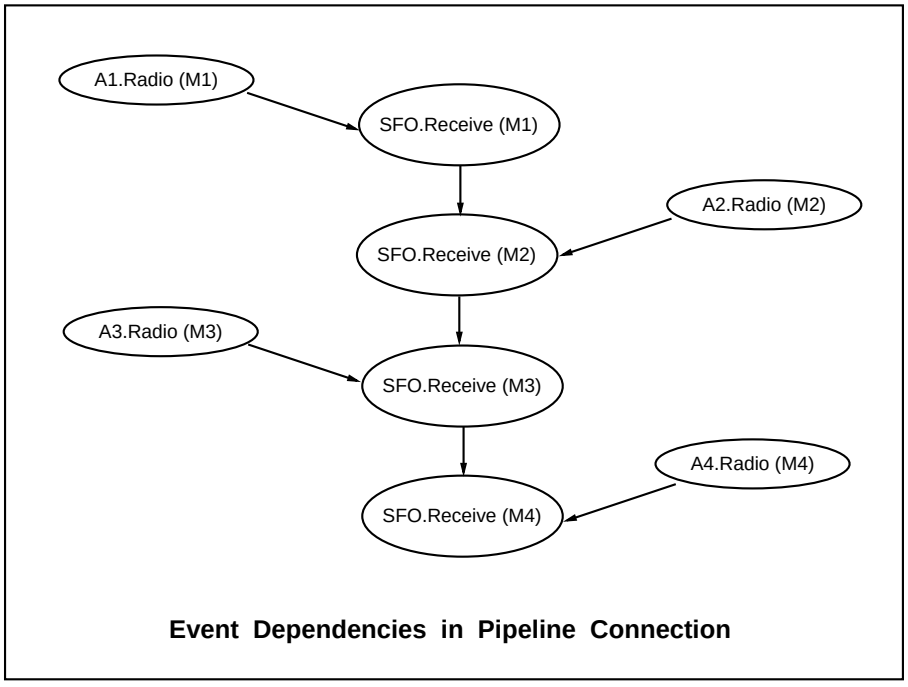

Fig. 4. Pipeline events.

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:* $P \,||\, P'$. 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 RAPIDE 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. *guarded pattern:* 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 V).

Examples

Simple Patterns

Here are some simple patterns and descriptions of the posets which would match them.

```
    A(?i) and B(?i);
--  An A event and a B event with the same parameter.
    A → B;
--  An A event and a B event which depends on the A event.
    A < B;
--  An A event and a B event which is temporally later
--  than the A event.
    A(?i) where ?i > 4;
--  An event whose parameter is greater than 4.
```

An iterator

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

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 interfaces to define behaviors, and in architectures to define connections. 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.

V. INTERFACES

An interface defines a type of components. It provides an abstract definition of externally visible behavior — i.e., the behavior that is visible to, and may be observed by, an architecture containing those components. An interface defines *(i)* the kinds of events that a component can observe or generate, *(ii)* the functions it provides to other components or requires from other components, *(iii)* its states and state transitions, and *(iv)* constraints on its external behavior. In general, components can be small static values (e.g., integers) or large dynamic systems with varying state (e.g., airplanes). Interface types provide *component abstraction* (Section II).

Structure of interface types

The syntax structure of interfaces is outlined in Figure 5. ("{ X }" means a list of 0 or more X's, and "[X]" means X is optional.) Standard Pascal-like types (Boolean, string, integer, array, record etc., but not pointers) are assumed here without definition. These are called the Procrustean types. [3] In RAPIDE *name types* (below) are used instead of pointers.

type_declaration ::=
    type *identifier* **is** *interface_expression* ';'
interface_expression ::=
    interface
        { *interface_constituent* }
    end [**interface**]

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

```
interface_constituent ::=
        provides { object_name_declaration }
|       requires { object_name_declaration }
|       in action { action_declaration }
|       out action { action_declaration }
|       service { service_declaration }
|       behavior
              [ declaration_list ]
        begin
              { state_transition_rule }
|       constraint pattern_constraint_list


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

service_declaration ::=
        name ['[' expr '..' expr ']']
              ':' [ dual ] interface_type_name

declarations ::=
        Procrustean type, object and function declarations,
        placeholder, iterator, and name declarations

state_transition_rule ::=
        trigger connection_op transition_body ';;'

connection_op ::= '=>' | '||>'

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

constraint ::= pattern ';'


transition_body ::= {state assignment}
        [restricted_pattern]
```

Fig. 5. Outline of *interface* syntax.

Many features are omitted from this overview. For example, interface type declarations can have type and object parameters (thus allowing polymorphic types), can inherit from other type declarations, and can refer to seperately compiled units.

Semantics of interfaces

An interface defines a type of object called a module. Modules that conform to an interface belong to its type. If an interface is a component of an architecture then a module of the interface's type can be a component of a system that conforms to that architecture. The module corresponds under the decomposition principle (Section II-B)to that interface. The architecture is called the *parent* of the interface, and the system is called an *instance* of the architecture.

First we describe how interfaces are used to define communication between modules of a conforming system.

An interface declares sets of *provides, requires, action* and *service* constituents. Those interface constituents are visible to the parent architecture. Other components of the architecture can be wired up (by *connections*, Section VI) to those interface constituents. Thus communication between interfaces is defined by the connections.

Object name declarations are declarations of objects of procrustean types (e.g, integers, arrays, etc.), interface types or function types. A module conforming to the interface must provide the objects and functions named in the *provides* constituents. Thus, for example, other components can be connected to call its provided functions. Conversely, a module may call *requires* functions of its interface and assume they are connected to call provided functions of other components. Connections between required and provided functions define synchronous communication.

An interface specifies the types of events its modules can observe and generate by declaring, respectively, *in* and *out actions*. Connections in the parent architecture can call the *in* actions of a module's interface, thus generating *in* events which the module can observe; conversely, the module can call its *out* actions, thereby generating events which the parent architecture can observe. Architecture connections between components' actions react to events generated by one component by generating events observed by other components – thus connections define asynchronous communication between components.

Any other constituents of a module (not in its interface) can be referred to only locally within the module. Conversely, a module can only refer to its interface constituents and its internal constituents. These visibility rules, together with parameter type restrictions (later) ensure communication integrity (Section II-B) of systems that are instances of an architecture. Thus, modules of a system communicate through the connections of the system's architecture.

Next we describe how the behaviors of interfaces execute as components of an interface connection architecture.

An interface optionally contains a *behavior* which consists of a set of types, objects, functions, and transition rules. Only procrustean objects are allowed. Objects declared in an interface model *state*. Transition rules model how modules of the type react to patterns of observed events by changing their states and generating events. A behavior is an abstraction of modules of that interface type.

An interface observes *in* events from the architecture. It reacts by executing its transition rules and generating *out* events which are sent to other components. An interface can also generate its own *in* events and observe its own *out* events. Similarly, an interface observes calls to its *provides* functions; the interface must declare functions in its behavior that are executed in response to these calls. When an interface calls its *requires* functions it depends upon the parent architecture to connect those calls to *provides* functions in other components.

There are two kinds of transition rules in behaviors. Those with the operator, =>, are called *pipes*; those with the operator, ||> are called *agents*. A pipe with a sequential pattern in its body specifies the behavior of a single thread of control, whereas an agent specifies the behavior of arbitrarily many threads (below).

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. A *provides* function body is treated as if it was a transition rule that is triggered by a function call; function calls are treated as events.

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

Next, the interface observes an event from the set of its events that have been generated and not yet observed. These events are queued awaiting observation. An event is observed by an interface at most once. If no events are queued, the module waits for one. Events are observed in a sequential order which is consistent with their temporal and causal orderings. [4]

Observing an event may *trigger* one or more transition rules. A rule R triggers if some subset of the events thus far observed but not yet used in triggering R match its pattern. If more than one set of observed events can trigger a rule, the earliest (in time and dependency) and largest is chosen. 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 one by one in some order.

The guards in pattern triggers are evaluated whenever an interface event is generated. Their values are associated with the event. In matching a pattern that has a guard (i.e., is restricted by a **where** condition – the guard), first the pattern is matched with the observed events and then the guard is evaluated. The value of the guard that is associated with the *last* event used in matching the pattern is taken as the value of the guard when the pattern is matched.

Executing a rule body consists of changing the state of the behavior part (by calling operations of objects declared there), generating the unique poset of new events defined by the instance of the restricted pattern, and adding the poset to the execution of the parent architecture. 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 rules to change any of the state objects that are referenced in executing the rule (e.g., in a guard in the trigger or in a parameter expression on the right side of the rule). In addition, if the rule is a *pipe*, then all of the generated events depend on all events generated by any previous triggering of the rule.

Execution of a behavior then continues with evaluating boolean triggers first, and then observing events, as above. Below is a schema for the above ordering of activities:

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

Any event may take part in triggering a given rule at most once, although it may take part in triggering several rules.

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.

The behavior of a module corresponding to an interface in an instance of an architecture is constrained to be consistent with both the interface's behavior and constraints – i.e., interface behaviors act as additional constraints on modules that conform to the interface. [5]

Services

A service names a group of constituents in an interface (note, it is not an object, it is a name.) Services provide a powerful notation for connecting large numbers of actions or functions (Section VI). Services also specify the types of (complementary) connections an interface "expects" from its parent architecture. So services specify, to some degree, the types of other components to which an interface expects to be connected.

A *service* S of type T in interface I declares that type I has the *provides* and *requires* functions and objects of T, and the *in* and *out* actions and services of T. To name them the name "S" must be appended with the usual "." notation before the name of the constituent. For example, if type T has an out action A, then "S.A" is a name of the out action A of I.

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.

A service S' is *dual* to S if its type is **dual** T. A dual type contains a set of constituents of I with the same names as constituents of T, but with dual roles: a *provides* function of T is a *requires* function of **dual** T, an *in* action of T is an *out* action of **dual** T, a type S service of T is a type **dual** S service of **dual** T, and conversely. Dual services of the same type in objects (usually of different types) may be connected together easily since they have complimentary constituents.

Name types

Instead of pointers, RAPIDE 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 O of the interface type T there is a corresponding object in the name type $\&T$ called the name

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

[5] In RAPIDE-1 modules can also be written in various programming languages.

of O. Each interface type T has an operator "&" defined on it which converts an object of T into the corresponding name in the name type, $\&T$. 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.

Only the parent architecture (Section VI) of T may dereference T's name. This restriction distinguishes name types from pointers; its purpose is to ensure communication integrity (Section II-B). An example of name types is given in Section VI.

Discussion

Component abstraction (Section II) is provided by interface types which include a powerful method of defining behavior. A single RAPIDE 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. Generally, many different modules or architectures will conform (Section VI) to an interface.

Communication integrity of architectures (Section II) is aided by style guidelines on the types of parameters of actions and functions, together with the visibility rules for modules and interfaces. Parameters of functions and actions should be of Procrustean types or name types. Consequently if a name of a component is passed, the receiver cannot dereference it to start direct communication with that component; only the parent architecture can communicate directly with its components. Style guidelines are not enforced by the RAPIDE compiler.

Behaviors are non determinisitic. In matching triggers of transition rules, events are selected one at a time and rules that trigger are executed one at a time in an arbitrary order. Because of shared state between rules, and since independent events may be selected in any order, the triggering on posets and order of execution of state transitions, as well as the parameter values of the events or function calls they generate, are all non deterministic.

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).

Complexity of Interfaces: Interfaces specify separately synchronous communication by function calls from asynchronous communication by events. Attempts to simplify interface design by using the same specification mechanism (e.g., *provides* and *requires* actions) lead users to incorrect expectations, both in subtyping of interfaces ([KLM94]), and in semantic equivalences between functions and actions. Interfaces also provide *behaviors* and *constraints* to satisfy *component abstraction* (Section II). Overall, RAPIDE interfaces are more complex than, say, class public parts in C++ or package specifications in Ada. Indeed, richer interfaces are needed for architecture definition since they must go beyond the traditional *information hiding* role of interfaces if they are to support component abstraction.

Examples

Example: Automobile controls.

```
interface AutoControls is
provides
   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 *provides* functions or *out* actions. An AutoControls component (an instance of the interface or a module of the type) must supply bodies or reactive rules for computing the functions. A user can call these functions, if its *requires* 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 observe *out* events if the architecture connects them to *in* events of the user.

Controls are specified as *in* actions. An AutoControls component 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.

AutoControls can output Warning_Light events with appropriate status data; the assumption is the parent architecture will do something useful with them.

Many different modules (or architectural designs) will conform (next section) to this interface.

Example: Specification of concurrent behavior.

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

Comments: This *agent* 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 observed by the AutoControl object itself. The other is an *out* event which will cause other events according to the architecture's rules.

The events generated by triggering this rule will depend on whatever events caused Speedometer > 55 (e.g., events causing some local state to change). An *agent* rule such as this one, does not impose any dependency between the events generated by different triggerings of the rule.

This rule abstracts a behavior of AutoControl modules – albeit one that most drivers wouldn't like.

Example: An RS-232 service

```
type   RS232 is interface
    out action TXD; -- Transmit Data.
    in  action RXD; -- Receive Data.
    out action RTS; -- Request to Send.
    in  action CTS; -- Clear to Send.
    in  action DSR; -- Data Set Ready.
    in  action DCD; -- Data Carrier Detect.
     ...
end RS232;

type Computer is interface
service S1, S2 : RS232;
    ...
end Computer;

type Modem is interface
service S : dual RS232;
     ...
end Modem;
```

Comments: RS-232 is a common interface used between computers and modems. It defines 25 signals, some of which are generated by the computer to the modem, and others from the modem to the computer. Here RS−232 is abstracted as an interface type with in and out actions corresponding to the 25 signals.

Using the service feature, a computer interface declares two RS−232 services. A modem interface declares a **dual** RS−232 service. Services in these interfaces express an important abstraction of the modules with these interfaces. Namely, the modules "expect" to be connected to other modules with RS−232 services, again illustrating support for component abstraction.

A computer and a modem can be connected in an architecture by a single connection statement, as shown in Section VI. This allows architectures with potentially large numbers of connections to be written with clarity and conciseness.

Note that a more ambitious interface would contain a behavior part defining RS−232 prototcols.

VI. Architectures

An interface connection architecture is a set of interfaces, a set of connection rules, and a set of constraints. Connection rules define relationships between events independently of any implementation; they are *communication abstraction* constructs (Section II). Connections are defined using event patterns. Event patterns provide the expressive power to define both static and dynamic architectures (Section II).

Syntax

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

Semantics

The optional **return** type name is the interface type of the architecture. An architecture defines a module of that interface type. If the return type is omitted, the empty interface type, Triv, is the default. Generally, in Rapide, architectures are parameterized, and are therefore *architecture generators* that, when called, return modules of the return type. Here, in order to focus on connection features, we have omitted parameterization.

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

declarations ::=
        components, placeholders and Pascal-like
        object and function declarations

connection ::=
        basic_pattern_list to basic_pattern_list ';'
        | basic_function_pattern to
                function_call_expression ';'
        | pattern connection_op restricted_pattern ';'
        | pattern connection_op component_generation ';'

basic_function_pattern ::=
        function_call_expression [where expression ]

connection_op ::= '=>' | '||>'

basic_pattern_list ::= basic_pattern
        | basic_pattern_list ',' basic_pattern

component_generation ::=
        new name

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

Fig. 6. Outline of the *architecture* syntax.

The Rapide architecture construct encapsulates both an interface connection architecture in which *all* components are interfaces, and instances of such an architecture, in which components may be module of the interface types. Types and components 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 interface types 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 or function calls, and creation rules define event conditions that lead to creation of new components.

A *connection rule* (Abbrev: *connection*) is composed of two patterns. The patterns are separated by a connection operator, (**to**, =>, ||>). As with transition rules, the left pattern of a connection is called its *trigger*. The right side

of a connection is called its *body*.

Connections are used to "wire up" components of an architecture as follows. A trigger must be a pattern of out events or requires function calls of components; a body must be a pattern of in events or provides function calls of components. A connection may also wire the architecture's interface to its components. In this case, the trigger is a pattern of *in events* and *provides* functions of the interface and the body is a pattern of *in* events and *provides* functions of components, or conversely, the trigger is a pattern of *out* events and *requires* functions of components and the body is a pattern of *out* events and *requires* functions of the interface.

The semantics of executing connections is as follows. Events or function calls are either generated by the components in the architecture or observed at the interface of 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. However, unlike transition rules, matching of triggers of different connections may take place independently or concurrently because there is no state shared between connections. 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.

The guards in the pattern triggers of connection rules are evaluated when an event is generated that can be observed by the architecture. Their values are associated with the event for future reference. During matching, if Pat is guarded by a **where** condition, the value of that guard that was associated with the last event to be selected in matching Pat is used as the value of the guard.

The number of guards that need be evaluated for any observable event can clearly be reduced in general. Any given event will be a potential participant in matching only a subset of the guarded patterns in the set of connection triggers. This is a compiler optimization.

Basic connections. A basic connection is a **to** connection 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 **to** 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.

Basic connections between functions. A basic connection defines an *alias* of a *requires* function of a component to a *provides* function of a component, and a sychronization at each call between the caller and callee. The following conditions must hold for a basic connection between functions to be correct: *(i)* the left pattern must match calls of the *requires* function and the right pattern must match calls to the *provides* function, *(ii)* the *provides* function must be a subtype of the *requires* function.

Evaluation of a call to the *requires* function triggers the connection. The resulting instance of the connection's right pattern must be a call to the *provides* function. The caller's execution is suspended, the call to the *provides* function is executed and any return object is returned to the caller as the value of the (triggering) *requires* function call.

By using guards in the triggering function call, the alias for a *requires* function can vary at runtime. If a *requires* function call has more than one alias, perhaps because a call triggers more than one connection, one of the return objects is the result.

Basic connections can also alias *provides* (or *requires*) functions of the architecture's interface to *provides* (or *requires*) functions of components, respectively.

Basic connections between services. A basic connection can be used to connect a service of a component to a dual service of a component. The connection defines a set of basic connections, one for each pair of constituents with the same name in the two services. It is bi-directional in the sense in each of the basic connections the *out* constituent is action or function name in the trigger and the *in* constituent is the action or function name in the body.

General connections. The semantics of a general connection, $Pat\ op\ Pat'$, where op is $=>$ or $||>$, are as follows. When Pat is matched, Pat' must define a unique poset (i.e., all placeholders in Pat' must be bound by the match). Then the connection is executed. The events in the instance of Pat' are generated together with the dependencies defined as follows:

(i) each event in Pat' depends on all events in the triggering poset,

(ii) each event depends on other events in Pat' as defined by the pattern, Pat',

(iii) if the connection operator is $=>$ (a pipe) then all the generated events depend on all events generated by previous triggerings of the connection.

The result is a new poset of *in* events of components and *out* events of the interface.

An architecture may be constrained by patterns in its constraint section. The sets of events generated by the architecture's interfaces and connections must match the constraint patterns. Constraints may, for example, require components to use a particular communication protocol. As discussed in Section V, constraints may be supported in different ways by various tools.

Conformance to an interface

If an architecture is bound to a non-trival interface it should *conform* to the interface. This means that :

1. calls to *provides* function names in the interface should result (if at all) in objects of the return type. To achieve this, the architecture should have basic connections aliasing *provides* function names in the interface to *provides* functions of its components, or alternatively it can declare an executable function body with the same name.
2. Any poset of interface events resulting from executing the architecture should satisfy constraints in the interface. That is, *in* events observed by the interface may trigger connections in the architecture, and result eventually in *out* events of the interface being generated by connections in the architecture. These *out* events will be related by dependencies and time to the *in* events, thus defining posets of interface events. The posets of interface events must satisfy the interface constraints.
3. An interface poset generated by an architecture must be a super poset of the poset generated by its interface behavior (i.e., a superset of the events with an identical dependency order on the common subset). In this sense, an interface behavior (Section V) acts as a constraint on an architecture of that interface type.

Discussion

Connection rules provide *communication abstraction* (Section II). They refer only to constituents (functions and actions) of interfaces of components and are independent of the modules implementing the components.

Basic connections are fundamental. A general connection is, in fact, an abstract interface expressed in a succinct notation. A general connection can be replaced in any architecture by a *connector* component (whose transition rule expresses the same connection relation between *in* and *out* events) together with basic connections between components and the connector.

Hierarchy (Section II) is provided by the ability to bind architectures to interfaces using connections, and by conformance criteria. Both interfaces and connection rules can be expanded into architectures (of lower level components).

Examples

Example: A basic connection.

```
  ...
  ?P : Person; ?B : Button;
connections
  ?P.Push(?B) to Button_Light_On(?B);;
```

Comments: This basic connection links pairs of events. An event of any person pushing a button triggers the rule and produces a new event denoting 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 and a particular control center as depicted in Figure 1, Section III. 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 generated, 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 is essentially a fan-in connection. It imposes dependencies between pairs of Radio and Receive events as shown in the poset Figure 2, Section III. In this figure, nodes are events and directed arcs represent dependency. The poset also shows that the Receive events are all independent. This implies that they could be observed by the control center concurrently.

To illustrate how posets distinguish between different architectures, we simply change the connection rule in the previous example to be a pipe instead of a basic connection.

Example: Pipelining air traffic control.

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

Comments: We have changed the air control sector architecture so that all radio events are observed at the control center through a pipe rule. Essentially, a pipe is used to connect airplanes and the control center as shown in Figure 3. Now all airplanes communicate with SFO by a single pipe instead of by broadcast.

Pipe connections order the events they generate into a linear dependency sequence. In this case, each generated event, SFO.Receive, depends on the ?A.Radio event that triggered the rule, and all previous SFO.Receive events.

The SFO.Receive events are all in a linear dependence chain, as shown in Figure 4. This means that messages can only be received at the control center one-by-one in their dependency order.

The semantic differences between the broadcast and pipeline architectures are shown clearly by the posets in the two figures, 2, 4.

Example: Using RS-232 to connect computers and modems

```
with Computer, Modem;
architecture Office is
      PC : Computer;
      Mod: Modem;
      ...
connect
      PC.S1 to Mod.S;   -- bi-directional flow of events.
      ...
end Office;
```

Comments: Following the RS−232 example (Section V), connecting a computer component PC to a modem component Mod in an office architecture requires a single connection rule between their RS−232 services (Figure 7). This connection expresses a set of 25 basic connections between pairs of RS−232 constituents with the same name; in each connection the **out** constituent is the action in the trigger pattern.

Example: An Intelligent Network Architecture.

An intelligent network is a dynamic architecture which works by passing the names of components to other components. The restrictions on name types (Section V) ensure that the Network Architecture's connection rules define all pairs of components that may participate directly in data transfer.

There are three types of components: providers, clients, and brokers. The numbers of components of each type can vary (although in our example they are fixed). Providers can Register with a broker, indicating the service they provide. The broker stores the names of providers (which are contained in Register events as the *actor* element) and the jobs they can perform.

A client can ask a broker for a provider of a job by calling Find_Provider. As a result the *name* of a provider is supplied by the broker to the client – not a provider itself. A client can then use the name of the provider to request a job. The client cannot dereference the name and get the provider, and then call the provider directly. A client must communicate a request for a job with the provider's name to the architecture. The parent architecture is the only module that can dereference the provider's name (in its connection rules) and generate the request to the provider.

```
interface Provider is
   provides function Do_Job(J : Job; P : Parameters)
                     return data;
   out action Register(J : Job);
   ...
end;

interface Client is
   requires   -- a list of functions
        function Request_Job(J : Job; P : Parameters;
                             Pn : &Provider)
                         return Data;
        function Find_Provider(J : Job)
                         return &Provider;
   ...
end;
```

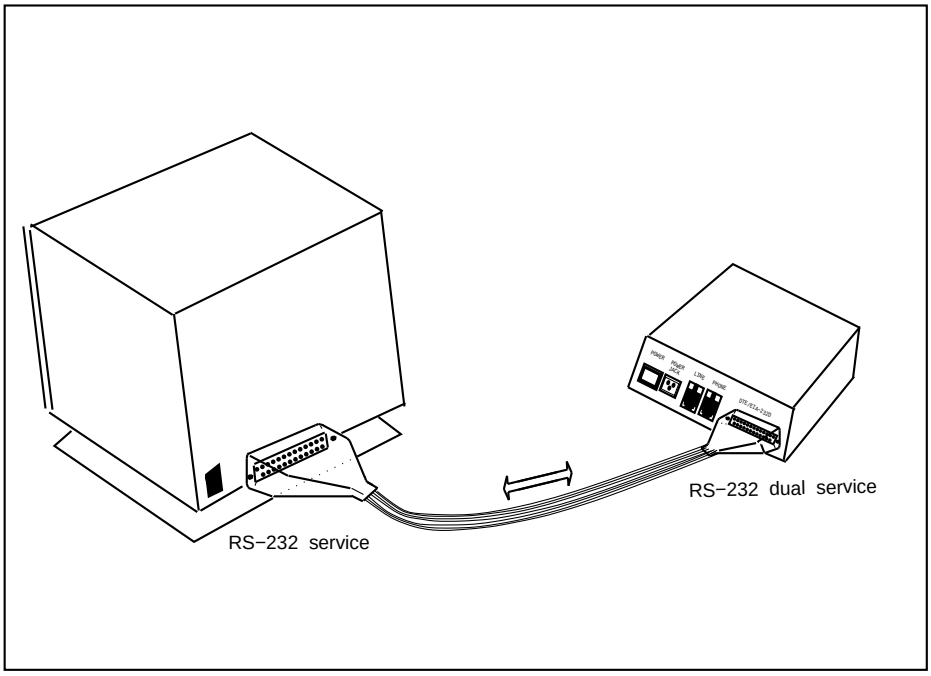

Fig. 7. An RS-232 connection.

```
interface Broker is
   provides  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;
```

The broker, client and provider components are declared in the network architecture (Figure 8). All of the direct communication among pairs of components is defined by three basic connection rules.

```
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;
connect
   ?P.Register(?J) to
          NTT.Register_Provider(?J, &?P);;

   ?C.Find_Provider(?J) to
          NTT.Provider_Lookup(?J);;

   ?C.Request_Job(?J, ?param, ?N) to
      *?N.Do_Job(?J, ?param);;
end Network;
```

The first rule connects any provider's **out** action Register with the broker's **in** action Register_Provider. This is a fan-in rule connecting many providers to a single broker; asynchronous actions are used so that providers don't block.

The second rule defines an alias between a call to a client's Find_Provider function and a call to the broker's Provider_Lookup function; the return value is the name of a provider. Again, it is fan-in rule connecting many clients to a broker. Synchronous function call is used because clients will need to wait for a return value. (If there were multiple brokers, this rule could be generalized to alias a call to Find_Provider to any broker, the result being one of the returned names.)

The third rule aliases a call to any client's Request_Job function to a call to the Do_Job function of the

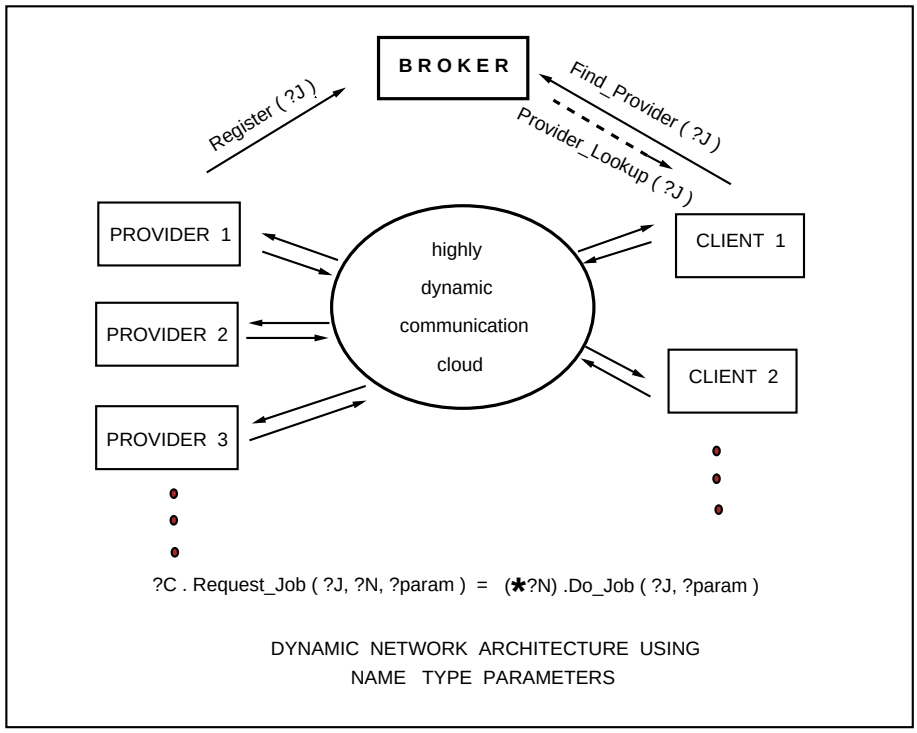

Fig. 8. An intelligent network.

provider whose name the client supplies. The rule dereferences the provider name in the client's call and calls the provider's Do_Job function. The rule expresses $Num\_Clients * Num\_Providers$ connections between pairs of components.

Communication integrity implies that these rules define all of the direct communication in the Network between pairs of clients, providers and the broker. They allow us to draw some conclusions without knowledge of the modules implementing these components. For example: *(i)* since none of the connection rules can be triggered by the broker, we know that the broker does not initiate any communication. *(ii)* Only providers can cause the broker's Register_Provider events. *(iii)* Clients do not communicate data directly to clients, and providers do not communicate data directly to providers. *(iv)* transfer of data from a provider to a client has to be caused by a function call from a client (i.e., limited junk mail rule).

These properties of the network communication could not be deduced if the integrity of the connection rules (Section II-B) could be subverted by passing components, or pointers to components, as parameters of their functions and actions. The deductions are valid even though component names are being passed between the components. This is because only the architecture Network may dereference the names of its components (as is done in the third connection rule). Thus, for example, the broker cannot use the name of providers it stores to communicate with them directly. This would not be true if pointers were used in place of names; the broker could dereference a pointer to a provider and call the provider's functions. In an ordinary programming language we would have to examine the implementations of all of the components.

VII. Mappings

Event pattern mappings (Abbrev: mappings) can be used to define how one architecture is related to another one, or how an architecture is related to an interface. 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.

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 .

Semantics. Maps have visibility into the declarations of their domain (**from** architecture) and range (**to** architecture). Mapping rules can trigger on events happening at the top level inside the domain, and generate events at the top level inside the range. Rules in a mapping have the same semantics as transition rules in components (or non basic connections in architectures) *except* that they do not define any causal relation between the triggering events (from the domain) and the events they generate (in the range).

```
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]
```

Fig. 9. *map* syntax.

A. Example: A simple microprocessor

This section gives an example of two RAPIDE architectures for a simple microprocessor and an event pattern mapping from one to the other. It shows some of the complexities of "real life" applications that require the power of an event pattern language.

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 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. [6] By using VAL 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 event pattern 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 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.

[6] 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!).

Below we give the RAPIDE 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 relationships between a set of RTL events that would trigger the map, resulting in a load event.

The RAPIDE 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 mapping 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 [Tea94a].

```
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;
```

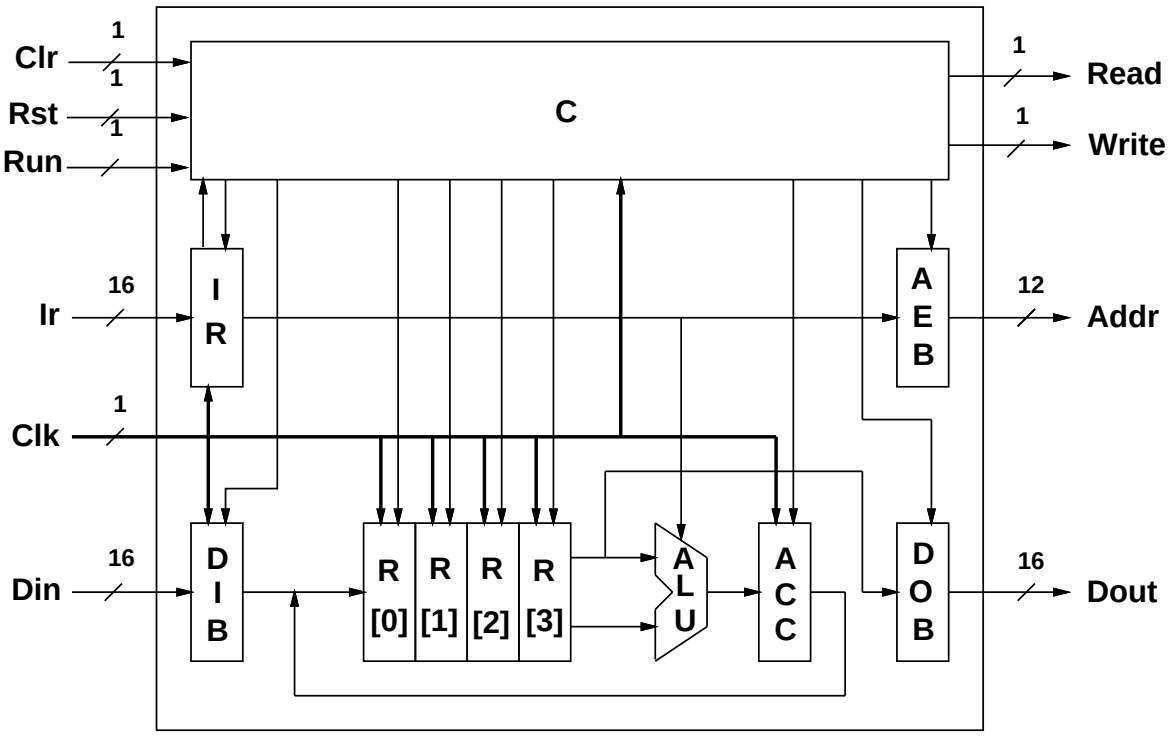

Fig. 10. CPU register level architecture.

```
with TypePreamble, Two_Output_Register_Logic;
interface Two_Output_Register is
   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;

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 actionirCE(val : bit),
             accCE(val : bit),
             accOE(val : bit),
             dinOE(val : bit),
             irRST(val : bit),
             readOE(val : bit),
             writeOE(val : bit);
    service  reg [0..3] : dual 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, ALU in the figure is a logic_unit, C is a Controller, and R is a bank of four Two_Output_Registers. The RAPIDE 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),
```

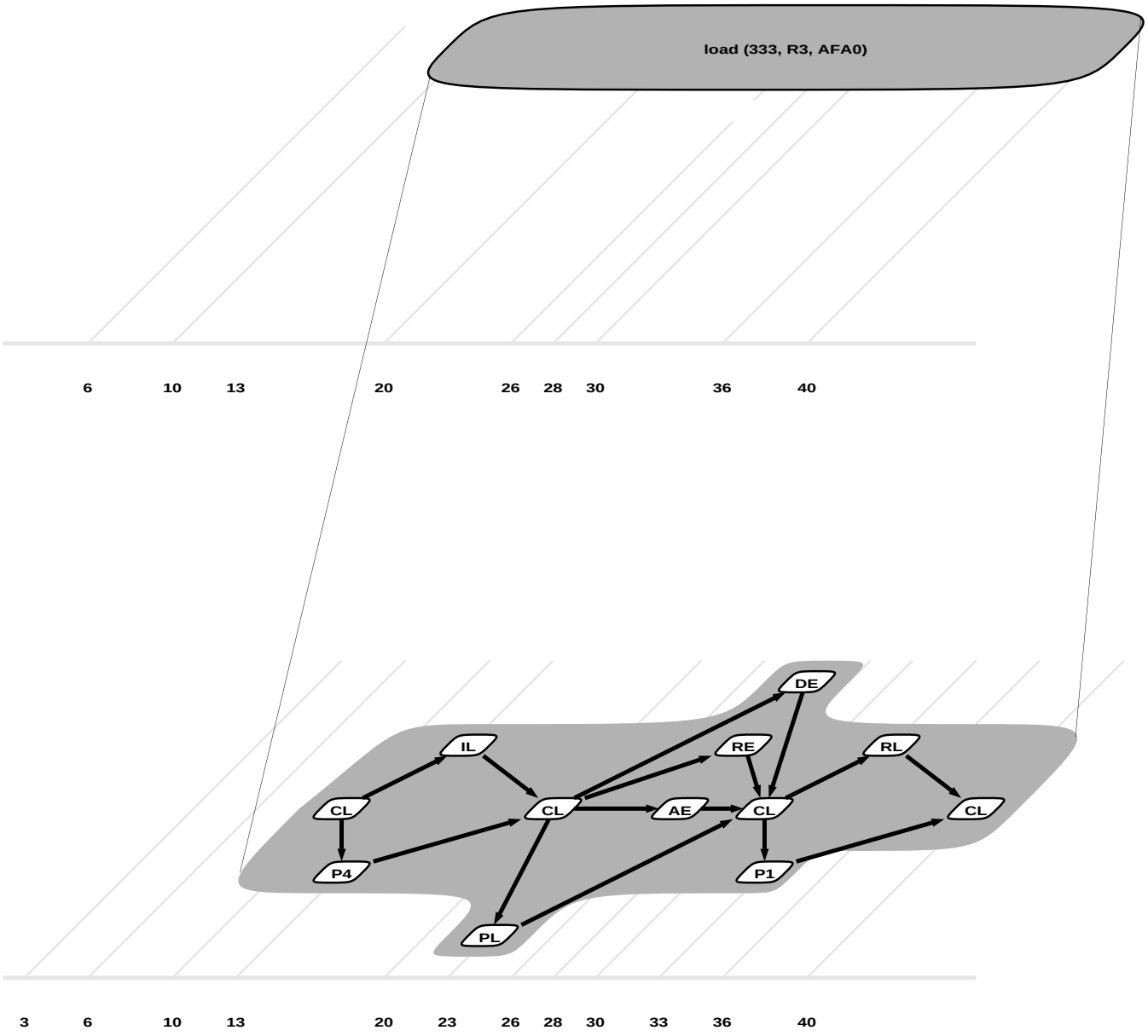

Fig. 11. Map from register events to Load instruction.

```
            Addr(val : bit12),
            Read(val : bit),
            Write(val : bit);
end RTL_CPU_Interface;
```

For simplicity, if the parameter profiles of the patterns on both sides of the "**to**" 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;

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

  -- architecture connections.
```

```
   C.dinOE to DIB.OE;

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

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

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

   IR.dout(?b16) to 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 to DOB.OE, AEB.OE;
   C.irCE to IR.CE;
   C.irRST to IR.Rst;

-- output bindings.
   C.readOE to Read;
   C.writeOE to Write;
   AEB.dout to Addr;
   DOB.dout to 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 guards, 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 RAPIDE 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 RAPIDE 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 RAPIDE) to the wires. Here, RAPIDE basic connections allow us to define the "connecting wires" directly. Also, RAPIDE 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 mapping SimRef. The SimRef maps finite sets of RTL events (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. In this example, the SimRef trigger uses only the timing relation between events; it does not use causality.

```
-- 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;

-- Event pattern 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 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 clock 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.

VIII. History and status of Rapide

Rapide has evolved from several sources: *(a)* VHDL (for event-based and architecture concepts), *(b)* ML [MTH90] and C++ [ES90] (for type systems), and *(c)* TSL [LHM+87] (for event patterns and formal constraints on concurrent behavior expressed in terms of patterns of events).

The evolutionary steps can be summarized as follows. First, Rapide departs from previous event-based languages in adopting the partially ordered set of events (*poset*) execution model in place of linear traces of events. Simulations in Rapide produce posets. The concept of posets has been described by Fidge [Fid88], and Mattern [Mat88], and was probably extant in the database literature since the 1970's. The first studies to our knowledge of the feasibility of implementing simulation languages to produce poset executions, and to check them for constraint violations, were done independently on the Rapide-0.2 project [Bry92],[MSV91], and the OEsim project [AB91].

Secondly, there are many event-based reactive languages in existence; a few of the ones that we have studied are VHDL [VHD87], Verilog [TM91], LOTOS [BB89], CSP [Hoa78], and Esterel [BCG87]. Most of these languages have simple forms of event patterns for triggering processes — e.g., VHDL has sensitivity lists which are disjunctions of events, and LOTOS has basic events with predicate guards. In Rapide we have introduced more powerful event patterns, as is appropriate for specifying posets. Event patterns play a basic role in features for defining both reactive behaviors and formal constraints. Pattern matching concepts go back at least to the *unification algorithm* in Resolution theorem-proving [Rob65], and their use in AI languages is typified in Planner [Hew71] and Prolog [CM84].

Third, the concepts of interface in Ada (package specification) and VHDL (entity interface), both of which we extended with formal annotations in prior work [Luc90], [ALG+90], have been extended to interface types in Rapide with a capability to specify concurrent behavior. In these earlier languages, interfaces were not types. Interface types can inherit from other types using object-oriented methods, and are related by a notion of structural subtyping [KLM94]. Much research is yet to be done on interface design and the interplay between subtyping and well-formedness of architectures.

Fourth, VHDL provided us with the best previous model of "architecture" which is a wiring of interfaces, totally separated from a binding of interfaces to implementations (configurations). Structural connections in VHDL, expressed by port maps that bind the ports of component interface instances to signals in an architecture, are generalized in Rapide to event pattern connection rules. This feature allows dynamic architectures.

Finally, event patterns are used in Rapide to define mappings between architectures, thus allowing for hierarchical and comparative simulation, as described in our earlier work on VAL+ [GL92].

At present a simulation toolset for Rapide-1.0 is being tested on industrial examples of software and hardware architectures of moderate complexity. The simulator produces posets. Analysis tools display simulator output graphically, automatically check output for violations of formal constraints, and allow simulations to be animated on pictures of the architecture that is being simulated. Input tools are being constructed to allow architectures to be input in various formalisms and translated to Rapide. The eventual goal is to develop an industry scale toolset.

Acknowledgements

Our thanks are due to many people who have collaborated in experiments with various versions of Rapide and its toolset. Especially, our thanks are due to John McHugh and John Munson (University of North Carolina and TRW), and members of the Rapide design team: John Kenney, Doug Bryan, Walter Mann, Alexandre Santoro, and Larry Augustin (Stanford), Sigurd Meldal (University of Bergen), and Frank Belz and Holly Hildreth (TRW).

References

[AB91] Tod Amon and Gaetano Borriello. OEsim: A simulator for timing behavior. *ACM/IEEE Design Automation Conference*, 28(1):656–661, June 1991.

[Ada94] Intermetrics Inc., Cambridge, Mass. *Ada 9X Reference Manual*, June 1994. ANSI/ISO Draft International Standard.

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

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

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

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

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, second edition, 1984.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

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

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

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

[Gro91] The Object Management Group. *The Common Object Request Broker: Architecture and Specification*. The Object Management Group, revision 1.1 edition, December 1991.

[Hew71] Carl Hewitt. *Description and Theoretical Analysis of Planner*. PhD thesis, MIT, 1971.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[KLM94] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *Proc. 21-st ACM Symp. on Principles of Programming Languages, Portland*, 1994.

[LHM+87] David C. Luckham, David P. Helmbold, Sigurd Meldal, Douglas L. Bryan, and Michael A. Haberler. Task sequencing language for specifying distributed Ada systems: TSL-1. In *Proceedings of PARLE: Conference on Parallel Architectures and Languages Europe. Lecture Notes in Computer Science. Number 259, Volume II: Parallel Languages*, pages 444–463, Eindhoven, The Netherlands, 15–19 June 1987. Springer-Verlag.

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

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

[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*, 21(3):253–265, June 1993.

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

[Mat88] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proceedings of Parallel and Distributed Algorithms*. Elsevier Science Publishers, 1988. Also in: Report No. SFB124P38/88, Dept. of Computer Science, University of Kaiserslautern.

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

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Tea94a] Rapide Design Team. *The Rapide-1 Executable Language Reference Manual*. Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[Tea94b] Rapide Design Team. *The Rapide-1 Specification Language Reference Manual*. Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[Tea94c] Rapide Design Team. *The Rapide-1 Types Reference Manual*. Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

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

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

[XoD92] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berkshire RGI 1AX, U.K. *Distributed Transaction Processing: The Peer–to–Peer Specification*, December 1992. Snapshot.