DRAFT
Guide to the Rapide 1.0 Language Reference Manuals

Rapide Design Team
Program Analysis and Verification Group
Computer Systems Lab
Stanford University

July 17, 1997

Preface

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

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

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

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

This Guide is intended to provide a short introduction to three topics:

1. what is meant by "*architecture*",
2. the concepts of RAPIDE for representing and prototyping architectures,
3. an overview of the RAPIDE Language Reference Manuals (Abbrev. LRMs).

We have also included a complete syntax for the language framework.

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

The general discussion of concepts of architecture is followed by a short overview of how the Rapide language and tools support architecture modelling, that is building models of system architectures, simulating architectures and analyzing simulation results.

Rapide is an event-based language and simulation toolset. We give a short but quite detailed overview of *events*, causal histories of events, and Rapide computations.

There are five Language Reference Manuals (LRMs) for Rapide, each devoted to a different sublanguage of the framework. Chapter 4 explains the rationale behind this organization. Chapter 5 gives a simple top-down graphical overview of the different language constructs and how they are related.

Availability of Documents

All Rapide LRMs are preliminary drafts. A *Primer* on Rapide is planned for the near future. Publications on Rapide, and Technical Reports on case studies of medium size system architectures from Industry and DoD, are available either from the Computer Systems Lab. at Stanford or from the Rapide Web page:

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

As our documentation for Rapide improves, the description of concepts will be moved into a *primer* about programming in Rapide. This document will become merely an overview of the LRMs.

Contents

List of Figures

Chapter 1

History and the Design Team

1.1 Roots

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] [MMM91]. 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.

1.2 History

RAPIDE was proposed to the Advanced Research Projects Agency (ARPA) in 1990 as a design for a systems prototyping language with supporting tools for simulation and analysis. This proposal was submitted as a teaming arrangement between Stanford University and TRW Inc. by David Luckham (Stanford) and Frank Belz (TRW). The team roles were initially that Stanford would provide language design and tools, and TRW would provide industry evaluation and criticism by applying the language and tools to modelling and simulation of DoD and Industry systems.

The initial proposal envisaged and proposed substantially the language breakdown into sub-languages for types, executable modules, architectures and constraints that is presented in the present set of LRM's. Discrete causal event semantics, with a rich event pattern language for constructing both executable reactive prototypes and formal constraints, was a major feature of the proposal. The ability to model and specify true concurrency behavior by patterns of causally related events was seen as a critical element in advancing beyond hardware modelling languages of the time, to something appropriate for large scale distributed systems. The use of event pattern mappings, which had been tested previously by Luckham and Gennart in the context of VHDL, was proposed as another step beyond current event-based simulation technology.

The proposal was initially funded and supported under the ARPA Prototech program, and later on, also by the ARPA Domain Specific Software Architectures (DSSA) and Software Composition programs. AFOSR has supported research into algorithms underlying poset generation, and applications of RAPIDE to modelling avionics and other distributed systems. The earliest publications were by Luckham and Belz concerning an early version of the language called "Reality". But this name was so disliked by the Prototech program manager, W. S. Scherlis, probably quite rightly, that it was later changed to RAPIDE (the moderately fast French trains).

A preliminary toolset supporting a subset of the language, RAPIDE-0.2, was implemented in the first two years of the project. This subset was specifically aimed at exploring architecture modelling techniques and implementation algorithms. It contained the architecture constructs and a very simple subset of the predefined types [1]. Small prototypes could be simulated with this toolset, but it did not (and was not intended to) scale to larger models.

RAPIDE-1.0 was finalized following the RAPIDE-0.2 project. Perhaps "finalized" is too strong a word since elements of the language are still considered "experimental", and the constraint and pattern languages in particular, are undergoing an extensive redesign for applicability outside the realm of RAPIDE simulations.

[1] A subset of those types found in all programming languages, called the "Procrustean types" by Bill Scherlis

1.3 Rapide Design Team

Over five years the membership of the Rapide design and implementation team has seen some changes, but, as is essential for a project of this nature and complexity, a substantial core of the team has remained constant, even from the earlier days of Anna and TSL. Several members graduated with M.S. and Ph.D. degrees during the project, and one graduating member has left and returned. We have also been fortunate to have the collaboration of Prof. Sigurd Meldal (University of Bergen and Stanford) over a number of years on both the TSL and Rapide projects, and Prof. John Mitchell (Stanford) on the Rapide type system.

1.3.1 Team Members

Present: *(Stanford):* **David Luckham, Walter Mann, John Kenny, James Vera, Sigurd Meldal, Neel Madhav, Alex Santoro, Francois Guimbretierre, Yung-Hsiang Lu, Woosang Park, Ernest Lam.**

(TRW): **Frank Belz, Holly Hildreth.**

Past: *(Stanford):* **John Mitchell, Larry Augustin, Doug Bryan, Dinesh Katiyar, Sriram Sankar, Alex Hsieh, Rob Chang.**

(TRW): **Mike Rowley, Paul Corneil.**

1.3.2 Activities

The design team leader and principle investigator is David C. Luckham. The TRW principle investigator is Frank Belz. Luckham, in collaboration with Belz and members of the design team, has promoted and encouraged the design vision of Rapide as explained in the previous section 1.2. Within this broad vision, many technical design and implementation issues were needed to make the vision a reality. Various issues have been solved by individual team members. Special design study groups for various language components have included: (present team members) Walter Mann, James Vera, Sigurd Meldal, John Kenny, Francois Guimbretierre and Yung-Hsiang Lu; (past team members) Doug Bryan, John Mitchell, Larry Augustin, Neel Madhav, Sriram Sankar, and Dinesh Katiyar. The suggestions, encouragement and constructive criticism from Frank Belz and the TRW team have been invaluable throughout.

The Rapide project is not without flaws. Luckham is responsible for what may come to be viewed in the future as errors in the overall conception and implementation of Rapide or lack of planning in the execution of the project.

The Type System for Rapide is based upon ML, and was designed in collaboration with John Mitchell. Sriram Sankar and Dinesh Katiyar wrote the Types LRM and implemented the basic type checker.

The implementation of the Rapide simulator, including both the Rapide-0.2 version and the Rapide-1.0 version, was undertaken by the core implementation team of Walter Mann and Doug Bryan, with contributions in defining the internal tree representation by Larry Augustin and James Vera, and in the areas of parsing, pattern matching and constraint checking, by Larry Augustin, John Kenny, and Neel Madhav. The front end parsing and static semantic checking is due to Walter Mann. The code generator, which translates from Rapide internal tree representation to C++ is due to Doug Bryan, aided by other team members including James Vera, Sigurd Meldal, John Kenny and Sriram Sankar.

The Run Time Scheduler (RTS) was implemented by James Vera and Doug Bryan. The algorithms for tracking causality between events were the subject of a publication by Vera, Meldal, and Sankar, [MSV91], and are still being improved for efficiency by Vera.

Various implementations of pattern matching have been undertaken by Doug Bryan, Neel Madhav and John Kenny, and is a continuing topic of research, particularly in its applications to constraint checking.

The RAPIDE Separate Compilation tool was implemented by Larry Augustin and Walter Mann.

The RTS uses a multi-threaded Garbage Collector from Xerox Parc that interfaces with a version of the POSIX threads interface as supported by Unix systems.

RAPIDE tools for analyzing simulation results include a Poset Browser (POB), the Rapide Animation System (Raptor), and a constraint checker. The POB was implemented by Doug Bryan and has proved so useful that a more general implementation for application to event generating systems in general is being implemented by Francois Guimbretierre and Ernest Lam. The Raptor implementation by James Vera and Alexandre Santoro, based on Tcl TK, has provided a much needed "bridge" between a complex poset output from the simulator and human visualization of the behavior that the poset records. Raptor is continuing to be expanded in its capabilities to handle new graphical metaphors, and to apply to other event generating systems (e.g., Verilog). The effectiveness of the present constraint checker for a subset of the constraint language, implemented by Doug Bryan, has been demonstrated in several case studies. The constraint language is being redesigned and the new checker will be implemented by the constraint language subgroup (below).

Case Studies are a crucial element in RAPIDE development. While small examples are important in teaching and publishing techniques of architecture prototyping, larger scale examples are needed to test tools performance and scalability issues, and to report user concerns. Larger scale examples that have been subject to ongoing prototyping in RAPIDE include:

1. Sparc Version 9 64 bit Instruction Set Architecture by Woosang Park and Alexandre Santoro.
2. X/Open Distributed Transaction Processing standard by John Kenny with collaboration from D. Luckham.
3. The ARPA Prototech Navy AEGIS architecture, a public version of part of the Navy's Aegis system, and alternative dynamic architectures, by TRW: Frank Belz, Holly Hildreth, and Mike Rowley.
4. The IBM DSSA ADAGE architecture, by TRW: Frank Belz, Holly Hildreth and Paul Corneil.
5. The DoD Advanced Distributed Simulation High Level Architecture, by Francois Guimbretierre, Yung-Hsiang Lu, and David Luckham.
6. the RAPIDE RTS, by James Vera.

These studies are the subject of various technical reports and publications. A video presentation of the AEGIS study is available from TRW.

There is a separate Language Reference Manual (LRM) for each RAPIDE language component: Patterns, Types, Executable Modules, Constraints, and Architectures. The style of these manuals varies somewhat according to the individuals responsible for writing them. In any event, they are not intended to be teaching documents, but rather guides to details of language semantics, and possibly aids to implementers. Everybody contributed to these documents, either as an author or by finding errors and omissions. Furthermore, it is true to say that the LRMs and the Toolset are being constructed together, sometimes in collaboration with each other, and sometimes not. The principle authors (to date) of each manual and supporting documents are as follows:

- Types: Sriram Sankar and Dinesh Katiyar.
- Predefined Types: Dinesh Katiyar, Neel Madhav, Doug Bryan, Walter Mann, Sriram Sankar.
- Executable Modules: Doug Bryan, James Vera, Larry Augustin, Walter Mann.
- Patterns: Walter Mann.
- Constraints (under revision): Walter Mann, John Kenny, Sigurd Meldal, Francois Guimbretierre, David Luckham.
- Architectures: David Luckham, James Vera, Sigurd Meldal.
- Combined Syntax: Larry Augustin, Walter Mann.
- Overview: David Luckham.

There are also Unix MAN pages for each tool which are maintained by the tool implementors.

This work continues. Case studies and toolset improvements are continuing at this time to support the Design Team, TRW and other outside groups in the very active research area of applying causal event simulation, together with animation and constraint checking, to *(i)* evolutionary development of complex system architectures, and *(ii)* standards definition and testing processes.

Further, the code of the RAPIDE simulator is being documented by Wolfgang Polak (of Polak Inc.). This will allow others to start more extensive and robust implementations.

Certain sublanguages and tools are being prepared for application to event generating systems in general, outside of the RAPIDE simulator:

- The Patterns and Constraint sublanguages are being redeisgned by a subgroup consisting of: Walter Mann, John Kenny, Sigurd Meldal, Francois Guimbretierre, Yung-Hsiang Lu, and David Luckham.
- The Raptor animation system is being extended by Alex Santoro and James Vera.
- A new causal event history browser is being designed and implemented by Francois Guimbretierre and Ernest Lam.
- A new constraint checker is being designed by Walter Mann, John Kenny, Sigurd Meldal, Neel Madhav and others.
- A RAPIDE Primer is being written by David Luckham with the help of everyone else.

Chapter 2

What Is Architecture?

Many large software systems are distributed object systems. Their components are modules (or "objects") that can compute independently but must also collaborate by interchanging data and synchronizing. *Architecture* is a plan of a distributed object system showing what types of modules are components of the system, how many components of each type there are, and how its components interact. Quite often "architecture" is no more than a pictorial repesentation. The purpose of RAPIDE is to provide facilities (language and tools) to allow an architectural plan to be used both to guide the "wiring-up" of the components of a system, and, more importantly, to prototype the behavior of the system before effort is put into building the modules (i.e., the system's *components*). Our goal is to support new methods of utilizing architectures to compose sets of objects into systems by building upon current object-oriented technology. Indeed, so-called "middleware technologies" based upon object-oriented concepts, such as CORBA [Gro91] for example, are currently being developed which allow distributed systems of interacting modules to be wired together over the Internet. But the architectural plan showing how the components are to be wired up is still lacking in such technologies.

The term, *architecture*, is very widely used even though it has no generally accepted definition at the present. If one asks, "what is architecture", the answer is usually a common maxim that architecture consists of interfaces and connections. Any details about the nature of interfaces and connections are left vague.

This Chapter gives a brief outline of two precise concepts of architecture. It discusses how architectures differ from systems of objects, and when a system can be said to have, or conform to, an architecture. Finally, it gives a brief outline of some of the features of RAPIDE that are designed to model and prototype architectures, and to help analyze their behavior.

2.1 Object Connection Architectures

"Every system has an architecture" is a popular belief. Figure 2.1 shows one view of the architecture of a system. The rectangles are interfaces of objects, the objects themselves are boxes below the interfaces, [1] and the spaghetti-like arcs denote an object "using" a feature declared in the interface of another object. "Uses" here can mean function call as in C++ or Ada, or it could be an asynchronous message passing mechanism such as a task entry call in Ada. The architecture consists of the interfaces and the "uses" links, and the "system" consists of the interfaces, modules and links. This kind of architecture is called an *object connection architecture* because the connections are from modules to (interface features of)

[1] I.e., the objects are the modules (also called components) of the system.

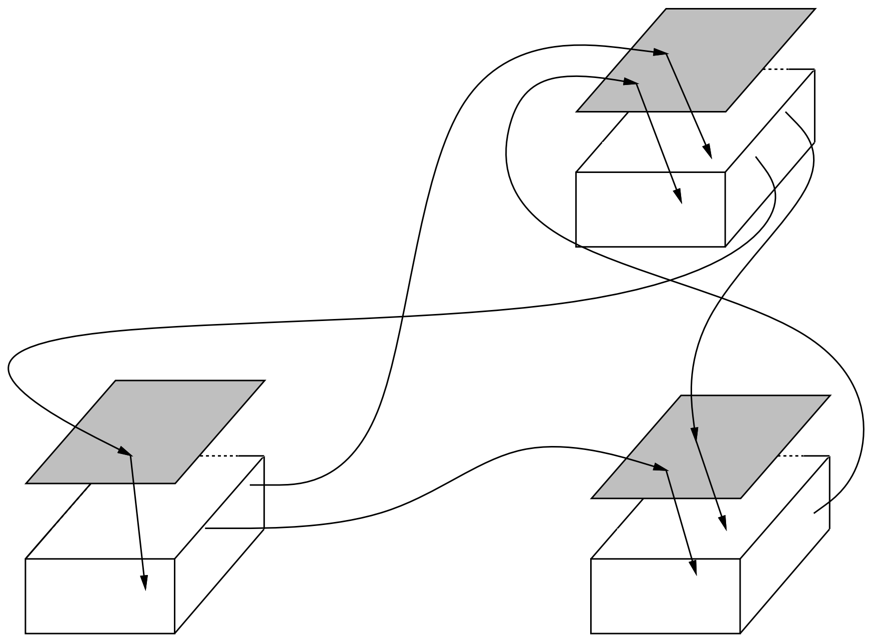

Figure 2.1: An object-oriented program and its architecture

modules. Object-oriented programming languages provide interfaces (of objects) sufficient to build systems with object connection architectures, see also, e.g., CORBA IDL [Gro91].

We can build object connection systems in Rapide, although we don't advise it for reasons given below. Here's an example that is almost self explanatory since it could be done in any object-oriented language.

Example: *An object connection system in* Rapide

```
type Parser is interface
provides
      function Initialize();
      function FileName() return String;
end Parser;

type Semanticizer is interface
provides
      function Semantize(Tree);
      function Incremental_Semantize(Context :Tree, Addition : Tree);
end Semanticizer;



type CodeGenerator is interface
provides
       function Generate(Tree);
end CodeGenerator;
```

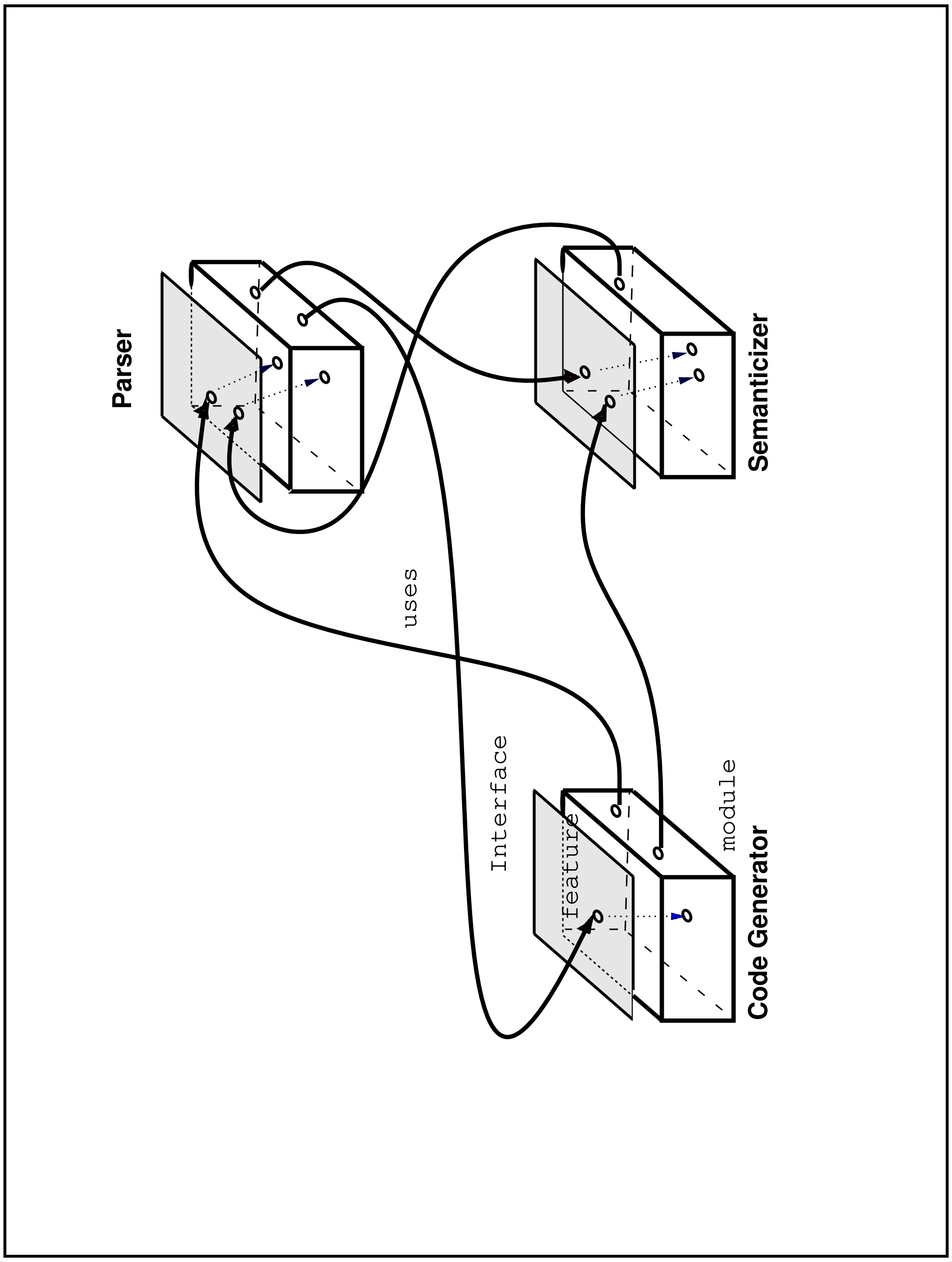

Figure 2.2: A Compiler as an object connection system

```
module PaserGen(S : Semanticizer; C : CodeGenerator) return Parser is
    function Initialize() is ... end;
    function FileName() return String is ... end;
begin
    ...
        S . Semantize(...);
        C . Generate(...);
    ...
end PaserGen;


module SemantizeGen(P : Parser) return Semanticizer is
    function Semantize(Tree) is ... end;
begin
    ...
        P . Initialize();
    ...
end


-- CodeGen module is omitted; it will contain calls to
-- parameters for Parser and Semanticizer modules.


module Compiler return CompilerInterface is
    -- Non linear visibility of declarations.
    Par : Parser is ParserGen(Sem, CG);
    Sem : Semanticize is SemanticizeGen(Par);
    CG  : Code_Generator is CodeGen(Par, Sem);
begin
    ...
end Compiler;
```

Commentary:

This is a typical example of an object connection system represented in RAPIDE, but it could just as easily be in C++.

The interfaces for Parser, Semanticizer, and CodeGenerator modules are declared separately from the module generators that define objects with those interfaces. Each interface declares the functions that its objects provide to other objects. Objects can call the provided functions in the interface of an object. These calls allow objects to communicate data and to synchronize. The compiler contains a component of each interface type, Par, Sem, and CG, each of which has the other components as parameters. So when the compiler is elaborated, the components will call each other as illustrated in Figure 2.2. For example, the semanticizer Sem will call the Initialize function of the parser Par.

The architecture consists of the interfaces and the function call graph of the compiler. The interfaces together with the call graph represent the connections between the modules. The components of the compiler communicate according to this architecture. They might, for example, execute as coroutines, or as concurrent objects. The correct functioning of the compiler might well depend upon Sem calling the parser Initialize at certain points. Suppose we tried to redesign the module SemanticizeGen, say to improve incremental semantics. The new module must provide the functions defined in the Semanticizer interface, but the interface does not require it to call the parser's Initialize, as does the present Sem. So, if we substitute the new semanticizer for the old one, it may satisfy the requirements expressed by the semanticizer interface but the compiler may not function properly.

□

In an object connection architecture the connections (e.g., function calls or message sending commands) are buried in the modules as illustrated in Figure 2.1 and in the modules in the compiler example. This has a number of disadvantages.

- First of all, the system and its architecture are not clearly separate entities. Although the interfaces can be defined before the modules, the connections are only defined when the modules are built. So, the architecture has likely evolved as the system was built and was not predefined as a plan or outline of the system.

- Secondly, the architecture does not remain invariant under changes to the system, even very simple changes. For example, if the system is modified by replacing one module by another, the architecture will change if the new module uses different features of other components than were used by the old module — i.e., the connections will change.

- Thirdly, object connection architectures cannot be used to check correctness of changes to the system. An interface in object connection architectures need only define which features a module with that interface provides to other component modules of the system; an interface does not define the features that are required from other modules. Consequently, a module conforming (in a sense unspecified here) to an interface cannot be replaced by another module that conforms to the same interface with the guarantee that the system will continue to work as before. The new module can conform to the interface without using the same features of other components as the old one. This could lead to the system with the new module not working because it is critical that a particular feature of some other module be used (as mentioned in the commentary to the compiler example).

Finally, there is a subtle issue about the use of parameters in object connection systems. Some people think that parameters in an interface [2] can play the role of *requires* declarations (as described in the next section). The thinking is that by parameterizing the interfaces in the example above instead of the module generators, we would achieve the effect of *requires* features. These people are mistaken. A generic parameter in an interface specifies a degree of freedom – e.g., "a bag of any-type-of-thing". A generic parameter in an interface does not specify a feature of objects with that interface that requires binding at runtime (possibly dynamically) to a similar feature supplied elsewhere – e.g., "an airplane needing a navigation beacon, maybe different beacons at different times". More importantly, the "use" (such as function call) of a generic parameter is buried in the object itself, as in the example above. Requires features used in the definition of interface connection architectures cannot be replaced by generic parameterization, as we shall see.

[2] As exmplified by Ada generic parameters, or template parameters in C++.

2.2 Interface Connection Architectures

A different concept of architecture is the *interface connection architecture*, so called because all communication between modules is explicitly defined by connections between interfaces — no longer are connections buried in the modules, but instead they are defined between the features in interfaces. Figure 2.3 shows this kind of architecture.

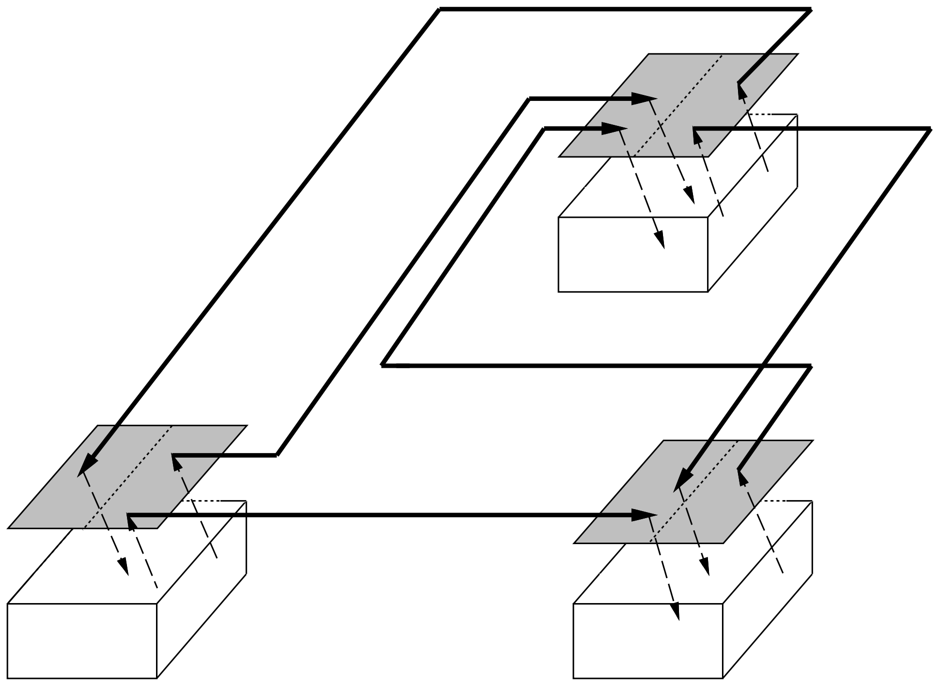

Figure 2.3: An interface connection architecture and conforming system

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

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

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

- *connections* between interfaces define relationships between the required features of interfaces and the provided features of the interfaces. The simplest kind of connection is *identification* between a required feature and a provided feature. [3] Identification connections have the effect that whenever a required feature is used then the connection invokes the provided feature in its place. More general kinds of connections allow sets of required features to be connected to sets of provided features.

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

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

We note that interfaces can contain executable behaviors (their effect is discussed in Section 2.5). Connections are also executable in the sense that whenever their required features are invoked, they result in execution of the provided features that they connect to. Consequently, in RAPIDE an interface connection architecture can be executed (or simulated) before modules (see figures 2.3, 2.4) are programmed for its interfaces.

Example: *An Interface Connection Architecture with asynchronous connections*

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

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


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

Commentary:

Here is a simple example of an interface connection architecture. There are two interface types, Producer [4] and Consumer. They define the interfaces of producer and

[3] Also called a *basic* connection.

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

consumer objects. These interfaces define asynchronous communication features called *actions*. The Producer interface contains two actions, an **out** action Send and an **in** action, Reply. Objects of this type can generate Send events and can receive Reply events. The Producer interface also contains reactive behavior transition rules. The first one triggers on a Start event (which all objects receive when they are elaborated) and reacts by generating an **out** event, Send(1). The second behavior rule triggers on Reply events containing integer data provided the data is less than Max, and generates a Send event containing the next integer.

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

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

The architecture defines the communication between the two components in terms of the actions in their interfaces. The architecture can be refined by introducing detailed modules for Prod and Cons. The refined system is called an *instance* of the interface connection architecture. As long as the new modules communicate only by calling the actions in their own interfaces, they will communicate in the instance only as defined by the connections in the architecture. The instance conforms to the architecture if its modules behave consistently with the behavior rules defined in their interfaces, and preserve its communication (see Section 2.3). Conformance implies that many properties of the system are defined by its architecture. For example, conformance implies that the instance will have the property that an interval of integers, $0, 1, 2, \ldots, Max - 1$ is communicated between the producer and consumer, according to a protocol whereby there is an acknowledgement from the consumer before the next integer is sent by the producer.

□

As another example, we can "re-architect" our simple compiler to have an interface connection architecture.

Example: *An interface connection architecture for the compiler*

```
type Parser is interface
provides
      function Initialize();
      function FileName() return String;
requires
      function Semantize(Tree);
      function Generate(Tree);
end Parser;
```

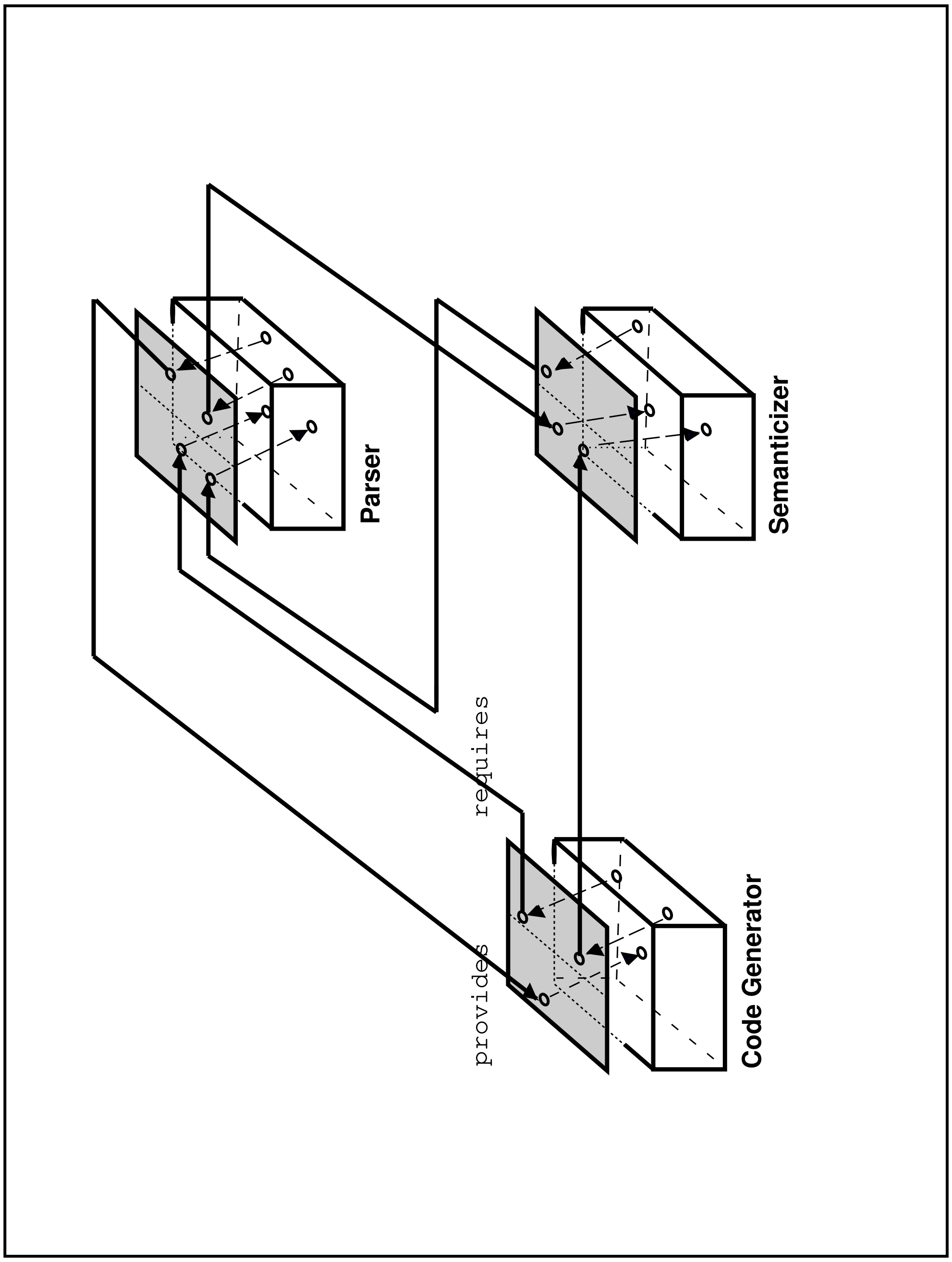

Figure 2.4: An interface connection architecture for the compiler

type Semanticizer **is interface**
provides
    function Semantize(Tree);
    function Incremental_Semantize(Context :Tree, Addition : Tree);
requires
    function FileName() **return** String;
    function Initialize_Parser();
end Semanticizer;

```
type CodeGenerator is interface
provides
     function Generate(Tree);
requires
     function Initialize_Parser();
     function Semantize(Context : Tree; Addition : Tree);
end CodeGenerator;
```

```
module PaserGen( ) return Parser is
     function Initialize() is ... end;
     function FileName() return String is ... end;
begin
     ...
     Semantize(...);
     ...
     Generate(...);
     ...
end PaserGen;
```

```
module SemantizeGen( ) return Semanticizer is
     function Semantize(Tree) is ... end;
begin
     ...
     Initialize_Parser();
     ...
end

-- CodeGen module is omitted; it will contain calls to
-- to the required functions in its interface.
```

```
architecture Compiler return CompilerInterface is
      Par : Parser is ParserGen( );
      Sem : Semanticize is SemanticizeGen( );
      CG  : Code_Generator is CodeGen( );
connect
      ?Tree, ?Tree' : Tree;
   Par.Semantize(?Tree) to Sem.Semantize(?Tree);
   Par.Generate(?Tree) to GC.Generate(?Tree);
   Sem.FileName() to Par.FileName();
   Sem. Initialize_Parser() to Par.Initialize();
   GC.Initialize_Parser() to Par.Initialize();
   GC.Semantize(?Tree, ?Tree') to Sem.Incremental_Semantize(?Tree, ?Tree');
end Compiler;
```

Commentary:

What is shown here is an instance of an interface connection architecture for our compiler in which modules have been assigned to its components. Functions in the component's interfaces are connected by the connection rules of the architecture. The connections are defined at the level of the interfaces, independently of any of the modules. We could have defined behaviors in the interfaces instead of assigning modules, thus showing an uninstantiated, executable interface connection arhcitecture. But we wish to also illustrate how the communication defined by the connections is maintained in instances (below).

Here the type interfaces for the compiler's components now define *requires* functions as well as *provides* functions. Requires functions are the functions that modules with those interfaces may assume are supplied by the environment (or system). These interfaces give an indication of context in which the modules can operate.

The architecture, Compiler declares its components, and defines connections between their interfaces: each connection connects a requires function in one interface with a provides function in another interface. Type correctness of connections is preserved. Connections here are "identification" connections (called *basic connections* in RAPIDE). For example there is a connection between the Par.Semantize function, which is a required function of Par and the Sem.Semantize function, which is a function provided by Sem. Each time Par calls its Semantize the connection will trigger and invoke a call with the same parameters to the Semantize function provided by Sem. This semantics is similar to remote function call. Par will block until a result is returned, so synchronization between caller and callee is implied.

Module generators call their own interface requires functions instead of calling directly the functions of other objects. [5] This discipline ensures that communication will be defined by the connections between the interfaces in the architecture, and not by individual modules. So, the architecture can be a plan for the communication between components of the system.

□

An interface connection architecture can be used to analyze properties of a system that conforms to that architecture — conformance is discussed in Section 2.3.

[5] As is done in object connection architectures.

For example, we can determine potential users of a provided feature in an interface simply by inspecting the architecture connections. Interfaces whose requires features are connected to the provided feature are interfaces of modules that are potential users. So connections are used to bound the search. For example to determine the users of function Semantize provided by the Semanticizer Sem, we check if there is a connection from a required function of either the parser Par or code generator, CG.

Note that type compatibility is a necessary condition for a connection to be correct, and is a compiletime check of the connections. [6] A connection from either of the requires functions, Semantize, in Par or CG, would satisfy this condition – a call to either of these would supply the Tree parameter needed for the successful evaluation of the provided function in the Semanticizer; however, a connection from, say the requires function Initialize_Parser in CG, could not be correct because that function is not a supertype of the provided function.

To determine if a potential user actually uses an interface feature we would need to analyze the constraints in the interface of the user — and assume that modules satisfy the constraints in their interfaces. In an object connection architecture we have to inspect all modules to find potential users because only provides features are specified in the interfaces.

An interface connection architecture can also be used to check if two modules for the same interface can be interchanged so as to preserve a requirement of the system's behavior. We must assume conformance to the architecture (Section 2.3), and also that the constraints in the interfaces and architecture are detailed enough to logically imply the behavioral requirement that is at issue.

For example, supose we want to replace the Sem module with a new one as before, and we are concerned about its effect on initializing the parser Par. The Semanticizer interface has a required function, Initialize_Parser, that is the same type as the provided function, Initialize, in the parser interface. A correct connection between those two functions is possible. If such a connection does not exist, then neither the old nor new Sem can initialize Par (since conformance implies that all connections are defined in the architecture). So the system must function correctly without initializing Par. If there is such a connection, then we inspect the constraints in the Sem interface. If they imply that a call must or must not be made, then both the new and the old Sem modules are obliged to satisfy the constraint – and we can be sure that the parser initialization is treated the same before and after an exchange. If there is no constraint about calling Initialize then we assume that the system will function correctly in either case.

To summarize the discussion, we argue that common system manipulation problems can be resolved in systems with interface connection architectures by analysis of only the architecture's interfaces and connections. This is not the case in object connection systems. However, using the architecture to make inferences about the system assumes that the system conforms to the architecture (see Section 2.3). Conformance can be difficult to check in general, perhaps even undecidable, depending upon the semantic constraints of the architecture. So it is important to develop methods of building systems that guarantee conformance to the architecture. The most important advantage is that interface connection architectures can be defined *before* a system is built, and used both to a prototype the behavior of conforming systems, and as a plan from which to implement conforming systems.

2.2.1 Dynamic Architectures

Although Figure 2.3 depicts connections between the interfaces as static wires, interface connection architectures in RAPIDE can be *dynamic* architectures. Typically, a *static* architecture

[6] A provides function must be a subtype of a requires function that is connected to it, see RAPIDE 1.0 Types Language Reference Manual.

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

Example: *A dynamic architecture*

```
type Airplane is ...
type  Control_Center is ...
type  Msg is ...
type EnRouteCenter is interface
     action in Accept(X : Airplane);
            out HandOff(X : Airplane);
end EnRouteCenter;

architecture FremontSector return EnRouteCenter is
     SFO : Control_Center;
     ...
connect
     ?A : Airplane; ?M : Msg;
     Accept(?A) => link(?A);
     ?A.Radio(?M) where ?A.InRange(SFO) ||> SFO.Receive(?M);
     HandOff(?A) => unlink(?A);
end FremontSector;
```

Commentary:

Assume the interface types of Airplane, Control_Center, and Msg are already defined. The interface for Airplanes, for example, may define that they can generate Radio events, provide a Position attribute (i.e., a function), etc.

The interface type for EnRouteCenter declares **in** and **out** actions that allow an EnRouteCenter to receive and send aircraft as parameters of **in** and **out** events. So new airplanes (modules) are continually being received and handed-off.

FremontSector is an architecture generator. When it is called it returns an architecture of type EnRouteCenter.

The FremontSector architecture defines the communication between the interfaces of airplanes and the interface of a Control_Center module by connection rules. The visibility rules of Rapide require the architecture to Link to an airplane that has been received as data of an Accept event before it can observe the events that airplane is generating. The first connection **link**s an airplane whenever an Accept event with that airplane is received. Events generated by the airplane will then be received by the AirControlSector. The second connection defines the following event-based communication:

> The rule triggers whenever any airplane (a match for ?A) generates a Radio event containing a message ?M and the InRange predicate of that airplane is true when the radio event is generated. ?A is bound to the airplane and ?M to the message. Then the rule calls SFO's Receive action, resulting in generating a Receive event with the same message that SFO receives. . The connection triggers only when an airplane is in range.

Finally, whenever the Sector hands off an airplane it **unlink**s from it, so communication from the airplane is no longer received.

This is a dynamic interface connection architecture. It defines communication between varying numbers of airplanes and a ControlCenter. The second connection rule defines a conditional broadcast between all airplanes and the control tower. It is essentially a fan-in connection. The architecture does not define any communication between airplanes. So, if modules assigned to SFO and various airplanes in a instance of the architecture conform to it, we can conclude that airplanes do not communicate directly with one another.

□

2.3 Conformance of a System to an Architecture

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

From now on we will assume that "architecture" means *interface connection architecture.*

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

1. **decomposition** : for each interface in the architecture there should be a unique module corresponding to it in the system (i.e., the component implementing that interface). [7]
2. **interface conformance**: each component in the system must conform (as described below) to its interface. Since RAPIDE behaviors and constraints can be part of interfaces, this conformance criterion is, in general, stronger than the syntactic interface conformance usually required by programming languages.
3. **communication integrity**: All communication between components is constrained by the architecture. Two components cannot communicate with each other directly unless a connection (possibly conditional on runtime parameters) between their interfaces is defined by the architecture.

Conformance of modules to interfaces in programming languages usually only requires a module to contain features that match names and parameter signatures of features in their interfaces — a simple compiletime check. In RAPIDE conformance of modules to interfaces requires satisfying both behavior and semantic constraints. In general, determining if a module conforms to an interface with semantic constraints is an undecidable problem, although there are many useful cases where it can be decided by practical methods. Developing tools for testing architecture simulations for conformance to constraints, and proving correctness of architectures, is a challenging activity at present. The RAPIDE toolset supports testing for interface conformance by both compiletime and runtime checking (see Section 2.6). In the future tools for applying proof techniques to interface conformance may be added.

Although determining interface conformance in the presence of behaviors and constraints is a tougher problem than the usual syntactic signature requirements between interfaces and modules, it allows us to conclude much more about the modules of a system. Interface constraints

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

allow us to specify modules sufficiently to ensure that any two modules conforming to an interface can be interchanged without changing the behavior of the system. Also, if the connections in an architecture are correct — i.e., the constraints on a provided feature logically imply the constraints on a required feature connected to it — then instances of the architecture where interfaces have been assigned modules conforming to those interfaces will also have correct connections.

There are many strategies one may adopt to try to ensure that a system satisfies communication integrity. For example, one may adopt restrictions on the coding of modules (called a *style guides*). A possible style guide could be that a module should be constrained to only communicate with other modules of the system, or its parent architecture (i.e., the architecture of which it is a component), through its own interface, as shown in (Figure 2.3). This style helps to ensure that only the communication defined in the architecture takes place between modules. [8] Sufficient conditions for communication integrity involve *(i)* restrictions of the RAPIDE visibility rules (see Chapter on Visibility in Executable LRM), and *(ii)* restrictions of the types of parameters of actions and functions so that particular types of objects cannot be passed between components.

2.4 The Role of Constraints in defining architectures

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

Here is a simple example of a constraint on the type Semanticizer. It requires any semanticizer module to always call its requires function Initialize_Parser before it does anything else.

Example: *A requirement on the Semanticizer for the compiler*

```
type Semanticizer is interface
provides
     function Semantize(T : Tree);
     function Incremental_Semantize(Context :Tree; Addition : Tree);
requires
     function FileName() return String;
     function Initialize_Parser();
constraint
     match Start –> Initialize_Parser'Call –> Any();
     ...
end Semanticizer;
```

Commentary:

Here, the constraint restricts the events that may happen at the interface of a semanticizer module. Intuitively, the constraint requires that the poset of all events that happen at the interface must *match* its pattern:

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

> The complete execution must consist of a Start event [9] which causes a call event to the Initialize_Parser function causally followed by any set of events.

It does not say anything about events outside the interface.

The constraint requires any semanticizer to call its requires Initialize_Parser function before it does anything else, and indeed, anything it does must be causally after this first event. So in the previous example of a compiler architecture, any semanticizer will always initialize the parser because a semanticizer must always call its Initialize_Parser function and the architecture connects that call to the parser's Initialize function.

□

Interface constraints can specify input/output conditions on functions, and other simple kinds of requirements; we omit examples of these here in order to focus on the sequencing aspects of event pattern constraints.

The following example of an architecture constraint refers back to the producer/consumer architecture example in Section 2.2.

Example: *A constraint on the architecture of Producer/Consumer.*

```
architecture ProdCon() return SomeType is
    Prod  : Producer(100);
    Cons  : Consumer;
connect
    (?n in Integer)
    Prod.Send(?n)  =>  Cons.Receive(?n);
    Cons.Ack(?n)   =>  Prod.Reply(?n);
constraint
    observe from Prod.Send, Cons.Ack
        match ((?X:Integer)(Prod.Send(?X)  ⟶  Cons.Ack(?X)))↑(⟶ *);
    end;
end architecture ProdCon;
```

Commentary:

The constraint applies only to the Send and Ack events generated by the producer and consumer. The **observe** part is called a *filter*. Its purpose is to specify a subset of events in the execution of the architecture — in this case, the list of events following **from** specifies all Send and Ack events in the execution. The constraint applies to that subset. The subset must satisfy the pattern after the **match**. The subset of Send and Ack events must be a set of pairs consisting of one Send causally followed by an Ack, (indicated by the $(Send \rightarrow Ack)$ pair). All the pairs must form a single causal chain of any length (indicated by the exponential notation, $\uparrow(\rightarrow *)$, similar to regular expressions, after the pair). The constraint requires the parameters of an Ack and Send in a pair to be the same, but different pairs can have different parameters.

So, the constraint describes an execution that conforms to a protocol requiring each Send from the producer to cause an Ack from the consumer with the same parameter, and the producer's next Send must await the Ack for the previous Send.

[9] all modules receive a Start event upon elaboration.

The constraint places restrictions not only on the architecture's connections, but also on the modules Prod and Cons. The semantics of the connections imply that each Send causes a Receive event, and each Ack causes a Reply (see Section 2.5.1). But the Cons module must react to the Receive by causing an Ack. In other words, the Ack can't be independently generated by "guessing" when a Receive arrives. Similarly, when the Prod gets a Reply it must react by causing the next Send if there is one.

This constraint does not say anything about the other events in an execution of the architecture.

□

The air traffic control sector architecture (Section 2.2.1) illustrates an issue in power of expression of constraint languages.

Example: *Constraining the unknown.*

```
architecture FremontSector return EnRouteCenter is
      SFO : Control_Center;
      ...
connect
      ?A : Airplane; ?M : Msg;
      Accept(?A) => link(?A);
      ?A.Radio(?M) where ?A.InRange(SFO) ||> SFO.Receive(?M);
      HandOff(?A) => unlink(?A);
constraint
    never (?e, ?e' in event(); ?a1, ?a2 in Airplane)
                ?e |> ?e' where ( ?e.from_module = ?a1 and
                                  ?e.owner_module = ?a2 and
                                  ?a1 /= ?a2 );

end FremontSector;
```

Commentary:

The constraint expresses a property of the FremontSector architecture that airplanes never communicate directly with one another. In this example the airplane interface is unspecified. We assume it is as general as possible, so it will be a supertype of any airplane type. This means that any specific type of airplane can be substituted for the type of airplane in FremontSector. We have no idea what kinds of events might be available for communication between airplanes, e.g., radio, radar, rockets, etc. The constraint must exclude direct communication by *any* kind of events. So, the constraint uses placeholders of the predefined **event** type which match any event. It expresses:

> The execution must never contain two events such that the first one is the direct cause of the second, the first is generated by an airplane and the second is received by a different airplane.

The pattern, ?e |> ?e', expresse "direct cause" by matching two events if they are causally related and there are no intermediate events that are causally between them. Events (see Chapter 3) contain information that can be accessed, such as the module that generated the event (the from_module), or the module that received the event (the owner_module).

□

2.4.1 Reference architectures

The term, "reference architecture", come up frequently in the literature on architectures. Generally it seems to mean a standard to which the architectures of systems in some class of systems should be related. It is never stated how a reference architecture differs from any other kind of architecture. And the kind of relationship that should hold between the reference and a particular system is usually also left vague.

Constraints allow a style of constraint-based architecture specification. Bounds on the behavior of a system architecture are specified by a set of constraints; executable behaviors are omitted. The resulting architecture is not necessarily executable, but instead defines a *reference standard* to which systems (or other architectures) claiming to have that architecture must conform by satisfying its constraints. Here is a constraint-based version of the Producer/Consumer example chosen here for its simplicity.

Example: *A Constraint-Based Architecture for Producer/Consumer*

```
type Producer(Max : Positive) is interface
    action out Send(N : Integer);
    action in  Reply(N  : Integer);
behavior
    -- the behavior part is empty.
constraint
    observe from Send, Reply
        match ( (?N in Integer)
                (Reply(?N) -> Send(?N+1)where ?N<Max))↑(→ *) ->
                (Reply or Empty);
    end;
end Producer;
```

```
type Consumer is interface
    action in Receive(N : Integer);
    action out Ack(N : Integer);
behavior
    -- the behavior part is empty.
constraint
    observe from Receive, Ack
        match ( (?N in Integer)
                (Receive(?N) -> Ack(?N)))↑(→ *)
    end;
end Consumer;
```

```
architecture ProdCon() return SomeType is
    Prod : Producer(100);
    Cons : Consumer;
connect
    -- connection rules are omitted.
constraint
    observe from Prod.Send, Cons.Receive
        match ( (?N in Integer)
                    (Prod.Send(?N) →
                    (Cons.Receive(?N) or Empty)))↑(~ *);
    end;
    observe from Cons.Ack, Prod.Reply
        match ( (?N in Integer)
                    (Cons.Ack(?N) →
                    (Prod.Reply(?N) or Empty)) )↑(~ *);
    end;
end architecture ProdCon;
```

Commentary:

The reactive interface behaviors and connection rules in the Producer/Consumer example of Section 2.2 have been deleted. Constraints have been put in their place. The architecture specifies allowable behaviors between its components, but does not implement any particular behavior.

The first architecture constraint in ProdCon requires a causal relation between each Prod.Send(?N) event and a Cons.Receive(?N) event. This was implemented by the first connection rule in the previous example. The constraint is more general in that the required causal relation between the two events, Send, Receive allows any number of intermediate events. It could be satisfied by a connector module (say a model for a communication network, mailer or ethernet), as well as the reactive connection rule. Similar remarks apply to the second architecture constraint in comparison with the second connection rule in the previous example.

The interfaces also specify sets of allowable behaviors but do not provide any particular executable behavior. For example, the constraint in the Producer interface requires each Reply event that is received by a producer to cause a Send event with the argument incremented unless it exceeds the Max bound. It also constrains the architecture containing the producer because it requires the Reply events that come back to be causally related to the Send events that they reply to. So, the interface constraint is constraining not only the modules that have that interface, but also the environments in which they are placed. [10]

□

The approach of using constraints in the definition of architecture standards is supported in RAPIDE not only by the constraint sublanguage (see Chapter 4), but also by mappings whose purpose is to define relationships between architectures and systems. Event pattern mappings are a very powerful feature for specifying in an executable way how the behaviors of one system or architecture are interpreted as posets of the events of another architecture or system. They allow executions of a system to be tested for conformance to the constraints in a standard.

[10] This is simply a more general instance of, say, an input condition on a function — which must be obeyed by callers.

Such mappings provide a technology basis for automating testing systems for compliance with standards.

2.5 How Rapide Architectures Execute

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 architecture is executed, it produces a simulation that shows not only the events that make up the architecture's behavior, and their timestamps, but also which events caused other events, and which events happened independently.

This kind of a causal history, produced by a RAPIDE execution, is often called a *poset* (partially ordered set of events) because the various relationships (i.e., cause and time) between events that are explicitly represented in such a history, are partial orderings of the events. Events are simply tuples of data; Chapter 3 describes events in detail.

The RAPIDE language constructs have a discrete event semantics. That is, the meanings of the constructs are defined in terms of an execution model in which events are generated together with their dependencies, timestamps, and arguments.

Consider interface connection architectures, such as Figures 2.4, 2.6, programmed in RAPIDE. The connections between the interfaces are defined by connection rules called *event pattern connections*. Event pattern connections link actions and functions declared in interfaces of components. A connection rule can be thought of as connecting "sending interfaces" to "receiving interfaces". Components can communicate through the connections by calling an action or a function declared in their own interface. When a component calls one of its actions (it must be an **out** action) it generates an event corresponding to that action. When a component calls one of its functions (it must be a **requires** function) it generates a function call event and waits for a function return event [11]

Example: *Asynchronous Architecture Connection rules*

```
connect
-- asynchronous connections between a Producer and a Consumer.
    (?n in Integer)
    Prod.Send(?n) => Cons.Receive(?n);

-- asynchronous connection between any airplance and a control sector.
    ?A : Airplane; ?M : Msg;
    ?A.Radio(?M) where ?A.InRange(SFO) ||> SFO.Receive(?M);
```

Commentary:

The first example is a connection between a producer, Prod, and a consumer, Cons (previous Producer/Consumer example). It executes by triggering whenever the Prod component generates a Send event (an instance of its **out** action). The connection then generates a Receive event with the same integer parameter that is an instance of the **in** action of the Cons component. The Receive event is received by the consumer. The producer may continue to execute subject to its own behavioral constraints while the connection or the consumer are executing.

[11] Actual implementations may optimize away the function call and return events.

Moreover, the connection imposes a *causal* relation between the triggering Send event and the generated Receive event. That is, the Send causes the Receive, and this will be recorded in the output from the RAPIDE simulator.

The second architecture connection triggers when any airplane that is a component of the architecture generates a Radio event when the InRange predicate is true. The connection generates a Receive event that is an instance of the **in** action of the SFO component. Again the connection is asynchronous, allowing the airplanes, connections and SFO to execute independently. Similarly, the triggering Radio event *causes* the generated Receive event.

□

Event pattern connections are *reactive rules*. They trigger when appropriate events are generated that match their pattern triggers. Events generated by components *trigger* event pattern connections between their interfaces. The effect of triggering a connection is to generate new events at the receiving interfaces. The result is that the components corresponding to the receiving interfaces receive those events (or function calls). Data is passed between components as parameters of the events; that is, connections have the ability to put data in the new events they generate, where that data is a function of the data in the events triggering the connections.

When a component generates an event by an action call it does not block, but simply continues execution. Thus, sending events between components by means of actions calls and connections is asynchronous communication of data between components.

Similarly, a function call can trigger connections and cause them to generate other function calls. But in this case the execution of the caller is blocked until the connection returns the return object of the generated call from the callee as the value of the triggering call. In this way, a connection can also connect functions in different components and synchronize them when the triggering function is called.

Example: *Synchronous function connection rules.*

```
-- synchronous connections between functions of compiler components.
   ?Tree, ?Tree' : Tree;
 Par.Semantize(?Tree) to Sem.Semantize(?Tree);
 Par.Generate(?Tree) to GC.Generate(?Tree);
 GC.Initialize_Parser() to Par.Initialize();
 GC.Semantize(?Tree, ?Tree') to Sem.Incremental_Semantize(?Tree, ?Tree');
```

Commentary:

Here are examples of architecture connections between **requires** functions and **provides** functions of components of the compiler. They alias the triggering **requires** calls to calls of **provides** functions. The caller blocks awaiting a return value, so these connections synchronize the caller and callee. They also imply a causal relation betweeen events associated with the two function calls; this relation is defined in detail in the Architectures LRM.

□

So, event pattern connections allow components that are executing independently to communicate either asynchronously by means of events or synchronously by means of function calls. It should be noted that connections can also be defined between the actions and functions in the interface of an architecture and the actions and functions of its components. So, a parent architecture can receive events and function calls from outside, through its interface, and pass them on to its components for processing. It can receive the results back from its components via connections from its components to its interface features, and then send them to its outside environment. Interactions between the parent architecture and its environment will be by means of connections in that (higher level) environment.

Note, also, (see Section 2.6.1) that interfaces can contain behavior rules which act like modules. Behaviors define the execution of interface objects. So an interface connection architecture consisting of only interfaces and connections can be executed, as well as any of its instances. In the case of an instance in which a module is a component corresponding to a particular interface type, the behavior in that interface defines a specification to which the module must conform.

2.5.1 How Causality is Modelled

An event in the execution of a RAPIDE model *causes* another event in specific situations; otherwise, any two events are independent.

- *Connections:* The events that are generated by a connection when it is triggered *depend* upon the events that triggered the connection. Connections execute independently – i.e., events generated by one connection will be independent of events generated by another connection unless the events triggering one connection depend upon the events generated by the other.
- *Behaviors:* Events generated by a behavior rule in an interface when it is triggered depend upon the events that triggered it. As with connections, behavior rules execute independently, so any dependency between the events generated by two rules must result from the events that trigger one rule depending upon the events generated by the other.
- *Processes:* Events generated by a process within a module when it is triggered depend upon the events that triggered it.
- *Sequential statements:* Events generated during the execution of a sequence of statements depend upon events generated earlier in the sequence.
- *Reading and Writing Objects:* Dependencies between events may also result from operations on a module — e.g., reading or writing some module.
- *Transitivity:* Dependency is a transitive relation between events.

RAPIDE provides a basic concept of causality and independence that expresses the reactive semantics of various concurrent constructs (connections, behavior transitions, etc.). Other causal models, for example causality resulting in Networking Systems, can be modelled on top of the basic RAPIDE causality by building architectures that model the behavior of such software or hardware systems. Causality in physical systems can probably also be modelled in RAPIDE by modules that model the rules for such systems (say, Newton's laws of motion), but such endeavors go beyond what RAPIDE was intended to achieve.

2.6 Architecture Simulation and Analysis

When a RAPIDE architecture is executed a causal event history of the behavior of the architecture is generated. The execution proceeds by first inputting calls to actions and functions in the interface of the parent architecture module. These inputs generate events at the interface. Connections between the interface and components of the architecture (inside the architecture module) then trigger and generate events at the interfaces of components. Components react to these events, and generate further events, which are communicated by connections to other components, and so on.

The simulation result is a poset showing the causal history of events in the execution, independent activity, timing, dataflow and other properties. This gives us the capability to simulate the behavior of a system architecture very early during the requirements analysis and design phases of the system.

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

- *Constraint Checkers.* Semantic constraints specify interface features architecture connections in RAPIDE. Constraints in interfaces restrict the visible behavior of components with those interfaces. Constraints on connections restrict the sequences in which connections can trigger and their timing, thereby specifying protocols, timing, and other important charactistics of communication between modules. Constraint checkers are used to automatically detect violations of constraints in the simulations.

- *Poset Browsers.* It is often necessary to browse a simulation simply to see how a given architectural design behaves. Indeed, one of the most important uses of early lifecycle simulation is experimental. Poset bowsers usually represent causal event simulations in a DAG form, nodes representing events and directed arcs representing causality. They supply a user interface with pattern directed operations for scalability to display pieces of large simulations, and to organize a display into a hierachy of sub-simulations.

- *Animation Tools.* The wealth of information in posets together with the somewhat abstract display formats provided by browsers, often results in the human viewer missing important properties. [12] Animation tools give the user a capability to animate in graphical forms the event activity in any poset. usually, a picture of the architecture is used as the basis for animation, and both the poset and the animation can be viewed simultaneously. The user is thereby given a human understandable intepretation of a poset simulation — in fact more than one animation view of the same poset can be provided.

2.6.1 Architecture-Driven System Development

RAPIDE is also intended to allow exploration of new methods of using architectures to build and test systems. In one approach to *architecture driven* system development, a RAPIDE interface connection architecture is used as a framework for building a system. One starts with an architecture consisting of interfaces and connections. The architecture is executed under different input scenarios to simulate the behavior of a system with that architecture. Assuming simulations of the architecture show behavior that meets the requirements for the system, , modules are then assigned to interfaces one at a time. Each module must conform to the interface it is assigned to (RAPIDE type rules help towards this, but as mentioned in Section 2.3, testing satisfaction of semantic constraints is a difficult problem.) When a module is assigned to an interface, the module is executed and the role of the interface behavior is to act as a constraint to which the module's behavior must conform (in addition to the semantic constraints). The result of assigning a module to an interface is called an *instance* of the architecture. Each instance is tested for comformance to the architecture's interface constraints, and also to the

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

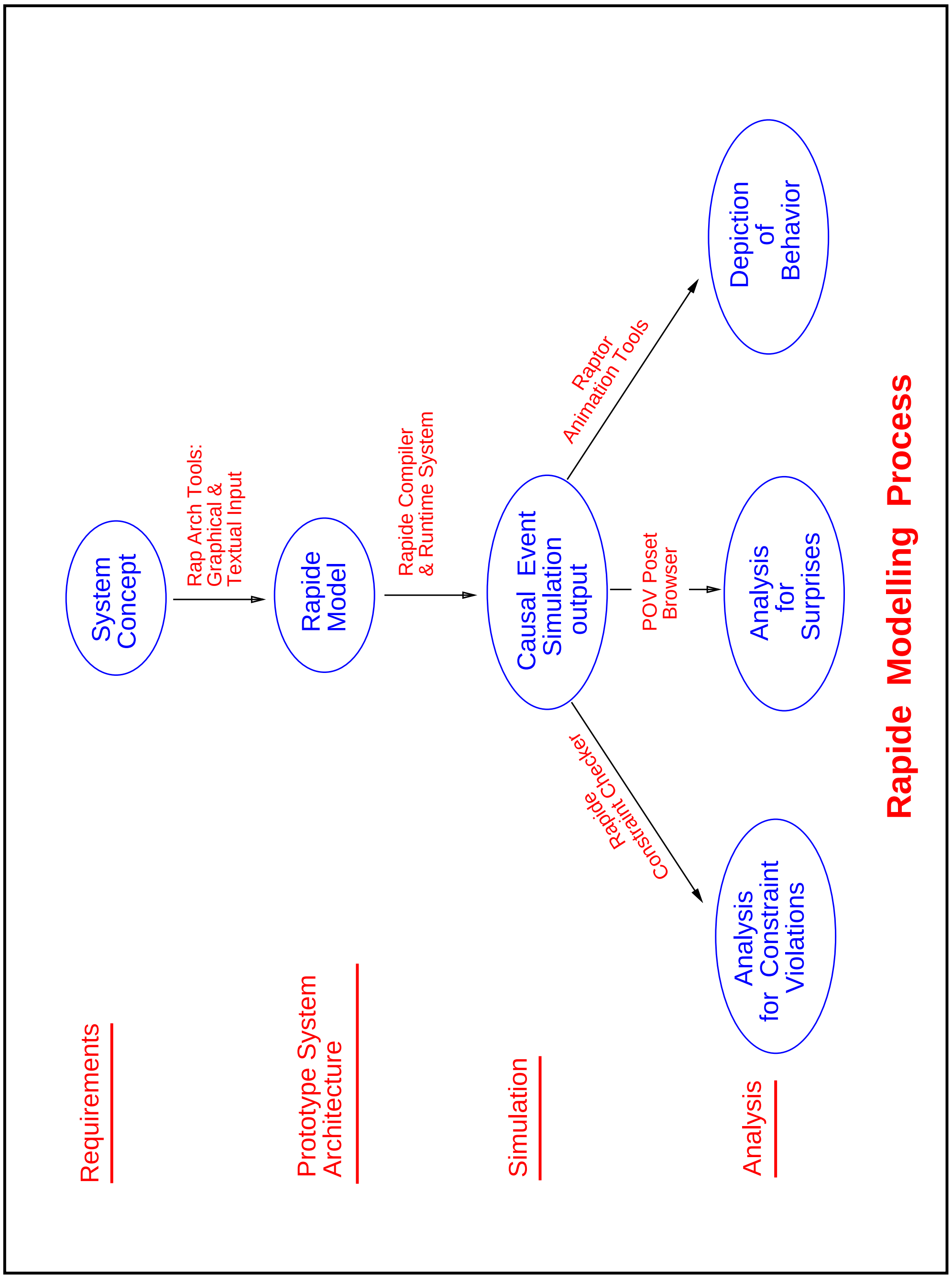

Figure 2.5: The Rapide simulation and analysis toolset

constraints on the architecture's connections. The final result should be a system of modules satisfying the architecture's interface and constraints.

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

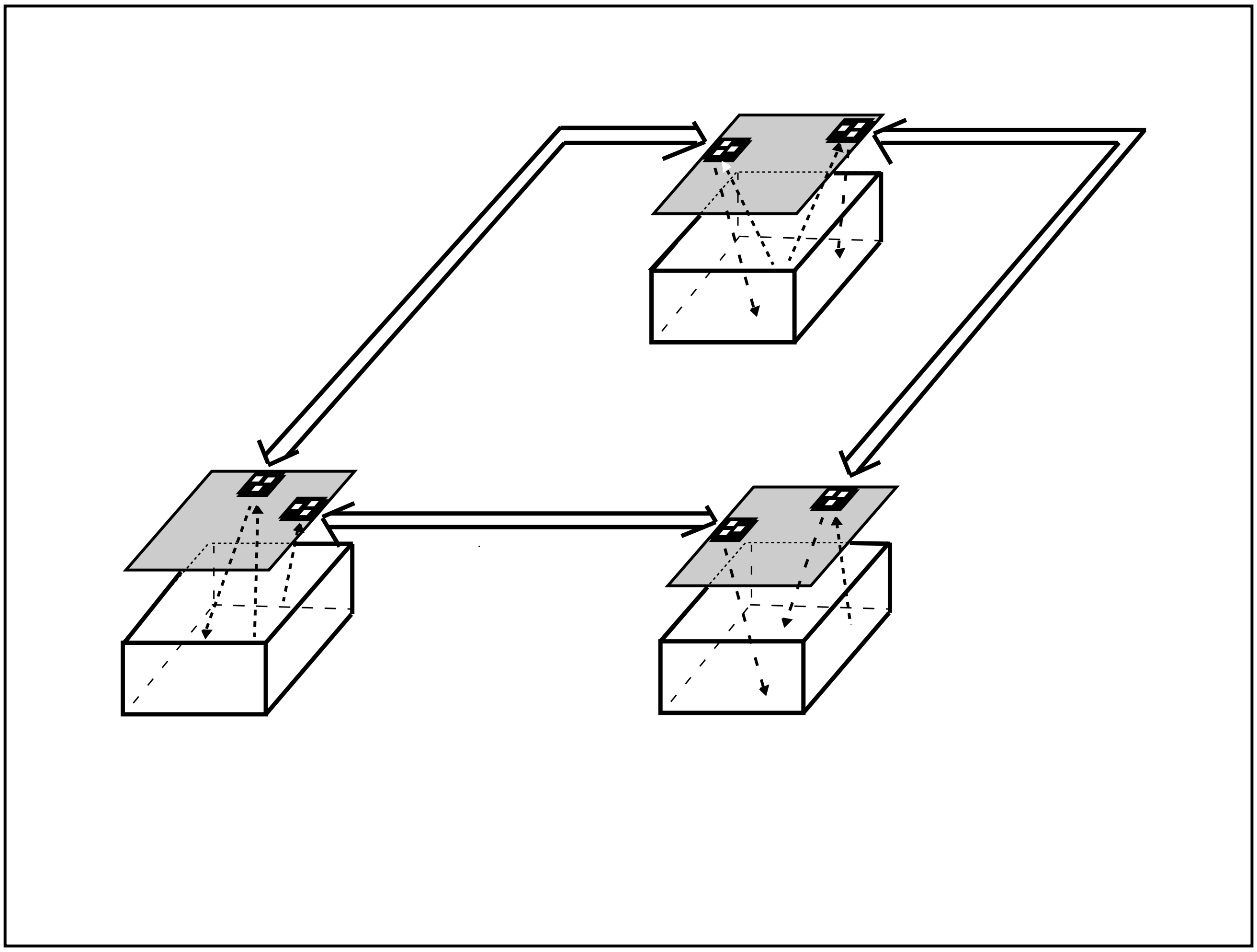

Figure 2.6: A single connection between two services (plugs) encapsulates many individual connections between functions and actions of the interfaces.

play the role of "wires" between the sockets.

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

2.7 Structuring Large Architectures

Large architectures have large numbers of components, sometimes varying numbers at runtime, and large numbers of connections. Issues related to representing large architectures are called *scalability* issues. There are various *scalability* problems with present architecture definition languages (ADLs). RAPIDE attempts to deal with some of them. For example, features for concise definition of large numbers of connections have been lacking in prior ADLs, notably in the hardware domain. [13]

RAPIDE provides *interface services* and the concept of *dual* services as an approach to scalability issues. Services are interfaces within interfaces. They provide a way to structure interfaces into sub-interfaces.

A service S is a *dual* of service T if S **provides** the **requires** features of T and **requires** the **provides** features of T, and similarly, the **out** actions of T are **in** actions of S and conversely.

There is a simple connection rule between services: a service in one interface may be connected to a dual service in another interface. One connection between two dual services in the interfaces of two components denotes all those connections between the pairs of interface functions or actions, one in the service and the other in the dual service, that have the same name (see Figure 2.6). So, services and connections between services can alleviate the problem of defining large numbers of connections. Also, connections between services are guaranteed to be correct if modules conform to their interfaces.

Advantages of services are:

- An interface can be structured into services which define related groups of features (services are often called *plugs* and the duals are called *sockets*). An RS232 plug or socket can be a service in a computer (or modem) interface, for example. The features in a service can be related by protocols that, e.g., constrain them to be used in particular orders.
- A service in an interface indicates which other types of interfaces it should be connected to in architectures — i.e., those with dual services. So services in an interface specify context in which the interface is intended to be used.
- A connection between dual services is shorthand syntax for the set of similar connections between the individual functions and actions with the same name in the services. So service connections structure the connections of a large architecture into sets of connections between related functions and actions of its components.
- Service connections are less prone to trivial syntactic errors because they connect dual services – not simply individual features.

Let us revisit our compiler example in Section 2.2. By looking at the interfaces of the compiler components one cannot tell which requires and provides functions are intended to be connected. In fact, we chose similar names in different interfaces to try to indicate intended connections. Suppose we had not done this. It would not be easy to tell which connections were intended or "reasonable" in some sense, except by type compatibility of the connected functions. For example, which component is intended to use the Parser's Initialize function? In general, if a connection is unintended (very common in hardware modelling languages) there is no easy way to detect it if the functions or actions it connects have compatible types..

The following example defines groups of functions in the component interfaces that we intend to be related. [14]

Example: *Services for communication between compiler modules.*

[13] VHDL, for example, has the *generate* statement for enumerating sets of connections that are indexed by an integer variable.

[14] Admittedly, our grouping here is for illustrative purposes and does not reflect any legimate theory of compiler construction.

```
type CodeGenerator_Parser is interface
provides
    function Initialize();
requires
    function Generate(Data : Tree);
constraint
    match Initialize'Call  → Generate'Call↑(~ *)
end;


type Parser_Semanticizer is interface
provides
    function Semantize(Data : Tree);
requires
    function FileName() return String;
end;


type CodeGenerator_Semanticizer is
provides
    function Incremental_Semantize(Data : Tree; Context : Tree);
end;
```

Commentary:

The CodeGenerator_Parser type specifies two functions. A service of this type says informally, "whatever interface contains me provides an Initialize function to other components and requires other components to provide a Generate function." We have also added a constraint requiring a call to Initialize before any calls to Generate. We will use this constraint to illustrate how service connections imply correct behavior if modules obey their interface constraints (next example). The other services can be explained similarly.

□

Using these services we can structure the interfaces of the compiler's components, and define a plug and socket interface connection architecture for it. Remember, the dual of a service has the **provides** and **requires** functions reversed, and a connection between a service and its dual connects the pairs of functions with the same name.

Example: *A new definition of the Compiler's interface connection architecture.*

```
type Parser is interface
 provides:
    dual_P_S : dual service Parser_Semanticizer;
    C_P      : service CodeGenerator_Parser;
    ...
end Parser;
```

```
type Semanticizer is interface
 provides:
    C_S : service CodeGenerator_Semanticizer;
    P_S : service Parser_Semanticizer;
    ...
end Semanticizer;


type Code_Generator is interface
 provides:
    dual_C_P : dual service CodeGenerator_Parser;
    dual_C_S : dual service CodeGenerator_Semanticizer;
    ...
end Code_Generator;


architecture Compiler return CompilerInterface is
      Par : Parser is ParserGen( );
      Sem : Semanticize is SemanticizeGen( );
      CG  : Code_Generator is CodeGen( );
connect
   Par.dual_P_S to Sem.P_S;
   CG.dual_C_P to Par.C_P;
   CG.dual_C_S to Sem.C_S;
end Compiler;
```

Commentary:

This plug and socket architecture defines the same interfaces and connections as the interface connection architecture given in Section 2.2. Here, each interface is structured into two services. Each service groups the functions needed for communication between a specific pair of interfaces; each pair of interfaces contains duals of the service defining the communication between them. There are three service connections (shown in Figure 2.6 as cables between the interfaces). They define the six connections of the previous compiler architecture (Figure 2.3). Moreover, the service connections in this example are less likely to be erroneous due to typos and other simple errors. RAPIDE static semantic checking will check duality of service connections.

Let us consider the service connection between the code generator and parser. The connection between the CodeGenerator_Parser service and dual service actually connects a pair of Initialize functions and a pair of Generate functions. The Parser provides an Initialize function which is required by the CodeGenerator. The constraint remains the same in the dual service as in the service. So, the CodeGenerator must call its required Initialize function before doing anything else after its Start event. This will cause the connection to generate an Initialize call to the Parser. Meanwhile, the Parser must await a call to its provides Initialize before it calls its required Generate function — which in turn will cause a call to the CodeGenerator's provides Generate function. So, if the two modules conform to their interfaces they will behave so as to obey the constraint. As a consequence, the two modules together with the service connection between them will generate an event behavior that is consistent with the constraint.

□

2.8 Comparing Architectures

Hierarchical design is a frequently used methodology for developing system architectures at different levels of detail, starting from a high level involving abstract concepts, and expanding the architecture at progressively lower levels involving more detailed concepts. This is typical in hardware or protocol design.

The architecture of a system at each level in the design hierarchy should be consistent with the architectures at other levels. To determine mutual consistency of architectures at different levels, one must be able to define how the architectures correspond. Often, the correspondence is complicated (e.g., between an instruction set architecture and a gate level architecture for a microprocessor). New language constructs are needed to define such correspondences.

Another area where comparison between architectures needs to be defined precisely is between architectures for different systems within a given domain. Examples of domains are control systems, or avionics, or transaction processing. Precise methods of architecture comparison should enable one to study various performance issues that are often thought to be critical in the choice of architecture. At present, ADLs do not provide any capability to compare architectures.

Maps and Bindings have been defined in RAPIDE to provide a capability to define relationships between architectures, and thus to automate comparative analysis in hierachical design and comparison of systems with reference architectures. The basic idea in both constructs is to used event patterns to define correspondences between events in two different architectures. This gives a capability to define either very simple (1-1) correspondences, or complex non determininsitic (many-many) correspondences between posets in either architecture. Maps in particular may be used to deal with some other problems of scale, notably analyzing large, low level simulations, by mapping them into smaller high level simulations. Maps and Bindings are considered to be experimental language constructs at the present time. See Architectures LRM for further details.

Chapter 3

Event-based Computation

This Chapter describes events and computations. It describes briefly how RAPIDE 1.0 programs generate events and interact with them. Its main purpose is to provide context for understanding the general concepts involved in defining the semantics of RAPIDE constructs — both the executable constructs given in the RAPIDE 1.0 Executable Language Reference Manualand RAPIDE 1.0 Architectures Reference Manual, and the constraints described in RAPIDE 1.0 Constraint Language Reference Manual.

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

3.1 Events

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

The constituents of an event are the name of the generating action, parameters, information [1] defining which events caused the event, and timestamps. Thus an event, may be defined as a tuple consisting of:

1. the action whose invocation led to the event,
2. parameters of the action,
3. information defining which other events must have happened in order for this event to happen (called its *dependencies*),
4. two values of any clock within whose scope the event was generated; for each clock, the first clock value, v_1, is the *start time* of the event, and the second value, v_2, is its *finish time*, where $v_1 \leq v_2$.

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

Not all of these constituents are visible to the user. Some constituents (such as the dependency information) are useable only through predefined operations. For the definition of the event type, see Chapter 19 in the RAPIDE 1.0 Predefined Types and Objects Reference Manual.

3.1.1 Operations on events

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

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

3.1.2 Examples of events

Example: *Events from a Communication system*

```
-- Given a module, Sender, that send messages by means of the ac-
-- tions:
     action out Transmit(Msg : String; To : Receiver);
     action in Continue(Reply : Bit);

-- the following events, E1, E2, E3 could occur during a computation:

    E1.Action_Name = "Transmit", E1.From_Module = Sender,
    E1.msg = "Hello", E1.To = Star_Satellite,
    SFO_Clock.Start(E1) = 1400, SFO_Clock.Finish(E1) = 1401,
    Greenwich_Clock.Start(E1) = 2200.30, Greenwich_Clock.Finish(E1) = 2201.29.

    E2.Action_Name = "Continue", E2.From_Module = Star_Satellite, E2.Reply = 1,
    SFO_Clock.Start(E2) = 1401.25, SFO_Clock.Finish(E2) = 1401.26,
    Greenwich_Clock.Start(E2) = 2201.54, Greenwich_Clock.Finish(E2) = 2201.55.

    E3.Action_Name = "Transmit", E1.From_Module = Sender,
    E3.msg = "Goodbye", E3.To = Star_Satellite,
    SFO_Clock.Start(E3) = 1402, SFO_Clock.Finish(E3) = 1403.
    Greenwich_Clock.Start(E3) = 2202.28, Greenwich_Clock.Finish(E1) = 2203.27.
```

Commentary:

Events E1 and E3 are generated by calls to the Transmit action, and E2 is generated by a call to the Continue action. E1 and E3 differ in their Msg parameter value and their times. In this example all of the events have timing values with respect to two

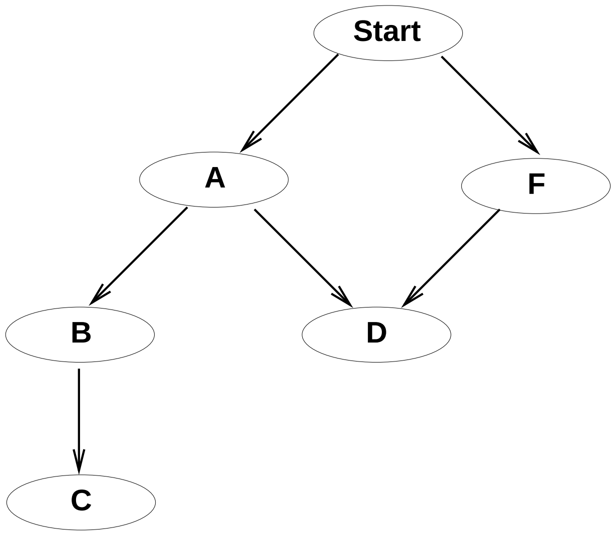

Figure 3.1: Dependent and independent events.

clocks, an SFO_Clock associated with the Sender, and a Greenwich_Clock associated with a more global scope containing the Sender; say, the world. Note that distinct events can have the same value (e. g., different invocations of the same action can have the same parameter values, dependencies, and times).

□

3.2 Relationships between events

RAPIDE 1.0 provides facilities for defining and referencing two kinds of relationships between events:

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

Both relationships are partial orderings of events.

Event E2 is *dependent* upon event E1 if E1 must have happened in order for E2 to have happened. Dependence is a partial ordering of the set of events. Dependence may be represented graphically as a directed acyclic graph (DAG) as shown in Figure 3.1. If two events are not related in this ordering they are *independent*. Some analysis tools supporting RAPIDE 1.0 use DAGs to represent dependence between events.

In figure 3.1 event C depends on both A and B but is independent of D and F; D is dependent on A and F.

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

3.2.1 Generating dependent events

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

- *reactive* rules and processes. These are rules and processes that have pattern triggers that define sets of events together with dependencies and timing. Whenever a set of events with the required relationships is observed [2] by a reactive rule or process, and *matches* its trigger, then that rule or process executes (i.e., the rule or process *triggers*). The events subsequently generated by the rule or process on that particular execution depend upon those events that triggered it.

  Reactive rules are transition rules in interface behaviors (Architectures LRM), connection rules in architectures (Architectures LRM), and mapping rules in maps (Architectures LRM). Reactive processes are **when** statements in modules (Executable LRM).

- *sequential* code — events generated by sequential executions have a strict linear dependence represented by their order of generation,

- *ref objects* — a **ref** type object defines dependence between the events generated by the processes that share the object. If an event results from a computation that dereferenced a ref object, then the event depends on the (unique) event that last changed the contents of that ref object — i.e., operations on ref objects are linearly ordered.

3.2.2 Generating timed events

A type or module can be *timed* by having a clock associated with it, or by being placed in the scope of a clock. The type Clock and several subtypes of it are predefined. Events receive start and finish time values for each clock within whose scope they are generated (see Chapter 17 of the RAPIDE 1.0 Predefined Types and Objects Reference Manual).

Action calls may specify the time taken for actions to be performed with respect to a clock (see Section 7.2 of the RAPIDE 1.0 Executable Language Reference Manual). If action A is defined to take duration d with respect to clock C, then for an event E generated by A, C.Start(E) + d = C.Finish(E). If no duration is defined, the events are generated infinitely fast with respect to any clock.

3.3 Observation of Events

The events generated by modules of a RAPIDE program are *visible* to, and can be *observed* by, other modules. Events are made *available* to a particular module for observation when other modules call its interface **provides** functions and **in** actions (see Architectures LRM).

The ability to call the interface functions and actions of a module depends upon visibility rules (Executable LRM, Chapter 2) of the language. For example, a typical way (but not the only way) that a module can communicate with another module is by generating events which trigger connection rules in the architecture in which both modules are components. Both components are visible to the architecture. The connection rules are then executed by the architecture and result in calls to the other module's interface functions or actions.

[2] as explained in Chapter 2 of the Executable LRM

3.3.1 Observing events by pattern matching

RAPIDE provides features for observing and reacting to sets of events and their orderings, thereby providing a powerful facility for modelling distributed communication.

Features for observing events, and the dependency ordering between events, are provided by the *pattern language*. Briefly, patterns are templates that allow the definition of sets of events. Patterns can specify relationships between events by means of dependence operators. For example, "A depends on B" is written as $\mathsf{A} \rightarrow \mathsf{B}$, and "A is independent of B" is written as $\mathsf{A} \mid\mid \mathsf{B}$ (see the RAPIDE 1.0 Event Pattern Language Reference Manual).

The timing parameters of events can be accessed by clock operations as described in Section 3.1.1. The pattern language also defines abstract pattern operations dealing with time and events.

Patterns can be used as the triggers of: *(i)* processes (Executable LRM), *(ii)* transition rules in interface behaviors, *(iii)* connections in architectures, and *(iv)* map rules (see the RAPIDE 1.0 Architectures Reference Manual). Events that are made available to a module are observed by *matching* the pattern triggers of the reactive rules or processes in the module. Pattern matching is explained in the Patterns LRM. If and when an event contributes to matching the pattern trigger of a rule or process, it can no longer be observed by that rule or process.

Finally, patterns can be used in formal constraints (see the RAPIDE 1.0 Constraint Language Reference Manual). Events that are available to a module are also observed by matching the constraints of that module. Violations of constraints are reported as they happen.

3.3.2 Orderly observation

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

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

3.4 Computations

The events generated by all the modules of a RAPIDE 1.0 program comprise the computation [3] generated by the program. A computation consists of a set of events, S, a dependence partial ordering, $\leq_d$, and timing partial orderings, $\leq_C$, for objects C of type Clock.

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

These computations are called *posets*, i.e., partially ordered sets of events. Posets generated by RAPIDE programs satisfy two invariants.

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

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

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

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

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

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

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

Similarly, the various time orderings obey a consistency invariant:

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

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

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

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

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

3.4.1 Levels of abstraction in Computations

A RAPIDE program often consists of modules at different levels of abstraction. Typically, the interfaces of the highest level modules will be connected in a top level interface connection architecture. Modules corresponding to these interfaces may themselves be architectures consisting of lower level modules, and so on. The computaton generated by a program will contain the events generated by all the modules at all levels.

A sub-computation corresponding to a given level of abstraction will consist of the subset of events corresponding to the actions and functions of the interfaces at that level. Events at both higher and lower levels are deleted. The dependency relation between the remaining events is defined by: an event E_1 depends upon event E_2 in the subcomputation if E_2 is in its dependency history in the full computation.

3.5 Examples

Example: *Producer and Consumer*

```
type Producer is interface action out Emit( n : Integer ); end Producer;
type Consumer is interface action in Source( n : Integer ); end Consumer;
```

```
module New_Producer( min, max : Integer ) return Producer is
    function Compute( n : Integer ) return Integer is
        begin return n + 1; end function Compute;
initial
    Emit(min);
parallel
    when (?x in Integer) Emit(?x) where ?x < max do
        Emit( Compute( ?x ));
    end when;
end module New_Producer;

module New_Consumer() return Consumer is
    function Use( n : Integer ) is begin null; end function;
parallel
    when (?y in Integer) Source(?y) do Use(?y); end when;
end module New_Consumer;

architecture ProdCon() is
    Prod : Producer is New_Producer(1, 5);
    Cons : Consumer is New_Consumer();
connect
    (?n in Integer)
    Prod.Emit(?n) => Cons.Source(?n);
end architecture ProdCon;
```

Commentary:

The architecture Prodcon in this example consists of two components, a Producer object and a Consumer object (called Prod and Cons respectively). They are declared in the declarative part of the architecture, before the **connect** part. The interface types of these objects define a Producer as being able to generate Emit events, and a Consumer as being able to observe Source events. These components of the architecture are assigned very simple modules. The producer module simply generates Emit events up to some maximum; the consumer module just uses Source events.

In the **connect** part, Prodcon connects the **out** Emit events of the Producer to the **in** Source events of the Consumer by a a *pipeline* connection, indicated by the "=>" operator. Whenever the connection is triggered by a Emit event, it executes by generating a Source event with the same integer argument. This connection imposes a causal relation between the event that triggers it and the event that it generates as a consequence. Also, since it is a pipeline connection, it imposes a causal relation between all the events it generates, in the order they are generated. The poset generated by executing the example is shown in Figure 3.2. Thus, Emit(1) causes Source(1), Emit(2) causes Source(2), and so on; also, all the Source events are causally ordered in the order they were generated. But what caused the causal ordering of the Emit events? This results from the semantics of the single sequential **when** process in the producer object (see the module generator, New_Producer). Note that the **parallel** part of this module generator allows for many sequential processes "in parallel", but in this case there is only one sequential process.

The connection can be changed from a pipeline to an *agent* connection,

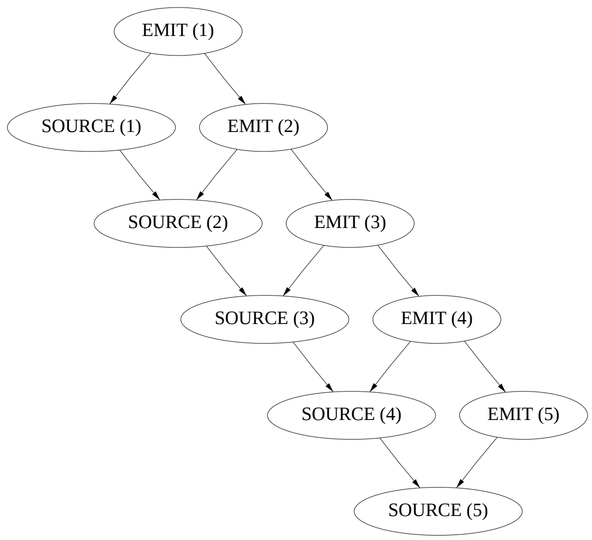

Figure 3.2: Poset generated by a pipeline connection in Producer and Consumer.

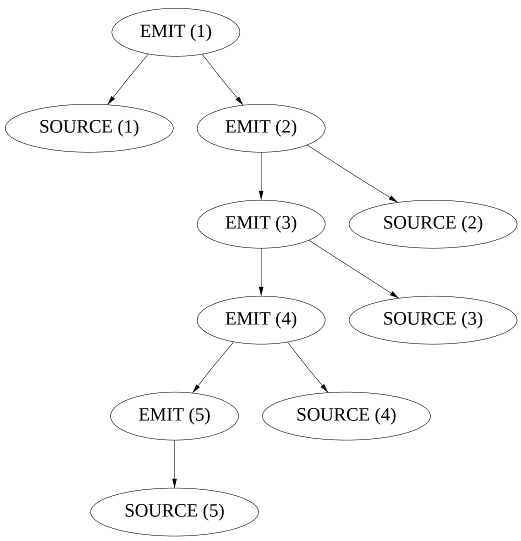

Figure 3.3: Poset generated by an agent connection in Producer and Consumer.

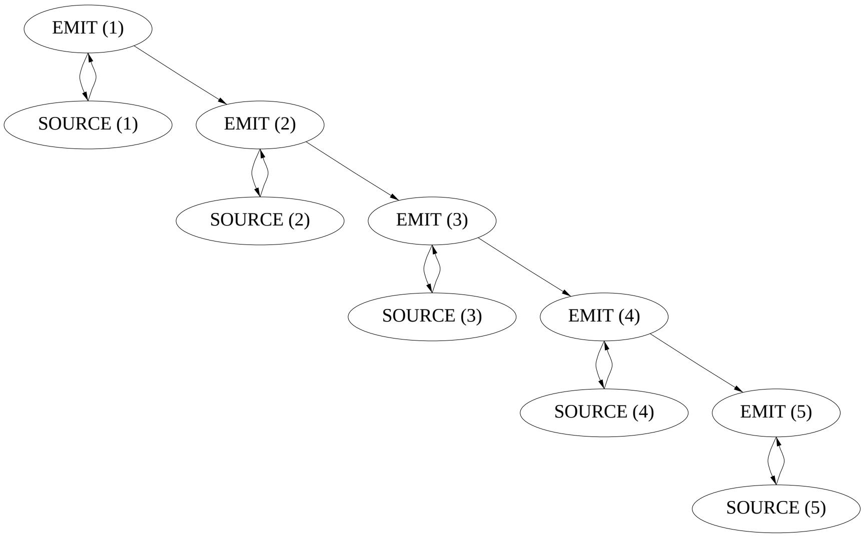

Figure 3.4: Poset generated by basic connections in Producer and Consumer.

connect
  (?n **in** Integer)
  Prod.Emit(?n) ||> Cons.Source(?n);

An agent connection has a semantics whereby each new triggering represents a new thread of control. Therefore it causally relates triggering events and generated events on each triggering, but does not relate events generated on separate triggerings. The resulting poset is shown in Figure 3.3. The difference between the two figures, apart from changes in layout, is that none of the Source events are causally related.

Finally, the connection can be a *basic* connection,
connect
  (?n **in** Integer)
  Prod.Emit(?n) **to** Cons.Source(?n);

The semantics of a basic connection identify the triggering and generated events. This means that they are indistinguishable by causal or time ordering. This is shown in Figure 3.4 by double arrows between the pairs of equivalent events.

□

Example: *Dining Philosophers.*

```
type Direction is enum Left, Right end enum;

type Philosopher is interface
    action   out EnterRoom();
             out ExitRoom();
             out RequestFork( fork : Direction );
             out ReleaseFork( fork : Direction );
             in  GetFork( fork : Direction );
             action out Think();
                    out Eat();
constraint
-- match ( Think -> EnterRoom -> RequestFork(Left) -> RequestFork(Right) -> Eat
--           -> ReleaseFork(Left) -> ReleaseFork(Right) -> ExitRoom ) ^ (->*);
    never ( Think || Think );
end Philosopher;

module New_Philosopher() return Philosopher is
parallel
    for I : Integer in 1..1 do
        Think();
        EnterRoom();
        RequestFork( Left  );
        await GetFork( Left  );
        RequestFork( Right );
        await GetFork( Right  );
        Eat();
        ReleaseFork( Left  );
        ReleaseFork( Right );
        ExitRoom();
    end do;
end New_Philosopher;


type Fork is interface
    action in PickUp(D : Direction);
           in PutDown();
           out Acknowledge(D : Direction);
           out IsDown();
constraint
-- match ( PickUp -> PutDown ) ^ (->*);
    never ( Pickup || PutDown );
    never (( Pickup -> PutDown ) || ( Pickup -> PutDown ));
end Fork;
```

```
module New_Fork() return Fork is
    action Free(), Taken();
parallel
    when Start do
        Free();
    end;
||
    when (?D in Direction) PickUp(?D) and Free do
        Acknowledge(?D);
        Taken();
    end;
||
    when PutDown() and Taken do
        IsDown();
        Free();
    end;
end Fork;


type Room is interface
    action in Enter();
           in Leave();
end Room;


architecture Dining( ) return Root is
   parts : Integer is 3;
   P   : array[ Integer ] of Philosopher is ( 0..parts-1, default is New_Philosopher());
   F   : array[ Integer ] of Fork is ( 0..parts-1, default is New_Fork());
   R   : Room;
connect
   for I : Integer in  0. .(parts-1) generate
       P[I].RequestFork( Right )   to  F[I].PickUp( Right);
       P[I].RequestFork( Left  )   to F[(I+1) mod parts].PickUp( Left);
       F[I].Acknowledge( Right )   to  P[I].GetFork( Right);
       F[I].Acknowledge( Left )    to P[(I-1+parts) mod parts].GetFork( Left);
       P[I].ReleaseFork( Right )   to F[I].PutDown;
       P[I].ReleaseFork( Left  )   to F[(I+1) mod parts].PutDown;
       P[I].EnterRoom()            to R.Enter();
       P[I].ExitRoom()             to R.Leave();
   end for;
end Dining;
```

Commentary:

The dining philosophers example is represented as an architecture, Dining, containing components of types Philosopher, Fork, and Room. Notice that all of these objects are active. Philosopher objects are instances of the module generator, New_Philosopher. They are a single sequential process that loops through a sequence of activities such as requesting forks and waiting to get them. Fork objects are instances of New_Fork. They are critical regions which must receive a PutDown event after each PickUp event before they will observe and respond to another PickUp event. When they respond to a PickUp event they send an Acknowledge event. A Fork consists of two concurrent processes, one reacts to PickUp events when the fork is Free and one reacts to PutDown events when it is Taken; the processes signal each other by means of the internal events Free and Taken. The Room is trivial in this version of the example.

Dining contains an array of 3 phiposophers, an array of 3 forks, and a room. The connections define a communication architecture as follows. RequestFork events from a philosopher are connected to PickUp events of a specific fork, depending upon whether the left or right fork is indicated in the request. Similarly, Acknowledge events from a fork are connected to GetFork events of the requesting philosopher. ReleaseFork events are connected to PutDown events of the fork being released. Consequently a philosopher interacts only with the forks on his left and right. For simplicity, only one kind of connections is used, basic connections (**to**).

A poset generated by executing the Dining Philosophers example is shown in Figure 3.5. Each event shows the thread of control that generated that event, and the action name; other data has been deleted. Also, some events have been deleted for simplicity. Note that causal arcs do not correspond necessarily to threads of control, but may contain events generated by different threads whenever those threads interact, e.g., by trggering on events generated by one another.

There are six independent objects active within the architecture (the room is also active but does nothing here). When each philosopher Starts as a result of its elaboration, it first generates a Think event. When each fork starts it generates a Free event. The causal path of events generated by a philosopher (a single process here) shows a sequence of Think, Enter, RequestFork events. The RequestFork(Left) events are connected by a basic (**to**) connection to PickUp events of some fork. When the basic connection is triggered it generates a PickUp event at a fork which is equivalent to the triggering RequestFork event. So only one of the events appears in this poset. This event *and* the Free event generated by the start process of that fork together trigger the pickup process of the fork, and cause an Acknowledge event to be generated. So, the two events cause an Acknowledge, which in turn causes a connection to generate a GetFork event at a philosopher. The connection is executed by the architecture, Dining, and results in the GetFork event. Also, the sequential pickup process within the fork follows the Acknowledge event by generating an internal event of the fork, Taken. Consequently, an Acknowledge causes two events, GetFork and Taken. The sequential process in a philosopher awaits the GetFork event.

In this layout of the poset, the threads of control of the single process in a philosopher and the pickup process in a fork are shown as crossing over at an Acknowledge event (which they cause together) and then continuing to execute independently. The "crossover" has no siginificance; it is purely a result of the graphical layout algorithm.

The poset shows the typical deadlock that Dijkstra originally constructed this example to illustrate. The concurrent activity leads to a situation in which each fork is taken and each philosopher is blocked waiting for its second RequestFork to be acknowledged.

□

Example: *Scheduled Dining Philosophers.*

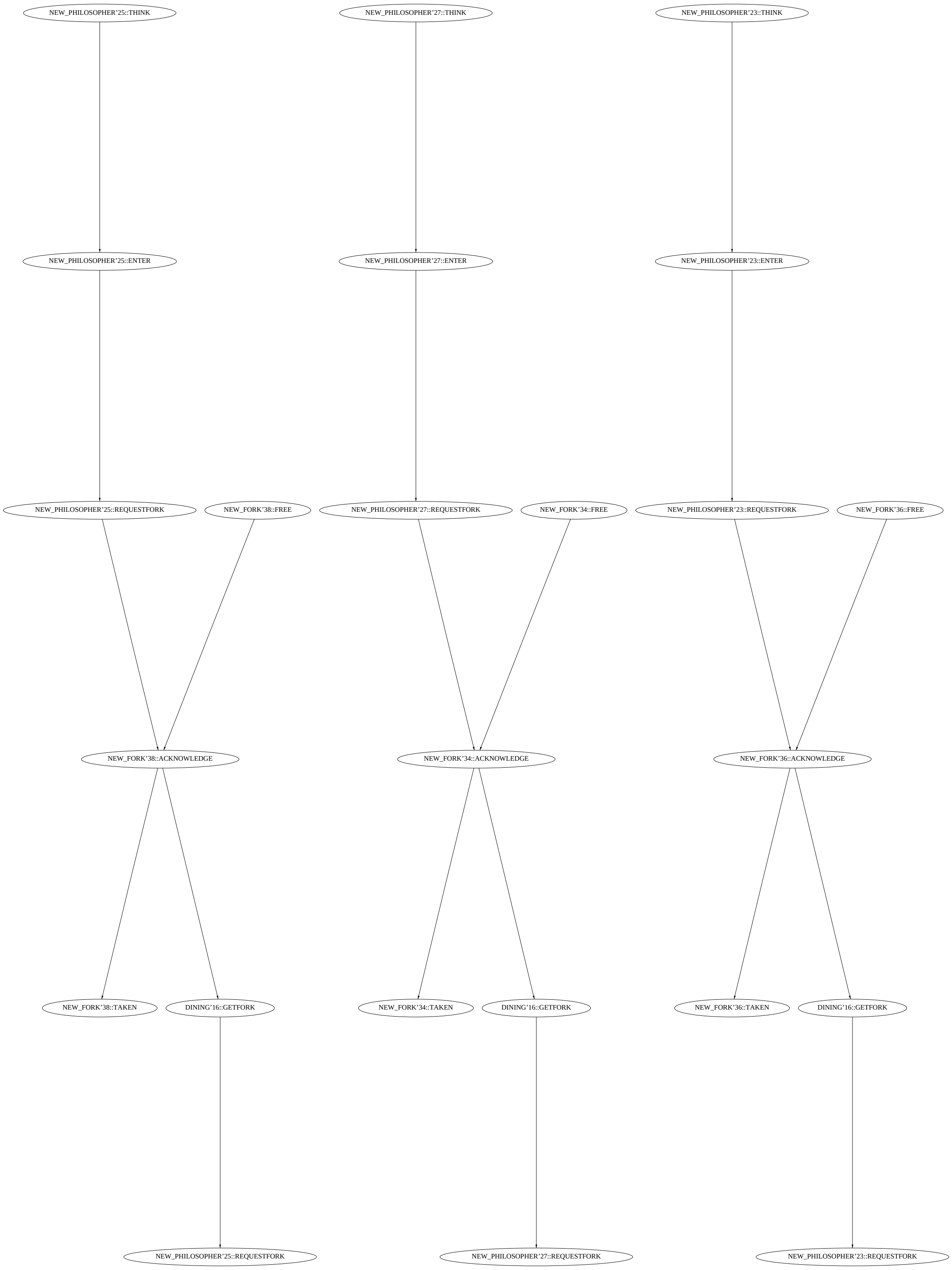

Figure 3.5: A poset generated by the Dining Philosophers.

```
type Direction is enum Left, Right end enum;

type Philosopher is interface
    action   out EnterRoom();
             in  InsideRoom();
             out ExitRoom();
             in OutsideRoom();
             out RequestFork( fork : Direction );
             out ReleaseFork( fork : Direction );
             in  GetFork( fork : Direction );
             action out Think();
                    out Eat();
constraint
    ...
end Philosopher;


module New_Philosopher() return Philosopher is
parallel
    for I : Integer in 1..1 do
        Think();
        EnterRoom();
        await InsideRoom();
        RequestFork( Left  );
        await GetFork( Left  );
        RequestFork( Right );
        await GetFork( Right  );
        Eat();
        ReleaseFork( Left  );
        ReleaseFork( Right );
        ExitRoom();
        await OutsideRoom();
    end do;
end New_Philosopher;


type Fork is interface
    ...
end Fork;


module New_Fork() return Fork is ... end;


type Room is interface
    action in Enter( P : Philosopher );
           out InRoom( P : Philosopher );
           in Leave( P : Philosopher );
           out OutOfRoom( P : Philosopher );
end Room;
```

```
module New_Room() return Room is
    WaitVal : var Integer := 1;
parallel
    when (?P in Philosopher) Enter(?P)   do
      InRoom(?P) pause $WaitVal;
      WaitVal := $WaitVal + 1;
    end;
||
    when (?P in Philosopher) Leave(?P)   do
      OutOfRoom(?P) pause 1;
    end;
end Room;


architecture Dining( ) return Root is
   parts : Integer is 2;
   P   : array[ Integer ] of Philosopher is ( 0..parts-1, default is New_Philosopher());
   F   : array[ Integer ] of Fork is ( 0..parts-1, default is New_Fork());
   R   : Room is New_Room();
connect
connect
   for I : Integer in  0. .(parts-1) generate
       P[I].RequestFork( Right )    to  F[I].PickUp( Right);
       P[I].RequestFork( Left  )    to F[(I+1) mod parts].PickUp( Left);
       F[I].Acknowledge( Right )    =>   P[I].GetFork( Right);
       F[I].Acknowledge( Left  )    to P[(I-1+parts) mod parts].GetFork( Left);
       P[I].ReleaseFork( Right )    to F[I].PutDown;
       P[I].ReleaseFork( Left  )    to F[(I+1) mod parts].PutDown;
       P[I].EnterRoom()             to R.Enter();
       P[I].ExitRoom()              to R.Leave();
   end for;
   (?P in Philosopher) ?P.EnterRoom()    to R.Enter(?P);
   (?P in Philosopher) R.InRoom(?P)       to ?P.InsideRoom();
   (?P in Philosopher) ?P.ExitRoom()     to R.Leave(?P);
   (?P in Philosopher) R.OutOfRoom(?P) to ?P.OutsideRoom();
end Dining;
```

Commentary:

The *scheduled dining philosophers* differs from the first version of dining philosophers in that the Room acts as a scheduler. Philosophers request to enter the Room as before, but they now wait for an InsideRoom event before requesting forks. The Room responds to their requests to enter one by one after one time unit delay. The example shows the changes to the philosophers' interface, the module generator for Rooms, and the Dining architecture with new connections between philosophers and the Room.

Figure 3.6 shows a poset generated by the scheduled dining philosophers. The layout algorthim for the DAG representation of the poset is not too smart. Even this small example has had some events deleted — e.g., Acknowledge events generated by forks. Generally, causal event simulations provide a lot of information about what happened — causal arcs between events are not contained in event traces. The layout tries to minimize the number of arc cross-overs. So, here we can see why automated constraint checking would be a helpful way to analyze simulation output.

If we look carefully we can see three philosopher threads, numbers $39, 37, 41$. They each start by generating a Think event. Then they generate Enter events of the Room A philosopher's EnterRoom event is equivalent to the corresponding Enter event of the room under the basic **to** connection. The Room is thread 54. request compete to enter the Room.

□

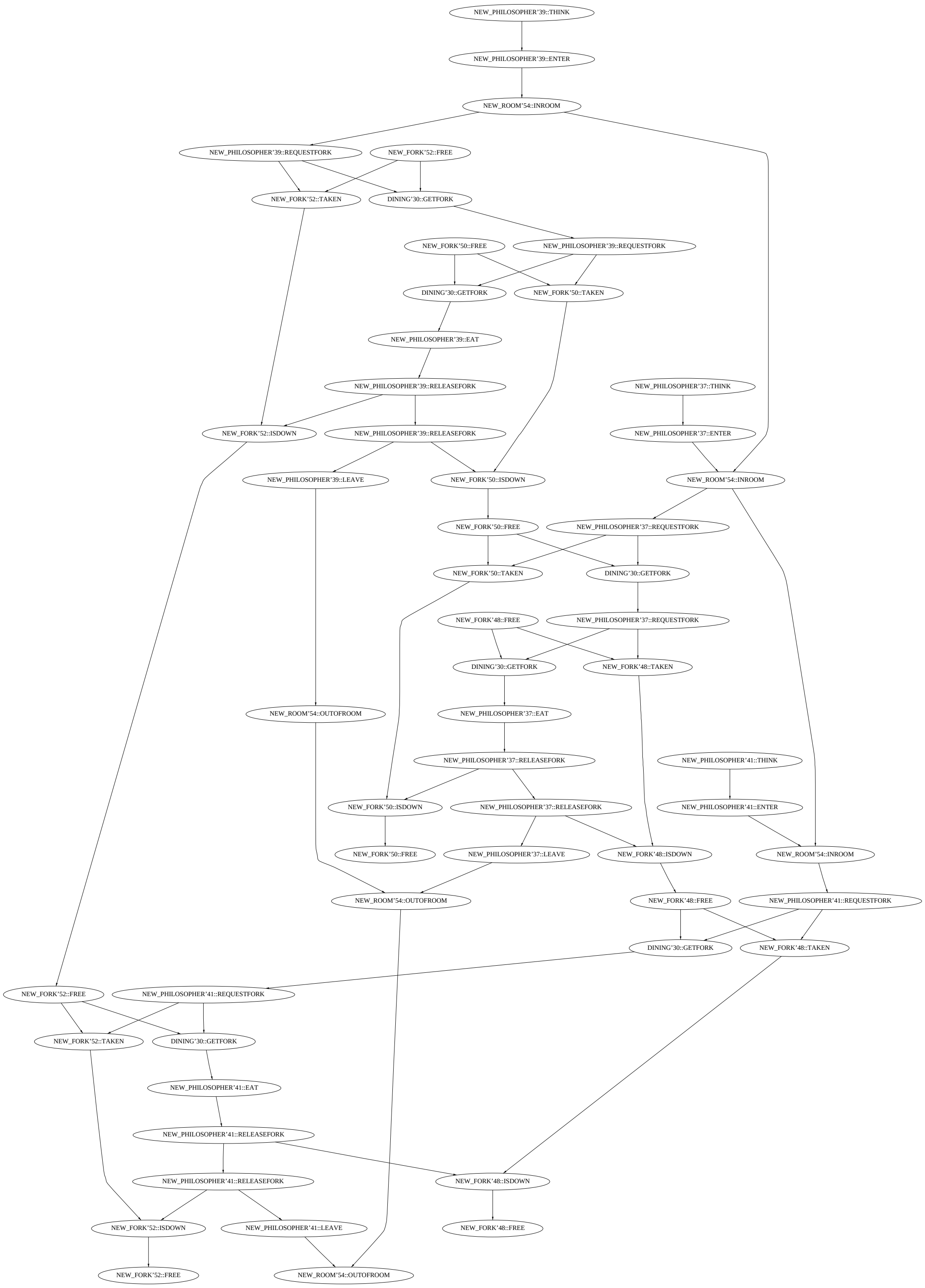

Figure 3.6: A poset generated by the Scheduled Dining Philosophers.

Chapter 4

An Overview of Rapide

The full RAPIDE language consists of five sublanguages. These are:

- the Types Language (Types LRM [Tea94f]) and the Predefined Types (Predefined Types LRM [Tea94d]) which are special cases of interface types,
- the Pattern Language (Patterns LRM [Tea94c]),
- the Executable Language (also called the Reactive Programming language, Executable LRM [Tea94b]),
- the Architecture Language (Architectures LRM [Tea94a]),
- the Constraint Language (Constraint LRM [Tea94e]).

These sublanguages and their relationships are shown in Figure 4.1. The *is-a-sublanguage* relationship is transitive. However, the picture does not give any details of how one language is a sublanguage of another.

The sublanguage relationship is *syntactic extension*; that is, the productions of a sublanguage are a subset of the productions of a super-language. But the actual details of a sublanguage relationship differ for each pair of languages. For example, all the languages contain the type declarations and object declarations of the Types language, and cannot declare objects in any other way (so all the languages are strongly typed). Type and object declarations can occur in any declarative region of any of the other languages. But, although constraints are also a sublanguage of the Executable and Architecture languages, they do not occur in the same places as type and object declarations — e.g., constraints are placed in the constraint region of an architecture. And patterns occur as the bodies of constraints in the Constraint language.

A human understandable syntax document for the full RAPIDE language is given in the RAPIDE 1.0 Full Syntax [Tea95]. This is probably the most useful quick reference document for users who are constructing RAPIDE models.

4.1 Rationale for SubLanguages

There are several motivations for dividing RAPIDE into sublanguages. Essentially, it is hoped that division into sublanguages makes changes to the language easier to manage, and facilitates reuse of parts of RAPIDE in other languages. We discuss some of the issues that led to our sublanguage approach below.

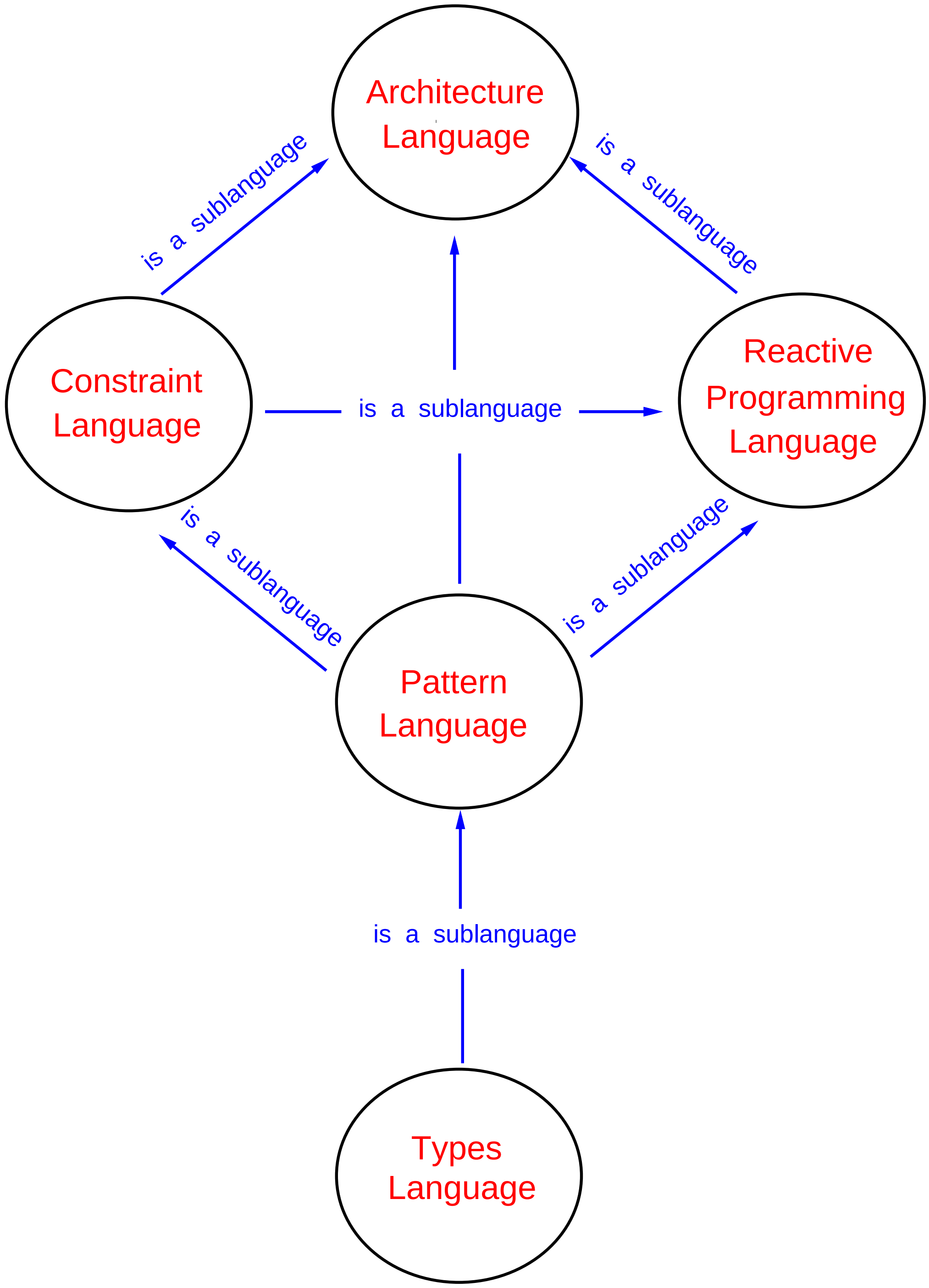

Figure 4.1: Sublanguages of Rapide and their Relationships.

4.1.1 Language Evolution

RAPIDE is an experimental ADL. It is intended for defining architectures and simulating their behavior by causal event simulation. There are many design and implementation issues that need to be researched, and for which RAPIDE can serve as a testbed. Some of these research issues are:

- design of languages for representing dynamic architectures.
- design of formal constraints for defining architectural standards.
- implementaton of constraint checking on causal event simulations for automated analysis of simulations.
- algorithms for efficiently representing causality in event-based simulations.
- design of mapping constructs for testing conformance of systems to architectures.
- interoperation of event-based languages and programming languages.

It is to be expected that various parts of RAPIDE will under go changes in the future based upon practical experience with applications. We hope that redesign of one sublanguage, e.g., either syntax or semantics of the constraint language, will have little or no impact on any of the other sublanguages.

As another example, we might expect to introduce new mapping constructs for testing conformance of actual systems to constraints that specify architectural standards. This would involve additions to the architecture sublanguage and also, possibly, changes to the Pattern sublanguage; beyond that the features of the other sublanguages should be not be impacted.

4.1.2 Types Language

The Types language was designed as a general language for defining types of modules in any modular language. The concept of module interface is equated with type. Subtyping between interface types is separated from any concept of code inheritance between modules. Checking that an interface is a subtype of another is done by a compiletime algorithm that is defined as part of the Types language.

If a module has an interface that is a subtype of another interface, then it can be substituted for any module with the super-type interface. This feature is intended to allow RAPIDE programs to be evolved from simple ones to more detailed ones quicky by developing more detailed modules and substituting them in place of earlier versions — this evolutionary development is thought to be important in "rapid prototyping". As experience with RAPIDE accumulates, it will become clear how much subtype substitution helps in evolving models of system architectures. This experience could feedback, e.g., to requiring more sophisticated subtyping that takes account of some kinds of semantic specifications associated with types. Changes to the Types language and subtyping should not impact any of the other sublanguages.

RAPIDE interface types define both the features that modules of that type *provide*, and also the features that modules of that type *require* from other modules. Thus, interfaces in RAPIDE differ from the usual programming languages concept of interface which specifies only the *provided* features. Specifying both provided and required features is an important requirement of interfaces in order to be able to define *interface connection* architectures (see section 2.2). The subtyping algorithm takes account of both kinds of features.

4.1.3 Predefined Types

Different sets of predefined types are needed for various domains of applications. Also, a federation of users that is organizing a set of programs, such as simulations, for inter-operation, will first need to agree upon a set of Predefined types for inter-operation. It seems therefore inadvisable to hardwire the predefined types into the design of a language like RAPIDE. Here, we have, by separating out the Predefined Types from the Types Language, emphasized that the predefined types are separate, special cases of interface types, and are open to alternative choices. Of course, it would be desirable to provide efficient implementations for any given set of predefined types, rather than to translate them into general interface types. This is considered a support tools issue.

4.1.4 Pattern Language

Patterns play many fundamental roles in RAPIDE, for example, in connections where they provide the expressive power to represent dynamic architectures, in formal constraints, and in reactive programming constructs. It was therefore thought appropriate to factor out patterns into a separate sublanguage.

Patterns provide a general language for defining event-based reactive constructs. Very elementary forms of patterns (usually single event triggers, or Boolean disjunctions of events in guards or sensitivity lists) are to be found in all event-based languages — e.g., VHDL, Verilog, and LOTOS. Prior to RAPIDE, patterns in previous event-based languages could only specify sets of events with no causal relationship. The RAPIDE pattern language provides the means to define patterns of events with causality, independence and timing relationships. A very simple pattern, for example, specifies a set of events, namely the events that *match* it. A complex pattern specifies the set of posets that match it.

Secondly, in future work it may be useful to introduce various subsets of the pattern language into new event-based language designs— e.g., the current CORBA IDL will probably need to be extended with some subset of event patterns in the future, e.g., to specify the asynchronous behavior of CORBA objects in their IDL interfaces. This was considered another argument for a separate pattern sublanguage.

Thirdly, in practice, it may turn out that the RAPIDE pattern language is something of an overkill in the sense that only a simple subset gets used in reactive rules (as opposed to constraints). Redesign or restriction of patterns in executable constructs can be achieved by defining subsets of the Pattern Language and referring to those subsets in the definition of the reactive rules. No other changes should be needed. Note that subsets of the Pattern Language may allow more efficient implementation of pattern matching than a general algorithm for all patterns.

Finally, patterns also play a major role in designing Constraint Languages for specifying formal constraints on event-based behavior. Constraint languages are orthogonal to executable simulation languages. It is therefore important to factor out the common features of both. The Pattern language does this.

4.1.5 Architecture Language

The Architecture sublanguage is intended for modelling interface connection architectures, as presented in Chapter 2. It defines three kinds of constructs: *(i)* behaviors and services which are additions to the Type language interfaces, *(ii)* connection rules for defining connections between interfaces in both static and dynamic architectures, and *(iii)* Maps and Bindings for comparing architectures.

Behaviors contain states (sets of objects) and reactive state transition rules. When a behavior is included, an interface can play a dual role of defining a type of module, and providing a simple module of that type (i.e., the behavior). [1] Interfaces with behaviors can be used as executable components of architectures.

Connection rules use patterns both as triggers to react to available events, and as event generators to generate new event sets. This provides a very powerful capability for defining dynamic architectures, and fan-in and fan-out connections between components. Architectures consisting of interfaces and connections are executable and can be used to simulate system architecture designs.

An interface can be structured into several groups of features. Such a group is called a *service*. Any service has a type. A type of service always has a *dual* type of service, analogously to a type of plug and the corresponding type of socket. A pair of services of dual types, say one in one interface and the other in a different interface, can be connected together in a natural and obvious way, very much like a plug and a corresponding socket.

Services in interfaces are intended to cope with *scale* of large architectures with many thousands of connections. Sets of related connections can be encapsulated by one connection between dual services – rather like a cable that bundles up a set of wires. *Duality* is defined in the Architectures LRM.

The following considerations led to separating the Architecture Language from, say, the Executable Language.

- The constructs for defining interface connection architectures should be clearly distinguished from standard kinds of programming constructs. The goal is to encourage a style in which a user restricts himself to types, patterns and architectures to model and simulate interface connection architectures. The communication protocols between components can be modelled at a global, system-wide level, and subjected to simulation and analysis, prior to considering issues of implementing individual modules.
- A style of gradual refinement of an architecture into a system is encouraged: executable modules are introduced one by one as replacements for each interface and connection in the architecture. At each step the new version of the system can be simulated and analyzed as described in Section 2.6.1 to see if it still conforms to its constraints.
- New language constructs for comparison of (possibly widely differing) architectures, and for testing conformance of systems to architectures (see Section 2.3), need to be designed. Separate Architecture and Constraint Languages provide better opportunities to do this than, say, putting everything, old and new, into one humongous language.
- Architecture features are clearly delimited from other language features for possible incorporation into other languages or systems.

4.1.6 Constraint Language

The Constraint Language provides features for defining formal constraints on the behavior of components and architectures. As shown in Figure 4.1, constraints are a sublanguage of architectures. Constraints on components can be included in their interfaces, and constraints on architectures can be included in a special constraint section of an architecture.

Constraints contain patterns defining posets that are either required or forbidden. Constraints on posets present a new research area, requiring investigation into syntax and semantics of

[1] Unfortunately, if equivalence of behaviors is considered a condition for subtyping, the subtype test becomes undecidable. So, at this time, behaviors are not considered in subtyping.

constraints, and implementation of checking algorithms. For this reason alone, it seemed appropriate to keep the design of the Constraint Language separate from Types and executable constructs.

4.1.7 Executable Language

The Executable Language contains common control structures found in many programming and simulation languages. We thought about eliminating these constructs completely. This would have required the user to define behaviors of components by means of the RAPIDE concurrent reactive rules which present a paradigm shift from conventional programming languages. New users of RAPIDE who are unfamiliar with concurrent reactive rules, particularly for event-based execution, appear more comfortable, at least initially, with writing modules to implement interfaces.

Also, the style of architecture driven system development described in Section 2.6.1 suggests a gradual replacement of behaviors in interfaces by modules containing "detailed" implementations. The Executable Language is therefore provided. It can easily be replaced by another programming language with diferent control structures. It should *not* be considered as an essential part of RAPIDE – another good reason to factor it out from other sublanguages. However, it does contain quite powerful concurrent processes which, for a new user coming from say a C++ background, can present a learning experience — especially when faced with a causal event simulation which really shows what a cocurrent program is doing.

Chapter 5

A Graphical Guide to Rapide

This chapter provides a top-down graphical summary of RAPIDE. It shows how the highest level language constructs are composed of constructs from the next lower level, and so on. RAPIDE has a very simple top-down structure which is, essentially, the view presented in Figure 4.1 reversed by going from top to bottom — except that types are top level constructs needed to define the components of architectures.

5.0.8 Architectures

Figure 5.1 shows the main elements in architectures: an architecture consists of components, connections between them, and constraints on the architecture's behavior. [1]

So, if we want to build an architecture, we must first define the types of its components, then some components, then connect them up, and finally constrain the activity of the connections, e.g., by adding, say, protocol constraints requiring connections to fire in particular orders.

A *Component* of an architecture is a module — i.e., an object of an interface type. It is shown as being of three kinds: an interface object (called simply an *interface*), an architecture that implements an interface, or a module that implements an interface.

An interface defines a type of objects if it is used in a type declaration to define a type. However, an object can be declared by an object declaration without any module implementation; in this case we also refer to such an object as "an interface". In this case, the behavior part of the interface (below) acts as the module implementing the object.

The second and third kinds of components are really the same kind: an architecture is a restricted kind of module that contains only some of the constructs normally allowed in modules.

Connections are shown in Figure 5.5. Connections in architectures are very similar to transition rules (in interface behaviors) and processes (in modules), so they are shown together in this figure. All three constructs are concurrent reactive rules. A reactive rule "reacts" when its pattern trigger is matched and then executes its body.

There are two kinds of connections, *basic* connections and *complex* connections. Basic connections define *identity* between pairs of interface features, e.g., a pair of functions, a pair of actions, or a pair of services. Identity is the strongest form of connection relationship. For example a basic connection between a required function of one component and a provided function of another defines the latter function as an alias for the former. Thus calls to the required

[1] See Figure 5.5 for the key to these diagrams.

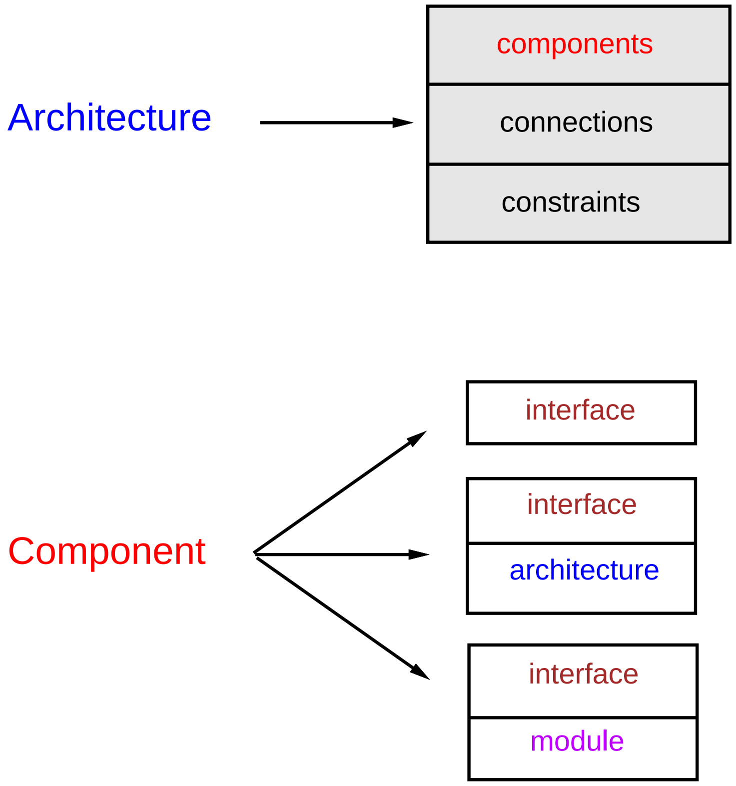

Figure 5.1: Graphical Synopsis of Rapide: Architectures

function are replaced by calls to the provided function. Basic connections between functions and actions are essential to defining connections between components of an architecture.

Complex connections define more general connections between components. They may trigger on posets of events generated by some components and then generate posets the events which are received by other components. Complex connections are essentially syntactic sugar for *connector* components that can be defined by interface types. There are two flavors of complex connections: *pipe* connections, which use the syntax "=>", and *agent* connections, which use the syntax "||>". Pipes are pipeline connectors and agents are multi-threaded connectors.

5.0.9 Interfaces

A module of an interface type must conform to the interface as discussed generally in Section 2.3. A module is also said to *have the interface* or to *implement the interface*.

An interface can have up to seven parts, all parts being optional, see Figure 5.2. The **provides** part declares the names and signatures of functions, modules and types that modules of that interface type provide to other modules. The **requires** part declares the names and signatures of functions, modules and types that will be required from other modules by modules of the interface type. A module can communicate synchronously with another module if it calls one of its interface required functions, and there is a connection that connects that function to a provided interface function of the other module.

The **action** part declares the **in** and **out** actions by means of which modules of the interface

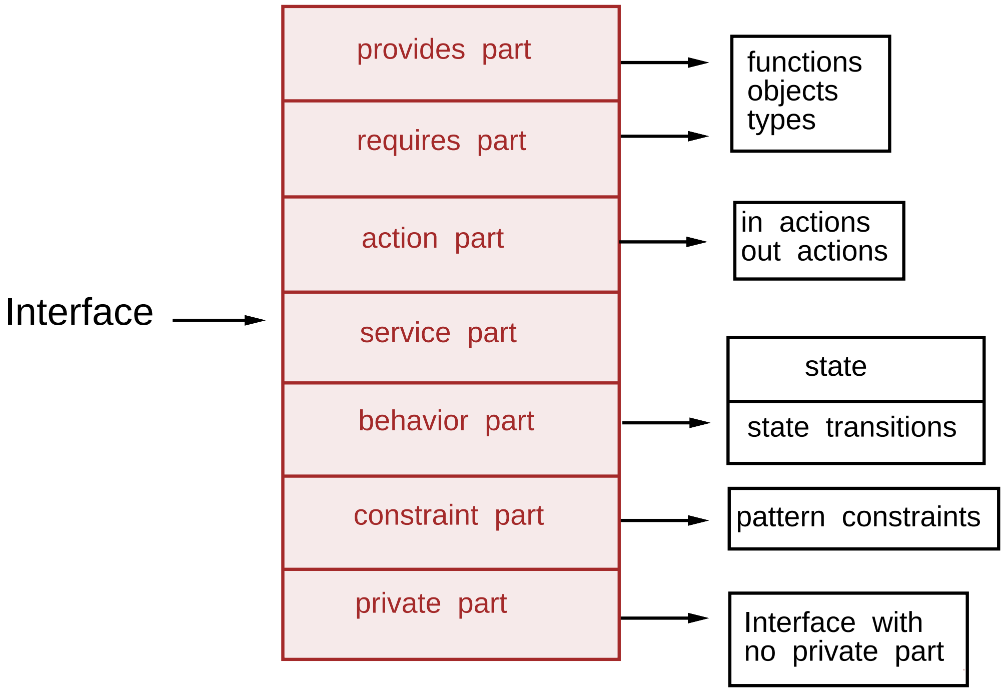

Figure 5.2: Graphical Synopsis of Rapide: Interfaces

type can communicate asynchronously with other modules. Communication is by sending and receiving events that are generated by calls to those actions, and then "transported" by connections.

The **service** part contains services (and **dual** services). Services are like sub-interfaces that define plugs or sockets — i.e., bundles of functions and actions that must be connected to dual bundles, rather like, say an RS-232 port. Services are explained in Architectures LRM.

The **behavior** part contains state (sets of objects) and transition rules that act on the state and also generate new events. Behaviors are rather simple kinds of modules that are defined in a very simple concurrent reactive language of pattern-triggered state-transition rules.

state transtions are shown in Figure 5.5. They are used in behaviors and maps. A state transtion consists of a pattern or a boolean condition that triggers the execution of a sequence of operations on the state of a behavior, followed by the generation of a pattern of events.

The **constraint** part contains constraints on the behavor of modules implementing the interface. Constraints may be input/output conditions on the parameters and return objects of functions, or they can be more general patterns of events that constrain the visible behavior (i.e., the sets of event visible at its interface) of the module.

The **private** part can contain interface parts, but no other private part. The visibility of declarations in a private part is limited to modules with the same interface. The purpose of private parts is to allow modules with the same interface to be implemented differently and to inter-operate as if they were instances of the same module.

5.0.10 Modules

Modules implement interface types. As shown in Figure 5.3 there are seven possible parts of a module, all optional. Note that all of the parts of an architecture are parts of a module. A module consisting of only these parts is called an architecture.

A module may also be an executable multi-threaded program, consisting of an initial part, executed first; then a process part, giving several processes that execute simultaneously; and a final part, to be executed when all processes have terminated. Constraints and exception handlers may also be parts of modules.

Processes (defined in modules) are shown in Figure 5.5. A process can consist of a simple list of statements, to be executed once, as well as reactive statements such as the **when** statement, which repeatedly waits for a pattern of events and triggers the execution of a body of statements when a match of the pattern is observed.

5.0.11 Maps

A map is a construct that defines mapping rules that transform the executions of one or more architectures (its domains) into an execution of another architecture or an interface (its range). The elements of maps are shown in Figure 5.4. The mapping rules define how sets of events generated in the domain map into events in the range.

Mappings rules are very similar to state transition rules in interface behaviors. So the same concept of pattern-triggered transition rules are used to transform the events generated by one architecture into events defined in another architecture. A map can be the domain of another map, so maps can be composed transitively. Maps may contain constraints that restrict the order in which their rules can trigger.

Maps are an experimental language construct for relating different levels in a design heirarchy. Their main application is intended to be in defining conformance of detailed architectures

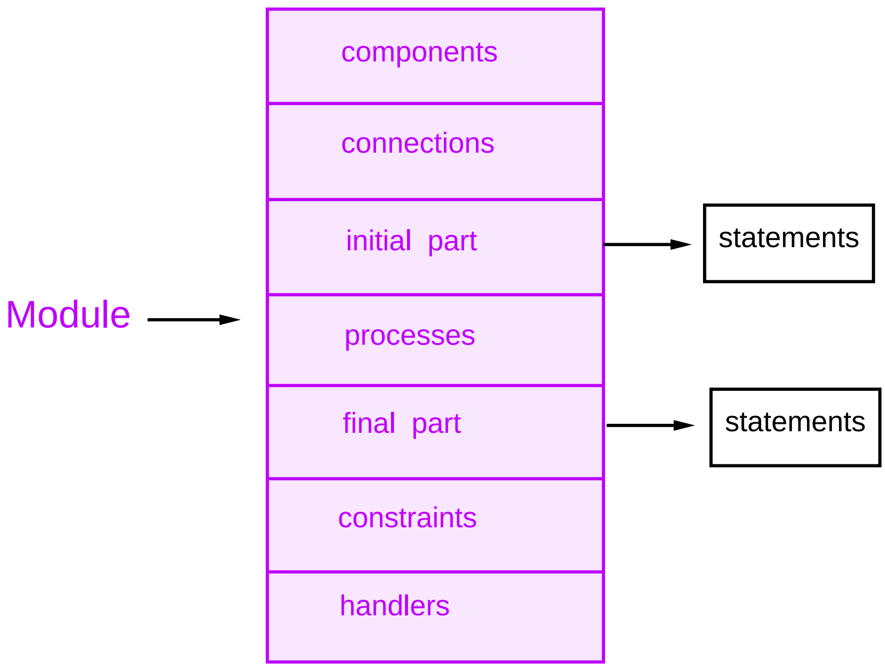

Figure 5.3: Graphical Synopsis of Rapide: Modules

to higher level, more abstract architectures. Compilation of maps allow them to be used as constraints to check conformance. Maps are defined in Architectures LRM.

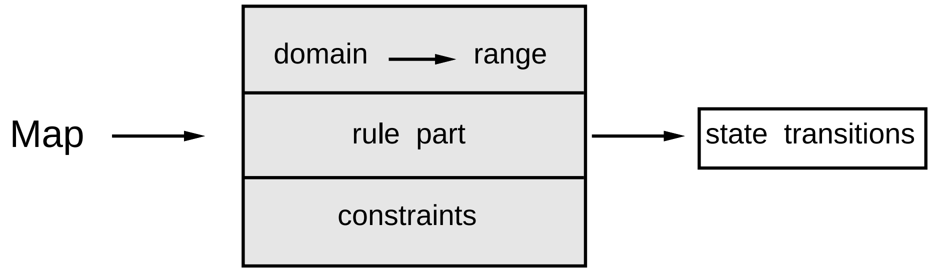

Figure 5.4: Graphical Synopsis of Rapide: Maps

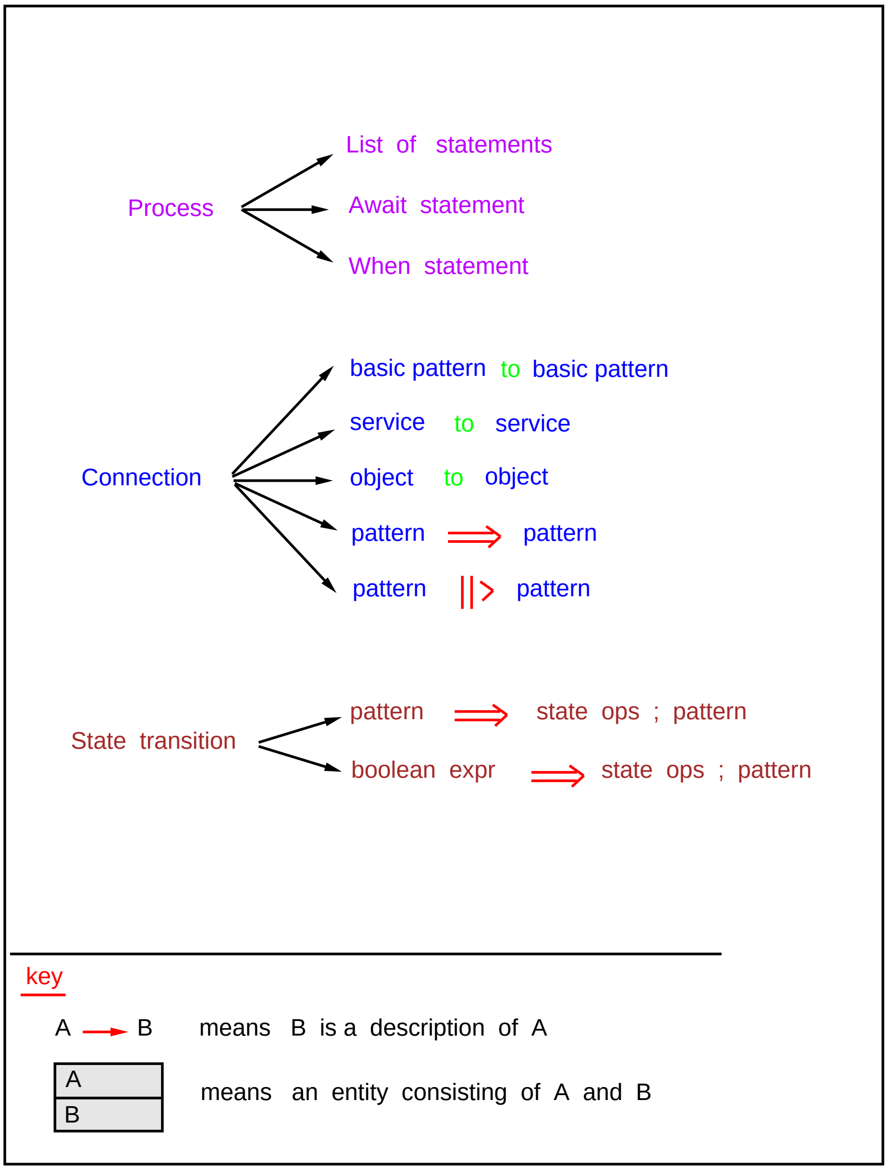

Figure 5.5: Graphical Synopsis of Rapide: Other Constructs

Glossary

ADL: Architecture Definition Language. See page 57.

Scalability: Issues related to representing large architectures are called *scalability* issues. See page 31.

communication integrity, of architectures: a property that an architecture defines all the possible communication paths between components. See page 20.

computation: A set of events together with the dependence and time partial orderings that relate events in the set. See page 37.

constraint: defines a patterns of events which must, or must not, occur during executions. See page 21.

dependence, between events: if event A causes B, then B is said to depend upn A. See page 39.

dual, of a service service S is a dual of service T if S provides the required features of T and requires the provides features of T. See page 31.

event pattern: a Pattern Language expression that matches sets of posets. See page 27.

instance, of an architecture: a system obtained by assigning a module to an interface of an interface connection architecture See page 29.

orderly observation, of events: events are observed for pattern matching in an order consistent with their causal order; an event cannot be observed before the events it depends upon. See page 41.

poset: a partially ordered set of events. See page 41.

poset: a set of events together with a partial ordering denoting the dependency (or causal) relation between the events. See page 26.

poset: a set of events together with a partial ordering denoting the dependency (or causal) relation between the events. See page i.

reactive process: a **when** statement in a module. See page 40.

reactive rule: a pattern triggered rule, i.e., a transition rule in a behavior, a connection in an architecture, or a map rule. See page 40.

service: An interface within another interface. See page 31.

sublanguange relation: L is a sublanguage of M if the syntax productions of L are a subset of the productions of M. See page 55.

Bibliography

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

[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.

[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.

[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.

[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.

[MMM91] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of standard ML modules with subtyping and inheritance. In *Proceedings of the ACM conference on POPL 1991.* ACM Press, January 1991. Also Stanford University Computer Systems Laboratory Technical Report No. CSL–TR–91–472.

[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 Architectures Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[Tea94b] 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.

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

[Tea94d] Rapide Design Team. *The Rapide-1 Predefined Types Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[Tea94e] 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.

[Tea94f] Rapide Design Team. *The Rapide-1 Types Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[Tea95] Rapide Design Team. *The Rapide-1 Full Syntax Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, August 1995.

[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.

Index