A RAPIDE-1.0 DEFINITION OF THE ADAGE AVIONICS SYSTEM

Walter Mann
Stanford University

Frank C. Belz and Paul Corneil
TRW Inc.

Technical Report: **CSL-TR-93-585**
(Program Analysis and Verification Group Report No. 66)

November 1993

This research was funded by ARPA under ONR contracts N00014-92-J-1928 and N00014-91-J-0173. Walter Mann was also funded by the Air Force Office of Scientific Research under Grant AFOSR-91-0354.

A Rapide-1.0 Definition of the ADAGE Avionics System

Walter Mann
Stanford University

Frank C. Belz and Paul Corneil
TRW Inc.

Technical Report: CSL-TR-93-585
Program Analysis and Verification Group Report No. 66

November 1993

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

We have used the Rapide prototyping-languages, developed by Stanford and TRW under the ARPA ProtoTech Program, in a series of exercises to model an early version of IBM's ADAGE software architecture for helicopter avionics systems. These exercises, conducted under the ARPA Domain Specific Software Architectures (DSSA) Program, also assisted the evolution of the Rapide languages. The resulting Rapide-1.0 model of the ADAGE architecture in this paper is substantially more succinct and illuminating than the original models, developed in Rapide-0.2 and Preliminary Rapide-1.0. All Rapide versions include these key features: *interfaces*, by which types of components and their possible interactions with other components are defined; *actions*, by which the events that can be observed or generated by such components are defined; and *pattern-based constraints*, which define properties of the computation of interacting components in terms of partially ordered sets of events. Key features of Rapide-1.0 include *services*, which abstract whole communication patterns between components; *behavior rules*, which provide a state-transition oriented specification of component behavior and from which computation component instances can be synthesized; and *architectures*, which describe implementations of components with a particular interface, by showing a composition of subordinate components and their interconnections. The Rapide-1.0 model is illustrated with corresponding diagrammatic representations.

Key Words and Phrases: formal specification, Rapide, avionics, architectural description

Walter Mann
Stanford University

Frank C. Belz and Paul Corneil
TRW Inc.

Contents

1 Introduction **1**
- 1.1 Rapide . . . 1
- 1.2 ADAGE . . . 1
- 1.3 Brief History . . . 2
- 1.4 Key Aspects of the Rapide-1.0 Language . . . 2
- 1.5 Status of the Model . . . 3
- 1.6 Structure of the Document . . . 3

2 The Top Level **5**
- 2.1 The Partial Top Level Interface . . . 6
- 2.2 The Partial Top Level Architecture . . . 9

3 Aircraft Sensors **13**
- 3.1 The Aircraft Sensor Interface . . . 13
- 3.2 The Aircraft Sensor Architecture . . . 14
  - 3.2.1 The Sensor Interface . . . 16

4 Data Sources **19**
- 4.1 The Data Sources Interface . . . 19
- 4.2 The Data Sources Architecture . . . 20
  - 4.2.1 The Device Driver Interface . . . 25
  - 4.2.2 The Data Source Interface . . . 28

5 Navigation **31**
- 5.1 The Navigation Interface . . . 31
- 5.2 The Navigation Architecture . . . 32
  - 5.2.1 The Earth Model Interface . . . 34
  - 5.2.2 The Atmosphere Model Interface . . . 34
  - 5.2.3 The Aircraft State Vector Model Interface . . . 35
  - 5.2.4 The VOR Radio Navigation Interface . . . 36

6 Guidance **38**
- 6.1 The Guidance Interface . . . 38
- 6.2 The Guidance Architecture . . . 39

7 Lateral Guidance **40**
- 7.1 The Lateral Guidance Interface . . . 40
- 7.2 The Lateral Guidance Architecture . . . 40
  - 7.2.1 The VOR Interface . . . 42
  - 7.2.2 The Direct Fixed-Point Interface . . . 45

7.2.3 The Direct Moving-Point Interface 48

8 Services **51**
8.1 Body Motion 51
8.2 Device Control 51
8.3 Navigation Control 51
8.4 Mission Objectives 52
8.5 Error Signals 52
8.6 Driver Sensor 52
8.7 Source Navigation Service 53
8.8 Aircraft State 53

9 Common Definitions **54**

10 Acknowledgements **58**

A The Original IBM ADAGE Specifications **59**

B An Alternative Sensor-Interface Model **73**

C A Unified Lateral-Guidance-Type Interface **77**

Chapter 1

Introduction

TRW and Stanford have conducted a series of exercises using their Rapide languages to define an early version of the ADAGE software architecture for helicopter avionics systems. This report provides a model for that definition.

1.1 Rapide

Rapide [LVB+93, MMM91] is a concurrent object-oriented language framework specifically designed for prototyping large concurrent systems. It consists of several sublanguages, including a *type language* for structuring systems into components, an *executable architecture definition language* for allowing sets of components to be combined and connected into larger systems, and a *constraint language* for specifying the behavior of components.

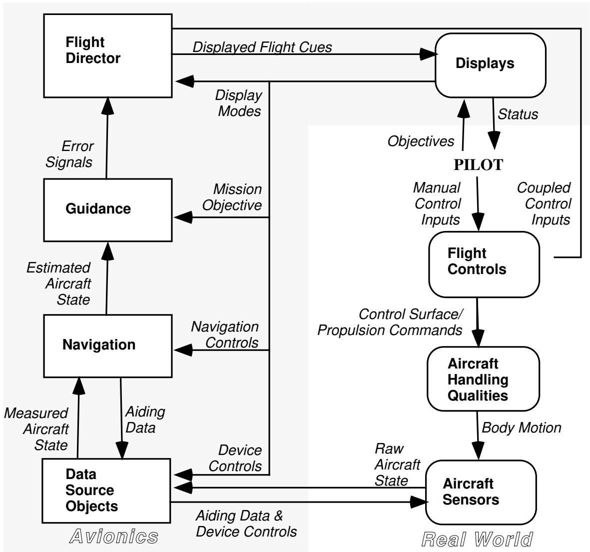
An initial version of the Rapide language, called Rapide-0.2, has previously been designed and implemented [Bry92, Hsi92]. Rapide-0.2 was used in our avionics exercises mainly to study the applicability of an execution model based on partially ordered events. Such models are actively being pursued in several research efforts worldwide [Fid91, SM91].

An architecture-definition language, μRapide [LV93], was also designed to analyze the requirements for executable architecture definitions.

These efforts have been combined into a first version of the full language, which is called Rapide-1.0 [Tea93b, Tea93a]. Development of tools for Rapide-1.0 is currently under way. Concurrently, preliminary investigation of a diagrammatic notation for Rapide-1.0 has begun. An initial notation, tentatively called Graphic Rapide, has been used for the diagrams in this document. The design and evaluation of the Rapide languages and their supporting tools is being conducted by Stanford and TRW as part of the Advanced Research Projects Agency's ProtoTech program and an Air Force Office of Scientific Research project on Foundations of Technology for Constructing Highly Reliable Distributed Realtime Systems.

1.2 ADAGE

ADAGE (Avionics Domain Application Generation Environment) is an environment for developing, specifying, and analyzing helicopter avionics software [TC93]. It is intended to be easily adaptable and reconfigurable. The ADAGE project is being conducted by an IBM-led team in the DSSA Program. The avionics software system architecture defined by IBM using the ADAGE environment includes Flight Director, Navigation, and Guidance subsystems and a collection of Data Source objects for managing data from external aircraft sensors (see figure).



1.3 Brief History

In May of 1992, Lou Coglianese of IBM sent to TRW a brief list of properties of a hypothesized avionics system architecture that was (at most) a precursor to the evolving ADAGE architecture. Lou wished to see what the expressive power of Rapide-0.2 was with respect to these properties. The items contained in this IBM communication are given in Appendix A (page 59), along with cross-references to the elements of the Rapide model described herein. A number of conversations ensued to resolve some, but not all, of the ambiguities in the list of properties.

The attempt to model a system with these properties in Rapide-0.2 stimulated an extensive analysis of Rapide-0.2 by TRW during a period in which Stanford was busily developing both a preliminary version of Rapide 1.0 and the preliminary architecture-definition language, μRapide. As a result, several fragmentary models were constructed by TRW, none corresponding completely to any of the versions of Rapide (none sufficed to capture all of the properties), but each correlated strongly with one of the languages. The expected value of the exercise to IBM has never been achieved; but the model has become a benchmark within the Stanford/TRW ProtoTech team by which language changes are judged. It has, therefore, had a significant impact on the evolution of Rapide.

1.4 Key Aspects of the Rapide-1.0 Language

Certain key aspects of Rapide-1.0 will be illustrated in subsequent sections of this document:

To represent architectural components of a system, the primary Rapide constructs are *interfaces*, for defining types of components, and *architectures*, for defining the structure of implementations of the components of each type. An interface defines what constituents of a component are visible to other components. A component's architecture declares other, subordinate, components (with defined interfaces, and possibly also having defined architectures) and defines connections between them. In general, an interface may be implemented by several architectures; that is, distinct components with different architectures may have the same interface (in Rapide jargon, they may be of the same interface type). In the ADAGE model presented here, each interface is implemented by at most one architecture.

Rapide languages are event-based; components communicate by generating and observing events. Each type of event a component can generate or observe is determined by an *action declaration* in the component's interface. Events generally represent communication and/or computation to be realized in the final implementation.

Sets of action declarations in an interface may be grouped into *services*; services are themselves defined by interfaces. Thus a related set of action declarations may be grouped into a single interface type, which is then used in a component's interface as a service. Services reduce the complexity of the higher-level descriptions and allow for later elaboration of the connections as requirements are refined.

The behavior of components (i.e., how they respond to patterns of observed events by producing patterns of generated events) can be defined using *behavior rules* in their interface definition. The behavior rules provide a simple state-machine definition which describes a component's reaction to patterns of events by describing how the component changes local states and generates further events. Behavior rules are designed to provide enough information that tools may derive prototype executable components of the type. Additionally, behavioral properties can be specified as *pattern constraints* defining required, permitted, and prohibited patterns of events.

1.5 Status of the Model

This report is an interim report on the partially rewritten model. It does not quite correspond to the current Rapide-1.0: in some cases we are behind the evolution of Rapide, not having exploited completely some powerful new features, such as type derivation, and in other cases we are ahead, having used some features that are on the list for introduction in Rapide in the next six months (without which the model fails to be adequately expressive). The model and the languages are expected to merge in the next few months.

We have included two appendices to give a flavor of directions that can be taken with the model to better take advantage of new features of Rapide. While the entire model should not be taken as authoritative in any measure with respect to avionics system architectures (we may be wrong on every detail that goes beyond the original properties, and some of those are probably incorrect), the appendices are particularly "bogus" in that they have never been considered to be valid architecturally. They exist not to explore architectural issues, but rather architecture *representation* issues.

1.6 Structure of the Document

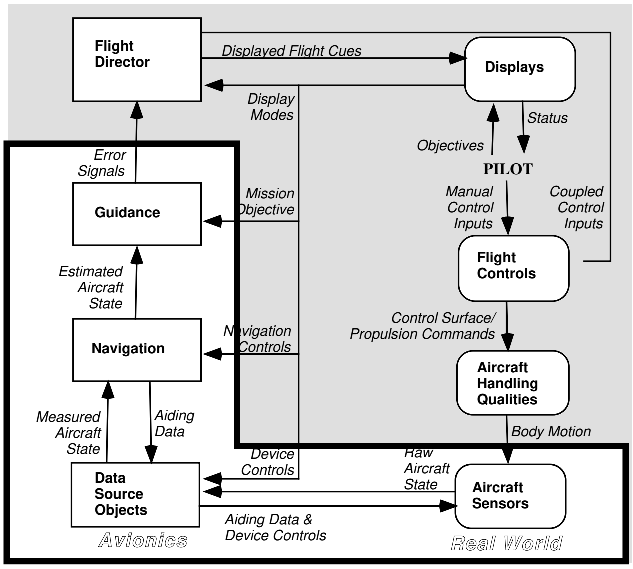
The chapters of this document are organized according to the structure of the ADAGE model. Each constituent interface and its architecture are sections of their respective chapter. An interface without corresponding architecture (a "leaf" interface in this "tree") is described in an appropriate subsection. Throughout, cross-references of the form "IBM 2.1.3.4" have been inserted into portions of our Rapide code referring to specific line numbers in Coglianese's specification; this therefore is the inverse of the mapping done in Appendix A, described above.

The ADAGE top-level interface and architecture are in Chapter 2. Chapters 3-6 (pages 13-38) put forth the four components of the top-level architecture—aircraft hardware sensors and three software components: data sources, navigation subsystem, and guidance subsystem. Lateral guidance, the only component of the guidance architecture actually elaborated in this model, is described in Chapter 7 (page 40). Communications services used are listed at their point of introduction but are also collected for reference in Chapter 8 (page 51). Common type and object declarations are collected in Chapter 9 (page 54).

Chapter 2

The Top Level

The topmost level of the ADAGE avionics, as given at the STARS '92 conference, is depicted in this figure (software on the left, hardware on the right). The Rapide model presented below addresses only the part of that architecture enclosed by the heavy outline.



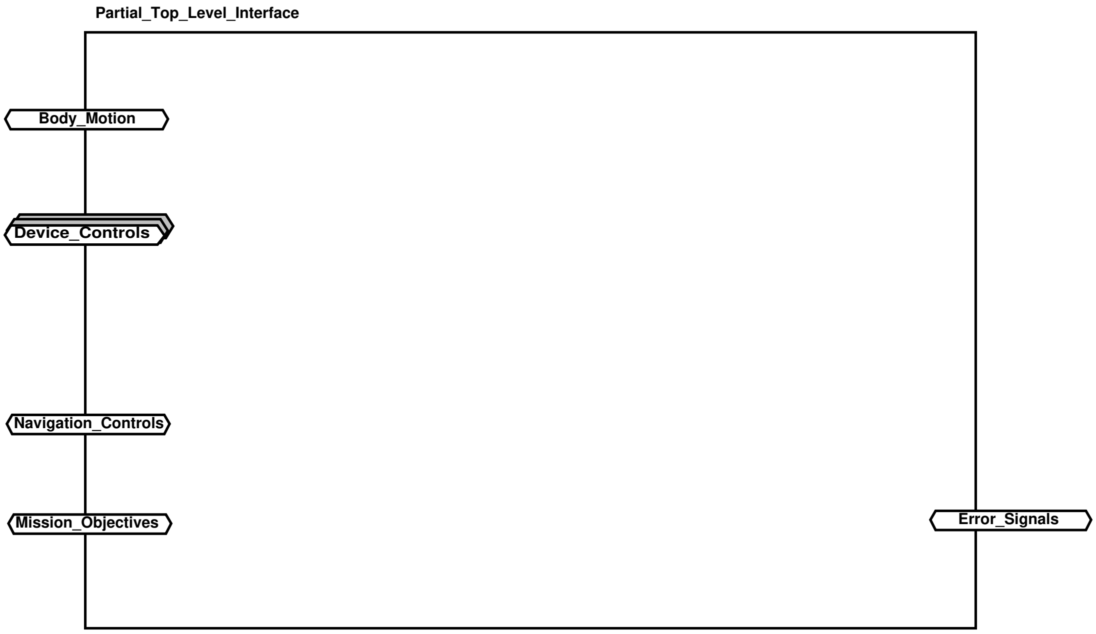
Furthermore, the lowest levels of this model are only sketched in; most of the detail of the actual architecture is elided. Finally, in the top level of the model presented below, the names of some elements will differ from those in the preceding diagrams, which portray an evolution of the ADAGE system somewhat beyond that originally presented to us for modelling.

The top-level description of the Rapide model includes its *interface* definition, which describes the model's interfaces with the rest of the ADAGE system, and its *architecture*, which specifies its internal components and their connections both with the external interfaces and with each other. As these make their appearance, each may be correlated with elements of the preceding figures.

2.1 The Partial Top Level Interface

Our Partial Top Level Interface shows the following external-communication interfaces, modeled by Rapide *services* whose types are defined later in this section:

- Body_Motion — a service of type Body_Motion_Service from outside the model to the aircraft sensors
- Device_Controls — an array of services of type Device_Control_Service, one for each kind of sensor, for controlling the data sources
- Navigation_Controls — a service of type Navigation_Control_Service, for communicating with navigation from outside
- Mission_Objectives — a service of type Mission_Objectives_Service which signals destination and arming information for the guidance subsystem
- Error_Signals — a service of type Error_Signals_Service for signals which originate in the guidance subsystem and determine the aircraft's status, its deviation from its intended heading.



The Rapide definition of the Partial Top Level Interface is then

```
type Partial_Top_Level_Interface is interface

    Body_Motion : service Body_Motion_Service;
    Device_Controls : service array(Sensor_Kind) of Device_Control_Service;
    Navigation_Controls : service Navigation_Controls_Service;
    Mission_Objectives : service Mission_Objectives_Service;

extern
    Error_Signals : service Error_Signals_Service;

end;
```

Services (and, below, actions) in an interface to which they are providing data are located in its public part (in which the keyword **public** may be omitted). Thus, a service (or action) regarded as providing data to an external component is defined there and thus must be in the **extern** part of any interface using it to provide the data. In the figures, "inputs" are located in the left walls of an interface and "outputs" in the right walls.

This model, extremely abbreviated in comparison to the full ADAGE avionics system, currently defines the above arrays of services in terms of just five sensor kinds, given by the enumerated elements in the Sensor_Kind type below:

```
    type Sensor_Kind is enum DNS, INU, GPS, ADC, VOR end;
```

- DNS — Doppler Navigation System
- INU — Inertial Navigation Unit
- GPS — Global Positioning System
- ADC — Air Data Computer
- VOR — VHF Omni-Range

This list, also contained in the Common_Definitions interface (page 54) may be extended, simply by adding further elements.

The five types of services (grouped together in Chapter 8, page 51) employed in the Partial Top Level Interface are defined below, also as interfaces, in terms of their constituent actions. Two, Body_Motion_Service and Navigation_Control_Service, are provided for future use but are presently empty, as we have no requirements from IBM for them.

```
type Body_Motion_Service is interface

end;


type Device_Control_Service is interface

    action Initialize_Sensor();
    action Align_Sensor();

    action Override_Data( S : Sensor_Data );
    action Select_Criteria( C : Selection_Criteria );
    action Aiding_Command( B : Boolean );

end;


type Navigation_Control_Service is interface

end;
```

```
type Mission_Objectives_Service is interface

public

    action Start_Guidance( K : Lateral_Guidance_Kind;
                           D : Lateral_Guidance_Data );
    action Stop_Guidance( K : Lateral_Guidance_Kind );
    action Arm_Guidance( K : Lateral_Guidance_Kind;
                         A : Lateral_Guidance_Arming_Data );
    action Disarm_Guidance( K : Lateral_Guidance_Kind );
    action Change_Guidance( K : Lateral_Guidance_Kind;
                            C : Lateral_Guidance_Revision_Data );

extern
    action Guidance_Started( K : Lateral_Guidance_Kind );
    action Guidance_Stopped( K : Lateral_Guidance_Kind );
    action Guidance_Armed( K : Lateral_Guidance_Kind );
    action Guidance_Disarmed( K : Lateral_Guidance_Kind );

end;


type Error_Signals_Service is interface

    action Heading_Error( D : Heading_Error_Type );

end;
```

Device_Control_Service above, in turn, uses Sensor_Data and Selection_Criteria types (which are included with other common definitions in Common_Definitions, Chapter 9, page 54):

```
    type Sensor_Data is record
        Quality : var Sensor_Data_Quality;  -- Initially Unusable;
        Measured_Aircraft_State : var Raw_Data; -- Initially Default_Measured_Aircraft_State;
    end record;

    type Sensor_Data_Quality is enum Usable, Degraded, Unusable end;

    type Raw_Data is record
        Position: var Position_Type;
        Velocity: var Velocity_Type;
        Attitude: var Attitude_Type;
    end record;

    type Position_Type is (TBD);
    type Velocity_Type is (TBD);
    type Attitude_Type is (TBD);

    Default_Measured_Aircraft_State : Raw_Data;  -- Initially Raw_Data'(TBD);
```

```
type Selection_Criteria is record
    Option : Selection_Options;
    Kind : Sensor_Kind;
    Sensor_Num : Positive;
end record;

type Selection_Options is enum External, Best_Available, Sensor end;
type Sensor_Kind is enum DNS, INU, GPS, ADC, VOR end;
```

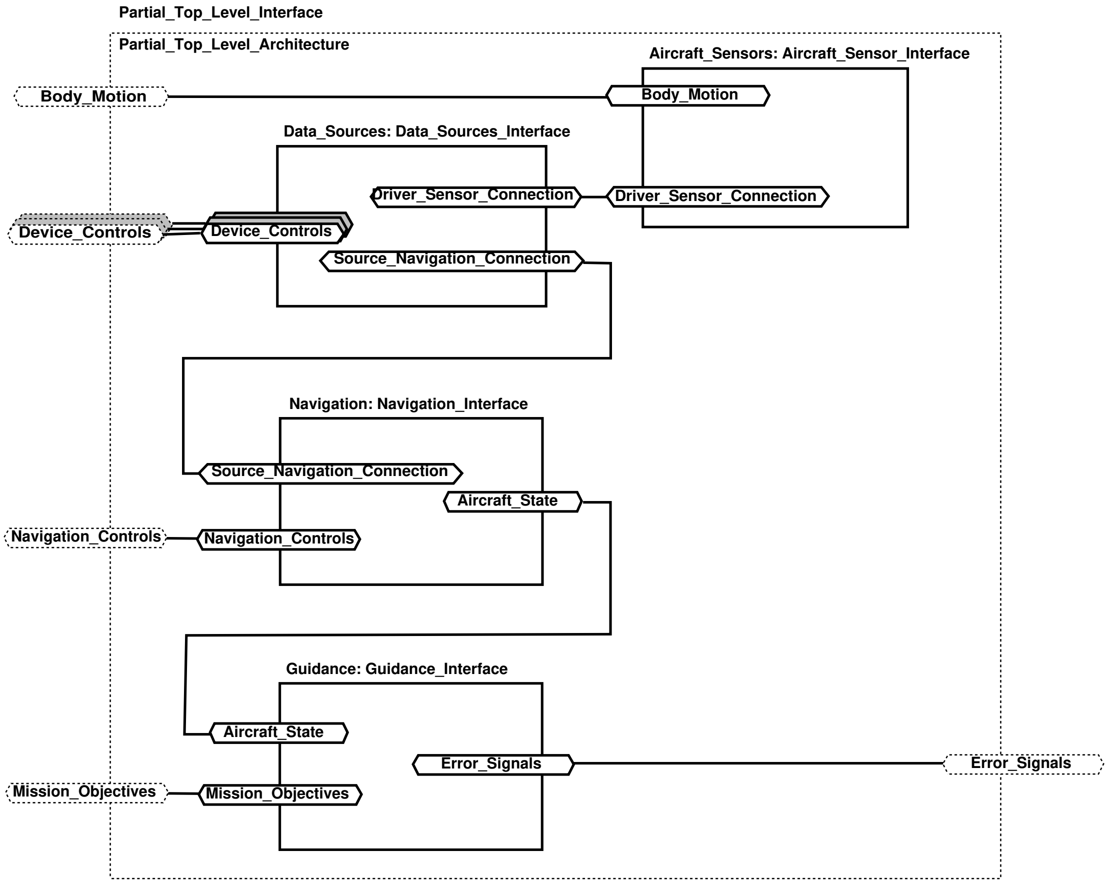
Mission_Objectives_Service defines its public actions in terms of a number of lateral-guidance types from the Common_Definitions interface:

```
type Lateral_Guidance_Kind is enum Direct_Fixed, Direct_Moving, VOR end;

type Lateral_Guidance_Data is record
    Destination : var Destination_Point;    -- for Direct_Fixed and Direct_Moving
    Heading : var Heading_Type;             -- for Direct_Moving
    Speed : var Speed_Type;                 -- for Direct_Moving
    Radial : var Radial_Type;               -- for VOR
end record;

type Lateral_Guidance_Arming_Data is record
    Destination : var Destination_Point;    -- for Direct_Fixed and Direct_Moving
    Capture : var Destination_Point;        -- for Direct_Fixed and Direct_Moving
    Heading : var Heading_Type;             -- for Direct_Moving
    Speed : var Speed_Type;                 -- for Direct_Moving
    Radial : var Radial_Type;               -- for VOR
end record;

type Lateral_Guidance_Revision_Data is record
    Destination : var Destination_Point;    -- for Direct_Moving
    Heading : var Heading_Type;             -- for Direct_Moving
    Speed : var Speed_Type;                 -- for Direct_Moving
end record;
```

These and Error_Signals_Service use several types which have for now been left undefined in the Rapide model: Heading_Type, Speed_Type, Radial_Type, and Heading_Error_Type.

2.2 The Partial Top Level Architecture

The four ADAGE components introduced in Chapter 1 comprise the Partial Top Level Architecture; each is a Rapide object whose interface type is described in its own chapter later: Aircraft Sensors (Section 3.1, page 13), Data Sources (Section 4.1, page 19), Navigation (Section 5.1, page 31), and Guidance (Section 6.1, page 38).



The Aircraft_Sensors component of this architecture presently provides only trivial communication with the environment external to the Rapide model using the Body_Motion service (which is empty, as explained in the Partial Top Level Interface, Section 2.1). The Data_Sources component communicates externally by way of the Device_Controls array of services previously introduced. Navigation would communicate externally via the Navigation_Controls service (if it also were not empty), as explained in the previous section. The Guidance component communicates with the external environment by way of the Mission_Objectives and Error_Signals services of the Partial Top Level Interface.

Then internally there are three more connections:

- Driver_Sensor_Connection — a service of type Driver_Sensor_Services between the data sources and the sensors, which abstracts the communication of commands and data between particular sensors and their drivers
- Source_Navigation_Connection — a service from the data sources to the navigation subsystem
- Aircraft_State — a service of type Aircraft_State_Service between the navigation and guidance interfaces

The Rapide definition of this architecture declares the components and institutes their connections in the **connect** part:

```
architecture Partial_Top_Level_Architecture() for Partial_Top_Level_Interface is

    Aircraft_Sensors : Aircraft_Sensor_Interface is Aircraft_Sensor_Architecture;
    Data_Sources : Data_Sources_Interface is Data_Sources_Architecture;
    Navigation : Navigation_Interface is Navigation_Architecture;   -- IBM 2.0[1]
    Guidance : Guidance_Interface is Guidance_Architecture;   -- IBM 3.0

connect

    ?K : Sensor_Kind;

    -- Architecture inputs to internal components:

    Device_Controls(?K) to Data_Sources.Device_Controls(?K);
    Navigation_Controls to Navigation.Navigation_Controls;
    Mission_Objectives to Guidance.Mission_Objectives;


    -- Internal components to architecture outputs:

    Guidance.Error_Signals to Error_Signals;


    -- Connections among internal components:

    Data_Sources.Driver_Sensor_Connection to Aircraft_Sensors.Driver_Sensor_Connection;
    Data_Sources.Source_Navigation_Connection to Navigation.Source_Navigation_Connection;
    Navigation.Aircraft_State to Guidance.Aircraft_State;

end;
```

[1] Comments containing IBM are cross-references to the corresponding original specifications from Lou Coglianese, as described in Section 1.6.

Since the Rapide definitions of the external communications services were in the previous section, we focus on the new internal communications.

The service Driver_Sensor_Connection used above is declared in both Aircraft_Sensor_Interface and Data_Sources_Interface to be of type Driver_Sensor_Services. Below is its Rapide definition, along with Driver_Sensor_Service, used in defining the elements of Driver_Sensor_Services type (both found in Section 8.6, page 52):

```
type Driver_Sensor_Services is interface

    DNS_DS : service array(DNS_Range) of Driver_Sensor_Service;
    INU_DS : service array(INU_Range) of Driver_Sensor_Service;
    GPS_DS : service array(GPS_Range) of Driver_Sensor_Service;
    ADC_DS : service array(ADC_Range) of Driver_Sensor_Service;
    VOR_DS : service array(VOR_Range) of Driver_Sensor_Service;

end;
```

```
type Driver_Sensor_Service is interface

    action Request_Sensor_Data();
    action Request_Init_Device();
    action Request_Align_Device();
    action Aiding_Data_To_Sensor( D : Sensor_Data );

extern
    action Sampled_Sensor_Data( R : Raw_Data );
    action Device_Initialized();
    action Device_Aligned();

end;
```

Driver_Sensor_Services is composed of a separate service for each kind of sensor (Sections 2.1 and 3). Each of these services is composed of an array of Driver_Sensor_Service elements, one element for each sensor actually defined in the architecture. For each kind of sensor, a range (found in the Common_Definitions interface, Chapter 9) defines the number of sensors of that kind present in the architecture. By convention, these ranges are from 1 through Nr_of_sensors of the given kind:

```
    type DNS_Range is Range(Integer, 1, 1);
    type INU_Range is Range(Integer, 1, 2);
    type GPS_Range is Range(Integer, 1, 1);
    type ADC_Range is Range(Integer, 1, 1);
    type VOR_Range is Range(Integer, 1, 1);
```

The service Source_Navigation_Connection is declared in both Data_Sources_Interface and in Navigation_Interface to be of type Source_Navigation_Service (Section 8.7, page 53):

```
type Source_Navigation_Service is interface

    action Aircraft_State_Data( K : Sensor_Kind; S : Sensor_Data );

extern
    action Navigation_Aiding_Data;

end;
```

Because we have no requirements from IBM for its action Navigation_Aiding_Data, it is presently not used.

The service Aircraft_State is declared in Navigation_Interface and Guidance_Interface to be of type Aircraft_State_Service (Section 8.8, page 53):

```
type Aircraft_State_Service is interface

    action Aircraft_State_Vector( V : State_Vector );
    action VOR_Relative_Navigation_Output();

end;
```

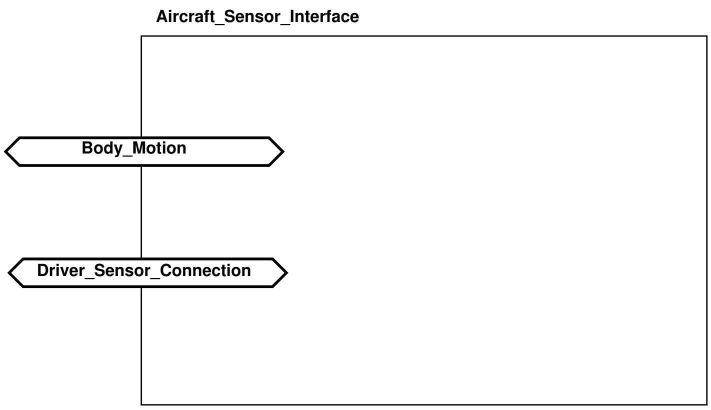
Chapter 3

Aircraft Sensors

The Aircraft Sensor component of the Partial Top Level Architecture represents the hardware sensors. Five kinds are defined currently: DNS (Doppler Navigation System), INU (Inertial Navigation Unit), GPS (Global Positioning System), ADC (Air Data Computer), and VOR (VHF Omni-Range). This list could easily be extended, by adding further elements to the Sensor_Kind type in the Common_Definitions interface (Section 9, page 54), which we saw in the Partial Top Level Interface (Section 2.1). Each kind of sensor has a sensor array, the size of which determines the number of sensors of that kind actually in the architecture. The range constants (from Common_Definitions, Section 9 [page 54]) were given in the preceding Section 2.2.

3.1 The Aircraft Sensor Interface

As we have seen (Section 2.1), the service Body_Motion is empty at present for lack of specifications from IBM; thus it does not connect with the underlying Aircraft Sensor Architecture. The service Driver_Sensor_Connection is the only constituent of the Aircraft Sensor Architecture visible outside the Aircraft Sensor Interface.

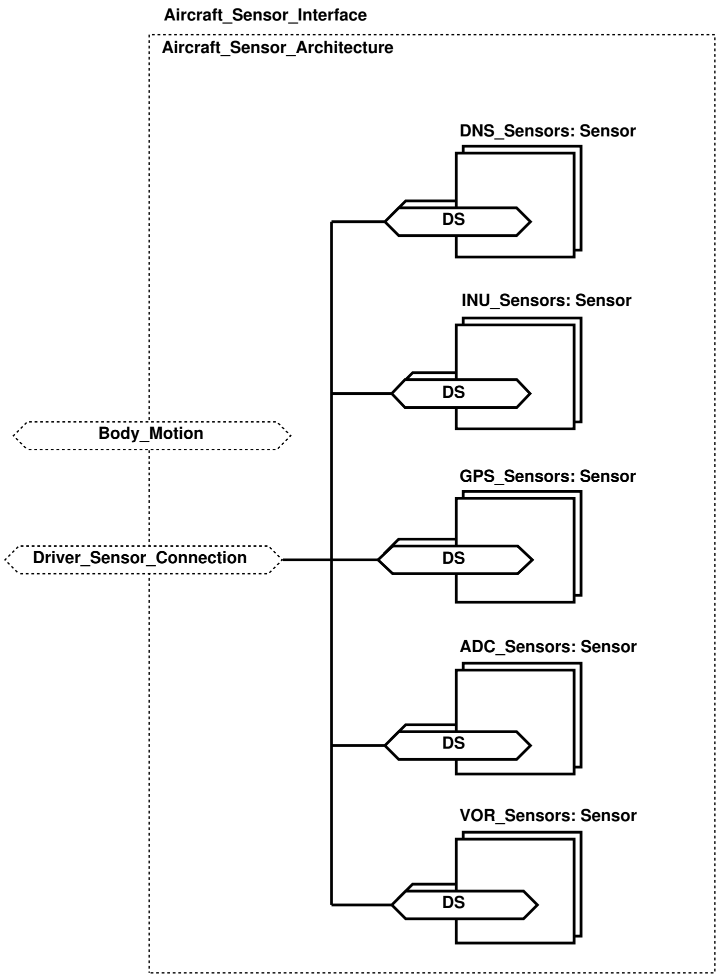


Driver_Sensor_Connection is a single service, of type Driver_Sensor_Services, whose Rapide code was given for the Top_Level_Architecture (page 9) and is also found in its section of the Services Chapter (Section 8.6, page 52). This service provides the communication path between each sensor and its driver. (Drivers are defined in the Data Sources Architecture [Section 4.2, page 20] and its Device Driver Interface [Section 4.2.1, page 25].)

```
type Aircraft_Sensor_Interface is interface

    Body_Motion : service Body_Motion_Service;
    Driver_Sensor_Connection : service Driver_Sensor_Services;

end;
```

3.2 The Aircraft Sensor Architecture

The arrays of sensors which make up this architecture are defined here.



The **connect** part of this architecture defines the fan-out from the single Driver_Sensor_Connection service to the individual sensors.

```
architecture Aircraft_Sensor_Architecture() for Aircraft_Sensor_Interface is

    DNS_Sensors : array(DNS_Range) of
        Sensor( DNS_Init_Duration, DNS_Align_Duration );
    INU_Sensors : array(INU_Range) of
        Sensor( INU_Init_Duration, INU_Align_Duration );
    GPS_Sensors : array(GPS_Range) of
        Sensor( GPS_Init_Duration, GPS_Align_Duration );
    ADC_Sensors : array(ADC_Range) of
        Sensor( ADC_Init_Duration, ADC_Align_Duration );
    VOR_Sensors : array(VOR_Range) of
        Sensor( VOR_Init_Duration, VOR_Tuning_Duration );

    ?D : DNS_Range;
    ?I : INU_Range;
    ?G : GPS_Range;
    ?A : ADC_Range;
    ?V : VOR_Range;

connect

    Driver_Sensor_Connection.DNS_DS(?D) to DNS_Sensors(?D).DS;
    Driver_Sensor_Connection.INU_DS(?I) to INU_Sensors(?I).DS;
    Driver_Sensor_Connection.GPS_DS(?G) to GPS_Sensors(?G).DS;
    Driver_Sensor_Connection.ADC_DS(?A) to ADC_Sensors(?A).DS;
    Driver_Sensor_Connection.VOR_DS(?V) to VOR_Sensors(?V).DS;

end;
```

The following selections from the common data declarations (page 54) describe data items above:

```
    type Sensor_Kind is enum DNS, INU, GPS, ADC, VOR end;

    type DNS_Range is Range(Integer, 1, 1);
    type INU_Range is Range(Integer, 1, 2);
    type GPS_Range is Range(Integer, 1, 1);
    type ADC_Range is Range(Integer, 1, 1);
    type VOR_Range is Range(Integer, 1, 1);

    Msec :   Time is 1;
    Sec :    Time is 1000;
    Never :  Time is (TBD);

    DNS_Init_Duration :     Time is 1 * Sec;
    DNS_Align_Duration :    Time is Never;

    INU_Init_Duration :     Time is 3 * Sec;
    INU_Align_Duration :    Time is 5 * Sec;

    GPS_Init_Duration :     Time is 10 * Sec;
    GPS_Align_Duration :    Time is Never;

    ADC_Init_Duration :     Time is 1 * Sec;
```

```
ADC_Align_Duration :      Time is Never;

VOR_Init_Duration :       Time is 1 * Sec;
VOR_Tuning_Duration :     Time is 100 * Msec;
```

This model assumes that all sensors of a particular kind have the same initialization and alignment properties; this may not be true of the actual ADAGE architecture. The above connection rules mean the following: the particular sensor-driver service (selected by the appropriate Range index) from the array of services for that kind of sensor connect to the DS service of the corresponding particular sensor (also selected by the appropriate Range index).

The only interface type used for objects in this architecture is Sensor, the type of the sensor-array elements. It is described in the next section.

3.2.1 The Sensor Interface

This interface type, used to define the elements of the Aircraft Sensor Architecture sensor arrays, includes a **behavior**, which defines rules for the behavior of objects of this type (that is, sensor objects). These rules are sufficiently complete to constitute an executable prototype of such sensors. The behavior defines abstract states and pattern-triggered *state-transition rules*. These rules consist of two components: the *trigger*, a pattern which is matched against events observed by interface objects, and the *transition statement*, defining a transition to be performed when the triggering pattern of events is observed. The transition statement can change the state and perform events in response to those which triggered the statement.

For example, in the code below, the first state-transition rule following the comments is triggered and the states are changed when an event Request_Init_Device is observed via the DS service (whose Driver_Sensor_Service type, with its concomitant actions, we have seen already, in Section 2.2, page 9). If, as in the third state-transition rule, a guard (a boolean statement following **where**) is present, the state is changed only if the guard is true when the event is observed. In these state changes, the global object Clock, defined as follows:

```
Clock : Clock_Pkg::Clock is Make_Clock();
```

is used to enforce specified time durations; it contains the function Now() by which one accesses the current time with respect to that clock object. Rapide can model time with one or more clocks, whose “ticks” have integer value.

Some sensors also require *alignment* before they may be used. The Sensor interface models sensors both with and without alignment. Sensors without alignment are instantiated from this interface by passing the constant Never as the value of formal parameter Align_Duration.

The behavior defined here is not necessarily the behavior of sensors assumed in the actual ADAGE architecture. Here the requirements provided by IBM were ambiguous, and we found it necessary to disambiguate. Per IBM, sensor data Some_Data are passed by DS.Sampled_Sensor_Data irrespective of the initialization or alignment state of the sensor; the data will then be labeled Usable or Unusable in the Device-Driver Interface (Section 4.2.1, page 25) according to the preparedness of the sensor.

An alternate, more modular, formulation of the sensor interface is provided in Appendix B (page 73).

```
type Sensor( Init_Duration, Align_Duration : Integer ) is interface

   DS : service Driver_Sensor_Service;
```

behavior

```
State : var Sensor_State_Type;   -- Initially want (Not_Initted, Not_Aligned);

Actual_Initialization_Complete_Time : var Time := Never;
Actual_Alignment_Complete_Time : var Time := Never;

Some_Data : function() return Raw_Data;

begin

-- IBM 2.1.2.2, 2.2.2.2, 2.3.2.2, 2.4.2.2, 2.5.2.2, 2.6.2.2
-- IBM 2.1.2.5, 2.2.2.5, 2.3.2.5, 2.4.2.5, 2.5.2.5, 2.6.2.5
-- When an initialization request comes in for a non-initialized sensor,
-- set a parameter to the time the init should finish.  Then perform
-- Device_Initialized only when the clock time equals that parameter
-- (any intervening init requests having reset its value).

DS.Request_Init_Device =>
    State := Sensor_State_Type'( Initting, Not_Aligned );
    Actual_Initialization_Complete_Time := Clock.Now() + Init_Duration;;

Clock.Now() = Actual_Initialization_Complete_Time and State.I = Initting =>
    State.I := Initted;
    Actual_Initialization_Complete_Time := Never;
    DS.Device_Initialized;;

-- IBM 2.1.3.2, 2.2.3.2, 2.3.3.2, 2.4.3.2, 2.5.3.2, 2.6.3.2
-- IBM 2.1.3.3, 2.2.3.3, 2.3.3.3, 2.4.3.3, 2.5.3.3, 2.6.3.3
-- IBM 2.1.3.5, 2.2.3.5, 2.3.3.5, 2.4.3.5, 2.5.3.5, 2.6.3.5
-- Similarly for alignment if implemented:

DS.Request_Align_Device where (State.I = Initted and Align_Duration /= Never) =>
    State.A := Aligning;
    Actual_Alignment_Complete_Time := Clock.Now() + Align_Duration;;

Clock.Now() = Actual_Alignment_Complete_Time and State.A = Aligning =>
    State.A := Aligned;
    Actual_Alignment_Complete_Time := Never;
    DS.Device_Aligned;;

DS.Request_Sensor_Data =>
    DS.Sampled_Sensor_Data(Some_Data());;


-- Initialization
Start =>
    State := Sensor_State_Type'(Not_Initted, Not_Aligned);;
```

end;

Again, from Common_Definitions (Section 9, page 54), we get the record used for defining State above:

```
type Sensor_State_Type is record
    I : var Init_State_Type; -- Initially Not_Initted;
    A : var Align_State_Type; -- Initially Not_Aligned;
end record;

type Init_State_Type is enum Not_Initted, Initting, Initted end;
type Align_State_Type is enum Not_Aligned, Aligning, Aligned end;
```

Raw_Data, used for defining Some_Data above, was introduced in the Partial Top Level Interface (Section 2.1, page 6).

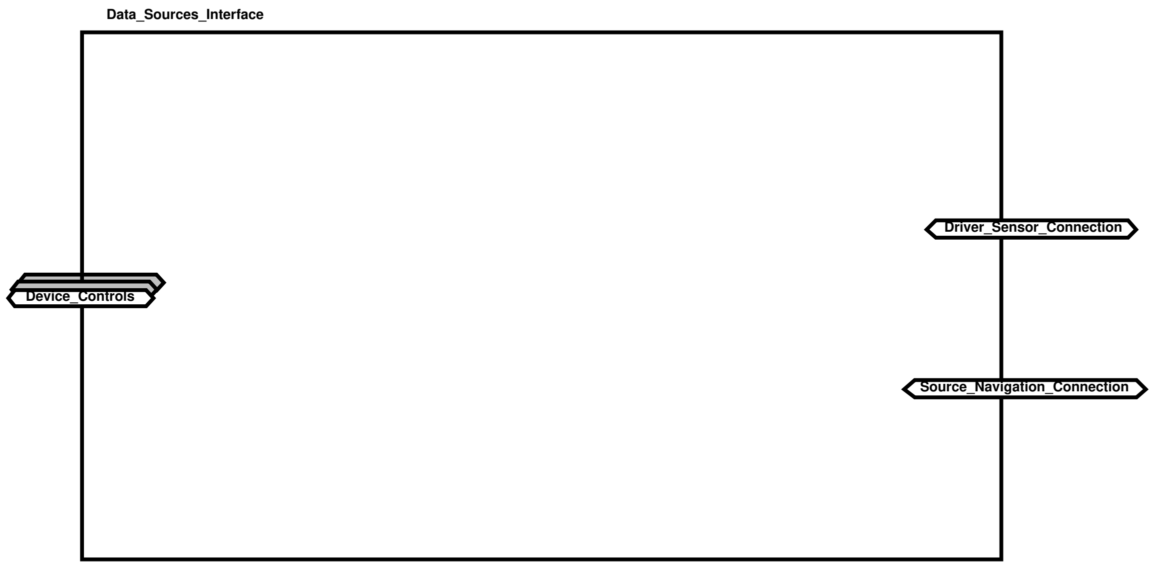
Chapter 4

Data Sources

As we have seen, in terms of communication interfaces this is the most complex component of the Partial Top Level Architecture (Section 2.2, page 9).

4.1 The Data Sources Interface

The figure shows the interface with its communications services already seen in the Partial Top Level Interface (Section 2.1, page 6) and Architecture.



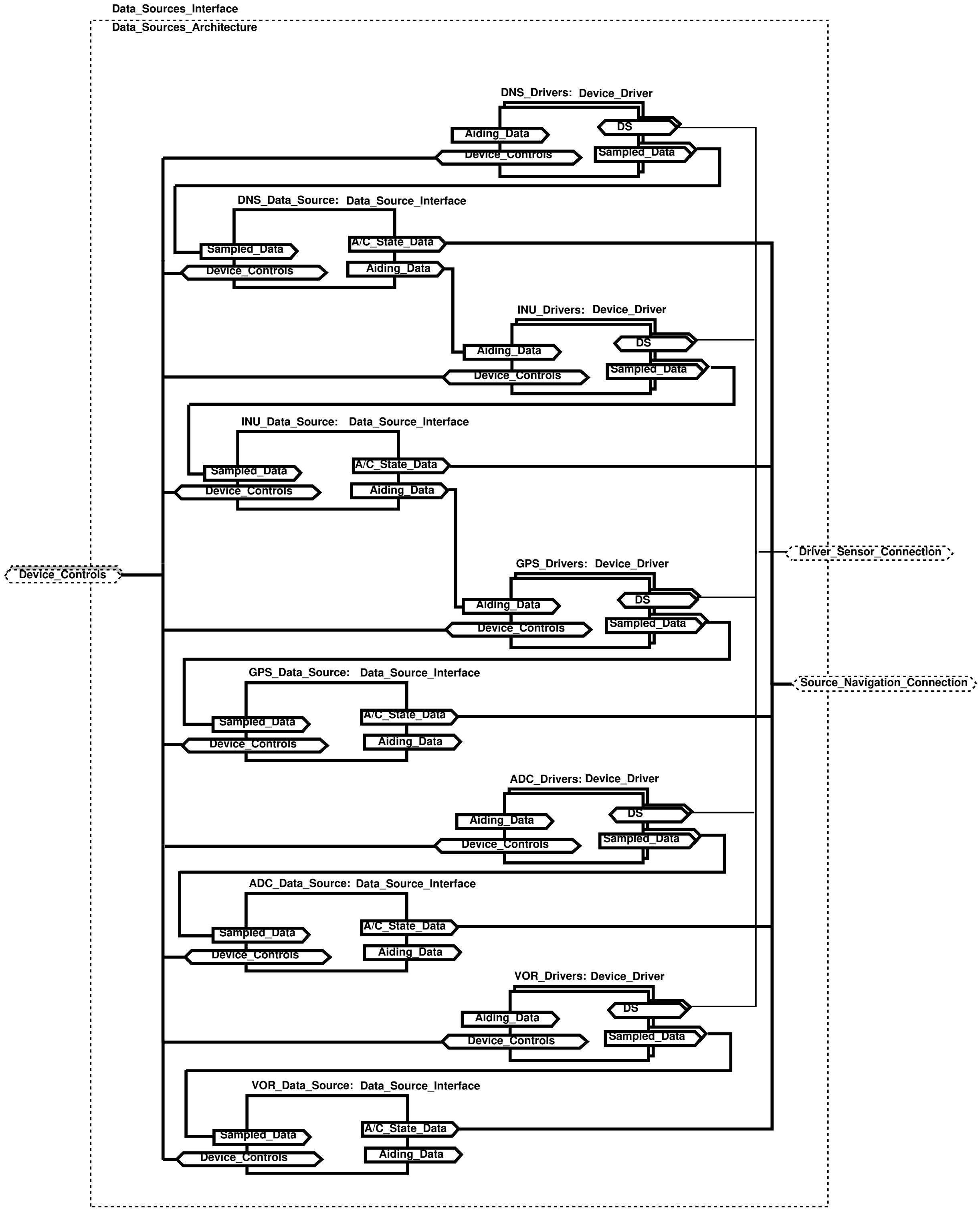
Thus the Rapide code for this interface contains nothing new to us:

```
type Data_Sources_Interface is interface

    Device_Controls : service array(Sensor_Kind) of Device_Control_Service;

extern
    Driver_Sensor_Connection : service Driver_Sensor_Services;
    Source_Navigation_Connection : service Source_Navigation_Service;

end;
```

The Driver_Sensor_Connection, defined in the Aircraft_Sensor_Interface (Section 3.1, page 13), is therefore **extern** here.

4.2 The Data Sources Architecture

This architecture declares the data sources and drivers for each sensor and establishes their connections.



The input service is a previously introduced (Section 2.1) array, Device_Controls, which defines the control events for the device drivers and controls selection criteria and aiding data.

For each kind of sensor, the architecture defines an array of device drivers, each an interface of type Device_Driver (Section 4.2.1, page 25), and one data source of type Data_Source_Interface (Section 4.2.2, page 28). The communication services for each driver are fanned in to the single service Driver_Sensor_Connection. Data from all data sources are fanned in to the single action Aircraft_State_Data of the service Source_Navigation_Connection. Both of these are familiar by now (Section 2.2, page 9).

Aiding data sent from some data sources to drivers are represented by the last two internal connections, between the components involved.

```
architecture Data_Sources_Architecture() for Data_Sources_Interface is

   -- IBM 2.4
   DNS_Drivers : array( DNS_Range ) of
       Device_Driver( DNS_First_Sample, DNS_Sampling_Interval,
                      DNS_Aiding_Interval, DNS_Align_Duration );
   DNS_Data_Source :
  Data_Source_Interface( DNS_Range, DNS, DNS_First_Sample,
                         DNS_Sampling_Interval, DNS_Aiding_Interval );

   -- IBM 2.1, 2.2
   INU_Drivers : array( INU_Range ) of
       Device_Driver( INU_First_Sample, INU_Sampling_Interval,
                      INU_Aiding_Interval, INU_Align_Duration );
   INU_Data_Source :
  Data_Source_Interface( INU_Range, INU, INU_First_Sample,
                         INU_Sampling_Interval, INU_Aiding_Interval );

   -- IBM 2.3
   GPS_Drivers : array( GPS_Range ) of
       Device_Driver( GPS_First_Sample, GPS_Sampling_Interval,
                      GPS_Aiding_Interval, GPS_Align_Duration );
   GPS_Data_Source :
  Data_Source_Interface( GPS_Range, GPS, GPS_First_Sample,
                         GPS_Sampling_Interval, GPS_Aiding_Interval );

   -- IBM 2.5
   ADC_Drivers : array( ADC_Range ) of
       Device_Driver( ADC_First_Sample, ADC_Sampling_Interval,
                      ADC_Aiding_Interval, ADC_Align_Duration );
   ADC_Data_Source :
  Data_Source_Interface( ADC_Range, ADC, ADC_First_Sample,
                         ADC_Sampling_Interval, ADC_Aiding_Interval );

   -- IBM 2.6
   VOR_Drivers : array( VOR_Range ) of
       Device_Driver( VOR_First_Sample, VOR_Sampling_Interval,
                      VOR_Aiding_Interval, VOR_Tuning_Duration );
   VOR_Data_Source :
  Data_Source_Interface( VOR_Range, VOR, VOR_First_Sample,
                         VOR_Sampling_Interval, VOR_Aiding_Interval );

   ?D : DNS_Range;
   ?I : INU_Range;
   ?G : GPS_Range;
```

```
    ?A  :  ADC_Range;
    ?V  :  VOR_Range;

    ?S  :  Sensor_Data;


connect

    -- Service connections:

    -- Architecture inputs to internal components:

    for DR : DNS_Range generate
        Device_Controls( DNS ) to DNS_Drivers( DR ).Device_Controls;
    for IR  : INS_Range generate
        Device_Controls( INU ) to INU_Drivers( IR ).Device_Controls;
    for GR : GNS_Range generate
        Device_Controls( GPS ) to GPS_Drivers( GR ).Device_Controls;
    for AR : ANS_Range generate
        Device_Controls( ADC ) to ADC_Drivers( AR ).Device_Controls;
    for VR : VNS_Range generate
        Device_Controls( VOR ) to VOR_Drivers( VR ).Device_Controls;

    Device_Controls( DNS ) to DNS_Data_Source.Device_Controls;
    Device_Controls( INU ) to INU_Data_Source.Device_Controls;
    Device_Controls( GPS ) to GPS_Data_Source.Device_Controls;
    Device_Controls( ADC ) to ADC_Data_Source.Device_Controls;
    Device_Controls( VOR ) to VOR_Data_Source.Device_Controls;


    -- Internal components to architecture outputs:

    DNS_Drivers( ?D ).DS to Driver_Sensor_Connection.DNS_DS( ?D );
    INU_Drivers( ?I ).DS to Driver_Sensor_Connection.INU_DS( ?I );
    GPS_Drivers( ?G ).DS to Driver_Sensor_Connection.GPS_DS( ?G );
    ADC_Drivers( ?A ).DS to Driver_Sensor_Connection.ADC_DS( ?A );
    VOR_Drivers( ?V ).DS to Driver_Sensor_Connection.VOR_DS( ?V );

    DNS_Data_Source.Aircraft_State_Data( ?S ) to
  Source_Navigation_Connection.Aircraft_State_Data( DNS, ?S );
    INU_Data_Source.Aircraft_State_Data( ?S ) to
  Source_Navigation_Connection.Aircraft_State_Data( INU, ?S );
    GPS_Data_Source.Aircraft_State_Data( ?S ) to
  Source_Navigation_Connection.Aircraft_State_Data( GPS, ?S );
    ADC_Data_Source.Aircraft_State_Data( ?S ) to
  Source_Navigation_Connection.Aircraft_State_Data( ADC, ?S );
    VOR_Data_Source.Aircraft_State_Data( ?S ) to
  Source_Navigation_Connection.Aircraft_State_Data( VOR, ?S );


    -- IBM 2.9.1.1, 2.7.1.1, 2.8.1.1, 2.10.1.1, 2.11.1.1
    -- Connections among internal components:

    DNS_Drivers( ?D ).Sampled_Data( ?S ) to DNS_Data_Source.Sampled_Data( ?S, ?D );
    INU_Drivers( ?I ).Sampled_Data( ?S ) to INU_Data_Source.Sampled_Data( ?S, ?I );
```

```
    GPS_Drivers( ?G ).Sampled_Data( ?S ) to GPS_Data_Source.Sampled_Data( ?S, ?G );
    ADC_Drivers( ?A ).Sampled_Data( ?S ) to ADC_Data_Source.Sampled_Data( ?S, ?A );
    VOR_Drivers( ?V ).Sampled_Data( ?S ) to VOR_Data_Source.Sampled_Data( ?S, ?V );

    for IR : INS_Range generate
        DNS_Data_Source.Aiding_Data( ?S ) to INU_Drivers( IR ).Aiding_Data;
    for GR : GNS_Range generate
        INU_Data_Source.Aiding_Data( ?S ) to GPS_Drivers( GR ).Aiding_Data;

end;
```

The following declarations from the common data declarations (page 54) describe data items above:

```
type Sensor_Kind is enum DNS, INU, GPS, ADC, VOR end;

type DNS_Range is Range(Integer, 1, 1);  -- IBM 2.4
type INU_Range is Range(Integer, 1, 2);  -- IBM 2.1, 2.2
type GPS_Range is Range(Integer, 1, 1);  -- IBM 2.3
type ADC_Range is Range(Integer, 1, 1);  -- IBM 2.5
type VOR_Range is Range(Integer, 1, 1);  -- IBM 2.6

Msec :  Time is 1;
Sec :   Time is 1000;
Never : Time is (TBD);

DNS_Align_Duration :     Time is Never;  -- IBM 2.4.3.5
DNS_First_Sample :       Time is (TBD);
DNS_Sampling_Interval :  Time is 125 * Msec;  -- IBM 2.4.1.1
DNS_Aiding_Interval :    Time is DNS_Sampling_Interval;  -- IBM 2.4.4

INU_Align_Duration :     Time is 5 * Sec;  -- IBM 2.1.3.5, 2.2.3.5
INU_First_Sample :       Time is (TBD);
INU_Sampling_Interval :  Time is 50 * Msec;  -- IBM 2.1.1.1, 2.2.1.1
INU_Aiding_Interval :    Time is INU_Sampling_Interval;  -- IBM 2.1.4, 2.2.4

GPS_Align_Duration :     Time is Never;  -- IBM 2.3.3.5
GPS_First_Sample :       Time is (TBD);
GPS_Sampling_Interval :  Time is 1000 * Msec;  -- IBM 2.3.1.1
GPS_Aiding_Interval :    Time is Never;  -- IBM 2.5.4

ADC_Align_Duration :     Time is Never;  -- IBM 2.5.3.5
ADC_First_Sample :       Time is (TBD);
ADC_Sampling_Interval :  Time is 200 * Msec;  -- IBM 2.5.1.1
ADC_Aiding_Interval :    Time is Never;  -- IBM 2.5.4

VOR_Tuning_Duration :    Time is 100 * Msec;  -- IBM 2.6.3.1
VOR_First_Sample :       Time is (TBD);
VOR_Sampling_Interval :  Time is 50 * Msec;  -- IBM 2.6.1.1
VOR_Aiding_Interval :    Time is Never;  -- IBM 2.6.4
```

4.2.1 The Device Driver Interface

The Device Driver Interface maintains the status of the sensor it controls. It uses the previously described (Sections 2.1 and 2.2, pages 6 and 9) interfaces Device_Control_Service (Section 8.2, page 51) and Driver_Sensor_Service (Section 8.6, page 52). To control the sensor (to request initialization or alignment, for example), it essentially relays the commands it receives by means of the Device_Controls service. Via service DS, of type Driver_Sensor_Service, it receives and stores data from the sensor; it periodically outputs the data and (using the sensor's state of readiness) the data's usability, in a Sampled_Data event (see Data_Sources_Architecture figure, page 21). If the device driver receives aiding data, these are sent to the sensor by triggering off either Aiding_Data or Sampled_Sensor_Data events, whichever are observed more often.

```
type Device_Driver( First_Sample_Interval, Sampling_Interval, Aiding_Interval, Align_Duration :
                    Time ) is interface

    action Aiding_Data( S : Sensor_Data );
    Device_Controls : service Device_Control_Service;

extern
    DS : service Driver_Sensor_Service;
    action Sampled_Data( S : Sensor_Data );


behavior

    ?R : Raw_Data;
    ?S : Sensor_Data;
    ?T : Time;

    State : var Sensor_State_Type;   -- Initially ( Not_Initted, Not_Aligned);

    Latest_Device_Data : var Sensor_Data;   -- Initially ( Unusable, Default_Aircraft_State );
    Latest_Aiding_Data : var Sensor_Data;

    -- Maintain device state and control initialization/alignment of device:

    begin

    Start =>
        State := Sensor_State_Type'( Not_Initted, Not_Aligned );
        Latest_Device_Data := Sensor_Data'( Unusable, Default_Aircraft_State );

    -- IBM 2.1.2.2, 2.2.2.2, 2.3.2.2, 2.4.2.2, 2.5.2.2, 2.6.2.2
    -- IBM 2.1.2.4, 2.2.2.4, 2.3.2.4, 2.4.2.4, 2.5.2.4, 2.6.2.4
    Device_Controls.Initialize_Sensor =>
        State := Sensor_State_Type'( Initting, Not_Aligned );
        Latest_Device_Data.Quality := Unusable;
        DS.Request_Init_Device;;

    DS.Device_Initialized =>
        State.I := Initted;;

    DS.Device_Initialized where Align_Duration = Never =>
        Latest_Device_Data.Quality := Usable;;
```

```
    -- IBM 2.1.3.2, 2.2.3.2, 2.3.3.2, 2.4.3.2, 2.5.3.2, 2.6.3.2
    -- IBM 2.1.3.3, 2.2.3.3, 2.3.3.3, 2.4.3.3, 2.5.3.3, 2.6.3.3
    -- IBM 2.1.3.4, 2.2.3.4, 2.3.3.4, 2.4.3.4, 2.5.3.4, 2.6.3.4
    Device_Controls.Align_Sensor where ( State.I = Initted and Align_Duration /= Never) =>
        State.A := Aligning;
        Latest_Device_Data.Quality := Unusable;
        DS.Request_Align_Device;;

    DS.Device_Aligned =>
        State.A := Aligned;
        Latest_Device_Data.Quality := Usable;;


    -- Sample and pass on the latest Sensor data; if no latest, reuse old data:
    Start =>
        Next_Data_Time := Clock.Now() + First_Sample_Interval;;

    Clock.Now() = Next_Data_Time =>
        Next_Data_Time := Clock.Now() + Sampling_Interval;
        DS.Request_Sensor_Data;;

    DS.Sampled_Sensor_Data(?R) =>
        Latest_Device_Data.Measured_Aircraft_State := ?R;
        Sampled_Data(Latest_Device_Data);;


    -- Aiding-data support:

    -- Save most recent aiding data:
    Aiding_Data(?S) =>
        Latest_Aiding_Data := ?S;;

    -- IBM 2.1.4.1, 2.2.4.1, 2.3.4.1, 2.4.4.1, 2.5.4.1, 2.6.4.1
    -- Send aiding data to sensor at maximum of Aiding and Sampling rates:
    ( (Aiding_Data where Aiding_Interval < Sampling_Interval) or
      (DS.Sampled_Sensor_Data where Aiding_Interval >= Sampling_Interval) )
    where ( State.I = Initted and (State.A = Aligned or Align_Duration = Never)
              and Aiding_Interval /= Never) =>
        DS.Aiding_Data_To_Sensor(Latest_Aiding_Data);;
    -- IBM 2.1.2.3, 2.2.2.3, 2.3.2.3, 2.4.2.3, 2.5.2.3, 2.6.2.3

constraint

    -- IBM 2.1.2.1, 2.2.2.1, 2.3.2.1, 2.4.2.1, 2.5.2.1, 2.6.2.1
    -- IBM 2.1.3.1, 2.2.3.1, 2.3.3.1, 2.4.3.1, 2.5.3.1, 2.6.3.1
    -- DS.Sampled_Sensor_Data arrives every Sampling_Interval.
    match Periodic( DS.Sampled_Sensor_Data, ?T, Sampling_Interval );

    -- Aiding_Data arrives every Aiding_Interval.
    match Periodic( Aiding_Data, ?T, Aiding_Interval );

end;
```

This interface uses the Sensor_State_Type, Sensor_Data, and Raw_Data types we saw in the top-level interface:

```
type Sensor_State_Type is record
    I : var Init_State_Type; -- Initially Not_Initted;
    A : var Align_State_Type; -- Initially Not_Aligned;
end record;

type Init_State_Type is enum Not_Initted, Initting, Initted end;
type Align_State_Type is enum Not_Aligned, Aligning, Aligned end;


type Sensor_Data is record
    Quality : var Sensor_Data_Quality;  -- Initially Unusable;
    Measured_Aircraft_State : var Raw_Data; -- Initially Default_Measured_Aircraft_State;
end record;

type Sensor_Data_Quality is enum Usable, Degraded, Unusable end;

type Raw_Data is record
    Position: var Position_Type;
    Velocity: var Velocity_Type;
    Attitude: var Attitude_Type;
end record;

type Position_Type is (TBD);
type Velocity_Type is (TBD);
type Attitude_Type is (TBD);

Default_Measured_Aircraft_State : Raw_Data;  -- Initially Raw_Data'(TBD);
```

and the Clock object, described in Section 3.2.1. To require the sensor and aiding data to arrive at specific rates, the constraints use the pattern macro Periodic, another item in the Common_Definitions interface (Section 9, page 54):

```
pattern Periodic( pattern P; Occurs, Period : Time ) is
    At( P, Occurs, Clock ) ->
      Periodic( P, Occurs+Period, Period );
```

This pattern macro takes as parameters a pattern expression and two expressions of type Time. The body of the macro is a pattern expression consisting of two sub-patterns: At(P, Occurs, Clock), matched by a computation all of whose elements match P at time Occurs with respect to clock Clock, followed in the partially ordered computation by Periodic(P, Occurs+Period, Period), a recursive call which depends on matching the At.

Thus, an instance of Periodic will match an ordered sequence of matches of P, beginning at time Occurs and occurring at intervals of width Period. In the instances shown above, the unbound placeholder ?T is passed as the value of Occurs, meaning the start time is irrelevant.

Finally, this and the next interface employ two common data elements:

```
Next_Data_Time :          var Time;
Next_Aiding_Data_Time :   var Time;
```

4.2.2 The Data Source Interface

The state-transition rules of the Data Source Interface specify the frequency with which source data are generated and how those data are related to input from the device drivers associated with that source, from external-override data, and from functions which determine the best available quality of data.

Some data sources generate *aiding data* for another device driver. The Data Source Interface implements both aiding and non-aiding data sources; no aiding data are produced if the interface is instantiated with Never as the value of the Aiding_Interval parameter (e.g., for GPS, ADC, and VOR) in Common_Definitions (Section 9, page 54).

```
type Data_Source_Interface( type Sensor_Range;
                            Kind : Sensor_Kind;
                            First_Sample_Interval, Sampling_Interval, Aiding_Interval : Time )
                            is interface

    Device_Controls : service Device_Control_Service;
    action Sampled_Data( S : Sensor_Data;   Sensor : Sensor_Range );

extern
    action Aircraft_State_Data( S : Sensor_Data);
    action Aiding_Data( S : Sensor_Data );


behavior

    ?S : Sensor_Data;
    ?R : Sensor_Range;
    ?C : Selection_Criteria;
    ?B : Boolean;

    Generates_Aiding_Data : var Boolean := (Aiding_Interval /= Never);
    Produce_Aiding_Data : var Boolean := False;

    Latest_Selection_Criteria : var Selection_Criteria;
    Latest_Data_Source_Data : var Sensor_Data; -- Initially (Unusable, Default_Aircraft_State)

    begin

    Start =>
        Latest_Data_Source_Data := Sensor_Data'( Unusable, Default_Aircraft_State );;

    -- IBM 2.7.1.3, 2.8.1.3, 2.9.1.3, 2.10.1.3, 2.11.1.3
    Device_Controls.Select_Criteria( ?C ) =>
        Latest_Selection_Criteria := ?C;;

    -- IBM 2.7.1.7, 2.8.1.7, 2.9.1.7, 2.10.1.7, 2.11.1.7
    -- Output data-source data every interval:

    Start =>
        Next_Data_Time := Clock.Now() + First_Sample_Interval;;

    Aircraft_State_Data =>
        Next_Data_Time := Clock.Now() + Sampling_Interval;;
```

```
Clock.Now() = Next_Data_Time =>
    Aircraft_State_Data( Latest_Data_Source_Data );;

-- Once the data are output, if selection criterion is set to Best_Available,
-- re-initialize the latest data to Unusable:
Aircraft_State_Data where Clock.Now() > Next_Data_Time
      and Latest_Selection_Criteria.Option = Best_Available =>
    Latest_Data_Source_Data.Quality := Unusable;;

-- IBM 2.7.1.6, 2.8.1.6, 2.9.1.6, 2.10.1.6, 2.11.1.6
-- If selection criterion is Best_Available, maintain and use best
-- available data:
( Sampled_Data( ?S, ?R ) or Device_Controls.Override_Data( ?S ) )
    where Latest_Selection_Criteria.Option = Best_Available
    and ?S.Quality > Latest_Data_Source_Data.Quality =>
    Latest_Data_Source_Data := ?S;;

-- IBM 2.7.1.2, 2.8.1.2, 2.9.1.2, 2.10.1.2, 2.11.1.2
-- IBM 2.7.1.5, 2.8.1.5, 2.9.1.5, 2.10.1.5, 2.11.1.5
-- If selection criterion is External, maintain and use latest override data:
Device_Controls.Override_Data( ?S )
    where Latest_Selection_Criteria.Option = External =>
    Latest_Data_Source_Data := ?S;;

-- IBM 2.7.1.4, 2.8.1.4, 2.9.1.4, 2.10.1.4, 2.11.1.4
-- If selection criterion is one of the sensors, output whenever data
-- come from it:
Sampled_Data( ?S, ?R )
    where Latest_Selection_Criteria = Selection_Criteria'( Sensor, Kind, ?R ) =>
    Latest_Data_Source_Data := ?S;;


-- Aiding data behavior:

-- IBM 2.7.2.1, 2.8.2.1, 2.9.2.1, 2.10.2.1, 2.11.2.1
Device_Controls.Aiding_Command( ?B ) where Generates_Aiding_Data =>
    Produce_Aiding_Data := ?B;
    Aiding_Data(Latest_Data_Source_Data) where ?B or empty where not ?B;;

-- IBM 2.7.2.2, 2.8.2.2, 2.9.2.2, 2.10.2.2, 2.11.2.2
Aiding_Data where (Generates_Aiding_Data and Produce_Aiding_Data) =>
    Next_Aiding_Data_Time := Clock.Now() + Aiding_Interval;;

Clock.Now() = Next_Aiding_Data_Time =>
    Aiding_Data(Latest_Data_Source_Data);;

end;
```

The selection criteria used to initialize Selection_Criteria also come from Common_Definitions, referenced earlier in this chapter:

```
Default_Selection_Criteria : Selection_Criteria is Selection_Criteria'( Best_Available, INU, 1 );
```

This interface uses Device_Control_Service, Sensor_Data, Selection_Criteria, and Sensor_Kind types, all seen before (Section 2.1, page 6). The service is repeated here:

```
type Device_Control_Service is interface

    action Initialize_Sensor();
    action Align_Sensor();

    action Override_Data( S : Sensor_Data );
    action Select_Criteria( C : Selection_Criteria );
    action Aiding_Command( B : Boolean );

end;
```

The Clock object, used for keeping time, was described in Section 3.2.1.

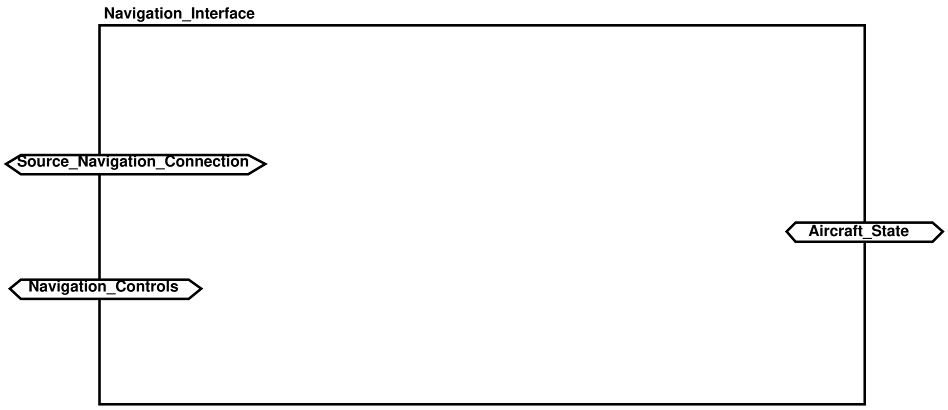
Chapter 5

Navigation

Three internal models of the aircraft and its environment — the Earth Model, the Atmosphere Model, and the Aircraft State Vector Model — and the VOR Radio Navigation component, which filters raw VOR sensor data, comprise Navigation.

5.1 The Navigation Interface

The interface is defined by its two communication channels, described before (Section 2.2, page 9). Input arrives from the data sources and is communicated to Guidance via the Aircraft_State service.



type Navigation_Interface **is** **interface**

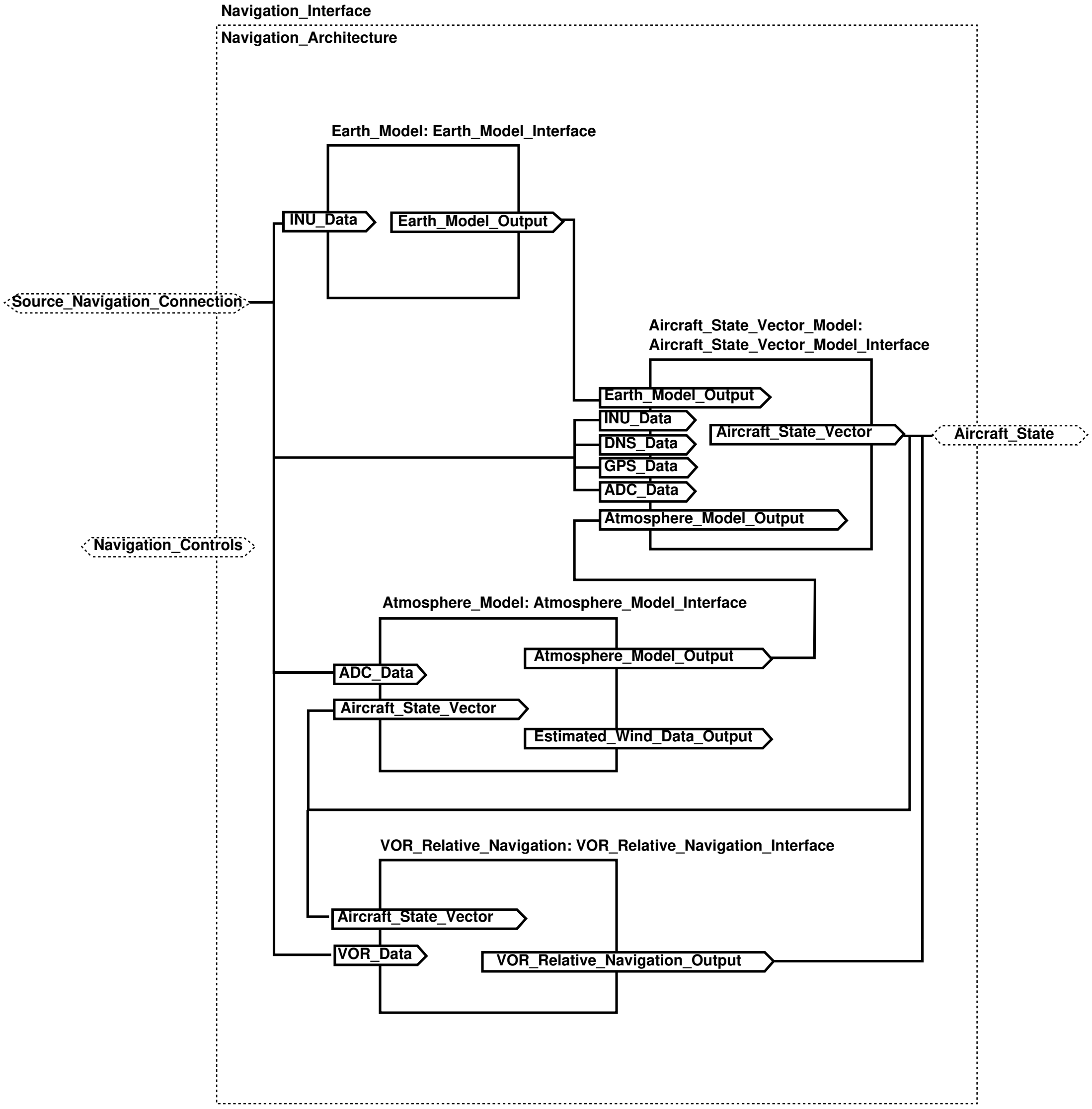
    Source_Navigation_Connection : **service** Source_Navigation_Service;
    Navigation_Controls : **service** Navigation_Control_Service;

extern
    Aircraft_State : **service** Aircraft_State_Service;

end;

5.2 The Navigation Architecture

The relationship of the relatively simple interface and rich internal architecture is shown in the figure.



Internal elements of the architecture are the three internal models of the aircraft and its environment:

- The Earth Model (Section 5.2.1, page 34)
- The Atmosphere Model (Section 5.2.2, page 34)
- The Aircraft State Vector Model (Section 5.2.3, page 35)

and the VOR Radio Navigation component (Section 5.2.4, page 36). All of these components use input from the data sources and together produce a single Aircraft_State service (of type

Aircraft_State_Service [Section 8.8, page 53]), which we saw in the Partial Top Level Architecture (Section 2.2, page 9).

Additionally, internal connections pass data from the Earth Model and Atmosphere Model to the Aircraft State Vector Model and from the Aircraft State Vector Model to the Atmosphere Model and the VOR Radio Navigation component.

```
architecture Navigation_Architecture() for Navigation_Interface is

   Earth_Model : Earth_Model_Interface;   -- IBM 2.12.0
   Atmosphere_Model : Atmosphere_Model_Interface;   -- IBM 2.12.1
   Aircraft_State_Vector_Model : Aircraft_State_Vector_Model_Interface;   -- IBM 2.12.2
   VOR_Relative_Navigation : VOR_Relative_Navigation_Interface;   -- IBM 2.13

   ?S : Sensor_Data;

connect

   -- Architecture inputs to internal components:

   Source_Navigation_Connection.Aircraft_State_Data( INU, ?S ) to
  Earth_Model.INU_Data( ?S ), Aircraft_State_Vector_Model.INU_Data( ?S );

   Source_Navigation_Connection.Aircraft_State_Data( DNS, ?S ) to
  Aircraft_State_Vector_Model.DNS_Data( ?S );

   Source_Navigation_Connection.Aircraft_State_Data( GPS, ?S ) to
  Aircraft_State_Vector_Model.GPS_Data( ?S );

   Source_Navigation_Connection.Aircraft_State_Data( ADC, ?S ) to
  Aircraft_State_Vector_Model.ADC_Data( ?S ), Atmosphere_Model.ADC_Data( ?S );

   Source_Navigation_Connection.Aircraft_State_Data( VOR, ?S ) to
  VOR_Relative_Navigation.VOR_Data( ?S );


   -- Internal components to architecture outputs:

   Aircraft_State_Vector_Model.Aircraft_State_Vector to
       Aircraft_State.Aircraft_State_Vector;

   VOR_Relative_Navigation.VOR_Relative_Navigation_Output to
       Aircraft_State.VOR_Relative_Navigation_Output;


   -- Connections among internal components:

   Earth_Model.Earth_Model_Output to
       Aircraft_State_Vector_Model.Earth_Model_Output;

   Atmosphere_Model.Atmosphere_Model_Output to
       Aircraft_State_Vector_Model.Atmosphere_Model_Output;

   Aircraft_State_Vector_Model.Aircraft_State_Vector to
       Atmosphere_Model.Aircraft_State_Vector,
```

```
    VOR_Relative_Navigation.Aircraft_State_Vector;
end;
```

The interfaces which represent the four internal components of the Navigation Architecture are described in the next four subsections.

5.2.1 The Earth Model Interface

The Earth Model uses information from the INU data source to produce an event which represents the computation of the model. In later versions, this event would be expanded and elaborated, as more detail of the model emerges.

```
type Earth_Model_Interface is interface

    action INU_Data( D : Sensor_Data );

extern
    action Earth_Model_Output();

behavior
    begin

    INU_Data => Earth_Model_Output;;   -- IBM 2.12.0.1

end;
```

5.2.2 The Atmosphere Model Interface

The Atmosphere Model uses input from the ADC data source as well as the Aircraft State Vector Model and produces events representing the Atmosphere Model computation and estimated wind data. Estimated wind data are currently unused.

```
type Atmosphere_Model_Interface is interface


    action ADC_Data( S : Sensor_Data );
    action Aircraft_State_Vector( V : State_Vector );

extern
    action Atmosphere_Model_Output();
    action Estimated_Wind_Data_Output();   -- Currently unused

behavior

    begin

    ADC_Data => Atmosphere_Model_Output;;   -- IBM 2.12.1.1
    ADC_Data and Aircraft_State_Vector => Estimated_Wind_Data_Output;;   -- IBM 2.12.1.2

end;
```

This and the next model use the State_Vector record, whose Sensor_Data_Quality type was given in the Partial Top Level Interface (Section 2.1, page 6):

```
type var State_Vector is record
    Quality : Sensor_Data_Quality;
    Vector : State_Vector_Info;
end record;
```

5.2.3 The Aircraft State Vector Model Interface

The Aircraft State Vector Model uses data from four data sources — the DNS, INU, GPS, and ADC (but not the VOR directly) — and from the Earth and Atmosphere models. It produces a single data representation of the state of the aircraft.

For the state vector to be usable, updated data must have been received from all input sources. This is modeled using the Data_Updated array; only if every element of this array is True will the Aircraft_State_Vector event include usable data.

```
type Aircraft_State_Vector_Model_Interface is interface

    action DNS_Data( S : Sensor_Data );
    action INU_Data( S : Sensor_Data );
    action GPS_Data( S : Sensor_Data );
    action ADC_Data( S : Sensor_Data );
    action Earth_Model_Output();
    action Atmosphere_Model_Output();

extern
    action Aircraft_State_Vector( V : State_Vector );

behavior

    -- Expandable to later implementations of data sources:
    type Input_Kind is enum DNS_In, INU_In, GPS_In, ADC_In,
                            Earth_Model_In, Atmosphere_Model_In end;

    type Updated_Vector is array(Input_Kind) of Boolean;
    Data_Updated : var Updated_Vector;

    ?K : Input_Kind;

    DNS_Data => Data_Updated(DNS_In) := True;;  -- IBM 2.12.2.3
    INU_Data => Data_Updated(INU_In) := True;;  -- IBM 2.12.2.1
    GPS_Data => Data_Updated(GPS_In) := True;;  -- IBM 2.12.2.2
    ADC_Data => Data_Updated(ADC_In) := True;;  -- IBM 2.12.2.4
    Earth_Model_Output => Data_Updated(Earth_Model_In) := True;;  -- IBM 2.12.2.5
    Atmosphere_Model_Output => Data_Updated(Atmosphere_Model_In) := True;;
        -- IBM 2.12.2.6

    function All_Elements_Updated() return Boolean is
    begin
        for I : Input_Kind do
            if not Data_Updated(I) then
                return False;  -- IBM 2.12.2.8
```

```
        end if;
    end loop;
    return True;
end function;

-- Shorthand for the computation that turns sensor inputs into real ASV output:
function Calculate_Vector_Value() return State_Vector is
    Some_Value : var State_Vector;  -- Represents some function on vector inputs
begin
    if not All_Elements_Updated() then
        Some_Value.Quality := Unusable;   -- IBM 2.12.2.9
    end if;
    return Some_Value;
end function;

begin

Start =>
    Data_Updated := Updated_Vector'(others => False);
    Next_Data_Time := Clock.Now() + Initial_ASV_Time;;

Clock.Now() = Next_Data_Time =>
    Aircraft_State_Vector(Calculate_Vector_Value());;

Aircraft_State_Vector =>
    Data_Updated(?K) := False;;
    Next_Data_Time := Clock.Now() + ASV_Interval;;

end;
```

This model uses three times from Common_Definitions (Section 9, page 54):

```
ASV_Interval :          Time is 50 * Msec;
Initial_ASV_Time :      Time is (TBD);

Next_Data_Time :        var Time;
```

5.2.4 The VOR Radio Navigation Interface

The VOR Radio Navigation component uses input from both the VOR data source and the Aircraft State Vector Model and produces filtered VOR data. As with other components of the navigational architecture, details of this output are omitted; the output is represented by the event VOR_Relative_Navigation_Output.

```
type VOR_Relative_Navigation_Interface is interface

    action Aircraft_State_Vector();
    action VOR_Data( S : Sensor_Data );

extern
    action VOR_Relative_Navigation_Output();


behavior

    begin

    Aircraft_State_Vector and VOR_Data => VOR_Relative_Navigation_Output;; -- IBM 2.13

end;
```

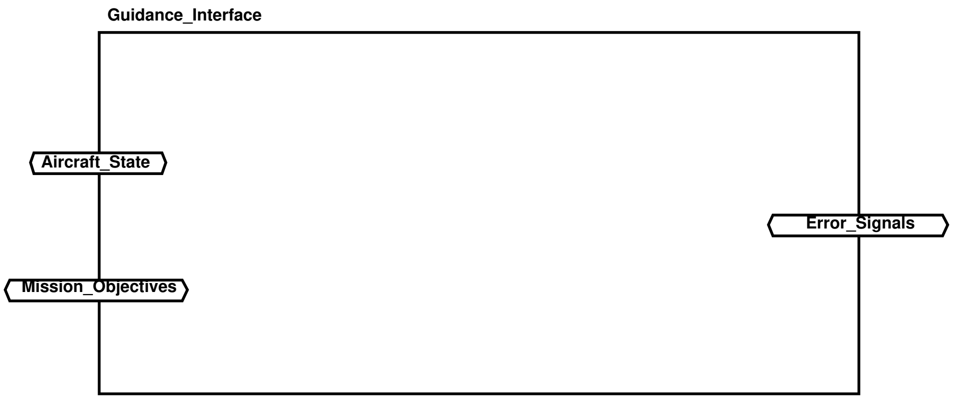
Chapter 6

Guidance

This component of the Partial Top Level Architecture is relatively simple itself but contains an architectural component of some complexity (Section 7).

6.1 The Guidance Interface

Inputs to the Guidance Interface are two services: Aircraft_State, which contains the current location and status of the aircraft, and Mission_Objectives, which manages enabling and disabling of all kinds of guidance. Output is the Error_Signals service, which gives the heading errors between the aircraft's current status and its desired destination.

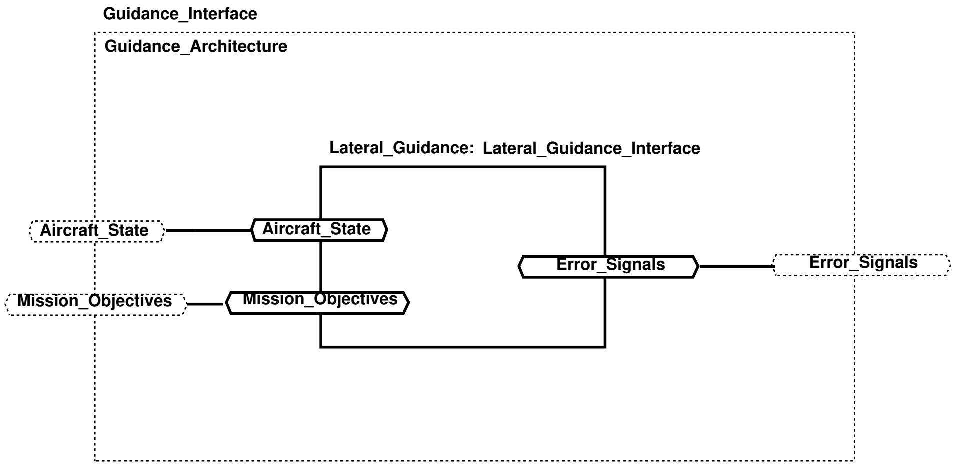


All three communication services have appeared in previous sections, the first being Partial Top Level Interface and Architecture (pages 6-9).

```
type Guidance_Interface is interface

    Aircraft_State : service Aircraft_State_Service;
    Mission_Objectives : service Mission_Objectives_Service;

extern
    Error_Signals : service Error_Signals_Service;

end;
```

6.2 The Guidance Architecture

Although there are various kinds of guidance, IBM provided us with definitions for Lateral Guidance only. Thus the only component of this architecture currently is Lateral_Guidance (Section 7), and the services are connected directly between the interface and this component.



architecture Guidance_Architecture() **for** Guidance_Interface **is**

    Lateral_Guidance : Lateral_Guidance_Interface
        is Lateral_Guidance_Architecture;

connect

    -- Service connections:

    Aircraft_State **to** Lateral_Guidance.Aircraft_State;
    Mission_Objectives **to** Lateral_Guidance.Mission_Objectives;
    Lateral_Guidance.Error_Signals **to** Error_Signals;

end;

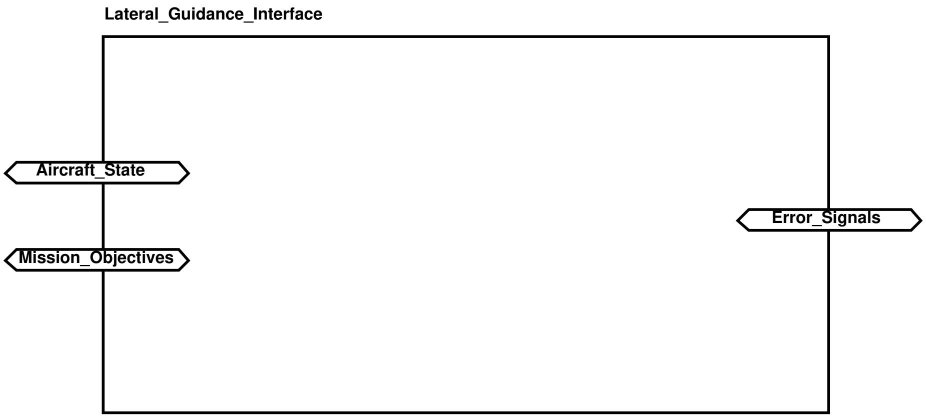
Chapter 7

Lateral Guidance

Lateral Guidance has the same interface services as the Guidance architecture itself.

7.1 The Lateral Guidance Interface

Therefore, we have seen all the services of this interface before.



type Lateral_Guidance_Interface **is** **interface**

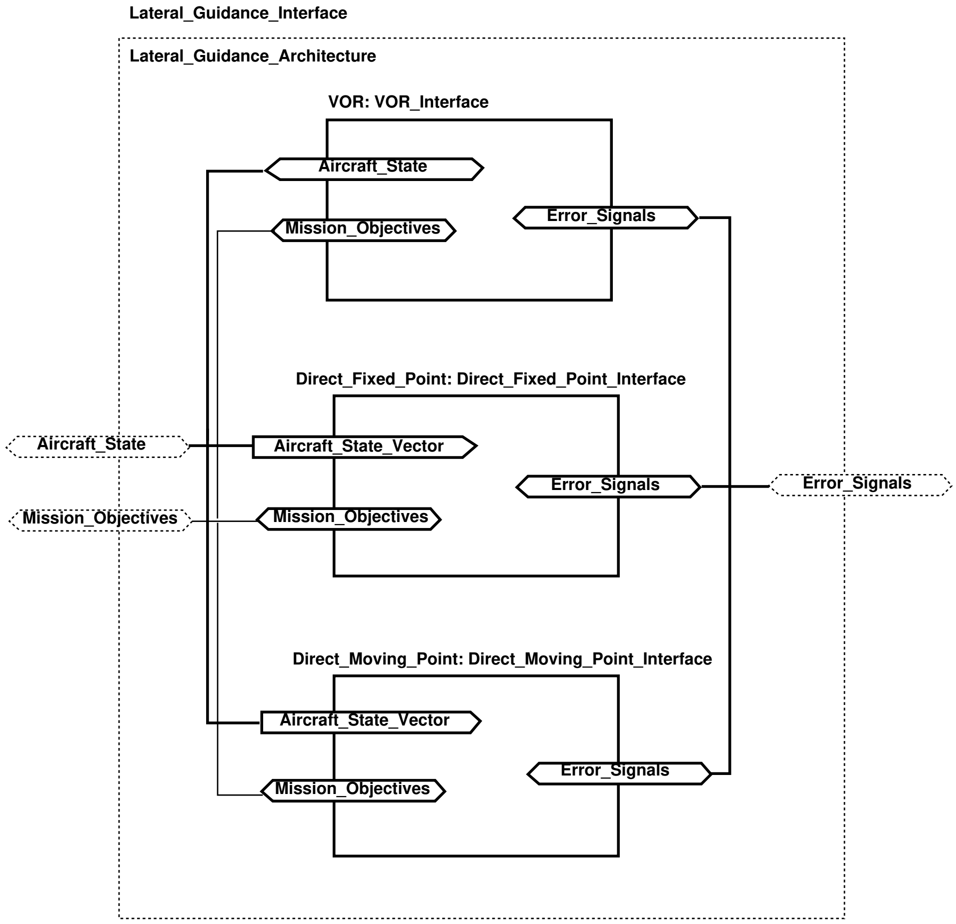
  Aircraft_State : **service** Aircraft_State_Service;
  Mission_Objectives : **service** Mission_Objectives_Service;

extern
  Error_Signals : **service** Error_Signals_Service;

end;

7.2 The Lateral Guidance Architecture

Lateral Guidance has three modes, described in successive subsections below: VOR (Section 7.2.1, page 42), Direct Fixed-Point (Section 7.2.2, page 45), and Direct Moving-Point (Section 7.2.3, page 48).



Each mode exploits Mission_Objectives to determine when its state is to change and uses the Aircraft_State in generating heading errors of the Error_Signals service, although Direct Fixed-Point and Moving-Point modes utilize only the Aircraft_State_Vector action of Aircraft_State.

```
architecture Lateral_Guidance_Architecture() for Lateral_Guidance_Interface is

   VOR : VOR_Interface;
   Direct_Fixed_Point : Direct_Fixed_Point_Interface;
   Direct_Moving_Point : Direct_Moving_Point_Interface;

connect

   -- Service connections:

   Mission_Objectives to
      VOR.Mission_Objectives,
      Direct_Fixed_Point.Mission_Objectives,
      Direct_Moving_Point.Mission_Objectives;

   Aircraft_State to VOR.Aircraft_State;
   Aircraft_State.Aircraft_State_Vector to
      Direct_Fixed_Point.Aircraft_State_Vector,
```

```
        Direct_Moving_Point.Aircraft_State_Vector;

    VOR.Error_Signals,  Direct_Fixed_Point.Error_Signals,
        Direct_Moving_Point.Error_Signals   to  Error_Signals;

end;
```

7.2.1 The VOR Interface

All three guidance behaviors have several state-transition rules which define conditions under which that guidance is started and stopped or is armed and disarmed.

A mode must stop itself when another mode is started. This is implemented by allowing all modes to observe the controlling signals of the other modes. Thus a mode which has been started will stop itself when it observes a Guidance_Started event not intended for that mode.

When VOR is armed and the aircraft reaches the current radial passed in by Mission Objectives, the mode starts itself by performing its own in-action. Connection rules of the parent cause this event to be visible to other guidance modes, and they will disable themselves if necessary, as just described. The pattern constraints below express the "liveness" properties of the original specification.

```
type VOR_Interface is interface

    Aircraft_State : service Aircraft_State_Service;
    Mission_Objectives : service Mission_Objectives_Service;

extern
    Error_Signals : service Error_Signals_Service;


behavior

    ?V : State_Vector;
    ?D : Lateral_Guidance_Data;
    ?M : Lateral_Guidance_Kind;
    ?A : Lateral_Guidance_Arming_Data;

    State : var Guidance_State_Type;   -- Initially ( False, False );

    Radial, Arm_Radial : var Radial_Type;

    Actual_Start_Time, Actual_Stop_Time : var Time := Never;
    Actual_Arm_Time, Actual_Disarm_Time : var Time := Never;

    -- Future duration functions to return the times needed to perform respective tasks:

    function Start_Duration() return Time is
    begin return TBD;
    end function;

    function Stop_Duration() return Time is
    begin return TBD;
    end function;
```

```
function Arm_Duration() return Time is
begin return TBD;
end function;

function Disarm_Duration() return Time is
begin return TBD;
end function;

-- Represents computation of current heading error:
function Current_Heading_Error() return Heading_Error_Type is
begin return TBD;
end function;

-- Represents computation of current radial:
function Current_Radial() return Radial_Type is
begin return TBD;
end function;


begin

Start =>
    State := Guidance_State_Type'( False, False );

-- IBM 3.3.1. Output heading error if Aircraft State Vector is
-- usable and state is started:
Aircraft_State.Aircraft_State_Vector( ?V ) where
    ?V.Quality = Usable and State.Started =>
    Actual_Start_Time := Never;
    Error_Signals.Heading_Error( Current_Heading_Error() );;

-- Placeholder for saving VOR data and using in output of the heading errors above:
Aircraft_State.VOR_Relative_Navigation_Output => empty;;

-- Behaviors to start and stop guidance:

-- IBM 3.3.2. Start if explicitly started, unless interrupted by another start event:
Mission_Objectives.Start_Guidance( VOR, ?D ) =>
    Radial := ?D.Radial;
    Actual_Start_Time := Clock.Now() + Start_Duration();;

Clock.Now() = Actual_Start_Time =>
    Mission_Objectives.Guidance_Started( VOR );
    State.Started := True;;

-- IBM 3.3.9. Start when arrive at radial:
Aircraft_State.Aircraft_State_Vector( ?V ) where
    ?V.Quality = Usable and State.Armed and Current_Radial() = Arm_Radial =>
    State.Armed := False;
    Actual_Arm_Time := Never;
    Mission_Objectives.Start_Guidance( VOR, Arm_Radial );;

-- IBM 3.3.4, 3.3.5. Stop if explicitly stopped:
Mission_Objectives.Stop_Guidance( VOR ) =>
```

```
        Actual_Stop_Time := Clock.Now() + Stop_Duration();;

    Clock.Now() = Actual_Stop_Time =>
        Actual_Stop_Time := Never;
        State.Started := False;
        Mission_Objectives.Guidance_Stopped( VOR );;

    -- IBM 3.3.6, 3.3.7. Stop immediately if Aircraft State Vector is unusable or another
    -- guidance mode is started:
    ( Aircraft_State.Aircraft_State_Vector( ?V ) where ?V.Quality = Unusable ) or
    ( Mission_Objectives.Guidance_Started( ?M ) where ?M /= VOR and State.Started ) =>
        State.Started := False;;

    -- IBM 3.3.8, 3.3.10. Behaviors to arm and disarm guidance:

    -- Arm within Max_Arm_Time, unless interrupted by another arm event:
    Mission_Objectives.Arm_Guidance( VOR, ?A ) =>
        Arm_Radial := ?A.Radial;
        Actual_Arm_Time := Clock.Now() + Arm_Duration();;

    Clock.Now() = Actual_Arm_Time =>
        State.Armed := True;
        Mission_Objectives.Guidance_Armed( VOR );;

    Mission_Objectives.Disarm_Guidance( VOR ) =>
        Actual_Disarm_Time := Clock.Now() + Disarm_Duration();;

    Clock.Now() = Actual_Disarm_Time =>
        State.Armed := False;
        Mission_Objectives.Guidance_Disarmed( VOR );;


constraint

    not match (Tick where
        Clock.Now() > Actual_Start_Time and Actual_Start_Time /= Never or
        Clock.Now() > Actual_Stop_Time and Actual_Stop_Time /= Never or
        Clock.Now() > Actual_Arm_Time and Actual_Arm_Time /= Never);

end;
```

This makes use of several types found in Common_Definitions (Section 9) — (1) various lateral-guidance types introduced in Partial Top Level Interface (Section 2.1), (2) a Radial_Type and a Heading_Error_Type (each yet to be defined), and (3) Guidance_State_Type:

```
    type Guidance_State_Type is record
        Started, Armed : Boolean is false;
    end record;
```

and three times defined in terms of Msec:

```
Max_Start_Time :   Time is 100 * Msec;
Max_Stop_Time  :   Time is 50 * Msec;
Max_Arm_Time   :   Time is 100 * Msec;

Msec :   Time is   1;
```

The Clock object, used for timing, was described in Section 3.2.1. Clock's function Now() is called explicitly in behavior patterns to fetch the current time with respect to Clock.

The *pattern constraint* at the end of VOR_Interface succinctly specifies the protocol which the interface must follow. It constrains only events of the actions mentioned. In this constraint, only Tick events are constrained; Disarm_Guidance events, *inter alia*, are not mentioned and thus not constrained.

The constraint checks that the guidance starting, stopping, and arming events occur within the time allotted for them. It is read as follows: allow no ticks of the timing clock being used (values of Clock.Now()) beyond the time set for starting, stopping, or arming this guidance mode unless that time is set to Never. In other words, signal a constraint error if the start, stop, or arm time has passed without the occurrence of the associated guidance event, which would have set the time indicator for that event back to its initial value of Never.

7.2.2 The Direct Fixed-Point Interface

Direct Fixed-Point and Moving-Point guidance modes are like VOR guidance, except that they calculate error headings using capture radius: when the mode is armed and the aircraft enters the capture radius of the current destination, the mode starts itself.

```
type Direct_Fixed_Point_Interface is interface

    action Aircraft_State_Vector( V : State_Vector );
    Mission_Objectives : service Mission_Objectives_Service;

extern
    Error_Signals : service Error_Signals_Service;


behavior

    ?V : State_Vector;
    ?D : Lateral_Guidance_Data;
    ?M : Lateral_Guidance_Kind;
    ?A : Lateral_Guidance_Arming_Data;

    State : var Guidance_State_Type;   -- Initially ( False, False );

    Destination, Arm_Destination, Arm_Capture_Point : var Destination_Point;

    Actual_Start_Time, Actual_Stop_Time : var Time := Never;
    Actual_Arm_Time, Actual_Disarm_Time : var Time := Never;

    -- Future duration functions to return the times needed to perform respective tasks:

    function Start_Duration() return Time is
    begin return TBD;
```

```
end function;

function Stop_Duration() return Time is
begin return TBD;
end function;

function Arm_Duration() return Time is
begin return TBD;
end function;

function Disarm_Duration() return Time is
begin return TBD;
end function;

-- Represents computation of current heading error:
function Current_Heading_Error() return Heading_Error_Type is
begin return TBD;
end function;

-- Represents computation of current location:
function Current_Location() return Destination_Point is
begin return TBD;
end function;


begin

Start =>
    State := Guidance_State_Type'( False, False );

-- Output heading error if Aircraft State Vector is usable,
-- state is started, and aircraft is outside capture radius:
Aircraft_State_Vector( ?V ) where
    (?V.Quality = Usable and State.Started
    and Current_Location() - Destination >= Capture_Radius ) =>
    Actual_Start_Time := Never;
    Error_Signals.Heading_Error( Current_Heading_Error() );;


-- Behaviors to start and stop guidance:

-- IBM 3.1.2, IBM 3.1.3. Start if explicitly started, unless
-- interrupted by another start event:
Mission_Objectives.Start_Guidance( Direct_Fixed, ?D ) =>
    Destination := ?D.Destination;
    Actual_Start_Time := Clock.Now() + Start_Duration();;

Clock.Now() = Actual_Start_Time =>
    Mission_Objectives.Guidance_Started( Direct_Fixed );
    State.Started := True;;

-- IBM 3.1.10. Start when enter capture radius of arming capture point:
Aircraft_State_Vector( ?V ) where
    (?V.Quality = Usable and State.Armed
    and Current_Location() - Arm_Capture_Point < Capture_Radius ) =>
```

```
    State.Armed := False;
    Mission_Objectives.Start_Guidance( Direct_Fixed, Arm_Destination );;

-- IBM 3.1.4, IBM 3.1.5. Stop if explicitly stopped, regardless of
-- additional Stop_Guidance events:
Mission_Objectives.Stop_Guidance( Direct_Fixed ) =>
    Actual_Stop_Time := Clock.Now() + Stop_Duration();;

Clock.Now() = Actual_Stop_Time =>
    Actual_Stop_Time := Never;
    Mission_Objectives.Guidance_Stopped( Direct_Fixed );
    State.Started := False;;

-- IBM 3.1.6, 3.1.7, 3.1.8. Stop immediately if Aircraft State Vector is unusable,
-- another guidance mode is started, or aircraft has entered the capture radius:
( Aircraft_State_Vector( ?V ) where ?V.Quality = Unusable ) or
( Aircraft_State_Vector( ?V ) where (?V.Quality = Usable and State.Started
    and Current_Location() - Destination < Capture_Radius ) ) or
Mission_Objectives.Guidance_Started( ?M ) where
    ?M /= Direct_Fixed and State.Started =>
    State.Started := False;;

-- IBM 3.1.9. Behaviors to arm and disarm guidance:

-- IBM 3.1.11. Arm within Max_Arm_Time, unless interrupted by another arm event:
Mission_Objectives.Arm_Guidance( Direct_Fixed, ?A ) =>
    Arm_Destination := ?A.Destination;
    Arm_Capture_Point := ?A.Capture;
    Actual_Arm_Time := Clock.Now() + Arm_Duration();;

Clock.Now() = Actual_Arm_Time =>
    Actual_Arm_Time := Never;
    Mission_Objectives.Guidance_Armed( Direct_Fixed );
    State.Armed := True;;

Mission_Objectives.Disarm_Guidance( Direct_Fixed ) =>
    Actual_Disarm_Time := Clock.Now() + Disarm_Duration();;

Clock.Now() = Actual_Disarm_Time =>
    Mission_Objectives.Guidance_Disarmed( Direct_Fixed );
    State.Armed := False;;


constraint

    not match (Tick where
        Clock.Now() > Actual_Start_Time and Actual_Start_Time /= Never or
        Clock.Now() > Actual_Stop_Time and Actual_Stop_Time /= Never or
        Clock.Now() > Actual_Arm_Time and Actual_Arm_Time /= Never);

end;
```

Except for a to-be-defined Capture_Radius type, the types and constraints are the same as for the VOR interface, Section 7.2.1 above.

7.2.3 The Direct Moving-Point Interface

Direct Moving-Point guidance is similar to Direct Fixed-Point guidance, but the Start_Guidance event now also includes the destination speed and heading, and a change of state may result from observing the command Change_Guidance (also with destination, speed, and heading).

```
type Direct_Moving_Point_Interface is interface

    action Aircraft_State_Vector( V : State_Vector );
    Mission_Objectives : service Mission_Objectives_Service;

extern
    Error_Signals : service Error_Signals_Service;


behavior

    ?R : Lateral_Guidance_Revision_Data;
    ?V : State_Vector;
    ?D : Lateral_Guidance_Data;
    ?M : Lateral_Guidance_Kind;
    ?A : Lateral_Guidance_Arming_Data;

    State : var Guidance_State_Type;   -- Initially ( False, False );

    Destination, Arm_Destination, Arm_Capture_Point : var Destination_Point;
    Destination_Speed : var Speed_Type;
    Destination_Heading : var Heading_Type;

    Actual_Start_Time, Actual_Stop_Time : var Time := Never;
    Actual_Arm_Time, Actual_Disarm_Time : var Time := Never;

    -- Future duration functions to return the times needed to perform respective tasks:

    function Start_Duration() return Time is
    begin return TBD;
    end function;

    function Stop_Duration() return Time is
    begin return TBD;
    end function;

    function Arm_Duration() return Time is
    begin return TBD;
    end function;

    function Disarm_Duration() return Time is
    begin return TBD;
    end function;

    -- Represents computation of current heading error:
    function Current_Heading_Error() return Heading_Error_Type is
    begin return TBD;
    end function;
```

```
-- Represents computation of current location:
function Current_Location() return Destination_Point is
begin return TBD;
end function;


begin

Start =>
    State := Guidance_State_Type'( False, False );

-- IBM 3.2.2. Change state of destination:
Mission_Objectives.Change_Guidance( Direct_Moving, ?R ) =>
    Destination := ?R.Destination;
    Destination_Speed := ?R.Speed;
    Destination_Heading := ?R.Heading;;

-- Output heading error if Aircraft State Vector is usable,
-- state is started, and aircraft is outside capture radius:
Aircraft_State_Vector( ?V ) where
    (?V.Quality = Usable and State.Started
    and Current_Location() - Destination >= Capture_Radius ) =>
    Actual_Start_Time := Never;
    Error_Signals.Heading_Error( Current_Heading_Error() );;


-- Behaviors to start and stop guidance:

-- IBM 3.2.1. Start within Max_Start_Time if explicitly started
-- unless interrupted by another start event:
Mission_Objectives.Start_Guidance( Direct_Moving, ?D ) =>
    Destination := ?D.Destination;
    Destination_Speed := ?D.Speed;
    Destination_Heading := ?D.Heading;
    Actual_Start_Time := Clock.Now() + Start_Duration();;

Clock.Now() = Actual_Start_Time =>
    Mission_Objectives.Guidance_Started( Direct_Moving );
    State.Started := True;;

-- Start when enter capture radius of arming capture point:
Aircraft_State_Vector( ?V ) where
    (?V.Quality = Usable and State.Armed
    and Current_Location() - Arm_Capture_Point < Capture_Radius ) =>
    State.Armed := False;
    Mission_Objectives.Start_Guidance( Direct_Moving, Arm_Destination );;

-- Stop within Max_Stop_Time, regardless of additional Stop_Guidance events:
Mission_Objectives.Stop_Guidance( Direct_Moving ) =>
    Actual_Stop_Time := Clock.Now() + Stop_Duration();;

Clock.Now() = Actual_Stop_Time =>
    Actual_Stop_Time := Never;
    Mission_Objectives.Guidance_Stopped( Direct_Moving );
```

```
    State.Started := False;;

-- Stop immediately if Aircraft State Vector is unusable, another
-- guidance mode is started, or aircraft has entered the capture radius:
( Aircraft_State_Vector( ?V ) where ?V.Quality = Unusable ) or
( Aircraft_State_Vector( ?V ) where (?V.Quality = Usable and State.Started
    and Current_Location() - Destination < Capture_Radius ) ) or
Mission_Objectives.Guidance_Started( ?M ) where
    ?M /= Direct_Moving and State.Started =>
    State.Started := False;;

-- Behaviors to arm and disarm guidance:

-- Arm within Max_Arm_Time, unless interrupted by another arm event:
Mission_Objectives.Arm_Guidance( Direct_Moving, ?A ) =>
    Arm_Destination := ?A.Destination;
    Arm_Capture_Point := ?A.Capture;
    Actual_Arm_Time := Clock.Now() + Arm_Duration();;

Clock.Now() = Actual_Arm_Time =>
    Actual_Arm_Time := Never;
    Mission_Objectives.Guidance_Armed( Direct_Moving );
    State.Armed := True;;

Mission_Objectives.Disarm_Guidance( Direct_Moving ) =>
    Actual_Disarm_Time := Clock.Now() + Disarm_Duration();;

Clock.Now() = Actual_Disarm_Time =>
    Mission_Objectives.Guidance_Disarmed( Direct_Moving );
    State.Armed := False;;


constraint

    not match (Tick where
        Clock.Now() > Actual_Start_Time and Actual_Start_Time /= Never or
        Clock.Now() > Actual_Stop_Time and Actual_Stop_Time /= Never or
        Clock.Now() > Actual_Arm_Time and Actual_Arm_Time /= Never);

    -- IBM 3.2.3. Require change-guidance event to cause the following heading-error report:
    observe Mission_Objectives.Change_Guidance( Direct_Moving ) < Error_Signals.Heading_Error
    match   Mission_Objectives.Change_Guidance -> Error_Signals.Heading_Error;
end;
```

In addition to the types used in the VOR and Direct Fixed Point interfaces above (Sections 7.2.1 and 7.2.2), Direct_Moving_Point_Interface also uses Heading_Type and Speed_Type, each yet to be defined.

The first constraint is identical to that for Direct Fixed-Point guidance. The last constraint uses causality to represent that the data from a Change_Guidance event must be used in generating the next Heading_Error: every Heading_Error which follows a Change_Guidance event in time (<) must also be caused by that event (− >).

Chapter 8

Services

Services are interfaces which encapsulate sets of action declarations shared by other interfaces. They allow a higher degree of data abstraction and are easily altered as communication requirements are refined. The Rapide code for each service used in the ADAGE avionics example is given in its section below.

8.1 Body Motion

```
type Body_Motion_Service is interface

end;
```

8.2 Device Control

```
type Device_Control_Service is interface

    action Initialize_Sensor();
    action Align_Sensor();

    action Override_Data( S : Sensor_Data );
    action Select_Criteria( C : Selection_Criteria );
    action Aiding_Command( B : Boolean );

end;
```

8.3 Navigation Control

```
type Navigation_Control_Service is interface

end;
```

8.4 Mission Objectives

```
type Mission_Objectives_Service is interface

public

    action Start_Guidance( K : Lateral_Guidance_Kind;
                           D : Lateral_Guidance_Data );
    action Stop_Guidance( K : Lateral_Guidance_Kind );
    action Arm_Guidance( K : Lateral_Guidance_Kind;
                         A : Lateral_Guidance_Arming_Data );
    action Disarm_Guidance( K : Lateral_Guidance_Kind );
    action Change_Guidance( K : Lateral_Guidance_Kind;
                            C : Lateral_Guidance_Revision_Data );

extern
    action Guidance_Started( K : Lateral_Guidance_Kind );
    action Guidance_Stopped( K : Lateral_Guidance_Kind );
    action Guidance_Armed( K : Lateral_Guidance_Kind );
    action Guidance_Disarmed( K : Lateral_Guidance_Kind );

end;
```

8.5 Error Signals

```
type Error_Signals_Service is interface

    action Heading_Error( D : Heading_Error_Type );

end;
```

8.6 Driver Sensor

```
type Driver_Sensor_Service is interface

    action Request_Sensor_Data();
    action Request_Init_Device();
    action Request_Align_Device();
    action Aiding_Data_To_Sensor( D : Sensor_Data );

extern
    action Sampled_Sensor_Data( R : Raw_Data );
    action Device_Initialized();
    action Device_Aligned();

end;
```

```
type Driver_Sensor_Services is interface

    DNS_DS : service array(DNS_Range) of Driver_Sensor_Service;
    INU_DS : service array(INU_Range) of Driver_Sensor_Service;
    GPS_DS : service array(GPS_Range) of Driver_Sensor_Service;
    ADC_DS : service array(ADC_Range) of Driver_Sensor_Service;
    VOR_DS : service array(VOR_Range) of Driver_Sensor_Service;

end;
```

8.7 Source Navigation Service

```
type Source_Navigation_Service is interface

    action Aircraft_State_Data( K : Sensor_Kind; S : Sensor_Data );

extern
    action Navigation_Aiding_Data;

end;
```

8.8 Aircraft State

```
type Aircraft_State_Service is interface

    action Aircraft_State_Vector( V : State_Vector );
    action VOR_Relative_Navigation_Output();

end;
```

Chapter 9

Common Definitions

This interface collects data-type, function, and constant definitions used throughout the ADAGE avionics example.

```
 Clock : Clock_Pkg::Clock is Make_Clock();

type Common_Definitions is interface

   -- Types for sensor data:

   type Sensor_Data_Quality is enum Usable, Degraded, Unusable end;

   type Position_Type is (TBD);
   type Velocity_Type is (TBD);
   type Attitude_Type is (TBD);

   type Raw_Data is record
      Position : var Position_Type;
      Velocity : var Velocity_Type;
      Attitude : var Attitude_Type;
   end record;

   Default_Aircraft_State : Raw_Data is Raw_Data'(TBD);

   type Sensor_Data is record
      Quality : var Sensor_Data_Quality;   -- Initially Unusable;
      Measured_Aircraft_State : var Raw_Data; -- Initially Default_Aircraft_State;
   end record;


   -- Types for the sensor state:

   type Init_State_Type is enum Not_Initted, Initting, Initted end;
   type Align_State_Type is enum Not_Aligned, Aligning, Aligned end;

   type Sensor_State_Type is record
      I : var Init_State_Type; -- Initially Not_Initted;
      A : var Align_State_Type; -- Initially Not_Aligned;
   end record;
```

```
-- Types for selection criteria:

type Selection_Options is enum External, Best_Available, Sensor end;

-- Types for the various kinds of sensors:

type Sensor_Kind is enum DNS, INU, GPS, ADC, VOR end;

type Selection_Criteria is record
    Option : var Selection_Options;
    Kind : var Sensor_Kind;
    Sensor_Num : var Positive;
end record;

Default_Selection_Criteria : Selection_Criteria is Selection_Criteria'( Best_Available, INU, 1 );

type State_Vector_Info is (TBD);

type State_Vector is record
    Quality : var Sensor_Data_Quality;
    Vector : var State_Vector_Info;
end record;

type DNS_Range is Range(Integer, 1, 1);  -- IBM 2.4
type INU_Range is Range(Integer, 1, 2);  -- IBM 2.1, 2.2
type GPS_Range is Range(Integer, 1, 1);  -- IBM 2.3
type ADC_Range is Range(Integer, 1, 1);  -- IBM 2.5
type VOR_Range is Range(Integer, 1, 1);  -- IBM 2.6

Msec :  Time is  1;
Sec :   Time is  1000;
Never : Time is  (TBD);

DNS_Init_Duration :      Time is  1 * Sec;  -- IBM 2.4.2.5
DNS_Align_Duration :     Time is  Never;  -- IBM 2.4.3.5
DNS_First_Sample :       Time is  (TBD);
DNS_Sampling_Interval :  Time is  125 * Msec;  -- IBM 2.4.1.1
DNS_Aiding_Interval :    Time is  DNS_Sampling_Interval;  -- IBM 2.4.4

INU_Init_Duration :      Time is  3 * Sec;  -- IBM 2.1.2.5, 2.2.2.5
INU_Align_Duration :     Time is  5 * Sec;  -- IBM 2.1.3.5, 2.2.3.5
INU_First_Sample :       Time is  (TBD);
INU_Sampling_Interval :  Time is  50 * Msec;  -- IBM 2.1.1.1, 2.2.1.1
INU_Aiding_Interval :    Time is  INU_Sampling_Interval;  -- IBM 2.1.4, 2.2.4

GPS_Init_Duration :      Time is  10 * Sec;  -- IBM 2.3.2.5
GPS_Align_Duration :     Time is  Never;  -- IBM 2.3.3.5
GPS_First_Sample :       Time is  (TBD);
GPS_Sampling_Interval :  Time is  1000 * Msec;  -- IBM 2.3.1.1
GPS_Aiding_Interval :    Time is  Never;  -- IBM 2.5.4

ADC_Init_Duration :      Time is  1 * Sec;  -- IBM 2.5.2.5
ADC_Align_Duration :     Time is  Never;  -- IBM 2.5.3.5
```

```
ADC_First_Sample :       Time is  (TBD);
ADC_Sampling_Interval :  Time is  200 * Msec;  -- IBM 2.5.1.1
ADC_Aiding_Interval :    Time is  Never;  -- IBM 2.5.4

VOR_Init_Duration :      Time is  1 * Sec;  -- IBM 2.6.2.5
VOR_Tuning_Duration :    Time is  100 * Msec;  -- IBM 2.6.3.5
VOR_First_Sample :       Time is  (TBD);
VOR_Sampling_Interval :  Time is  50 * Msec;  -- IBM 2.6.1.1
VOR_Aiding_Interval :    Time is  Never;  -- IBM 2.6.4

ASV_Interval :           Time is  50 * Msec;
Initial_ASV_Time :       Time is  (TBD);

Next_Data_Time :         var Time;
Next_Aiding_Data_Time :  var Time;


-- Types for guidance data:

Capture_Radius :  Distance_Type is  (TBD);
Max_Start_Time :  Time is  100 * Msec;  -- IBM 3.1.3, 3.3.3
Max_Stop_Time :  Time is  50 * Msec;  -- IBM 3.1.5, 3.3.5
Max_Arm_Time :  Time is  100 * Msec;  -- IBM 3.1.11, 3.3.10

type Heading_Type is (TBD);
type Speed_Type is (TBD);
type Radial_Type is (TBD);
type Heading_Error_Type is (TBD);

type Destination_Point is (TBD);
type Distance_Type is (TBD);

function < ( D1, D2 : Distance_Type ) return Boolean;
function Distance( D1, D2 : Destination_Point ) return Distance_Type;

type Lateral_Guidance_Kind is enum Direct_Fixed, Direct_Moving, VOR end;

type Lateral_Guidance_Data is record
    Destination : var Destination_Point;    -- for Direct_Fixed and Direct_Moving
    Heading : var Heading_Type;             -- for Direct_Moving
    Speed : var Speed_Type;                 -- for Direct_Moving
    Radial : var Radial_Type;               -- for VOR
end record;

type Lateral_Guidance_Arming_Data is record
    Destination : var Destination_Point;    -- for Direct_Fixed and Direct_Moving
    Capture : var Destination_Point;        -- for Direct_Fixed and Direct_Moving
    Heading : var Heading_Type;             -- for Direct_Moving
    Speed : var Speed_Type;                 -- for Direct_Moving
    Radial : var Radial_Type;               -- for VOR
end record;

type Lateral_Guidance_Revision_Data is record
    Destination : var Destination_Point;    -- for Direct_Moving
    Heading : var Heading_Type;             -- for Direct_Moving
```

```
        Speed : var Speed_Type;            -- for Direct_Moving
    end record;


    -- Types for guidance state:

    type Guidance_State_Type is record
        Started, Armed : var Boolean;   -- Initially False;
    end record;

constraint

    -- This pattern macro is used to define the intervals at which
    -- certain events are to be produced:

    pattern Periodic( pattern P; Occurs, Period : Time ) is
        At( P, Occurs, Clock ) ->
          Periodic( P, Occurs+Period, Period );

end;
```

Chapter 10

Acknowledgements

We wish to recognize the contributions of Allen Adams (TRW) to previous versions of the sensor model and the contributions of Lou Coglianese (IBM), who answered innumerable questions that Allen posed. Neither, of course, is in any way responsible for any errors that may exist herein.

We also acknowledge the countless interactions with the entire Rapide team at Stanford, led by Prof. David C. Luckham.

Appendix A

The Original IBM ADAGE Specifications

Following is an enumeration of the features of the ADAGE avionics system as described in IBM document ADAGE-IBM-92-10 from Louis H. Coglianese to TRW, 15 May 1992. To the right of each feature is a cross-reference to the section of the present document in which this Rapide model establishes that feature.

The reader can gain insight into the way behaviors are represented in Rapide's pattern language by comparing the Rapide **behaviors** in the sections referenced in the following tables with the corresponding IBM natural-language English specifications in the first column of the tables.

IBM Memo:		This paper:		
Paragr.	Subject or Feature No.	Sect.	Compilation Unit	Component
1.0	Common Definitions			
1.1	Sensor data	9	Common_Definitions	Sensor_Data
1.1	Quality (either Usable, Degraded, or Unusable)	9	Common_Definitions	Sensor_Data_Quality[1]
1.1	Position, velocity, attitude	9	Common_Definitions	Raw_Data
1.2	Functional data	9	Data_Sources_Interface	**action** Source_Navigation_Connection
1.3	Selection criteria (EXTERNAL, BEST_AVAILABLE, or one of the sensors)	9	Common_Definitions	Selection_Options
2.0	Navigation	2.2	Partial_Top_Level_Architecture	Navigation

[1]The IBM memo listed but did not use the quality "Degraded".

IBM Memo:		This paper:		
Paragr.	Subject or Feature No.	Sect.	Compilation Unit	Component
2.1	INU-1 Device Driver	4.2	Data_Sources_Architecture	INU_Drivers
2.1.1	General			
2.1.1.1	Sample and output INU-1 sensor data	4.2.1	Device_Driver	
2.1.1.1	20-Hz INU-1 sensor data rate	9	Common_Definitions	INU_Sampling_Interval
2.1.1.2	Report output sensor-data quality as Usable or Unusable	4.2.1	Device_Driver	
2.1.1.3	Sensor data initial value Unusable	9	Common_Definitions	Sensor_Data_Quality
2.1.2	Initialization			
2.1.2.1	Accept commands to initialize INU-1 any time	4.2.1, 3.2.1	Device_Driver, Sensor	No constraints on Initialize_Sensor means it can occur any time
2.1.2.2	Start INU-1 initialization before the next sampling interval	4.2.1, 3.2.1	Device_Driver, Sensor	Begins immediately
2.1.2.3	Output aiding data only after INU-1 has been initialized	4.2.1	Device_Driver	**behavior**
2.1.2.4	Mark as Unusable output sensor data sampled during initialization	4.2.1	Device_Driver	**behavior**
2.1.2.5	INU-1 initialization takes 3 seconds	9, 3.2.1	Common_Definitions, Sensor	INU_Init_Duration
2.1.3	Device-Specific Controls			
2.1.3.1	Accept commands to align INU-1 any time	4.2.1, 3.2.1	Device_Driver, Sensor	Align_Sensor command may be observed any time
2.1.3.2	Start INU-1 alignment before the next sampling interval	4.2.1, 3.2.1	Device_Driver, Sensor	Begins immediately
2.1.3.3	Start alignment only after initialization is complete	4.2.1	Device_Driver	**constraint**
2.1.3.4	Mark as Unusable output sensor data sampled during alignment	4.2.1	Device_Driver	**behavior**
2.1.3.5	INU-1 alignment takes 5 seconds	9, 3.2.1	Common_Definitions, Sensor	INU_Align_Duration
2.1.4	Aiding Data			
2.1.4.1	Aiding data sent to INU-1 at the maximum of the sampling rate (see 2.1.1.1) and the rate at which aiding data arrives	4.2.1	Device_Driver	**behavior**
2.1.4.2	Aiding data not sent during initialization	4.2.1	Device_Driver	**behavior**: aiding data sent only after initialization
2.1.4.3	Output aiding data only after alignment is complete	4.2.1	Device_Driver	**behavior**
2.1.4.4	Identical to 2.1.3.4 above	4.2.1	Device_Driver	**behavior**
2.1.4.5	Aiding data not sent during alignment	4.2.1	Device_Driver	**behavior**: see 2.1.4.3 above

IBM Memo:		This paper:		
Paragr.	Subject or Feature No.	Sect.	Compilation Unit	Component
2.2	INU-2 Device Driver	4.2	Data_Sources_Architecture	INU_Drivers
2.2.1	General			
2.2.1.1	Sample and output INU-2 sensor data	4.2.1	Device_Driver	
2.2.1.1	20-Hz INU-2 sensor data rate	9	Common_Definitions	INU_Sampling_Interval
2.2.1.2	Report output sensor-data quality as Usable or Unusable	4.2.1	Device_Driver	
2.2.1.3	Sensor data initial value Unusable	9	Common_Definitions	Sensor_Data_Quality
2.2.2	Initialization			
2.2.2.1	Accept commands to initialize INU-2 any time	4.2.1, 3.2.1	Device_Driver, Sensor	No constraints on Initialize_Sensor means it can occur any time
2.2.2.2	Start INU-2 initialization before the next sampling interval	4.2.1, 3.2.1	Device_Driver, Sensor	Begins immediately
2.2.2.3	Output aiding data only after INU-2 has been initialized	4.2.1	Device_Driver	**behavior**
2.2.2.4	Mark as Unusable output sensor data sampled during initialization	4.2.1	Device_Driver	**behavior**
2.2.2.5	INU-2 initialization takes 3 seconds	9, 3.2.1	Common_Definitions, Sensor	INU_Init_Duration
2.2.3	Device-Specific Controls			
2.2.3.1	Accept commands to align INU-2 any time	4.2.1, 3.2.1	Device_Driver, Sensor	Align_Sensor command may be observed any time
2.2.3.2	Start INU-2 alignment before the next sampling interval	4.2.1, 3.2.1	Device_Driver, Sensor	Begins immediately
2.2.3.3	Start alignment only after initialization is complete	4.2.1	Device_Driver	**constraint**
2.2.3.4	Mark as Unusable output sensor data sampled during alignment	4.2.1	Device_Driver	**behavior**
2.2.3.5	INU-2 alignment takes 5 seconds	9, 3.2.1	Common_Definitions, Sensor	INU_Align_Duration
2.2.4	Aiding Data			
2.2.4.1	Aiding data sent to INU-2 at the maximum of the sampling rate (see 2.2.1.1) and the rate at which aiding data arrives	4.2.1	Device_Driver	**behavior**
2.2.4.2	Aiding data not sent during initialization	4.2.1	Device_Driver	**behavior**: aiding data sent only after initialization
2.2.4.3	Output aiding data only after alignment is complete	4.2.1	Device_Driver	**behavior**
2.2.4.4	Identical to 2.2.3.4 above	4.2.1	Device_Driver	**behavior**
2.2.4.5	Aiding data not sent during alignment	4.2.1	Device_Driver	**behavior**: see 2.2.4.3 above

IBM Memo:		This paper:		
Paragr.	Subject or Feature No.	Sect.	Compilation Unit	Component
2.3	GPS Device Driver	4.2	Data_Sources_Architecture	GPS_Drivers
2.3.1	General			
2.3.1.1	Sample and output GPS sensor data	4.2.1	Device_Driver	
2.3.1.1	1-Hz GPS sensor data rate	9	Common_Definitions	GPS_Sampling_Interval
2.3.1.2	Report output sensor-data quality as Usable or Unusable	4.2.1	Device_Driver	
2.3.1.3	Sensor data initial value Unusable	9	Common_Definitions	Sensor_Data_Quality
2.3.2	Initialization			
2.3.2.1	Accept commands to initialize GPS any time	4.2.1, 3.2.1	Device_Driver, Sensor	No constraints on Initialize_Sensor means it can occur any time
2.3.2.2	Start GPS initialization before the next sampling interval	4.2.1, 3.2.1	Device_Driver, Sensor	Begins immediately
2.3.2.3	Output aiding data only after GPS has been initialized	4.2.1	Device_Driver	**behavior**
2.3.2.4	Mark as Unusable output sensor data sampled during initialization	4.2.1	Device_Driver	**behavior**
2.3.2.5	GPS initialization takes 10 seconds	9, 3.2.1	Common_Definitions, Sensor	GPS_Init_Duration
2.3.3	Device-Specific Controls			
2.3.3	GPS alignment is not defined	9, 3.2.1	Common_Definitions, Sensor	GPS_Align_Duration = Never
2.3.4	Aiding Data			
2.3.4.1	Aiding data sent to GPS at the maximum of the sampling rate (see 2.3.1.1) and the rate at which aiding data arrives	4.2.1	Device_Driver	**behavior**
2.3.4.2	Aiding data not sent during initialization	4.2.1	Device_Driver	**behavior**: aiding data sent only after initialization
2.3.4.3	Output aiding data only after alignment is complete	4.2.1	Device_Driver	**behavior**
2.3.4.4	Identical to 2.3.3.4 above	4.2.1	Device_Driver	**behavior**
2.3.4.5	Aiding data not sent during alignment	4.2.1	Device_Driver	**behavior**: see 2.3.4.3 above

IBM Memo:		This paper:		
Paragr.	Subject or Feature No.	Sect.	Compilation Unit	Component
2.4	DNS Device Driver	4.2	Data_Sources_Architecture	DNS_Drivers
2.4.1	General			
2.4.1.1	Sample and output DNS sensor data	4.2.1	Device_Driver	
2.4.1.1	8-Hz DNS sensor data rate	9	Common_Definitions	DNS_Sampling_Interval
2.4.1.2	Report output sensor-data quality as Usable or Unusable	4.2.1	Device_Driver	
2.4.1.3	Sensor data initial value Unusable	9	Common_Definitions	Sensor_Data_Quality
2.4.2	Initialization			
2.4.2.1	Accept commands to initialize DNS any time	4.2.1, 3.2.1	Device_Driver, Sensor	No constraints on Initialize_Sensor means it can occur any time
2.4.2.2	Start DNS initialization before the next sampling interval	4.2.1, 3.2.1	Device_Driver, Sensor	Begins immediately
2.4.2.3	Output aiding data only after DNS has been initialized	4.2.1	Device_Driver	**behavior**
2.4.2.4	Mark as Unusable output sensor data sampled during initialization	4.2.1	Device_Driver	**behavior**
2.4.2.5	DNS initialization takes 1 second	9, 3.2.1	Common_Definitions, Sensor	DNS_Init_Duration
2.4.3	Device-Specific Controls			
2.4.3	DNS alignment is undefined	9, 3.2.1	Common_Definitions, Sensor	DNS_Align_Duration = Never
2.4.4	Aiding Data			
2.4.4.1	Aiding data sent to DNS at the maximum of the sampling rate (see 2.4.1.1) and the rate at which aiding data arrives	4.2.1	Device_Driver	**behavior**
2.4.4.2	Aiding data not sent during initialization	4.2.1	Device_Driver	**behavior**: aiding data sent only after initialization
2.4.4.3	Output aiding data only after alignment is complete	4.2.1	Device_Driver	**behavior**
2.4.4.4	Identical to 2.4.3.4 above	4.2.1	Device_Driver	**behavior**
2.4.4.5	Aiding data not sent during alignment	4.2.1	Device_Driver	**behavior**: see 2.4.4.3 above

IBM Memo:		This paper:		
Paragr.	Subject or Feature No.	Sect.	Compilation Unit	Component
2.5	ADC Device Driver	4.2	Data_Sources_Architecture	ADC_Drivers
2.5.1	General			
2.5.1.1	Sample and output ADC sensor data	4.2.1	Device_Driver	
2.5.1.1	5-Hz ADC sensor data rate	9	Common_Definitions	ADC_Sampling_Interval
2.5.1.2	Report output sensor-data quality as Usable or Unusable	4.2.1	Device_Driver	
2.5.1.3	Sensor data initial value Unusable	9	Common_Definitions	Sensor_Data_Quality
2.5.2	Initialization			
2.5.2.1	Accept commands to initialize ADC any time	4.2.1, 3.2.1	Device_Driver, Sensor	No constraints on Initialize_Sensor means it can occur any time
2.5.2.2	Start ADC initialization before the next sampling interval	4.2.1, 3.2.1	Device_Driver, Sensor	Begins immediately
2.5.2.3	Output aiding data only after ADC has been initialized	4.2.1	Device_Driver	**behavior**
2.5.2.4	Mark as Unusable output sensor data sampled during initialization	4.2.1	Device_Driver	**behavior**
2.5.2.5	ADC initialization takes 1 second	9, 3.2.1	Common_Definitions, Sensor	ADC_Init_Duration
2.5.3	Device-Specific Controls			
2.5.3	ADC alignment is undefined	9, 3.2.1	Common_Definitions, Sensor	ADC_Align_Duration = Never
2.5.4	Aiding Data			
2.5.4.1	Aiding data sent to ADC at the maximum of the sampling rate (see 2.5.1.1) and the rate at which aiding data arrives	4.2.1	Device_Driver	**behavior**
2.5.4.2	Aiding data not sent during initialization	4.2.1	Device_Driver	**behavior**: aiding data sent only after initialization
2.5.4.3	Output aiding data only after alignment is complete	4.2.1	Device_Driver	**behavior**
2.5.4.4	Identical to 2.5.3.4 above	4.2.1	Device_Driver	**behavior**
2.5.4.5	Aiding data not sent during alignment	4.2.1	Device_Driver	**behavior**: see 2.5.4.3 above

IBM Memo:		This paper:		
Paragr.	Subject or Feature No.	Sect.	Compilation Unit	Component
2.6	VOR Device Driver	4.2	Data_Sources_Architecture	VOR_Drivers
2.6.1	General			
2.6.1.1	Sample and output VOR sensor data	4.2.1	Device_Driver	
2.6.1.1	10-Hz VOR sensor data rate	9	Common_Definitions	VOR_Sampling_Interval
2.6.1.2	Report output sensor-data quality as Usable or Unusable	4.2.1	Device_Driver	
2.6.1.3	Sensor data initial value Unusable	9	Common_Definitions	Sensor_Data_Quality
2.6.2	Initialization			
2.6.2.1	Accept commands to initialize VOR any time	4.2.1, 3.2.1	Device_Driver, Sensor	No constraints on Initialize_Sensor means it can occur any time
2.6.2.2	Start VOR initialization before the next sampling interval	4.2.1, 3.2.1	Device_Driver, Sensor	Begins immediately
2.6.2.3	Output aiding data only after VOR has been initialized	4.2.1	Device_Driver	**behavior**
2.6.2.4	Mark as Unusable output sensor data sampled during initialization	4.2.1	Device_Driver	**behavior**
2.6.2.5	VOR initialization takes 1 second	9, 3.2.1	Common_Definitions, Sensor	VOR_Init_Duration
2.6.3	Device-Specific Controls			
2.6.3.1	Accept commands to tune VOR any time	—	Modelled like sensor alignment	—
2.6.3.2	Start VOR tuning before the next sampling interval	4.2.1, 3.2.1	Device_Driver, Sensor	Begins immediately
2.6.3.3	Start tuning only after initialization is complete	4.2.1	Device_Driver	**constraint**
2.6.3.4	Mark as Unusable output sensor data sampled during tuning	4.2.1	Device_Driver	**behavior**
2.6.3.5	VOR tuning takes 0.1 seconds	9, 3.2.1	Common_Definitions, Sensor	VOR_Tuning_Duration = 0.1
2.6.4	Aiding Data			
2.6.4.1	Aiding data sent to VOR at the maximum of the sampling rate (see 2.6.1.1) and the rate at which aiding data arrives	4.2.1	Device_Driver	**behavior**
2.6.4.2	Aiding data not sent during initialization	4.2.1	Device_Driver	**behavior**: aiding data sent only after initialization
2.6.4.3	Output aiding data only after alignment is complete	4.2.1	Device_Driver	**behavior**
2.6.4.4	Identical to 2.6.3.4 above	4.2.1	Device_Driver	**behavior**
2.6.4.5	Aiding data not sent during alignment	4.2.1	Device_Driver	**behavior**: see 2.6.4.3 above

IBM Memo:		This paper:		
Paragr.	Subject or Feature No.	Sect.	Compilation Unit	Component
2.7	INU Functional-Data Source	4.2	Data_Sources_Architecture	INU_Data_Source
2.7.1	General			
2.7.1.1	Accept data from both INU-1 and INU-2 device drivers	4.2	Data_Sources_Architecture	Connections among internal components
2.7.1.2	Accept override data from external source at any time	4.2.2	Data_Source_Interface	**behavior**
2.7.1.3	Accept selection criteria from external source at any time	4.2.2	Data_Source_Interface	**behavior**
2.7.1.4	Output sensor data from INU-1 or INU-2 when selection criterion is that sensor	4.2.2	Data_Source_Interface	**behavior**
2.7.1.5	Output sensor data from external override when selection criterion is External	4.2.2	Data_Source_Interface	**behavior**
2.7.1.6	Output sensor data from best-quality source when selection criterion is Best_Available	4.2.2	Data_Source_Interface	**behavior**
2.7.1.7	Output sensor data at the same rate as the INUs produce output	4.2.2	Data_Source_Interface	**behavior**
2.7.2	Aiding			
2.7.2.1	Accept commands to start or stop aiding data	4.2	Data_Source_Interface	**behavior**
2.7.2.2	Outputs functional sensor data to the GPS device driver	4.2	Data_Sources_Architecture	Connections among internal components
2.7.2.2	Outputs functional sensor data at the rate it is produced	4.2.2	Data_Source_Interface	**behavior**

IBM Memo:		This paper:		
Paragr.	Subject or Feature No.	Sect.	Compilation Unit	Component
2.8	GPS Functional-Data Source	4.2	Data_Sources_Architecture	GPS_Data_Source
2.8.1	General			
2.8.1.1	Accept data from GPS device driver	4.2	Data_Sources_Architecture	Connections among internal components
2.8.1.2	Accept override data from external source at any time	4.2.2	Data_Source_Interface	**behavior**
2.8.1.3	Accept selection criteria from external source at any time	4.2.2	Data_Source_Interface	**behavior**
2.8.1.4	Output sensor data from GPS when selection criterion is that sensor	4.2.2	Data_Source_Interface	**behavior**
2.8.1.5	Output sensor data from external override when selection criterion is External	4.2.2	Data_Source_Interface	**behavior**
2.8.1.6	Output sensor data from best-quality source when selection criterion is Best_Available	4.2.2	Data_Source_Interface	**behavior**
2.8.1.7	Output sensor data at the same rate as the GPS produces output	4.2.2	Data_Source_Interface	**behavior**
2.8.2	Aiding			
2.8.2.1, 2.8.2.2	No aiding data	4.2.2	Data_Source_Interface	**behavior**: Generates_Aiding_Data = False

IBM Memo:		This paper:		
Paragr.	Subject or Feature No.	Sect.	Compilation Unit	Component
2.9	DNS Functional-Data Source	4.2	Data_Sources_Architecture	DNS_Data_Source
2.9.1	General			
2.9.1.1	Accept data from DNS device driver	4.2	Data_Sources_Architecture	Connections among internal components
2.9.1.2	Accept override data from external source at any time	4.2.2	Data_Source_Interface	**behavior**
2.9.1.3	Accept selection criteria from external source at any time	4.2.2	Data_Source_Interface	**behavior**
2.9.1.4	Output sensor data from DNS when selection criterion is that sensor	4.2.2	Data_Source_Interface	**behavior**
2.9.1.5	Output sensor data from external override when selection criterion is External	4.2.2	Data_Source_Interface	**behavior**
2.9.1.6	Output sensor data from best-quality source when selection criterion is Best_Available	4.2.2	Data_Source_Interface	**behavior**
2.9.1.7	Output sensor data at the same rate as the DNS produces output	4.2.2	Data_Source_Interface	**behavior**
2.9.2	Aiding			
2.9.2.1	Accept commands to start or stop aiding data	4.2.2	Data_Source_Interface	**behavior**
2.9.2.2	Outputs functional sensor data to either or both INU device drivers	4.2	Data_Sources_Architecture	Connections among internal components
2.9.2.2	Outputs functional sensor data at the rate it is produced	4.2.2	Data_Source_Interface	**behavior**

IBM Memo:		This paper:		
Paragr.	Subject or Feature No.	Sect.	Compilation Unit	Component
2.10	ADC Functional-Data Source	4.2	Data_Sources_Architecture	ADC_Data_Source
2.10.1	General			
2.10.1.1	Accept data from ADC device driver	4.2	Data_Sources_Architecture	Connections among internal components
2.10.1.2	Accept override data from external source at any time	4.2.2	Data_Source_Interface	**behavior**
2.10.1.3	Accept selection criteria from external source at any time	4.2.2	Data_Source_Interface	**behavior**
2.10.1.4	Output sensor data from ADC when selection criterion is that sensor	4.2.2	Data_Source_Interface	**behavior**
2.10.1.5	Output sensor data from external override when selection criterion is External	4.2.2	Data_Source_Interface	**behavior**
2.10.1.6	Output sensor data from best-quality source when selection criterion is Best_Available	4.2.2	Data_Source_Interface	**behavior**
2.10.1.7	Output sensor data at the same rate as the ADC produces output	4.2.2	Data_Source_Interface	**behavior**
2.10.2	Aiding			
2.10.2.1, 2.10.2.2	No aiding data	4.2.2	Data_Source_Interface	**behavior**: Generates_Aiding_Data = False
2.11	VOR Functional-Data Source	4.2	Data_Sources_Architecture	VOR_Data_Source
2.11.1	General			
2.11.1.1	Accept data from VOR device driver	4.2	Data_Sources_Architecture	Connections among internal components
2.11.1.2	Accept override data from external source at any time	4.2.2	Data_Source_Interface	**behavior**
2.11.1.3	Accept selection criteria from external source at any time	4.2.2	Data_Source_Interface	**behavior**
2.11.1.4	Output sensor data from VOR when selection criterion is that sensor	4.2.2	Data_Source_Interface	**behavior**
2.11.1.5	Output sensor data from external override when selection criterion is External	4.2.2	Data_Source_Interface	**behavior**
2.11.1.6	Output sensor data from best-quality source when selection criterion is Best_Available	4.2.2	Data_Source_Interface	**behavior**
2.11.1.7	Output sensor data at the same rate as the VOR produces output	4.2.2	Data_Source_Interface	**behavior**
2.11.2	Aiding			
2.11.2.1, 2.11.2.2	No aiding data	4.2.2	Data_Source_Interface	**behavior**: Generates_Aiding_Data = False

IBM Memo:		This paper:		
Paragr.	Subject or Feature No.	Sect.	Compilation Unit	Component
2.12.0	Earth Model	5.2	Navigation_Architecture	Earth_Model
2.12.0.1	Output earth-based data using INU functional data when produced	5.2.1	Earth_Model_Interface	**behavior**
2.12.1	Atmosphere Model	5.2	Navigation_Architecture	Atmosphere_Model
2.12.1.1	Output atmosphere-based data using ADC functional data when produced	5.2.2	Atmosphere_Model_Interface	**behavior**
2.12.1.2	Output estimated wind data using ADC functional data when produced	5.2.2	Atmosphere_Model_Interface	**behavior**
2.12.2	Aircraft State Vector Model	5.2	Navigation_Architecture	Aircraft_State_Vector_Model
2.12.2.1	Accept functional data from INU at rate produced	5.2.3	Aircraft_State_Vector_Model_Interface	**behavior**
2.12.2.2	Accept functional data from GPS at rate produced	5.2.3	Aircraft_State_Vector_Model_Interface	**behavior**
2.12.2.3	Accept functional data from DNS at rate produced	5.2.3	Aircraft_State_Vector_Model_Interface	**behavior**
2.12.2.4	Accept functional data from ADC at rate produced	5.2.3	Aircraft_State_Vector_Model_Interface	**behavior**
2.12.2.5	Accept earth-based data from earth model at rate produced	5.2.3	Aircraft_State_Vector_Model_Interface	**behavior**
2.12.2.6	Accept atmosphere-based data from atmosphere model at rate produced	5.2.3	Aircraft_State_Vector_Model_Interface	**behavior**
2.12.2.7	Compute and output aircraft-state vector at 20 Hz	9	Common_Definitions	ASV_Interval
2.12.2.8	Run only after all data have been updated	5.2.3	Aircraft_State_Vector_Model_Interface	**behavior**: **function** All_Elements_Updated
2.12.2.9	Mark vector unusable if no functional data sources are usable	5.2.3	Aircraft_State_Vector_Model_Interface	**behavior**: **function** Calculate_Vector_Value
2.13	VOR Radio Navigation	5.2	Navigation_Architecture	VOR_Relative_Navigation
2.13.1	Output filtered VOR data using VOR functional data when produced	5.2.4	VOR_Relative_Navigation_Interface	**behavior**

IBM Memo:		This paper:		
Paragr.	Subject or Feature No.	Sect.	Compilation Unit	Component
3.0	Guidance	2.2	Partial_Top_Level_Architecture	Guidance
3.1	Direct Lateral Fixed Point Guidance	7.2	Lateral_Guidance_Architecture	Direct_Fixed_Point
3.1.1	Output heading errors when usable aircraft-state vector produced	7.2.2	Direct_Fixed_Point_Interface	**behavior**: Current_Heading_Error
3.1.2	Accept commands to start Direct Lateral Fixed Point Guidance, including destination point, at any time	7.2.2	Direct_Fixed_Point_Interface	**behavior**: Destination
3.1.3	Start outputting heading errors within 0.1 seconds of start command	9, 7.2.2	Common_Definitions, Direct_Fixed_Point_Interface	**constraint**: Max_Start_Time = 100 msec
3.1.4	Accept commands to stop Direct Lateral Fixed Point Guidance at any time	7.2.2	Direct_Fixed_Point_Interface	**behavior**
3.1.5	Stop outputting heading errors within 0.05 seconds of start command	9, 7.2.2	Common_Definitions, Direct_Fixed_Point_Interface	**constraint**: Max_Start_Time = 50 msec
3.1.6	Stop outputting heading errors when aircraft-state vector becomes unusable	7.2.2	Direct_Fixed_Point_Interface	**behavior**: stop condition
3.1.7	Stop outputting heading errors whenever another lateral-guidance mode starts	7.2.2	Direct_Fixed_Point_Interface	**behavior**: stop condition
3.1.8	Stop outputting heading errors when aircraft arrived within capture radius of destination	7.2.2	Direct_Fixed_Point_Interface	**behavior**: stop condition
3.1.9	Accept commands to arm Direct Lateral Fixed Point Guidance, including destination point and capture point, at any time	7.2.2	Direct_Fixed_Point_Interface	**behavior**: Destination, Capture_Radius
3.1.10	Start outputting heading errors within whenever armed and arrived within the capture radius of capture point	7.2.2	Direct_Fixed_Point_Interface	**behavior**: start condition
3.1.11	Start estimating distance to capture point within 0.1 seconds of arm command	9, 7.2.2	Common_Definitions, Direct_Fixed_Point_Interface	**constraint**: Max_Arm_Time = 100 msec

IBM Memo:		This paper:		
Paragr.	Subject or Feature No.	Sect.	Compilation Unit	Component
3.2	Direct Lateral Moving Point Guidance	7.2	Lateral_Guidance_Architecture	Direct_Moving_Point
3.2.1	Accept commands to start Direct Lateral Moving Point Guidance, including destination point, speed, and heading	7.2.3	Direct_Moving_Point_Interface	**behavior**: Destination, Destination_Speed, Destination_Heading
3.2.2	Accept commands to change destination point, speed, and heading, at any time	7.2.3	Direct_Moving_Point_Interface	**behavior**: change state
3.2.3	Start outputting heading errors within 0.1 seconds of start command	9, 7.2.3	Common_Definitions, Direct_Moving_Point_Interface	Last **constraint**
3.3	VOR Lateral Guidance	7.2	Lateral_Guidance_Architecture	VOR
3.3.1	Output heading errors when usable aircraft-state vector produced	7.2.1	VOR_Interface	**behavior**: Current_Heading_Error
3.3.2	Accept commands to start VOR Lateral Guidance, including radial, at any time	7.2.1	VOR_Interface	**behavior**: Radial
3.3.3	Start outputting heading errors within 0.1 seconds of start command	9, 7.2.1	Common_Definitions, VOR_Interface	**constraint**: Max_Start_Time = 100 msec
3.3.4	Accept commands to stop VOR Lateral Guidance at any time	7.2.1	VOR_Interface	**behavior**
3.3.5	Stop outputting heading errors within 0.05 seconds of stop command	9, 7.2.1	Common_Definitions, VOR_Interface	**constraint**: Max_Start_Time = 50 msec
3.3.6	Stop outputting heading errors when aircraft-state vector becomes unusable	7.2.1	VOR_Interface	**behavior**: stop condition
3.3.7	Stop outputting heading errors whenever another lateral-guidance mode starts	7.2.1	VOR_Interface	**behavior**: stop condition
3.3.8	Accept commands to arm VOR Lateral Guidance, including radial	7.2.1	VOR_Interface	**behavior**: Radial
3.3.9	Start outputting heading errors within whenever armed and arrived at the radial	7.2.1	VOR_Interface	**behavior**: start condition
3.3.10	Start estimating distance to capture point within 0.1 seconds of arm command	9, 7.2.1	Common_Definitions, VOR_Interface	**constraint**: Max_Arm_Time = 100 msec

Appendix B

An Alternative Sensor-Interface Model

The omnibus definition of a sensor as described originally in Section 3.2.1 (page 16) may be decomposed, using Rapide's inheritability properties, so as to allow modular production of both basic, initializable, alignable, and complete sensors, the last having both initializing and aligning actions along with their proper logical/temporal relationships.

Here we are assuming a feature of Rapide which we hope will be present in the future in some form but does not currently exist: having one **include** inherit all components of one type into another with the various parts of the inherited type incorporated into the respective corresponding parts of the inheriting type. (This now requires a separate **include** in each part, which inherits components only from its parent's corresponding part.)

The exercise consists in creating some eight types so that combinations of them may be **include**d into others to fabricate all desired sensor behaviors with no redundant code. Each type thus contains just the right subset of sensor properties, so that, this "sensor kit" can, for example, build the full initializable and alignable sensor as well as, say, an initializable-only sensor. The three basic units and what they contain are as follows:

- Simplest_Sensor_Service — only the two actions of sensor-sampling request and sampled-data return
- Simplest_Sensor_Declarations — instantiations of the simplest-sensor service and its data
- Simplest_Sensor — the sensor-sampling construct itself

```
type   Simplest_Sensor_Service   is  interface

    action   Request();

extern
    action   Sampled(  R  :  Raw_Data  );

end;


type   Simplest_Sensor_Declarations  is  interface
```

```
    Sensor_Data : service  Simplest_Sensor_Service;

behavior

    Some_Data : Raw_Data;

end;


type  Simplest_Sensor is interface

include  Simplest_Sensor_Declarations;

behavior

    begin

    -- When sensor-data Request event is detected, pass some data via
    -- sensor-data Sampled event:

    Sensor_Data.Request => Sensor_Data.Sampled(Some_Data);;

end;
```

We then introduce the general concept of a "serviceable" sensor — one which may be initialized, aligned, or anything else the designer might need. The four nested boxes embodying this concept are the following:

- Serviceable_Sensor_Service — analogously to the Simplest_Sensor_Service, only two actions: the service request and notification that the service is complete
- Serviceable_Sensor_Declarations — the entities needed for the service and initial values for two of them
- Serviceable_Sensor_Task — the tidy execution of the requested task itself, with protection against multiple requests during task execution
- Serviceable_Sensor — the sensor sampling and data return, conditional upon this task having been properly concluded (the Serviceable_Sensor being included into instantiations of initializable or alignable sensors with appropriate **replaces**)

```
type  Serviceable_Sensor_Service  is interface

    action  Request();

extern
    action  Complete();

end;


type  Serviceable_Sensor_Declarations  is interface
```

```
    Task : service   Serviceable_Sensor_Service;

behavior

    Task_State_Type is enum (Not_Done, Doing, Done) end;
    Task_State : var Task_State_Type := Not_Done;
    Task_Complete_Time : var Time := Never;

end;


type  Serviceable_Sensor_Task(Task_Duration : Time) is interface
    include   Serviceable_Sensor_Declarations;

behavior

    begin

    -- When task is requested, set Task_Complete_Time to the time task should
    -- finish.  Perform Complete only if Task_Complete_Time has not been
    -- overwritten to another value by a later request for the same task.

    Task.Request   where Task_State = Not_Done   =>
        Task_State := Doing;
        Task_Complete_Time := Clock + Task_Duration;;

    Clock = Task_Complete_Time and Task_State = Doing =>
        Task_Complete_Time := Never;
        Task_State := Done;
        Task.Complete;;

end;


type  Serviceable_Sensor(Task_Duration : Time) is interface
    include   Simplest_Sensor_Declarations;
    include   Serviceable_Sensor_Declarations;

behavior

    begin

    -- Pass data upon request if task is done:

    Sensor_Data.Request   where (Task_State = Done) =>
        Sensor_Data.Sampled(Sample_Data);;

end;
```

Finally, we may now construct the full sensor with the prescribed logical/temporal relationship between initialization and alignment (not hitherto specified, of course) prior to sensor sampling.

```
type  Full_Sensor(Init_Duration, Align_Duration : Time ) is interface
```

```
    -- Import sensor data and service actions:
    include Simplest_Sensor_Declarations;

    -- Import initializable-sensor declarations and initialization task:
    include Serviceable_Sensor_Task(Init_Duration)
        replace
            Task                    by Init,
            Task_State              by Init_State,
            Task_State_Type         by Init_State_Type,
            Task_Complete_Time      by Init_Complete_Time,

    -- Import alignable-sensor declarations:
    include Serviceable_Sensor_Task(Align_Duration)
        replace
            Task                    by Align,
            Task_State              by Align_State,
            Task_State_Type         by Align_State_Type,
            Task_Complete_Time      by Align_Complete_Time,


behavior

    begin

    Init.Request   where Align_State /= Not_Done   =>
        Align_State := Not_Done;

    Align.Request   where Align_State = Not_Done   and   Init_State = Done   =>
        Align_State := Doing;
        Align.Complete   at Clock + Align_Duration;;

    Align.Complete   where Align_State = Doing   =>
        Align_State := Done;
        Align.Complete;;

    Sensor_Data.Request where Init_State = Done and Align_State = Done   =>
        Sensor_Data.Sampled(Some_Data);;

constraint

    -- The protocol followed by the sensor:
    match
        ( Init.Done^(<+) < Align.Done^(<+) where Align_Duration /= Never
          or null where Align_Duration = Never)^(<*);

end;
```

Appendix C

A Unified Lateral-Guidance-Type Interface

Among the three lateral-guidance interfaces (Sections 7.2.1-7.2.3, pages 42-48), so much of the Rapide code is identical that it is tempting to try to construct a single guidance interface in which the differences are selected by guidance type:

```
type Guidance_Mode_Interface_Generator( Guidance_Kind : Lateral_Guidance_Kind)
          is interface

    Aircraft_State : service Aircraft_State_Service;
    Mission_Objectives : service Mission_Objectives_Service;

extern
    Error_Signals : service Error_Signals_Service;


behavior

    ?V : State_Vector;
    ?D : Lateral_Guidance_Data;
    ?M : Lateral_Guidance_Kind;
    ?A : Lateral_Guidance_Arming_Data;

    State : var Guidance_State_Type;   -- Initially ( False, False );

    Radial, Arm_Radial : var Radial_Type;
    Destination, Arm_Destination, Arm_Capture_Point : var Destination_Point;
    Destination_Speed : var Speed_Type;
    Destination_Heading : var Heading_Type;

    Actual_Start_Time, Actual_Stop_Time : var Time := Never;
    Actual_Arm_Time, Actual_Disarm_Time : var Time := Never;

    -- Future duration functions to return the times needed to perform respective tasks:

    function Start_Duration() return Time is
    begin return TBD;
```

```
end function;

function Stop_Duration() return Time is
begin return TBD;
end function;

function Arm_Duration() return Time is
begin return TBD;
end function;

function Disarm_Duration() return Time is
begin return TBD;
end function;

-- Represents computation of current heading error:
function Current_Heading_Error() return Heading_Error_Type is
begin return TBD;
end function;

-- Represents computation of current radial:
function Current_Radial() return Radial_Type is
begin return TBD;
end function;

-- Represents computation of current location:
function Current_Location() return Destination_Point is
begin return TBD;
end function;


begin

Start =>
    State := Guidance_State_Type'( False, False );

-- Output heading error if Aircraft State Vector is usable and state is started:
Aircraft_State.Aircraft_State_Vector( ?V ) where
    ?V.Quality = Usable and State.Started and
    if Guidance_Kind = VOR then
      True
    elsif (Guidance_Kind = Direct_Fixed or Guidance_Kind = Direct_Moving) then
      (Current_Location() - Destination >= Capture_Radius)
    else False
    end if =>
    Actual_Start_Time := Never;
    Error_Signals.Heading_Error( Current_Heading_Error() );;


-- Behaviors to start and stop guidance:

-- Start if explicitly started and begin outputting heading errors
-- within Max_Start_Time, unless interrupted by another start event:
Mission_Objectives.Start_Guidance( Guidance_Kind, ?D ) =>
    if Guidance_Kind = VOR then
        Radial := ?D.Radial;
```

```
    elsif Guidance_Kind = Direct_Fixed or Guidance_Kind = Direct_Moving then
        Destination := ?D.Destination;
    end if
    if Guidance_Kind = Direct_Moving then
        Destination_Speed := ?D.Speed;
        Destination_Heading := ?D.Heading;
    end if
    Actual_Start_Time := Clock.Now() + Start_Duration();;

Clock.Now() = Actual_Start_Time =>
    Mission_Objectives.Guidance_Started( Guidance_Kind );
    State.Started := True;;

-- Start when arrive at start condition:
Aircraft_State.Aircraft_State_Vector( ?V ) where
    ?V.Quality = Usable and State.Armed and
    if Guidance_Kind = VOR then
      (Current_Radial() = Arm_Radial)
    elsif (Guidance_Kind = Direct_Fixed or Guidance_Kind = Direct_Moving) then
      (Current_Location() - Arm_Capture_Point < Capture_Radius)
    else False
    end if =>
        State.Armed := False;
        if Guidance_Kind = VOR then
           Mission_Objectives.Start_Guidance( Guidance_Kind, Arm_Radial );
        elsif (Guidance_Kind = Direct_Fixed or Guidance_Kind = Direct_Moving) then
           Mission_Objectives.Start_Guidance( Guidance_Kind, Arm_Destination );
        end if;;

-- Stop within Max_Stop_Time, regardless of additional Stop_Guidance events:
Mission_Objectives.Stop_Guidance( Guidance_Kind ) =>
    Actual_Stop_Time := Clock.Now() + Stop_Duration();;

Clock.Now() = Actual_Stop_Time =>
    Actual_Stop_Time := Never;
    Mission_Objectives.Guidance_Stopped( Guidance_Kind );
    State.Started := False;;

-- Stop immediately if Aircraft State Vector is unusable or if another guidance mode is
-- started or, if direct-fixed or direct-moving, aircraft has entered the capture radius:
( Aircraft_State.Aircraft_State_Vector( ?V ) where ?V.Quality = Unusable or
    ( if (Guidance_Kind = Direct_Fixed or Guidance_Kind = Direct_Moving) then
    ?V.Quality = Usable and State.Started and
        (Current_Location() - Destination < Capture_Radius)
    else False
    end if ) )
or Mission_Objectives.Guidance_Started( ?M ) where
      ?M /= Guidance_Kind and State.Started =>
      State.Started := False;;

-- Behaviors to arm and disarm guidance:

-- Arm within Max_Arm_Time, unless interrupted by another arm event:
Mission_Objectives.Arm_Guidance( Guidance_Kind, ?A ) =>
    if Guidance_Kind = VOR then
```

```
            Arm_Radial := ?A.Radial;
        elsif (Guidance_Kind = Direct_Fixed or Guidance_Kind = Direct_Moving) then
            Arm_Destination     := ?A.Destination;
            Arm_Capture_Point := ?A.Capture;
        end if
        Actual_Arm_Time := Clock.Now() + Arm_Duration();;

    Clock.Now() = Actual_Arm_Time =>
        Actual_Arm_Time := Never;
        Mission_Objectives.Guidance_Armed( Guidance_Kind );
        State.Armed := True;;

    Mission_Objectives.Disarm_Guidance( Guidance_Kind ) =>
        Actual_Disarm_Time := Clock.Now() + Disarm_Duration();;

    Clock.Now() = Actual_Disarm_Time =>
        Mission_Objectives.Guidance_Disarmed( Guidance_Kind );
        State.Armed := False;;


constraint

    not match (Tick where
        Clock.Now() > Actual_Start_Time and Actual_Start_Time /= Never or
        Clock.Now() > Actual_Stop_Time and Actual_Stop_Time /= Never or
        Clock.Now() > Actual_Arm_Time and Actual_Arm_Time /= Never);

end;



type VOR_Interface is interface
    include Guidance_Mode_Interface_Generator(VOR, Radial);

behavior

    begin

    -- Placeholder for saving VOR data and using in output of the heading errors:
    Aircraft_State.VOR_Relative_Navigation_Output => empty;;

end interface;



type Direct_Fixed_Point_Interface is interface
    include Guidance_Mode_Interface_Generator(Direct_Fixed, Destination);

end interface;



type Direct_Moving_Point_Interface is interface
    include Guidance_Mode_Interface_Generator(Direct_Moving, Destination);
```

```
behavior

    ?R : Lateral_Guidance_Revision_Data;

    begin

    -- Change state of destination:
    Mission_Objectives.Change_Guidance( Direct_Moving, ?R ) =>
        Destination := ?R.Destination;
        Destination_Speed := ?R.Speed;
        Destination_Heading := ?R.Heading;;

constraint

    -- Require change-guidance event to cause the following heading-error report:
    observe Mission_Objectives.Change_Guidance( Direct_Moving ) < Error_Signals.Heading_Error
    match   Mission_Objectives.Change_Guidance -> Error_Signals.Heading_Error;
end interface;
```

Whereas VOR guidance uses the complete Aircraft_State while Direct Fixed-Point and Moving-Point modes need only the Aircraft_State_Vector from Aircraft_State_Service, for further simplicity Guidance_Mode_Interface uses the complete Aircraft_State for all three modes.

Bibliography

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

[Fid91] Colin J. Fidge. Logical time in distributed systems. *Computer*, 24(8):28–33, August 1991.

[Hsi92] Alexander Hsieh. Rapide-0.2 examples. Technical Report CSL–TR–92–510, Computer Systems Lab, Stanford University, February 1992.

[LV93] David C. Luckham and James Vera. Event based concepts and language for system architecture. In *Proceedings of the Workshop on Studies of Software Design*, May 1993.

[LVB+93] David C. Luckham, James Vera, Doug Bryan, Larry Augustin, and Frank Belz. Partial orderings of event sets and their application to prototyping concurrent, timed systems. *Journal of Systems and Software*, 21(3):253–265, June 1993.

[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.

[SM91] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. Technical Report 215/91, University of Kaiserslautern, November 1991.

[TC93] W. Tracz and L. Coglianese. An adaptable software architecture for integrated avionics. In *Proceedings of NAECON 93*, pages 1161–1168, Dayton, Ohio, May 1993. IEEE.

[Tea93a] Rapide Design Team. *The Rapide-1 Executable Language Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, March 1993.

[Tea93b] Rapide Design Team. *The Rapide-1 Types Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, March 1993.