Technical Report
CMU/SEI-91-TR-20
ESD-91-TR-20

Design Specifications for Adaptive Real-Time Systems

Randall W. Lichota
Alice H. Muntz

December 1991

Technical Report
CMU/SEI-91-TR-20
ESD-91-TR-20
December 1991

Design Specifications for Adaptive Real-Time Systems

Randall W. Lichota

Hughes Aircraft Company, Ground Systems Group

Alice H. Muntz

Hughes Aircraft Company, Space and Communication Group

Distributed Systems Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Table of Contents

List of Figures

Design Specifications for Adaptive Real-Time Systems

Abstract: The design specification method described in this report treats a software architecture as a set of runtime entities, including tasks and external input/output elements, which interact either via messages or shared data structures. Tasks have a single thread of execution and represent program units that may be executed concurrently. External input elements produce input requests that in turn trigger a set of low-level activities to be executed by tasks. External output elements consume results that are produced by tasks. The specification method discussed here facilitates the description of the dynamic structure of runtime entities, the synchronization and communication between these entities, and their resource consumption and production properties (which include timing and sizing).

1 Introduction

The Specification Methodology for Adaptive Real-Time Systems (SMARTS) was developed by analyzing the computational characteristics of advanced radar systems [3]. In SMARTS, a software architecture is specified in three parts: the data model, the control flow model, and the sequential program model. The data model is expressed with a restricted data graph. The control flow model is expressed with a task graph in which each vertex represents a runtime entity, i.e., a task or an input/output data element, and each arc represents the data production/consumption relations between two communicating entities. The sequential program model is expressed with a restricted activity graph which is a subgraph of an activity graph representing the requirements specification [7]. Graphical notations are used for representing relations between data items in a data model, relations between two communicating entities and the temporal/spatial parallelism of inputs/outputs of each entity in a control flow model, and the internal computational structure of a sequential program model (i.e., a task). Textual notations are used for specifying timing constraints, pre-conditions, and post-conditions.

We are currently extending the Durra tool suite to support the specification and validation of software architectures using these notations. Durra itself is a non-procedural, task-description language specifically designed to support the development of large-grained distributed applications [2]. A task-level application description prescribes a way to manage system resources and includes behavioral and structural descriptions of the tasks, their mapping to processors, and their communication characteristics. Expressing a software architecture in Durra is reasonably straightforward because the semantics defined by our models map reasonably well to those defined by Durra. Moreover, the Durra runtime system supports the construction of distributed Ada programs, thus providing a mechanism for prototyping applications for a distributed environment.

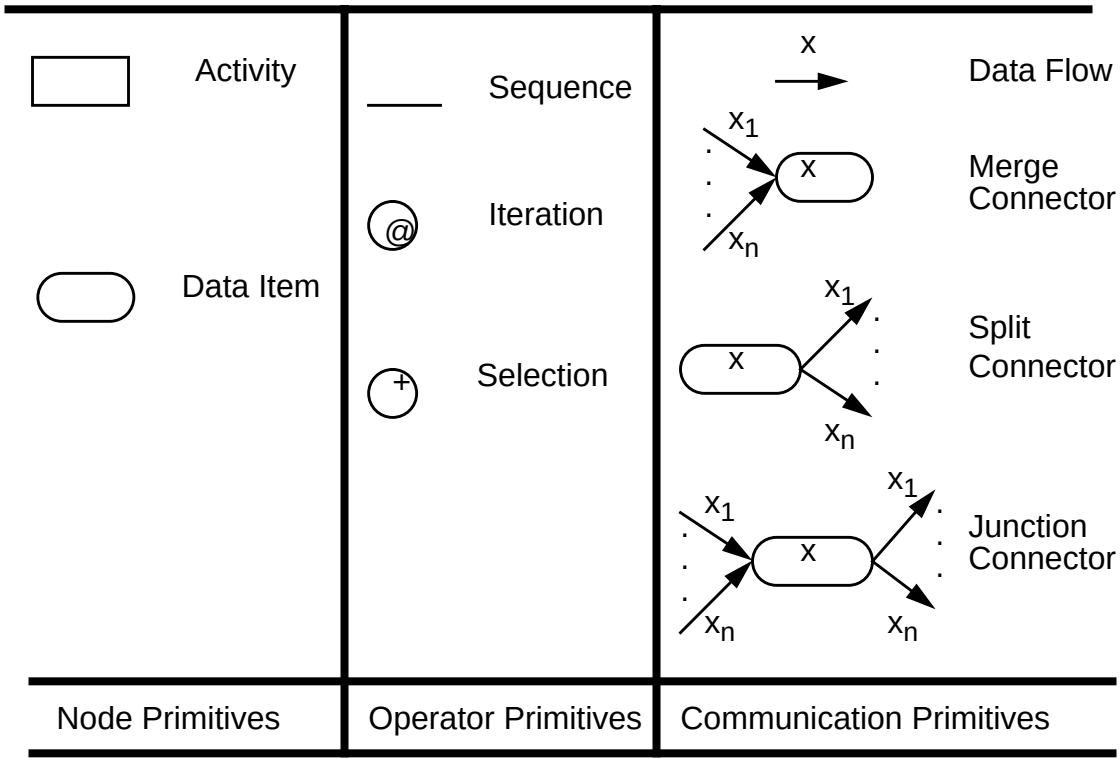

Figure 1-1 Graphical Notations for Restricted Activity/Data Graphs

2 Graphical Primitives for Specifying Design

The graphical notations presented in this section are for specification of restricted data graphs, restricted activity graphs, and task graphs. Figure 1-1 contains the graphical primitives for specification of restricted data/activity graphs. The node and operator primitives have been adopted from Alford's graph model [1] and the communication primitives have been added to represent interactions between activities. Node primitives include graphical representations for both activities and data items. Operator primitives are used to express sequential relationships among activities or data items. The sequence operator is used to express an order relation with respect to time. A pair of iteration operators encompassing a set of activities (or data items) can be used to represent either repetitive occurrences or to indicate periodic invocation of activities (or consumption of data items). A pair of selection operators encompassing a set of activities (or data items) specifies their mutual exclusive occurrence.

A data flow consists of a unidirectional arc and a label (indicated by 'X' in Figure 1-1). This arc represents one-to-one communication, the activity connected to the tail of the arc being the producer of X while the activity at the head being its consumer. The label attached to an arc may denote either a composite or an atomic data item. When the latter is the case, it is not necessary for the producer or consumer activities to be atomic. However, if both the consumer and producer activities are atomic then the data item denoted by the label must also be atomic.

A merge connector is composed of n labeled fan-in arcs x_1, ..., x_n, where n > 0 and a composite data item x has component data items: x_1, x_2, ..., and x_n. For each labeled arc x_i, there must be an activity at the opposite end of the data item x to write or access the component data item x_i. Write behavior is denoted by associating a unidirectional arc with x_i, placing an activity at the tail of the arc, and the data item x at the head. Access behavior of x_i is denoted by associating a bi-directional arc with x_i. The spatial and temporal parallelism and sequencing of data items x_1, ..., x_n are represented by the data graph associated with x.

A split connector is composed of n labeled fan-out arcs x_1, ..., x_n and a composite data item x with components: x_1, x_2, ..., and x_n. For each labeled arc x_i, there must be an activity to read or access the component data item x_i. Read behavior is denoted by associating a unidirectional arc with x_i, placing an activity at the head of the arc, and the data item x at the tail. Access behavior is denoted by associating a bi-directional arc with x_i. The spatial and temporal parallelism and sequencing of data items x_1, ..., x_n are represented by the data graph associated with x.

For both the split connector and the merge connector, the fan-in arc is unlabeled when n is equal to 1 since x and its component item x_1 are identical. The single arc merge (or split) connector identifies x as an input to the activity if the latter is at the head of the unidirectional arc. If the activity is at the head of the unidirectional arc, the single arc merge (or split) connector defines x as an output to the activity. Finally, if the activity is connected with x via a bi-directional arc then x is considered to represent a buffer.

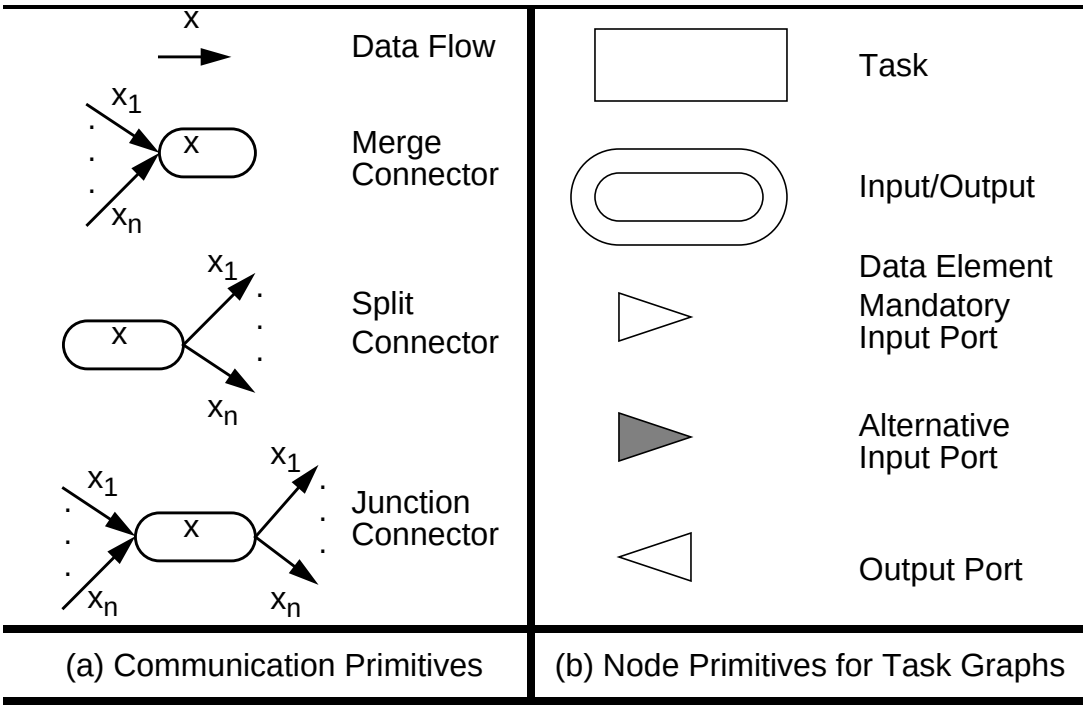

Figure 2-1 Graphical Notations for Task Graphs

A junction connector is composed of n labeled fan-in arcs (x_1, ..., x_n), a composite data item x (with component data items x_1, x_2, ..., and x_n), and n labeled fan-out arcs (x_1, ..., x_n). For each fan-in labeled arc x_i, there must be an activity to write or access the component data item x_i. For each fan-out labeled arc x_i, there must be an activity to read or access the component data item x_i. Similarly, the spatial and temporal parallelism and sequencing of data items x_1, ..., x_n are represented by the data graph associated with x.

A task graph is defined as a directed graph whose nodes represent tasks or data elements. An arc between two nodes represents a data flow, and the direction of the data flow corresponds to that of the arc. A channel is an interface program unit which implements the communication primitive connecting two or more vertices in a task graph representing a task or a data element. A task is described by a task template which includes the following types of information: ports—its interfaces to other tasks, external devices, and to the runtime system; attributes—importance level, state variables and their properties; behavior—its associated restricted activity graphs, which include functional specifications (e.g., pre- and post- condition formulas) and timing constraints (e.g., state-dependent execution time); and structure—its component connections. We have chosen to represent individual task templates using the Durra language [2]. During execution time, instances of tasks may run on separate processors and may communicate with each other according to a specific set of data flows, which may be implemented as messages. Task templates serve as components (along with maximum restricted activity graphs, data graphs, and sequential program models) of a design specification.

In our method, a runtime entity is a set of activities comprising a single execution thread. This set of activities is encapsulated by the task primitive in Figure 2-1, and the interacting behaviors of the encapsulated activities are defined using the communication primitives in Figure 2-1. For each fan-in arc labeled x_i of either the merge connector or the junction connector, there must be a task at the opposite end of the data item x to write or access data item x_i. For each fan-out arc labeled x_i of the split connector or the junction connector, there must be a task at the opposite end of data item x to read or access the component data item x_i.

The node primitives set in Figure 2-1(b) contains five elements: task, input/output data element, mandatory input port, alternative input port, and output port. A task represents a program unit that has a single execution thread and consumes a bounded amount of resources, such as processor cycles and memory capacities. Unlike data items which are passive entities, input/output data elements are active entities that do not consume processor resources. An input data element only has a set of output ports. Each input data element represents a source of instances of a specific set of input data items. An antenna receiver, for instance, can be represented as an input data element. An output data element only has a set of input ports. Each output data element represents the repository of instances of a specific set of output data items. An example of an output data element is a display device such as a color monitor.

The input and output ports define the interface of a task. Data arriving at an input port serves to enable the activities that define the task's behavior. A mandatory input port represents a logical location to which data must be directed in order for the task to be executed. By contrast, the failure of data to arrive at an alternative input port need not necessarily block the progress of the task. Rather, this corresponds to one possible branch through the task's restricted activity graph; the arrival of data is only required when it serves to enable an activity that has *arrived* [7]. An output port represents a logical location through which outputs are to be delivered upon execution of a given activity.

It should be noted that activity and data graphs are qualified by the term *restricted* partly to distinguish them from the graphs used to define software requirements. In SMARTS, the transition from requirements to design consists of partitioning an activity graph into a set of restricted activity graphs [5]. When one's design objective is to minimize overhead due to task context switch, one or more of these restricted activity graphs should be mapped to a task graph to minimize the number of tasks. In this case, the notion of a "maximal restricted activity graph" is useful for defining the criteria for deriving a task graph from a given activity graph. The algorithm for deriving a set of maximal restricted activity graphs from a given activity graph is described in [8].

3 Textual Specifications

The graphical representations presented in the preceding section are intended to aid a software engineer in visualizing the structural, behavioral, and functional aspects of a software design. In addition to supporting design specification, however, these representations are intended to serve as input to an Ada source code generator. To facilitate the construction of such a tool, we have defined textual grammar for the restricted activity and data graph models. Textual representations of these models are combined to form a complete behavioral description. A restricted activity graph may be referenced in the **behavior** part of a Durra task description. The name of the specification is “ACTIVITY” and is followed by the value of the specification proper. To avoid obscuring the readability of Durra task descriptions, the specification value is constrained to be a string literal representing a file name. However, textual representations of resource definitions, restricted data graphs, and type definitions are required to be placed in files that are not directly tied to a Durra task description. By convention, type definitions are placed in a file named “.TYPES”, data graph descriptions are placed in a file named “.DATA_ITEMS”, and resource definitions are placed in a file named “.RESOURCES”.

Syntax

Restricted Activity Graph Specification::=
ACTIVITY **=”** StringLiteral

Type Definitions ::= **‘**TypeDefList

Resource Definitions ::= **‘**ResourceDefList

Restricted Data Graph Specification::=
‘DataItemList

Example

behavior `activity = “/usr/rwl/sensor.g”;`

Meaning

The value of a restricted activity graph specification is a file name written as a string literal (i.e., a sequence of characters enclosed in double quotation marks “ “ “ “).

3.1 Notes on Syntax

The syntax of our language is described using standard Backus-Naur-Form (BNF), with the following conventions:

- Terminal symbols are enclosed in quotation marks (“and”), but the quotation marks do not belong to the terminal.
- No distinction is made between upper and lower case letters in terminals and non-terminals.

- Vertical bars (“|”) separate alternative productions. Braces (‘{“ and “}”) indicate optional components of a production.
- Comments start with a double hyphen (“—”). Any text between the double hyphen and the end of the line is ignored.
- Identifiers are sequences of letters, digits, and underscores (“_”). An identifier must begin with a letter and end with a letter or a digit. Consecutive underscores in the middle of an identifier are not allowed.
- Strings are sequences of ASCII printable characters, enclosed in double quotation marks (“). A double quotation mark inside a string must be written as two consecutive double quotation marks:
  “A string with a double quotation mark, “”, inside.”
- Integer and real numbers are always decimal, i.e., base 10. A real number can terminate with a period (“.”) without a fractional part.

3.2 Keywords and Predefined Identifiers

Keywords are highlighted in normal text by writing them in **bold face**. Predefined identifiers are highlighted in normal text by writing them inside quotation “marks.”

The following words are keywords in the language: **active, activity, and, behavior, concurrent, constraint, data, else, end, ensure, exclusive, first, float, for, is, item, last, level, local, loop, matrix, not, of, or, port, positive, preemption, range, repeat, require, requirements, resource, select, schedule, shared, state, stream, then, times, timing, uses, variables, vector, while.**

The identifiers “Boolean,” “Digits,” “False,” “Float,” “Integer,” “Natural,” and “True” are predefined in the language.

4 Specification

A restricted activity graph specification consists of five separate parts: a type definition portion, a resource definition portion, a list of data item definitions, a list of activity definitions, and a port mapping list. The type and resource definition parts are optional; each of the other parts must be specified.

Syntax

Activity ::= ActivityList TypeMappingList

Example

Note that this example is intended to illustrate what a specification might look like and doesn't represent an actual system.

Type Definitions

```
Mode Is (Planning, Tracking);
Boundary_Value Is Digits 7;
Angle Is Digits 5;
Distance Is Digits 6 Range 0.0..5000.0;
```

Resource Definitions

```
VAXCPU Is Shared(1..1000);
Data Item Msg Is
   Select
      When B1 Active
         Data Item Search Is
            Concurrent
               Data Item Azimuth Is
                  Angle
               End Data Item;
               Data Item Elevation
                  Distance
               End Data Item;
            End Concurrent;
         End Data Item;
      When B2 Active
         Data Item Front Is
            Concurrent
               Data Item Number Is
                  Natural
               End Data Item;
               Data Item Resource_Bound Is
                  Boundary_Value
               End Data Item;
               Data Item Importance_Level Is
                  Natural
               End Data Item;
            End Concurrent;
```

```
            End Data Item;
      End Select;
End Data Item;
Activity Planner Is
      Select
         Activity Front_End_Planning(Front) Is
            State Variables
               Planning_Directive : Directive;
               System_Mode : Mode Is Shared;
            Require System_Mode = Planning;
            Ensure True; Preemption True;
            Importance Level 7;
            Timing Constraint e < Front.Number * 0.02;
            Resource Requirements VAXCPU = 18;
            Behavior
               Planning_Directive := Generate_Plan();
         End Activity;
         Activity Search_Frame_Planning(Search) Is
            State Variables
               Frame_Size : Natural;
               System_Mode : Mode Is Shared;
            Require System_Mode = Planning;
            Ensure True; Preemption True;
            Importance Level 5;
            Timing Constraint e < Frame_Size * 0.01;
            Resource Requirements VAXCPU = 10;
            Behavior
               Frame_Size := DeriveSize(Search.Azimuth,
                                        Search.Elevation);
               SA_act := Generate_SA_Request(Frame_Size);
         End Activity;
      End Select;
End Activity;
```

4.1 Literal and Non-Literal Values

Syntax

BooleanLiteral	::=	“True” | “False”
IntegerLiteral	::=	{“+” | “-” } sequence_of_digits
RealLiteral	::=	{“+” | “-”} sequence_of_digits {“.” { sequence_of_digits} }
BooleanValue	::=	BooleanLiteral | FunctionCall | VariableRef | DataItemRef

IntegerValue	::=	IntegerLiteral | FunctionCall | VariableRef | DataItemRef
RealValue	::=	RealLiteral | FunctionCall | VariableRef | DataItemRef
Value	::=	BooleanValue | IntegerValue | RealValue | “(” ArithExpr “)”

Meaning

Literals denote constants of the appropriate type. An IntegerLiteral is a non-empty sequence of digits. A RealLiteral is also a non-empty sequence of digits with an embedded decimal point. Each of the non-terminals BooleanValue, IntegerValue, and RealValue stands for literals (constants) of the appropriate kind, calls to functions returning values of the appropriate kind, or references to variables or data items of the appropriate kind.

4.2 Expressions

Syntax

Expression	::=	{ **NOT** } Term { **OR** Expression }
Term	::=	BooleanValue | Relation { A**ND** Term }
Relation	::=	ArithExpr { RelOp ArithExpr }
RelOp	::=	“=” | “/=” | “>” | “>=” | “<” | “<=”
ArithExpr	::=	{ “+” | “-” } Factor { “+” | “-” Factor}
Factor	::=	Value { “*” | “/” Value }
IntegerRange	::=	{ “+” | “-” } IntegerLiteral “..” { “+” | “-” } IntegerLiteral

Meaning

Expressions are used to specify conditions and values used within Activity and Data Item definitions.

4.3 Type Definitions

Syntax

TypeDef	::=	IntegerTypeDef | RealTypeDef | EnumerationTypeDef

IntegerTypeDef ::= IntegerTypeName
IS “INTEGER”
{**RANGE** IntegerRange}

IntegerTypeName ::= Identifier

RealTypeDef ::= RealTypeName
IS “DIGITS” PositiveValue
{**RANGE** IntegerRange}

RealTypeName ::= Identifier

EnumerationTypeDef ::= EnumerationTypeName
IS “(“ IdentifierList “)”

EnumerationTypeName::= Identifier

Meaning

Each atomic data item must be associated with some abstract data type. Type definitions provide a means for defining application-specific data types based on a subset of those which are defined by the Ada language.

4.4 Activity Definition

Syntax

ActivityDef ::= **ACTIVITY** ActivityName
{“(“ InputDataItemList “)”}
IS ActivityBody
END ACTIVITY “;”

InputDataItemList ::= DataItemName {“,” InputDataItemList}

ActivityBody ::= ActivityList |
ActivityIteration |
ActivitySelection |
AtomicActivityDef |
ActivitySchedule

ActivityIteration ::= **REPEAT** ArithExpr
TIMES ActivityList
END REPEAT “;”

ActivitySchedule ::= **FOR** Relation
SCHEDULE ActivityList
END FOR “;”

ActivitySelection ::= **SELECT** ActivityList **END SELECT** “;”

ActivityName ::= Identifier

ActivityList ::= ActivityDef {ActivityList}

AtomicActivityDef ::= {**STATE VARIABLES** StateVariableList}
{**LOCAL VARIABLES** LocalVariables}
REQUIRE Expression “;”
ENSURE Expression “;”

		PREEMPTION Expression “;” **IMPORTANCE LEVEL** ArithExpr “;” **TIMING CONSTRAINT** Expression “;” **RESOURCE REQUIREMENTS** Expression “;” **BEHAVIOR** StatementList
StateVariableList	::=	StateVariableName “:” VariableType {**IS SHARED**} “;” {StateVariableList}
StateVariableName	::=	Identifier
IntegerList	::=	IntegerValue {“,” IntegerList}
LocalVariables	::=	LocalVariableName “:” VariableType “;” {LocalVariables}
LocalVariableName	::=	Identifier

Meaning

This part of the BNF provides a textual analogue to the restricted activity graph (RAG). Productions are provided for each of the nodes in a RAG: iteration, selection, sequence (the latter is expressed as ActivityList), activity (allows recursive definitions), and atomic activity. Note that for the iteration node two forms of iteration are supported (ActivityIteration and ActivitySchedule), allowing one to specify either a sequence of invocations or an invocation rate. Also note that an activity’s parameters denote data items referenced as input to the activity. In lieu of “out parameters,” data items are directly assigned values in an (atomic) activity’s behavior section. The reason for this asymmetry is that the input parameters are needed to identify the data items that form part of an atomic activity’s enabling condition.

State variables provide a means for defining mode-dependent operation and for specifying adaptive behavior. The scope of variables names is considered, with one exception, to be restricted to the atomic activity in which it is declared and any encompassing activity. The exception occurs when a state variable is qualified as *shared*. Shared state variables with the same name are considered to denote a single variable for tasks within the same clusters [2]. State variables that have the same name but are not denoted as shared or that are defined in atomic activities associated with tasks in different clusters are considered to be separate, distinct variables.

In addition to state variables, it is possible to define a set of variables that are local to a specific code body. These are intended only to serve as a means for denoting the intermediate results of an atomic activity’s computations and not as a means for holding state information.

4.5 Data Item Definition

Syntax

DataItemDef	::=	**DATA ITEM** DataItemName **IS** DataItemBody **END DATA ITEM** ";"
DataItemName	::=	Identifier
DataItemBody	::=	DataItemList | DataItemIteration | DataItemSelection | DataItemConcurrency | AtomicDataItemDef | DataItemSchedule
DataItemSchedule	::=	**FOR** Relation **SCHEDULE** DataItemList **END FOR** ";"
DataItemIteration	::=	**REPEAT** ArithExpr **TIMES** DataItemList **END REPEAT** ";"
DataItemList	::=	DataItemDef {DataItemList}
DataItemName	::=	Identifier
DataItemConcurrency	::=	**CONCURRENT** DataItemList **END CONCURRENT** ";"
DataItemSelection	::=	**SELECT** DataItemSelectionClause **END SELECT** ";"
DataItemSelectionClause	::=	**WHEN** ActivityName **ACTIVE** DataItemDef {DataItemSelectionClause}
AtomicDataItemDef	::=	ScalarType | StreamType

Meaning

This part of the BNF defines the restricted data graph (RDG) which is supplied along with a RAG. The two graphs resemble each other in having many of the same types of nodes (e.g., iteration, sequence, selection). The RDG is somewhat less "restrictive" than the RAG in that it includes a concurrency operator (to support specification of parallel data streams). This may be considered a refinement in that it was not originally defined as part of the RDG operators in earlier versions of SMARTS. Generally speaking, an RAG and RDG serve to complement one another, with the parameters specified for an activity corresponding to some set of data items.

4.6 Statements

Syntax

StatementList	::=	Statement “;” {StatementList}
Statement	::=	Assignment | Conditional | Iteration | While
Assignment	::=	DataItemRef | VariableRef “:=” ArithExpr
DataItemRef	::=	DataItemName {“.” DataItemRef}
VariableRef	::=	StateVariableRef | LocalVariableRef
StateVariableRef	::=	StateVariableName {**“[“ Identifier “]” |** “[“ Identifier “,” Identifier “]”}
LocalVariableRef	::=	LocalVariableName {**“[“ Identifier “]” |** “[“ Identifier “,” Identifier “]”}
Iteration	::=	**FORALL** IdentifierList **“:”** StatementList **END FORALL**
While	::=	**WHILE** Relation **LOOP** StatementList **END LOOP**
Conditional	::=	**IF** Relation **THEN** Statement { **ELSE** Statement} **END IF**
FunctionCall	::=	Identifier “(“ {ExpressionList} “)”
ExpressionList	::=	Expression | Expression “,” {ExpressionList}

Meaning

This section of the BNF is intended to be used for expressing the behavior of an atomic activity. Generally speaking, this may be considered as a form of Program Description Language (PDL) that includes a simplified version of many of the basic Ada language statements. The most notable deviation from the normal Ada practice is in the use of matrix/vector subscripts. When employed within the context of an iteration statement (FORALL), each element of the matrix or vector will be referenced. For example, consider a matrix A and two vectors B and C. The statement “`forall i,j: A[i,j] := B[i] + C[j] end forall;`” means that the element at the ith row and jth column of A is equal to the sum of the ith element of B and the jth element of C.

The remaining kinds of statements follow the Ada conventions. A WHILE statement specifies a sequence of statements to be repeated while a user specified condition specified by a relational expression is true. The conditional statement lists a sequence of statements to be executed if a specified relational expression is true. An optional ELSE clause specifies an alternative sequence of statements to be executed if the expression is false.

A means for specifying calls to external procedures has been provided primarily to facilitate interfacing with reuse libraries.

4.7 Type References

Syntax

VariableType	::=	ScalarType | VectorType | MatrixType
ScalarType	::=	IntegerType | RealType | EnumerationType
IntegerType	::=	IntegerTypeName | “NATURAL”
RealType	::=	RealTypeName | “FLOAT”
EnumerationType	::=	”BOOLEAN” | EnumerationTypeName
StreamType	::=	**STREAM** “(“ IntegerRange “)” **OF** ScalarType
MatrixType	::=	**MATRIX** “(“ IntegerRange “,” IntegerRange “)” **OF** ScalarType
VectorType	::=	**VECTOR** “(“ IntegerRange “)” **OF** ScalarType
TypeDefList	::=	TypeDef “;” {TypeDefList}

Meaning

Data items may be defined as either scalar types (e.g., integer or float) or as streams. For example, a sequence of data items denotes the arrival of the constituent items in the specified order (e.g., from a sensor). Similarly, if a set of data items were specified in parallel it would mean that they all would be assessable at the same time (this is a case where some form of automated consistency checking could be used to ensure this property would actually hold for a given specification).

By defining a data item as a stream or scalar, data requirements may be defined on the basis of characteristics of the target environment. State variables of an atomic activity must be de-

clared using one of two alternative data types (matrix and vector) in lieu of the stream. This is more in spirit with the expected usage of state variables (i.e., to form part of timing, resource requirement, and functional equations). Many of the kinds of activities which will likely be implemented will probably make extensive use of matrix and vector computations (e.g., Kalman Filtering).

4.8 Resource Definitions

Syntax

ResourceDefList	::=	ResourceDef ";" {ResourceDefList}
ResourceDef	::=	ResourceName **IS** ResourceUsage
ResourceName	::=	Identifier
ResourceUsage	::=	**EXCLUSIVE** | **SHARED** "(" IntegerRange ")"

Meaning

Resource definitions provide a means for identifying the kinds of resources that exist within a system and quantifying how they can be utilized. Each resource name is associated with an integer range that defines the units in which it may be utilized. In addition, a resource may be defined to be exclusive (meaning it may be utilized by only one atomic activity during a specified period of time) or shared. While resource definitions can obviously be associated with processors and memory buffers, these definitions represent abstractions and are not dependent on a given system architecture for their meaning (i.e., there are no "predefined" resources).

4.9 Type Mappings

Syntax

TypeMapping	::=	DataItemName **USES** TypeName
TypeMappingList	::=	TypeMapping ";" {TypeMappingList}

Meaning

A software architecture is expressed as a task graph in which tasks are interconnected via channels. Since the latest version of the Durra language has incorporated elements of the task graph formalism, it has been used to represent much of the task graph notation originally defined by SMARTS. The additional notation which is defined here is intended to provide a means for mapping data items to messages that are transferred between Durra ports. In Durra, each port is associated with a named type which defines the size of messages that may be transferred through the port. Message size may be specified as a fixed number of bits, a range, an indefinite length, or a set of alternative lengths. Types that are declared as a sequence of bits of fixed or variable length are known as scalar types. In addition, Durra permits types to

be declared as a union of previously declared types. In this latter case, messages passed through a port of this type could be one of any of its member types.

In the case of scalar types, a one-to-one mapping must be established between the type and a specific data item. Union types inherit the mappings established for their member type. Thus while a union type need not be given an explicit mapping, a mapping must be defined for each of its member types. With regards to scalar types, the data storage sizes implied by the data item's type must match the explicit size defined in the type declaration. The implied storage size is not defined here as it is implementation dependent.

Note that the non-terminal "TypeName" is defined by the Durra language. It thus serves as a unifying element which is shared between Durra and the restricted activity/data graph languages. Finally, it should be noted that the term "type" as used within the context of Durra has a different meaning from the usage typical in programming language circles. The Durra reference manual contains information on how types may be declared.

We should note that this language does not provide an equivalent of the *Get_Port* and *Send_Port* operations used by the Durra runtime interface to implement port input and output operations. The reason is that these are considered low-level operations, which is implied by the enabling conditions of a restricted activity graph.

5 Implementation Strategies

Restricted activity and data graphs may be used to generate an Ada implementation of a Durra task as a set of subprograms. By importing a package that implements various auxiliary types and subprograms, including the Durra communication procedure and additional support packages, the procedure could then be compiled and linked by a suitable Ada development system and stored in the Durra task implementation library. Similarly, a task graph may be expressed as a Durra application description from which the Durra compiler can generate an Ada package body for each cluster[1] defined in the application. The package body imports implementations of Durra tasks and channels, creates Ada tasks to serve as threads for the implementations, and includes code to evaluate reconfiguration conditions (if any are specified) and modify the cluster configuration as required. The package body also defines a set of tables, common to all clusters in the application, that describe the complete application structure. A hardware configuration table, which defines the environment in which the distributed application will run, may also be included. As part of the support environment, a set of runtime support packages are provided for use by all clusters. These support packages, the cluster-specific package body, and a driver subprogram are combined into a single Ada program that implements the cluster.

The basic premise underlying the code generation process is that the control and structural information provided by a set of task/activity graphs can guide the selection of implementation decisions. By rigorously defining the behavior of each activity, we are able to generate a significant portion of an implementation. This is because the task/activity graph topology can be used to identify many of the control and data structures as well as providing a basis for defining Ada packages. The actual Ada structures are derived from the topology of the data graph, the data item type definitions, and the resource requirement attributes. Each operator in the data graph represents a distinct time-ordered availability of the data items to which it is applied and suggests a distinct mapping into a particular form of Ada data types and objects. These mappings along with their rationale are described in [5].

One of the principle difficulties faced in code generation is the selection of an appropriate program element naming convention. Often times the software produced by a general-purpose code generator can be difficult to maintain due to the lack of coherent type, variable, and subprogram names. While our formal method of specification can significantly reduce this problem, it cannot eliminate it since the relevancy of a name can depend on context. Moreover, the descriptive names found in task, activity, and data graphs represent a limited source of information for denoting elements within an Ada program. For instance, when a for-loop statement is derived from an activity graph, information might not be available for naming the identifier in the loop parameter specification. In other instances, the length of the names derived from

1. The Durra language allows one to group tasks together to comprise a set of clusters. In the general case, scheduling would be conducted at two levels with a cluster scheduler operating in concert with a set of task schedulers, each assigned to a separate cluster.This report considers the case where only one cluster is specified per processor.

graph annotations may cause the generated software to appear cumbersome and difficult to read. To avoid these problems, we plan to provide standard default naming schemes which can be interactively overridden during the generation process. For example, the default name for an identifier in a loop parameter specification would be *Index,* with a number appended in order to establish uniqueness.

The Ada language, as currently defined, provides a programmer with the choice of assigning a task a fixed priority or leaving it be undefined. Selective use of fixed and undefined priorities form the basis for strategies such as priority inheritance, which is intended to ensure predictability [4]. By assigning a task a fixed priority, however, one is forced to permit preemption in the general case. As a consequence, a programmer has only a limited set of language features through which to control scheduling of activities based on its preemptability, importance level, timing constraint, and resource requirements. Our experience has led us to conclude that except for the simplest cases, it would be very difficult to construct a software implementation from a design specification using the existing language features.

One property of atomic activities which can be particularly difficult to implement in software is the activity precondition. In a programming language such as Ada, this would most likely be implemented using an availability control to prevent task synchronization from occurring until the precondition is satisfied [4]. Unfortunately, the straightforward solution of using a selective wait usually leads to a polling situation with unacceptable overhead. This problem, along with those mentioned in the previous paragraph, have led to a set of proposed extensions to Ada described in the following section.

5.1 Extensions to Ada

In considering how Ada might be extended to better support the implementation of a software architecture, we will follow the classification scheme described in [4] for examining specific language constructs. In this **classification**, scheduling and decision-making capabilities of a language were defined as **race controls** and **availability controls**, respectively. Availability controls are only relevant to one of the properties of an atomic activity: the precondition. We believe that adding generalized guards to the Ada accept statement, as pointed out in [4], would be sufficient to implement activity enablement. By contrast, race controls can be used to implement three of the additional four properties (importance level, timing constraint, and resource requirements) of an atomic activity. We find that the remaining property, preemptability, is not addressed by either the availability or race controls. The remainder of this subsection examines the relationship between the race controls defined in [4] and the i**mportance level, resource requirements**, and **timing constraints** of an atomic activity.

The E&D comprehensive set of race controls is divided into **forerunner control, preference control,** and **dynamic priority control**. Forerunner control represents a means by which pending entry calls may be ordered according to a specified criteria. Within the context of SMARTS, this would be most applicable to the implementation of data managers [5]. By han-

Assumptions:

Activity A	Activity B
deadline = 9 msec.	deadline = 4 msec
importance level = 7	importance level = 3

Priority = (100/(deadline + 1)) + importance level

Current Clock Time = 0

Maximum Importance Level = 10

Resulting Priority of the Corresponding Ada Tasks:

Task A: 17

Task B: 23

Figure 5-1 Task Priority Determination

dling calls to a data manager via an accept ordered according to priority, one would be assured that transactions could be carried out in order of importance during periods of heavy loading.

Preference controls are used to specify the criteria for choosing between the alternatives within a select statement. This would, not surprisingly, appear to be most applicable to cases where a selection between multiple activities is specified. Although examples of activity selection in prior papers used preconditions that were both mutually exclusive and collectively complete, this need not necessarily be the case [6]. The syntax and semantics of activity graphs does not preclude the case where more than one precondition may be satisfied. In this circumstance, one of these activities would be selected based on their relative importance level. The preference control described in [4] would handle the case in which the **relative** importance of the constituent activities does not change.

Dynamic priority controls probably bear the most complex relationship with respect to the activity characteristics mentioned above. This is because SMARTS defines three separate characteristics (importance level, resource requirements, and timing constraints) that can influence the scheduling of an atomic activity. By contrast Ada provides a single mechanism, priority, which can be used to influence task scheduling. Thus in the most general case, a hierarchical resource management scheme is needed to determine task priority based on the characteristics of individual activities [6]. It is possible, however, that special cases might exist in which a resource management hierarchy might be unnecessary. For example, suppose that one was to use an Earliest Deadline First (EDF) scheduling strategy and that resource contention does not occur. In this case, a task's priority could be expressed as a function of its constituent activity's scheduling deadline and importance level. Figure 5-1 illustrates one way in which this function might be implemented. In this case, the scheduling deadline is the most significant determinant of priority; the importance level will affect relative priority only when activity scheduling deadlines fall within a small, predefined window. Notice that the priority is inversely proportional to the length of the scheduling deadline and the importance level affects the priority less significantly than the scheduling deadline.

5.2 Emulating Race Controls Within Durra

While the E &D comprehensive set of race controls offers promise for improving a programmer's control over Ada task scheduling, it will likely be several years before these language extensions are incorporated as part of the Ada 9x revision process. We assert, however, that a software architecture derived using SMARTS can be effectively implemented within the framework of Durra. The latter implements a **buffered message passing** model of interprocess communication. Durra channels may have a "bound" attribute whose value determines the size (in number of messages) of the buffer associated with each input port of the channel. When a task attempts to send data to a channel, the task will block (i.e., be suspended) if the buffer associated with the port is full. Conversely, a task attempting to read data from a channel will block if the buffer is empty. Setting the buffer bound to zero forces synchronous communication, since either task will block until the other is ready to send or receive data. For practical purposes, one can achieve asynchronous communication by setting the bound to a very large number.

In the following sections, we explain how the effect of individual race controls may be emulated using Durra in conjunction with a conventional Ada compiler. This requires us to recast the notion of forerunner, preference and dynamic priority controls within the context of restricted activity graphs, and the Durra tasks that serve to encapsulate them. As an example, consider Figure 5-2. Task A is declared to have two ports, A1 and A2, through which data items are transferred to their constituent activities AA, AB, and AC. Port A1 is defined to be of scalar type (a fixed or variable sequence of bits) that must match the data storage requirements of the data item (X) which passes through the port. In this case, Port A1 is connected to a channel which serves as a conduit for transferring instances of Data Item X to Task A. When multiple sources of X exist, a class of channel known as a *merge* may be used to transfer each to its destination (in this case Task A). We may apply the notion of a forerunner control within the context of Durra (rather than Ada) by noting that it is possible to implement a merge channel in a variety of ways. Possibly the simplest implementation would be for the channel to queue data items using a FIFO ordering scheme. Alternatively, a channel may determine queuing order based on the priority of the sending task or on the value of the data item (or in the case of a composite, one of its constituent data items).

5.2.1 Emulating Preference Control

An analogous form of preference control may be illustrated by the connection of Port A2 to the activities AB and AC in Figure 5-2. Port A2 is defined to be of union type, indicating that the data items which pass through it must correspond to one of its member types. In this example, Port A2 has two member types corresponding to the data items Y and Z. Let us assume that the execution of Task A had been blocked, waiting for the arrival of a message at Port A2. Now suppose a message is sent to Port A2. Task A will resume execution in accordance with its priority with respect to tasks within the same cluster. Once this has occurred, the message is obtained from Port A2, and its associated data item is checked to determine whether it is intended as input for activity AB or AC. This is then used to update the enabling condition of the

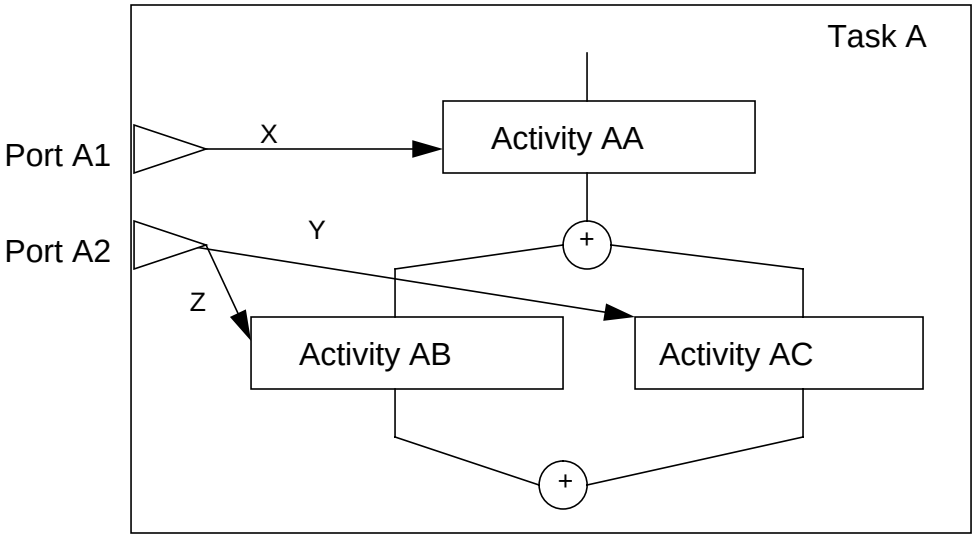

Figure 5-2 A Sample Task and its Restricted Activity Graph

respective activity. It should be noted that whether AB or AC is actually enabled for execution depends on properties of the associated activities (e.g., their preconditions) as well as the availability of input. In this context, preference control is much more restrictive becuase it is unlikely that all activities involved in a select will have their availability and preconditions satisfied at the same time.

The preference control specified above is a form of restricted availability control in which activity AB or AC is selected based on the appearance of its input data item at the port. Now suppose that Z is replaced by Y so both activities (activity AB and activity AC) the same data item Y as input. In this case, the behavior of the system when there is a data item available at the port depends on whether or not the precondition of activity AB or AC is true. If exactly one of the activities is found to have a precondition which evaluates to true, then that activity would be provided the data item. Otherwise, if more than one activity is found to have a precondition that is true, the activity with the greatest importance level should receive it. In the case where more than one of these activities have the same importance level, one is selected on the basis of its physical position within the specification. In the textual representation, this would be the top-most of the identified activities. In the graphical specification, it would be the left-most.

By combining precondition evaluation with the availability of input, we have introduced the use of private control. In effect, we would be choosing an alternative based both on consensus control (i.e., communication request through a port, in the form of a data item) and on the evaluation of a private control expression. In this case, the control expression would be evaluated first. Only those activities whose expressions evaluate to true would be allowed to participate in the selection. As stated previously, the highest priority activity whose input data items are available would be selected.

As shown in Figure 5-2, ports of union type (e.g., A2) are used to implement the activity selection operation so as to distinguish between inputs for the various activities. The use of a separate port to receive input for each activity within a selection (e.g., a separate port for X and for Y) is proscribed for the following reason. In Durra, a task would have to perform a “test port” operation for each to check on input availability, with a “get port” being invoked to obtain the message for each activity whose precondition was true. A problem occurs, however, in the case where none of the inputs are available. Because the order in which the inputs arrive is not usually predictable, it cannot generally be determined at which port the task should block waiting for input. On the other hand, a channel connected to a port of union type could order messages as they arrive according to their type. By relating type with the importance level of the activity that takes its associated data item as input, the presentation of input may be facilitated.

5.2.2 Emulating Dynamic Priorities

The Durra runtime environment establishes communications between clusters, starts and terminates Durra processes and links, transports data between Durra processes and links, evaluates reconfiguration conditions, and performs reconfigurations. The Durra runtime, however, has no responsibility for scheduling Durra processes and links other than starting and terminating them. Since they are implemented by Ada tasks, they are scheduled by the Ada runtime. Consequently, we must add an additional level of control in order to support dynamic task priorities. To ensure that task scheduling is carried out in accordance with the changes in importance level inherited via the encapsulated activities, we employ an additional task that determines the order in which application tasks are executed and thus serves as an application-level scheduler. While this task does not appear within a Durra application description, the latter task, along with its encapsulated restricted activity graphs, contains most of the necessary information needed to synthesize an application-level scheduler. For consistency with earlier papers, we refer to this task as a *global scheduler*.

In order to properly schedule the execution of application tasks, a

should be notified of the following conditions:

- A specific activity has arrived.
- All of the input data items are available for a given activity.
- The precondition associated with an activity has just become true.

A global scheduler would be notified of the first two types of conditions via reference to a special set of program variables which are shared with the activities it controls. The third condition would be determined by the global scheduler itself.As shown in Figure 5-3, a global scheduler is connected to each of its controlled tasks via a deal channel. By assigning the global scheduler the lowest possible priority, it can be guaranteed to execute only when all application tasks in its cluster have blocked. The global scheduler operates in a cyclic fashion. During each cycle the following actions occur:

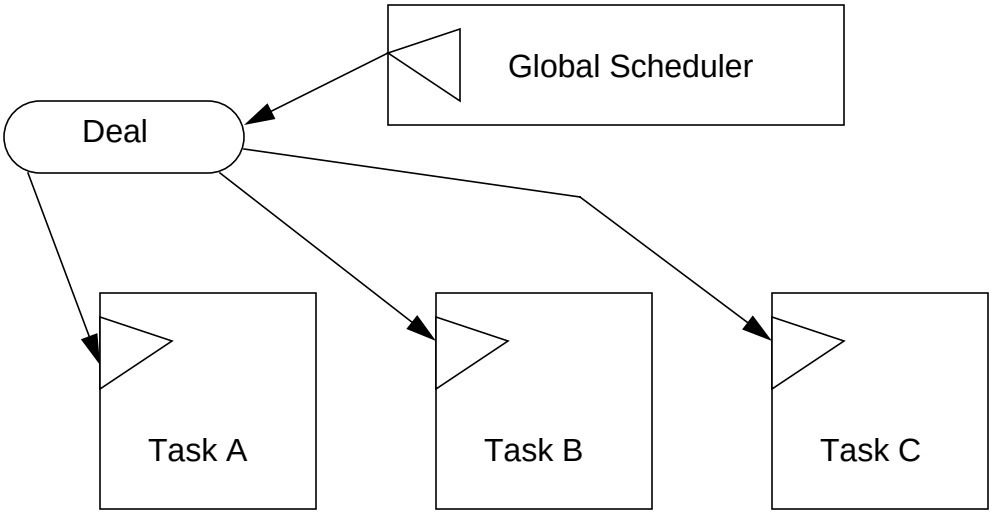

Figure 5-3 Relation Between A Global Scheduler and the Application Tasks it Controls

- First, the global scheduler checks for the presence of newly-arrived activities by scanning through its list of shared variables. Those whose value are found to have been reset are placed on an arrival list.
- Next, the global scheduler checks whether each activity on the arrival list has a true precondition. Each activity that passes is further considered in order of its importance level. Then, starting with the “most important” activity, the global scheduler checks in descending order of importance if all of an activity’s input data items were available from its associated input port. The highest priority activity for which its inputs are available is selected. Once selected, the activity is considered enabled.
- Those activities which become enabled are transferred from the arrival list to the ready list.
- Finally, the global scheduler examines the timing constraint properties of the activities on the ready list and selects a candidate for execution (and removes it from the ready list).

To assure proper operation, each of the application tasks must follow the same protocol in interacting with the global scheduler. Upon completion of an activity, the task sets a shared variable to indicate that the activity’s execution is finished. In addition, the task determines which of its activities would subsequently have arrived and sets its shared variables accordingly. The task then initiates a series of *Get_Port* operations (Durra communication primitives) to retrieve the inputs required by its newly-arrived activity (or activities). Once all of the inputs for one activity have been obtained, the task sets a shared variable to notify the global scheduler that this has occurred. Finally, the task performs a *Get_Port* operation on the port connected to the global scheduler’s deal channel. This will cause the task to block until the global scheduler has identified one of the task’s enabled activities for execution.

The protocol described in the preceding paragraphs, along with the properties of the individual atomic activities, provides the information necessary to synthesize a template corresponding to a global scheduler for each cluster within a system. A complete implementation of a global scheduler may be obtained by adding the code which implement one's desired scheduling policy.

Durra actually has no notion of a shared variable. Instead, tasks communicate by passing messages through interconnecting channels. We added shared variables to SMARTS, however, because this provides a quicker and more efficient way of transferring operational status between tasks within a task cluster. Moreover, global states or modes of operation can be very useful in describing system functionality. In a requirements specification, it is preferable not to specify the details of how each activity is informed about mode changes. Rather, one is concerned when mode changes occur and which activities need to know about it. Thus in the activity graphs, a mode is represented by a state variable shared between two or more activities.

In SMARTS, the progression from requirements to design involves partitioning the activity graph into a set of interconnected restricted activity graphs. Each of the latter is allocated to a Durra task. The interconnections between restricted activity graphs serve to define the interface for each Durra task (i.e., its ports). The state variables shared between activities must conform with the Durra semantics. In the case where a state variable is shared between activities within a single Durra task or within a cluster of Durra tasks, the update of the variable's value is straightforward. The state variables shared within a single cluster would be declared in a separate Ada package, which is "withed" each of the Ada packages that implement the activities and global scheduler for that cluster. The value of each variable would be updated by its respective activity. The global scheduler could evaluate the precondition of any activity that has arrived (inputs not yet available and/or precondition last evaluated to false) or is enabled (the inputs are available and the precondition last evaluated to true). The precondition evaluation itself could be done either in real or delayed time. We have not prescribed (nor proscribed) either alternative.Rather, we permit either as an application-specific design decision.

When a state variable is shared between activities allocated to tasks in different clusters (possibly on different machines), the update process can be much more complicated. In this case, one must explicitly describe a means for transferring the value between clusters. Clearly, this could also be accomplished using a real-time or delayed time semantics. In the case of the latter, updates to shared state variables would be recorded by the activities which implement the updates. Whenever the global scheduler would execute (e.g., the application tasks are blocked or waiting) it would check if any shared variables have been updated. If some have, their identity and new values would be sent to the other Global Schedulers as well. The latter would determine whether the state variables identified in the message are actually shared by its cluster. If this is the case, the value of each variable will be updated accordingly.

Acknowledgments

The work described in this report represents the culmination of three years of research in the adaptive systems domain. The graphical representations which we have presented were influenced by the graphical modeling primitives developed by Mac Alford. Kai Wang of Hughes Radar Systems Group helped us to enhance the activity timing annotations based on his experience with the airborne radar domain. Tzilla Elrad and Ufuk Verun of the Illinois Institute of Technology helped to clarify the semantics of the activity graph formalism. Michael Gardner, a resident affiliate from the Westinghouse Electric Company, provided insight into many of the salient runtime issues. Finally, we would like to mention Mario Barbacci, Dennis Doubleday, and Charles Weinstock of the Distributed Systems project at the SEI. Their efforts made it possible for us to effectively integrate SMARTS into the Durra environment at the SEI.

References

[1] M. W. Alford. “A Graph Model Based Approach to Specification.” In *Lecture Notes in Computer Science*, Vol. 190, Distributed Systems Methods and Tools for Specification, Chapter 4, pages131–200.

[2] M.R. Barbacci, D. L. Doubleday, M. Gardner, R.W. Lichota, and C.B. Weinstock, “D*urra: A Task-Level Description Language Reference Manual (Version 3)*,” Software Engineering Institute, Draft Report, April 1991.

[3] J.V. Berk, A. H. Muntz, and P.R. Stevens, “Software Architecture Concepts for Avionics,” IEEE National Aerospace and Electronics Conference, May 1989, 88CH2759-9, vol. 4, pages 1900–1905.

[4] T. Elrad, “Comprehensive Scheduling Controls for Ada Tasking,” *Proceedings of the International Workshop on Real-Time Ada Issues*, Devon, UK, June 1988.

[5] R.W. Lichota and A. H. Muntz, “Specification Methods and Mapping Techniques for Transitioning from Requirements to Implementation,” *Proceedings of the 3rd Workshop on Large Grain Parallelism*, Pittsburgh, Pa., October 1989.

[6] A. H. Muntz and E. Horowitz, “A Framework for Specification and Design of Software For Adaptive Sensor Systems,” IEEE Real-Time Systems Symposium 1989.

[7] A. H. Muntz and R. W. Lichota, “A Requirements Specification Method for Adaptive Real-Time Systems,” submitted to the 1991 IEEE Real-Time System Symposium.

[8] A. H. Muntz, “Specification and Design Methodologies for Semi-Hard Real-Time Systems,” Doctoral Dissertation, Department of Computer Science, University of Southern California, 1990.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

4. PERFORMING ORGANIZATION REPORT NUMBER(S
aCMU/SEI-91-TR-20

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESD-91-TR-20

6a. NAME OF PERFORMING ORGANIZATION
Software Engineering Institute

6b. OFFICE SYMBOL (if applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI Joint Program Office

6c. ADDRESS (City, State and ZIP Code)
Carnegie Mellon University
Pittsburgh PA 15213

7b. ADDRESS (City, State and ZIP Code)
ESD/AVS
Hanscom Air Force Base, MA 01731

8a. NAME OFFUNDING/SPONSORING ORGANIZATION
SEI Joint Program Office

8b. OFFICE SYMBOL (if applicable)
ESD/AVS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F1962890C0003

8c. ADDRESS (City, State and ZIP Code)
Carnegie Mellon University
Pittsburgh PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT NO.
63756E	N/A	N/A	N/A

11. TITLE (Include Security Classification)
{Insert title line 1}

12. PERSONAL AUTHOR(S)
Randall W. Lichota

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM TO	December 1991	38

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse of necessary and identify by block number)
FIELD	GROUP	SUB. GR.	real-time systems specifications software architecture

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The design specification method described in this report treats a software architecture as a set of runtime entities, including tasks and external input/output elements, which interact either via messages or shared data structures. Tasks have a single thread of execution and represent program units that may be executed concurrently. External input elements produce input requests thta in turn trigger a set of low level activities to be executed by tasks. External output elements consume results which are produced by tasks. The specification method discussed here facilitates the description of the dynamic structure of runtime entities, the synchronization and communication between these entities, and their resource consumption and production properties (which include timing and sizing).

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ■ SAME AS RPT □ DTIC USERS ■

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified, Unlimited Distribution

22a. NAME OF RESPONSIBLE INDIVIDUAL
Charles J. Ryan, Major, USAF

22b. TELEPHONE NUMBER (Include Area Code)
(412) 268-7631

22c. OFFICE SYMBOL
ESD/AVS (SEI)

ABSTRACT —continued from page one, block 19