Technical Report
CMU/SEI-91-TR-21
ESD-91-TR-21

Durra: An Integrated Approach to Software Specification, Modeling, and Rapid Prototyping

Mario R. Barbacci
Dennis L. Doubleday
Charles B. Weinstock
Randall W. Lichota

September 1991

Technical Report

CMU/SEI-91-TR-21

ESD-91-TR-21

September 1991

Durra: An Integrated Approach to Software Specification, Modeling, and Rapid Prototyping

Mario R. Barbacci

Dennis Doubleday

Charles B. Weinstock

Randall W. Lichota

Distributed Systems

Software Engineering Institute

Carnegie Mellon University

Pittsburgh, Pennsylvania 15213

This document was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

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

Review and Approval

This document has been reviewed and is approved for publication.

FOR THE COMMANDER

[illegible]
Charles J. Ryan, Major, USAF
SEI Joint Program Office

The Software Engineering Institute is sponsored by the U.S. Department of Defense. This report was funded by the Department of Defense.

Table of Contents

List of Figures

Durra: An Integrated Approach to Software Specification, Modeling, and Rapid Prototyping

Abstract: Software specification, modeling, and prototyping activities are often performed at different stages in a software development project by individuals who use different specialized notations. The need to manually interpret and transform information passed between stages can significantly decrease productivity and can serve as a potential source of error. Durra is a non-procedural language designed to support the development of distributed applications consisting of multiple, concurrent, large-grained tasks executing in a heterogeneous network. Durra provides a framework through which one can specify the structure of an application in conjunction with its behavior, timing, and implementation dependencies. These specifications may be validated by passing behavioral and timing information associated with each Durra task description to a run-time interpreter. Similarly, software prototypes may be constructed by directing this information to a suitable source code generator. We have already developed an interpreter and source code translator for a language based on simple timing expressions. We are presently constructing a source code generator for a more complex language defined by SMARTS (the Specification Methodology for Adaptive Real-Time systems developed by Hughes Aircraft Company). [1]

1 Introduction

This paper discusses the relationship between software specification, modeling and prototyping activities as part of a real-time system development strategy. Most often these activities are performed at different stages in a software development project by individuals who use different specialized notations. The need to manually interpret and transform information passed between stages can significantly decrease productivity and can serve as a potential source of error. Tne recent development of commercial executable specification tools represents an initial semi-automated link between specification, modeling and prototyping activities. Many of these tools use a graphical notation based on real-time structured analysis to represent software models and provide a built-in simulation capability [7]. Unfortunately, these tools have been shown to be relatively ineffective for performance modeling where computational accuracy, response time and resource contention are of principle concern [8]. Moreover, the prototypes generated from realistic software specifications tend to be much too inefficient for real-time applications.

1. An earlier version of this paper was presented at the 2nd International Workshop on Rapid System Prototyping and will appear in the workshop proceedings.

We believe that these problems are largely due to the fact that the formalisms upon which most executable specification tools are based represent too high a level of abstraction. We feel that to effectively link specification, modeling, and prototyping activities, integration must occur at the level of a *technical architecture*. This corresponds to a software architecture augmented to include formal descriptions of task behavior. We believe that Durra, currently under development at the Software Engineering Institute, can provide this integration. Durra is a task-description language intended for developing distributed applications implemented by large-grained tasks [4]. This is a non-procedural language, separate from the various programming languages used to develop the component tasks. Using Durra, the developer specifies the application structure and the resources allocated to the component tasks independently from the coding of the individual components. Durra also allows one to specify the conditions for system reconfiguration as well as their nature (e.g., to implement mode changes and/or fault recovery).

Although Durra is hardware independent and can be used on a variety of processors and communication networks, for the purpose of our explanation we assume an abstract machine with multiple non-homogeneous processors, a local area net to allow direct communication between processors, and an operating system or runtime executive running in each processor and providing reliable communications facilities.

2 An Overview of Durra

Durra was designed to support the development of distributed applications [3]. An application is specified in Durra as a set of *task descriptions* that prescribes a way to manage the resources of a heterogeneous machine network. Syntactically, an application description is a single task description and can be used as a component task for larger applications. As shown in Figure 2-1, a Durra task description is a compilable template that defines the properties of a task's implementation. Task descriptions provide information about the interface, behavior, attributes, and internal structure of a task. They may be used as building blocks for application-descriptions or larger, compound task-descriptions.

task *task-name* (*parameter-list*)

ports

port-declarations

behavior

specification-list

attributes

attribute-value-pairs

components

components-declaration

structures

component-connections

reconfigurations

condition-action-pairs

clusters

cluster_component_associations

end *taskname*;

Figure 2-1 Durra Task Description Template

In Durra, interconnections between tasks are denoted by links each representing an instance of a channel. A channel may be considered a conduit through which data is passed from one task to another. Channel descriptions are compilation units that define the properties of channel implementations and are syntactically similar to task descriptions, the only difference being that they lack the components, structures, and reconfigurations sections. Channel descriptions are used to construct application descriptions and compound task descriptions.

Interface information takes the forn of unidirectional, typed port declarations as shown on the following page.

```
ports
in1: in heads;
out1, out2: out tails;
```

A port declaration specifies the direction and type of data moving through the port. An *in* port takes input data from a channel; an *out* port deposits data into a channel. Note that the port types are not built into the language; they must be declared by the user.

The behavior part consists of a list of name/value pairs denoting formal properties of the component. For demonstration purposes, we have developed a simple language based on timing expressions which describes the behavior of a task in terms of the operations it performs on its input and output ports. This language is described in more detail in the section which follows.

As shown below, the attribute part consists of a list of name/value pairs denoting miscellaneous properties of a component (i.e., task or channel). Attributes play a central role in the construction of software prototypes by providing a link to pre-existing library components. Each component within a library is assumed to have both a description (specified in the Durra language) and an implementation (expressed in a suitable programming language). When entering a component into the library, the developer assigns each attribute an property value. When a task or channel name is referenced within the component section of a compound task description, the desired value of the property is specified. Example attributes include author, version number, programming language. procedure or package name (associated wit the implementation), and processor type:

```
attributes
author = "rwl";
package_name = "Data_Manager";
Sorting_Algorithm = "Quicksort";
```

Descriptions of simple tasks and channels (but not compound tasks) may be customized through the use of formal parameters. The latter consists of a list of typed parameter names that can be used in simple task or channel descriptions anywhere a value of the appropriate type is allowed. The actual parameter value is specified as part of the task or channel instantiation. The parameter types (INTEGER, REAL, STRING, and IDENTIFIER) are built into the language and should not be confused with port types which must be specified by the user.

Parameters allows one to abstract away some of the details of interface declarations, thus providing an additional measure of flexibility in constructing application descriptions. Moreover, parameters provide a means for constructing generic channel descriptions in which the port type and bounds need not be fixed. When completed, our implementation of the Durra compiler and library will include a set of generic channel descriptions (and their associated implementations) that will support some of the commonly-used forms of inter-task communication. As an example, a description of a generic broadcast channel is shown below.

```
channel broadcast (number_of_sources: integer,
port_type: identifier)
ports
input: In port_type;
output[1..number_of_sources]: out port_type;
end broadcast;
```

Structural information describes the internal components of a compound task and serves to distinguish a compound task from a simple task or a channel (i.e., channel descriptions cannot contain structural information). This information is comprised of three parts:

1. The component section enumerates the task *and channel instantiations used* as internal components.
2. The structure section describes how these internal components are connected to form a configuration.
3. The reconfiguration section describes the conditions under which the structure of an application can vary during execution (there can be one or more of these configurations in a structure section).

Durra provides developers with a means for specifying both the conditions for reconfiguration and the nature of the reconfiguration.

As noted earlier, the Durra language provides a mechanism for identifying and selecting components from a library. These are identified via templates of desired properties that the Durra compiler matches against templates of reusable program elements. Components are entered into a library by compiling their Durra description, and are subsequently selected by submitting an application or compound task description to the Durra compiler. The components referenced within this description are specified as component *selections*. These are expressed as templates which syntactically resemble a primitive form of component description (i.e., the interface, attribute, and behavior parts may be present but the component, structure, and reconfiguration information may not be specified).

In general, a given task might have multiple implementations that differ according to the algorithm used, performance, processor requirements, or version of the code. In order to select from alternative implementations, a task selection must be provided which lists the desirable features of a suitable implementation. A task description matches a selection if the ports have similar message types and direction (input or output), if the behavioral specifications of the task description imply the behavioral specifications of the task selection, and if the attribute expression in the task selection (a predicate on attribute names and values) yields true when evaluated in the context of the task description attributes (list of name/value pairs). During a library search, zero, one, or more than one candidate task description can be found to match a given task selection. Anything but exactly one match is considered an error although the severity of the error varies with the nature of the mismatch.

3 Behavioral Specifications

In principle, a wide variety of formal methods may be used to define the behavior of Durra tasks. To demonstrate the potential utility of Durra as a specification and modeling tool, we have initially focused on the implementation of a simple timing expression language which describes the behavior of a task in terms of the operations it performs on its input and output ports [5]. A timing expression may be used to describe the patterns of execution of operations on the input and output ports of a task and, thus, specifies its behavior as seen from the outside.

Timing	::=	Statement_List$_{semicolon}$
Statement	::=	loop_statement |
		repeat_statement |
		while_statement |
		if_statement |
		put_statement |
		get_statement |
		wait_statement |
		delay_statement |
		signal_statement
loop_statement	::=	**LOOP** Statement_List$_{semicolon}$ **END LOOP**
repeat_statement	::=	**REPEAT** IntegerExpr loop_statement
while_statement	::=	**WHILE** BooleanExpr loop_statement
if_statement	::=	**IF** BooleanExpr
		THEN Statement_List$_{semicolon}$
		{ **ELSE** Statement_List$_{semicolon}$ }
wait_statement	::=	**WAIT** ‘“’ “BooleanExpr “,” ArithExpr “)”
put_statement	::=	**PUT** ‘“’ “PortName { “,” TypeName { “,”’ IntegerExpr } } “)”
get_statement	::=	**GET** ‘“’ “PortName { “,”’ TypeName { “,”’ IntegerExpr } } “)”
delay_statement	::=	**DELAY** ‘“’ “ArithExpr { “,” ArithExpr } “)”
signal_statement	::=	***SIGNAL*** ‘“’ “IntegerExpr “)”

Figure 3-1 Timing Expression Language Syntax

As shown in Figure 3-1, the timing specification language consists of a small number of imperative statements which may be used to specify a sequence of event expressions. A basic event expression is either a channel operation (specified using the GET and PUT primitives), a SIGNAL, or a DELAY directive. The latter causes the execution of a task to be delayed an amount of time that may be fixed or may correspond to a random number, drawn from a uniform distribution between two specified limits. The SIGNAL statement sends a specified signal to the Durra runtime system and is intended to facilitate reconfiguration. An example of a simple timing expression appears in Figure 3-2.

The iteration statements (LOOP, REPEAT, and WHILE) indicate sequences of statements to be executed multiple times. These may be used to specify that a sequence of statements is to be repeated while a condition is true (WHILE), for a fixed number of times (REPEAT), or indefinitely (LOOP). The WAIT statement delays the execution of the task implementation until a specified Boolean expression is true. This expression is evaluated repeatedly until it yields true (the amount of time between retries may be specified).

The GET and PUT statements specify the port input and output operations respectively. A GET statement reads a message from an input port. If the type name is specified, a check is made that the message received is of that type. If the message size is also specified, a test is made to see whether the message received has the specified size. A PUT statement generates a message of a given type and size and sends it to an output port. If the port is declared in the Durra task description to be of a union type (i.e., messages of more that one type may be accepted), the type name denotes the actual message type. Similarly, if the type defines a data representation of variable size, the message size denotes the actual size. Note that both the message type and size can be left unspecified if the Durra compiler would be able to deduce this information from the port declaration.

```
signal(0);
 if sizeof(t1) > 20 then
  repeat 20 loop
    put(p1,t1,20);
    get(p2,t2);
    end loop;
 wait(20 > current_ptime, 10);
 delay(20,30);
 end if;
```

Figure 3-2 Example of a Simple Timing Expression

Timing specifications may be used to drive a run-time interpreter or a source-code generator that translates the specification into a corresponding Ada implementation of the task. The code generator produces the main unit of the program (a procedure) and imports additional support packages (e.g., Durra interface, Calendar, System). The procedure can be compiled and linked by a suitable Ada development system and stored in an object code library. This is discussed in more detail in the following section.

4 Durra Development Methodology

4.1 Application Development

There are three phases in the development of an application using Durra: (1) library creation, (2) application creation, and (3) application execution. During the first phase, the developer creates implementations for the various tasks that will be executed in the heterogeneous machine. For a given task, there may be many implementations differing in programming language (e.g., C or Ada), processor type (e.g., Motorola 68020), performance characteristics, or other properties. For each implementation, a task description must be written in Durra, compiled, and entered in the library. The description may include specifications of a task implementation's performance and functionality, the types of data it produces or consumes, the ports it uses to communicate with other tasks, and other miscellaneous attributes of the implementation.

During the second phase, the user writes an application description. Syntactically, an application description is a single task description and could be stored in the library as a new task which could then be used as a component task for larger applications. The Durra compiler uses a set of rules to select task descriptions (and their associated implementation) from a library baseo on the specified interfaces (typed message ports), attributes (name/value pairs), and behaviors (formal specifications). As shown in Figure 4-1, the Durra compiler groups the application components into clusters, generates "code frames" that link each cluster into one Ada program, and generates the main units using structural information contained in the application descriptions. This imposes a restriction that components must be written in Ada or in a language that can be linked with and invoked from Ada procedures.

To retain the advantages of reusability and reconfigurability, all information about network resources and application components and structure is hidden in the body of the generated frames (the main unit in each cluster). Application components communicate using Durra message-passing primitives but these are implemented either as local procedure calls (when communicating with tasks in the same cluster) or remote procedure calls (when communicating with tasks in a different cluster). The decision as to which version is used is based on the current configuration of the application and is transparent to the application tasks.

During the final stage the application is executed. A set of *cluster launchers* (one per machine) start the task implementations. The Durra runtime supports communication between tasks and provides for dynamic reconfiguration.

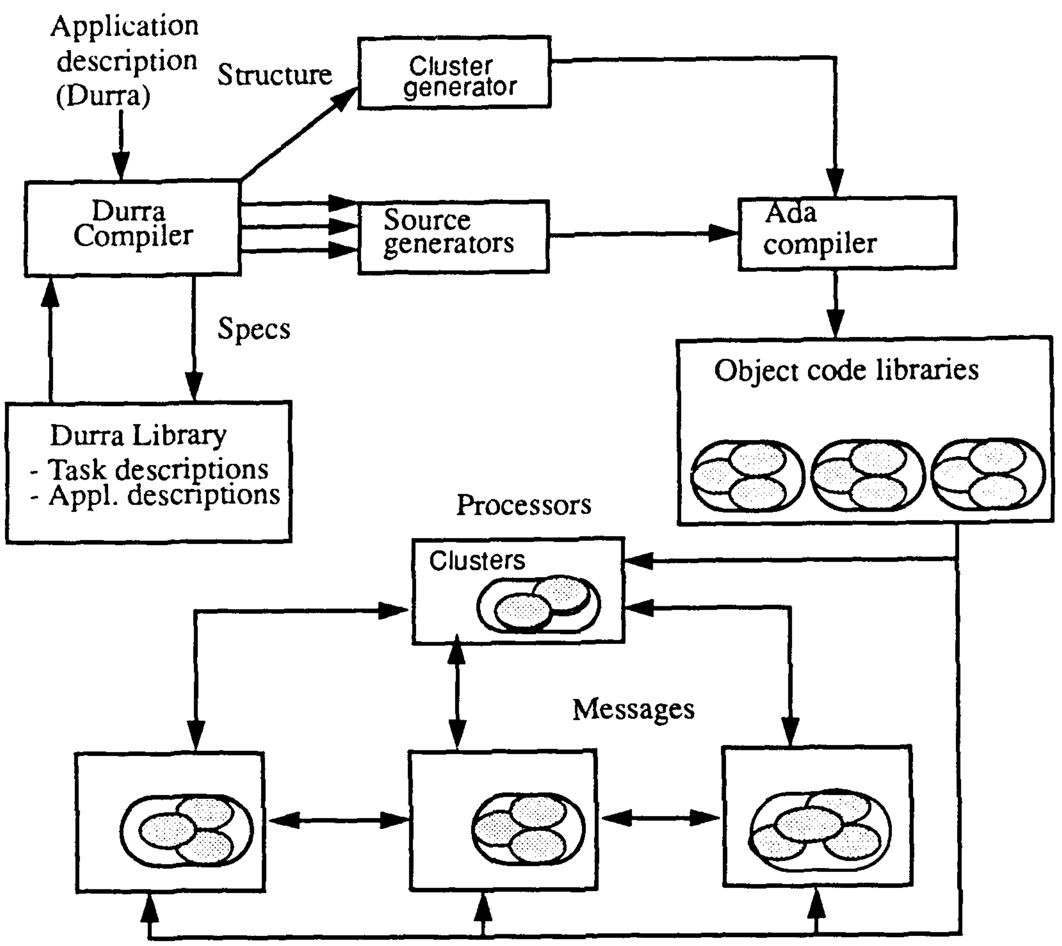

Figure 4-1 Durra Development Model

4.2 Prototype Development

The task matching mechanism outlined above works exclusively with Durra task selections and task descriptions. The task implementations (i.e. the real programs) are not part of the selection process. An important consequence of this separation between the task description and its implementation is that it is not necessary to have implementations of all the compo

nents to construct an application prototype. These can be built using early implementations of component tasks or component emulators driven by behavioral specifications. Component emulation can be used to model software designs and to experiment with alternative specifications of critical components. To illustrate the approach we have created an Ada program that interprets timing expressions of the form described in Chapter 3 [2].

The task emulator described above provides natural support for system development methodologies based on successive refinements, such as the Spiral method [6]. Users of the spiral model selectively identify high-risk components of the product, establish their requirements, and then carry out the design, coding, and testing phases. It is not necessary that this process be carried out through the testing phase -- higher-risk components might be identified in the process and these components must be given higher priority for development. Durra allows the designer to build mock-ups of an application decomposed into tasks specified by their interface and behavioral properties. Subsequently, an application can be emulated by interpreting the timing expressions for the yet-to-be written implementations.

The result of the emulation would identify those tasks whose timing expressions suggest are more critical or demanding and thus more likely to affect the performance of the entire system. The designers can then experiment by writing alternative behavioral specifications for each of these tasks until a satisfactory specification (i.e., template) is obtained. Once this is achieved, the designers can proceed by replacing the original task descriptions with more detailed templates, using the structural description features of Durra. These more refined application descriptions can be emulated again and alternative behavioral specifications of the internal tasks created until a satisfactory internal structure (i.e., decomposition) has been achieved. This process can be repeated as often as necessary, varying the degree of refinement of the tasks, and even backtracking if the timing constraints become unsatisfiable. It is not necessary to start coding a task until later, when its specifications are acceptable, and when further decomposition of the task is not required.

Once experimentation with component emulators has helped to establish the principle timing trade-offs, a prototype can evolve to include real component implementations. If the tasks were to be handcoded, errors might be introduced if the coder misunderstood the behavioral specifications used to drive the early prototypes. This can be avoided if formal specifications of behavior can be processed by suitable source code generators. Using Durra as a driver, a component can be generated by extracting the formal specification of the component from a task description, feeding the specification to the source code generator, and storing the resulting program in a library. To demonstrate the concept we have built a tool that generates Ada programs whose behavior is also specified by timing expressions. These programs perform input and output operations but do not execute code between input or output operations (timing expressions do not describe algorithms). These programs are useful nevertheless as an intermediate step between application prototypes using emulated components and the final application, and can be used as guides to programmers.

Since code frames are generated automatically, changes in the application structure and available resources can be captured in a Durra application description and a new set of frames can be generated, linking together the (unmodified) application components. This capability, in conjunction with support for behavioral specifications based on timing expressions, permits evolutionary prototyping to be carried out. Each of the application tasks would be distributed among multiple machines just as for the target system (except for possible differences in the processor/interconnection). Thus much of the structure of the application is directly implemented in a distributed environment. Initially, the computational and logical components of each task would be simulated. Gradually they would be replaced by implementations that have been derived using source code generation.

In most instances the time required to emulate the behavior of a task will differ from the execution time of the task's ultimate implementation. Accordingly, we will be implementing a global simulated clock as part of a prototype application. As the prototype evolves to include a greater number of task implementations, the simulated clock will become an increasingly less important factor in timing validations. Eventually, when these prototypes are transitioned to a production system, the simulated clock will disappear altogether.

5 Related Work

At the technical architecture layer, there has been very little work directed towards developing a comprehensive env. 'onment which will aid in the specification, implementation, and validation of a software architecture for real-time distributed applications. CONIC [10], for instance, focuses primarily on the problem of dynamic reconfiguration of real-time systems. Originally, CONIC restricted tasks to be programmed in a fixed language (an extension to Pascal with message passing primitives) running on homogeneous workstations. This restriction was later relaxed to support multiple programming languages.

MINION [11] consists of a language for describing distributed applications and a graphics editor for interactive modification of the application structure. MINION allows a user to expand, contract, or reconfigure an application in arbitrary ways during execution time. Hermes [13] hides from the programmers all knowledge about storage layout, persistency of objects or even operating system primitives. Processes communicate through ports, connected via message queues although the semantic of queue operations are similar to an Ada entry call/accept mechanisms, albeit the binding of processes to ports is dynamic, as in Durra, CONIC, and MINION.

RNET [12] is a language for building distributed real-time programs. An RNET program consists of a configuration specification and the procedural code, which is compiled, linked with a run-time kernel, and loaded onto the target system for execution. The language provides facilities for specifying real-time properties, such as deadlines and delays that are used for monitoring and scheduling the processes. These features place RNET at a lower level of abstraction, and thus RNET cannot be compared directly to Durra. Rather, it can be considered as a suitable language for developing the runtime executive required by Durra and other languages in which the concurrent tasks are treated as black boxes.

Maruti [14] is an environment intended to support hard real-time distributed applications that have security and/or fault tolerance requirements. Maruti also supports heterogeneous operation through a common message interface. While these represent important similarities with Durra, it should be pointed out that Maruti differs in that it constitutes an operating system in its own right. By contrast, Durra is intended to support the development of distributed software architectures for a variety of possible machine configurations.

MIDAS [1] represents an approach to the design of distributed real-time systems based on the iterative refinement of performance models. While this approach is not specifically described as a form of prototyping, it shares with Durra the concept of combining task implementations with simulated components. MIDAS also uses the concept of a simulated clock but differs in supporting general-purpose distributed simulation through checkpointing and rollback (Durra as yet only supports distributed simulation for cases where timing dependencies can be resolved). In addition, MIDAS does not support the generation of code from model specifications, it does not provide a means to specify reconfiguration, and it does not support the use of multiple behavioral specification languages.

Techniques for generating large-scale multitasking Ada applications has been developed by Noah Prywes and others at the University of Pennsylvania. Individual tasks are described in a non-procedural language based on algebraic specification methods. This language forms part of MODEL which provides automated checking and compilation of specifications into sequential Ada, PL/1, or C code [15]. Communication between tasks is specified in a dataflow language called CSL. A Configurator program is used to validate CSL dataflow graphs and generate an Ada shell to control task execution. This shell also implements communication between tasks (including interprocessor communication).

Prywes approach shares several common elements with Durra. Each module produced by the MODEL compiler is considered a separate distinct procedure. These are subsequently equated with tasks during the construction of CSL descriptions. Moreover, CSL tasks may be manually constructed as well as generated from model descriptions. In effect, these would correspond to Durra Task Implementations. Other commonalities with Durra include the use of standardized interface procedures to implement intertask communication and the provision for alternative communication implementations selectable via attributes.

Prywes approach differs from Durra primarily in the areas of communication and reconfiguration. In regard to the former, Prywes employs a number of built-in assumptions regarding the manner in which tasks communicate. For example, files (which correspond to Durra channels) can implement the transfer of data between tasks in one of four predefined ways. Durra, while also providing a small set of predefined channel behaviors, permits a user to define additional channel implementations. Because the Configurator is intended to automatically synthesize CSL descriptions into a set of communications procedures and tasks, arbitrary file behavior cannot be supported. Thus Durra provides the potential for greater tailoring of communications-oriented software at the expense of automatic generation. It is possible, however, to "re-use" additional channel implementations by placing these into the Durra library and defining for each a corresponding attribute.

In addition to the differences in intertask communication, Durra differs from CSL by providing a means to describe reconfiguration actions to be carried out at run time. This allows one to specify the migration of tasks between processors to support load balancing or recovery from run-time faults. The version of the Durra compiler under development will initially generate Ada software that supports one model of task migration.

6 Conclusion and Future Extensions

Durra is a non-procedural task-description language specifically designed to support the development of large-grained distributed applications. 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 structure of an application is expressed separate from its behavior. In addition, the Durra runtime system supports the construction of distributed Ada programs and thus provides a mechanism for prototyping applications for a distributed environment. Consequently, an application can evolve as its requirements change or are better understood. This may entail changes in the application description, selection of alternative task implementations from a library, and their connection to reflect alternative designs.

Task emulation and source code generation represents two methods for using Durra as a prototyping tool. Generated programs can serve as an intermediate step between application prototypes using emulated components and the final application. While our demonstration system uses timing expressions as the behavioral specification language, Durra can be easily tailored to support the use of other languages for which interpreters or source code generators exist. We are in the process of modifying the Durra tool suite to permit inclusion of other kinds of behavioral specifications in task descriptions in lieu of timing expressions. The next type of behavioral specification which will be supported is the Restricted Activity Graph notation defined by SMARTS [9].

References

1 R.L. Bagrodia and C.C. Shen. "MIDAS: Integrated Design and Simulation of Distributed Systems." *IEEE Transactions on Software Engineering.* To appear in the October 1991 issue.

2 M.R. Barbacci. "MasterTask: The Durra Task Emulator." CMU/SEI-88-TR-20 (DTIC: ADA199 429). Software Engineering Institute, Carnegie Mellon University, July 1988.

3 M.R. Barbacci, D.L. Doubleday, and C.B. Weinstock. "Application-Level Programming." *Proceedings of the 10th International Conference on Distributed Computing Systems.* Paris, France. May 1990.

4 M.R. Barbacci and J. M. Wing. "Durra: A Task-Level Description Language." *Proceedings of the 16th International Conference on Parallel Processing.* St. Charles, Illinois. August, 1987.

5 M.R. Barbacci and J.M. Wing. "Specifying Functional and Timing Behavior for Real-time Applications." *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE).* Springer-Verlag, *Lecture Notes in Computer Science,* volume 259, part 2. 1987. pp. 124-140.

6 B. W. Boehm. "A Spiral Model of Software Development and Enhancement." *Computer*, volume 21, number 5. May 1988.

7 D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman and A. Shtul-Trauring. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." *Proceedings of the 10th International Conference on Software Engineering.* 1988. pp. 396-406.

8 A. H. Muntz. "Specification and Design Methodologies for Semi-Hard Real-Time Control Systems." Doctoral Dissertation. University of Southern California, Los Angeles, California. May 1990.

9 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 10-11, 1989.

10 J. Kramer and J. Magee. "A Model for Change Management." *Proceedings of the IEEE Workshop on Trends for Distributed Computing Systems in the 1990's.* September 1988. pp. 286-295.

11 J.M. Purtilo and P. Jalote. "An Environment for Prototyping Distributed Applications." *Proceedings of the Ninth International Conference on Distributed Computing Systems.* Newport Beach, CA: IEEE Computer Society. June 1989. pp. 588-594.

12 C. Belzile, M. Coulas, G.H. MacEwen, and G. Marquis. "RNET: A Hard Real Time Distributed Programming System." *Proceedings of the 1986 Real-Time Systems Symposium.* IEEE Computer Society Press. December 1986. pp. 2-13.

13 D.F. Bacon, R.E. Strom, and S.A. Yemini. *Hermes User Manual.* IBM Thomas J. Watson Research Center, 1988.

14 O. Gudmundsson, D. Mosse, K. T. Ko, A. Agrawala, and S. Tripathi. "MARUTI: A Platform for Hard Real-Time Applications." 1989 Workshop on Operating Systems for Mission Critical Computing. September 19-21, 1989.

15 Y. Shi and N. Prywes. "Generating Multitasking Ada Programs from High-Level Specifications." *Proceedings of the Third International Conference on Ada Applications and Environments*. Manchester, New Hampshire. May 23-25, 1988. pp. 137-149.

16 Y. Shi and N. Prywes, "Generating Multitasking Ada Programs from High-Level Specifications," Proceedings of the Third International Conference on Ada Applications and Environments, Manchester, New Hampshire, May 23-25, 1988, pages 137-149.

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

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 CMU/SEI-91-TR-21	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-91-TR-21
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 OF FUNDING/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 152	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO 63756E; PROJECT NO. N/A; TASK NO N/A; WORK UNIT NO. N/A

11 TITLE (Include Security Classification)
Durra: An Integrated Approach to Software Specification, Modeling, and Rapid Prototyping

12 PERSONAL AUTHOR(S)
Mario R. Barbacci, Dennis Doubleday, Charles B. Weinstock, and Randall W. Lichota

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

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18. SUBJECT TERMS (Continue on reverse of necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Durra, software specification, software modeling, prototypes, real-time systems, distributed systems

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

Software specification, modeling, and prototyping activities are often performed at different stages in a software development project by individuals who use different specialized notations. The need to manually interpret and transform information passed between stages can significantly decrease productivity and can serve as a potential source of error. Durra is a non-procedural language designed to support the development of distributed applications consisting of multiple, concurrent, large-grained tasks executing in a heterogeneous network. Durra provides a framework through which one can specify the structure of an application in conjunction with its behavior, timing, and implementation

(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

dependencies. These specifications may be validated by passing behavioral and timing information associated with each Durra task description to a run-time interpreter. Similarly, software prototypes may be constructed by directing this information to a suitable source code generator. We have already developed an interpreter and source code translator for a language based on simple timing expressions. We are presently constructing a source code generator for a more complex language defined by SMARTS (the Specification Methodology for Adaptive Real-Time systems developed by Hughes Aircraft Company). [1]

1. An earlier version of this paper was presented at the 2nd International Workshop on Rapid System Prototyping and will appear in the workshop proceedings.