1994

A Specification Environment That Supports the Prototyping of Distributed Systems Using an Object-Oriented Model.

Abbas Dehkhoda
Louisiana State University and Agricultural & Mechanical College

Order Number 9425219

A specification environment that supports the prototyping of distributed systems using an object oriented model

Dehkhoda, Abbas, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A SPECIFICATION ENVIRONMENT THAT SUPPORTS THE PROTOTYPING OF DISTRIBUTED SYSTEMS USING AN OBJECT ORIENTED MODEL

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Abbas Dehkhoda
B.A., National University of Iran, 1972
M.S., Southern Methodist University, 1980
May 1994

Acknowledgments

After the long process of writing a doctoral dissertation, it's difficult to remember all the people to thank. But it's easy to know where to start. I would like to thank my advisor, Dr. Doris L. Carver, for being a constant source of advice and support during this process. She knew there would be substantial design, implementation, debugging, and experiments that preceded even the most preliminary results. She has carefully guided me and anticipated the problems, pitfalls, impasses, and logistic problems. No one could ask for a better advisor.

There are a number of other people to thank. Dr. Dennis Kafura, for numerous discussions we had that were invaluable. The committee members: Drs. Ahmeh El-Amawy, S. Sitharam Iyengar, Donald H. Kraft and Bush Jones for their contributions to this research. In addition, I would like to thank the software engineering students for their support.

I wish to express my special thank to my wife Fatemeh Ghassemi, my son Mohammad Taher, and my parents for their love and patience. I am always grateful to them.

Finally, I would like to thank God for making this work possible.

Table of Contents

List of Figures

Abstract

High-speed computer networking, interactive service, and incremental growth for computing are some of the motivations for developing a distributed system. Despite the inherent benefits of a distributed system, the development of software support is more difficult for distributed systems than for sequential systems. In either case, difficulties may arise from the communication problems between two groups of people with different backgrounds trying to formulate requirements for the system. This process depends on feedback and may take many iterations to converge. Customers can usually recognize the features they need when they start using a system, which makes prototyping an important tool in requirement analysis.

Many prototyping goals, objectives, and approaches are possible. Executable formal specifications are the most attractive one. This unification of specification and prototyping by having code generators has advantages of providing consistency and prototyping at higher levels of abstraction. Thus, a methodology for executing the **DOSL** (Distributed Object-based Specification Language) is defined and a prototype system is developed. **DOSL** is extended as a new formal distributed object-oriented specification language, **DOSL-II**. **DOSL-II** is object-oriented rather than object-based, and includes class, inheritance, simple I/O, stream I/O, concurrent I/O, and new constructs for object communication.

Chapter 1
Introduction

1.1 Overview

The current high costs, long development times, and unpredictable quality of software indicate that there are some difficult problems in software development. Some of these problems are technical and others involve human factors and economics. Many of them are linked to difficulties in dealing with uncertain information, communication problems, and the labor-intensive nature of current software development practices.

Schedule and cost overruns are common problems in software development. The effort for constructing a software system is very hard to predict based on the functional specifications, because many tasks in the development are unknown at that stage and small changes in the requirements can lead to a large difference in cost. Effort is also hard to predict because the ratio between the productivities of the best and worst programmers in a team can vary by at least a factor of 10 [Luq89]. The earliest time that accurate estimates (10-20%) are likely is during architectural design, when all of the modules to be built have been identified.

Repeated reestimation and rescheduling are often needed as the project proceeds and more information become available. This process usually requires flexibility in either schedule, cost, or functionality of the product to be delivered. Such flexibility is not always provided by contracts for software development. Large systems should be delivered as a series of relatively small enhancements to a simple kernel system; this allows the delivery of the system that performs some useful functions in a reasonable amount of time. Small enhancements can be delivered with less risk of exceeding the schedule and budget. There is also less risk of the customer's

perceptions of the problem changing so much that the system is obsolete before it is delivered.

The theory of software engineering is incomplete in the sense that there are no universal methods that guarantee a working system will appear after a finite number of steps. In practice, there are usually places where the developer must throw away the original design and start over. There is rarely enough room in the schedule for doing that unless the problem is recognized before a large amount of effort has been invested in the faulty design decision. Current wisdom is to invest heavily in design reviews early in the project.

The quality of software products has been unstable and difficult to predict, partially because it is difficult to determine accurate requirements for a software system. Communication problems are one source for this difficulty. Most customers can explain the symptoms of their problems, but they have difficulty in understanding the underlying causes or in explaining what the system must do to solve their problems. Reaching an agreement between two groups of people with very different backgrounds and formulating the requirements accurately is a time-consuming process that depends on feedback, and may take many iterations to converge. Customers can usually recognize what they need when they start using a sample of working system, which makes prototyping an important tool in the requirement analysis.

It is both difficult and expensive to produce high-quality software. One solution to help alleviate this problem is the use of ***software engineering environments*** that integrate a number of tools, methods and, data structures to provide support for program development and/or maintenance [Hai85][Sta84]. To summarize, successful automation of software development depends on research and development efforts, investment policies, and training.

The need for distributed systems is increasing. High-speed computer networking, interactive service, and incremental growth for computing are among the motivations for developing distributed systems. Despite the inherent benefits of a distributed system, the development of software support is more difficult for distributed systems than for sequential systems. In sections 1.2, 1.3, 1.4, 1.5 formal methods, specification languages, prototyping, and motivation are presented, respectively. Finally the scope of this research is presented in section 1.6.

1.2 Formal Methods

Formal software specification implies that the specification is expressed in a notation which is mathematically sound. This means that both the syntax and the semantics of the specification language should be formally defined so that the meaning of a specification can be determined by reference to the specification language definition.

The syntax of the language is usually presented formally in Backus-Naur Form (**BNF**); however, the problem of defining the semantics or meaning of the language constructs is a much more difficult one. There are three distinct approaches to this problem.

The operational approach: In this approach to semantic definition, an abstract machine is defined and the language semantics are expressed in terms of abstract machine operations. This technique has been used to define the semantics of the programming language **PL/1** using a notation called **VDL** [Pag81]. It has also been used to define the Distributed Object-based Specification Language (**DOSL**) [Lee91]. Other descriptions of the operational approach are given by [Ber82][Geh85][Cor90].

The problem which arises with this approach is that it relies on the operations of the underlying abstract machine being unambiguous and well understood. Using an operational model to define semantics simply pushes the problem down a level so that instead of language semantics the semantics of the abstract machine operations must be defined.

The denotational approach: The denotational approach to the definition of programming language semantics has its foundations in the lambda-calculus, which is a calculus of mathematical logic. The fundamental work in applying the lambda-calculus to the definition of programming languages has been carried out by Strachey and Scott and is described by Strachey and Milne [Str76] and by Stoy [Sto77].

While the operational approach maps programming language constructs onto abstract machine states, the denotational approach is based on functions which map constructs onto an abstract value space. The values in this space are mathematical objects such as integers, truth values and functions, so that mathematical techniques can be used to reason about their properties.

The denotational approach is the basis for a specification method called the Vienna Development Method (VDM) which has an associated specification language called **META-IV**. It has been suggested that this may be useful in the specification of large software systems, but the problem is that the associated notation is very complex [Jon80][Bjo82][Win90].

The axiomatic approach: The axiomatic approach to the definition of programming language semantics has been developed by Hoare [Hoa69]. It is unlike the denotational or operational approaches in that it is not based on some model underlying the programming language. Rather, it is founded on the idea that each programming construct should have associated axioms which state what may be

asserted after execution of that construct. These assertions are made in terms of what is true before execution.

This approach is the foundation for a great deal of work on formal program verification. However, it does have the disadvantage that axioms for complex programming language constructs are difficult to devise. The axiomatic approach has been used to define a subset of Pascal [Wir73] but it is better suited to the definition of simpler languages than most of today's widely used programming languages.

1.3 Specification Languages

Various specification languages have been developed to specify a specification of a system concisely and abstractly. A specification language is a very high level, abstract language. It may or may not be executable. It contains features which can identify the desired system's behavior, structural properties or/and constraints formally and abstractly [Win90].

A specification is formal if it is written entirely in a language with an explicitly and precisely defined syntax and semantics. There are advantages in using formal, rather than informal specifications. Formal specifications can be studied mathematically while informal specifications cannot. For example, a correct program can be proven to meet its specifications, or two alternative sets of specifications can be proven equivalent. Formal specification can also be meaningfully processed by a computer. Certain forms of inconsistency or incompleteness in the specification can be detected automatically [Gut75]. Since this processing can be done in advance of implementation, it can be a valuable aid to program design. In addition, formal specifications can sometimes be realized automatically, although the resulting implementation may not be as efficient as one designed by a programmer.

Even in cases where these mathematical tools will not be used, formal specifications are advantageous. When specifications are used as a communication

medium among programmers during system design and implementation, it is essential that the programmers reading a specification all agree on what that specification means. This is more likely when the specification is formal, for two reasons. First, there is only one way to interpret a formal specification, because of the well-defined and unambiguous semantics of the specification language. Second, the formality of the language encourages greater rigor in the definitions. The formal specification always can be extended with informal specification as comments. In this way, the reader can get the idea of the specification quickly and easily, but also have sufficient information to understand fully what is meant.

1.4 Prototyping

Prototyping is the process of quickly producing a software system that approximates a proposed system. The prototype exhibits the functional behavior of the target system, but may not meet all the real-time requirements. Using the prototype provides feedback to the software designers as to the suitability of the system, and also gives valuable early experience to future users [Red81][Bal89][Wan90][Hek88][Lew89].

Prototyping results in the early establishment of more complete and correct requirement and design [Gom81][Luq93][Luq92][Luq88][Zel80][Kal82][Ber90]. The overall effect of a prototype is to make the software development lifecycle more cost-effective.

Prototyping has a direct impact on the software engineering lifecycle. Since prototyping allows users to interact with the system, requirements deficiencies can be discovered early.

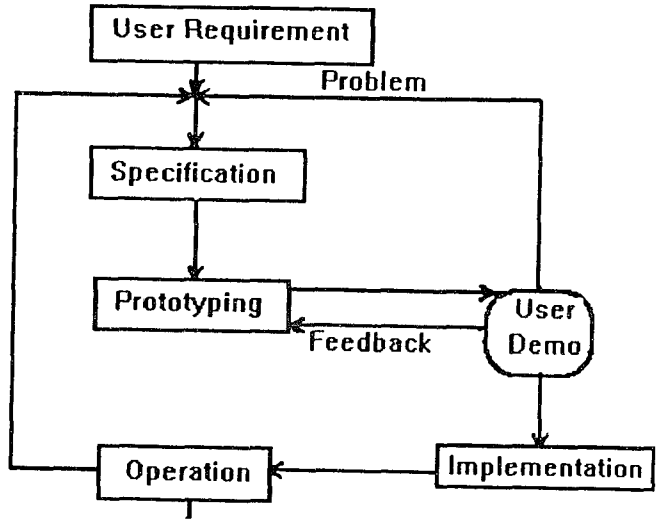

Figure 1.1 The Prototyping Process

When deficiencies are discovered sooner, time and money are saved. The prototyping system is not without cost, of course, but these additional costs at the beginning of lifecycle will improve requirements definition, design, and coding process such that the overall cost is reduced. The prototying process is illustrated in Figure. 1.1 [Ber90][Luq89][Mye92].

1.5 Motivation

It is widely acknowledged that providing software is both difficult and expensive. To help remedy this situation, many methods for specifying [Geh86][Gut78][Gut85][Kem83] and verifying [Gut78][Hoa69][Jon80][Loe84] software have been developed. One partial solution to this problem is the use of software engineering environments that integrate a number of tools, methods, and data structures to provide support for program development and/or maintenance

[Laf85][Sta84]. A research environment based on an object-oriented model has been initiated with the definition of **DOSL** [Lee90]. A formal specification language and integrated object-based environment for distributed system have been defined. At this stage there are no existing tools to support prototyping. In this research we define a specification environment that supports the prototyping of a distributed system using an object oriented model. We thus provide an environment that allows the system designer to work at the specification level [Dol90]. A prototyping environment provides the possibility for exploring numerous techniques. For example, we can develop a library of objects that can be verified using the prototyping tools and then saved for future reuse. **DOSL** models can be tested using the prototype techniques.

1.6 Scope of Research

We have presented the need for an integrated software development that includes a formal specification language and a prototyping system. Requirement documents often use natural language which is imprecise and ambiguous. Formal specification languages provide notations which can give unambiguous descriptions of the specifications. Formal specifications aim to increase the quality and reliability of software products by better being able to detect and correct conceptual flaws. In addition to the formal specification language, a prototype system that can execute the formal specification language provides a better environment to detect ambiguous problem. If the formal specification language is object-oriented, then the specifiers can define a system as a set of modules which provide services. The object-oriented specification encourages the development of reusable modules.

This research was initiated by a study of the formal Distributed Object-based Specification Language (**DOSL**), an object based formal specification language. The focus of this research is to:

1) modify **DOSL** specification from an object-based to an object-oriented;
2) extend the definition of the **DOSL** to include the definition of class;
3) provide necessary constructs to support the inheritance and object communication;
4) introduce standard Input/Output (**I/O**), stream and concurrent I/O;
5) define a methodology for executing **DOSL**; and
6) develop a prototype system that executes **DOSL** specification language.

An object-oriented specification language is a specification language that is object-based while also maintaining the idea of inheritance. **DOSL** is an object-based language that does not support inheritance.

The outline of this dissertation is as follows. In Chapter 2, related works are discussed. The **DOSL** extensions are given in Chapter 3 where we present an overview of **DOSL**, and the formal definition of the class object of **DOSL-II**. We also show a sample problem in **DOSL-II** using class object and inheritance. In sections 3.5 , we present new **I/O** constructs and discuss general problems associated with concurrent **I/O**. Concurrent **I/O** support for **DOSL-II** is presented and an example is given in section 3.11. In section 3.6, we introduce new communication constructs for **DOSL-II**, and an example is given in section 3.7. The methodology for executing **DOSL-II** is introduced in Chapter 4 where an overview of **ACT++** along with an example of concurrent I/O in **ACT++** is presented. In section 4.3 we have introduced a methodology for executing **DOSL-II** specification languages. We also have discussed LR(1) parsing algorithm and introduced a series of procedures which recognizes the **DOSL-II** specification language according to its syntax and then transforms it into **ACT++** code. **DOSL-II** constructs and its transformation code in **ACT++** are presented in section 4.6. In Chapter 5 the testing of the transformation process for the prototyping system is described. Finally, the summary and future research are presented in Chapter 6.

Chapter 2

A Specification Environment That Supports the Prototyping of Distributed Systems

2.1 Introduction

Large system development such as distributed systems requires a proper problem understanding and requirements statement before costly development starts. If complete requirements exist and developers completely understand those requirements, then they can more easily develop the correct system; however, requirements evolve and change over time.

Traditional approaches generally result in misunderstandings in the analysis stage being propagated to mistakes in the final system [Bis89]. The fixing of faults in the latter stages of the development lifecycle is expensive. Application prototyping is an alternative to reduce the development errors and thus the cost [Asu93][Did93][Kor93][Deh93][kor92]. Prototyping encourages the customers to take an active part in the development process and increase the likelihood that the final system will be validated to meet the customers' needs.

Many prototyping goals, objectives, and approaches are possible [Luq93][Luq92]. Executable formal specifications are the most attractive one. This unification of specification and prototyping by having code generators has advantages of providing consistency and prototyping at higher levels of abstraction.

2.2 Related Works

2.2.1 Related Actor-Based Concurrent Languages

The main focus of this research is i) to define an executable model for Distributed Object-based Specification Language (**DOSL**) [Lee91]; and ii) to expand **DOSL** to support inheritance and concurrent **I/O**. A brief discussion of actor-based concurrent languages is presented below.

The actor model of concurrent computation was first introduced by Hewitt [Hew77] and extended by many others [Bak77][Atk79]. More recently, [Agh86] has extended the actor model with a small number of powerful primitives. An actor has its own mail box where messages are queued. The behavior of an actor, called a *script*, performs an actions according to the message sent to it. The actor model provides "inherent concurrency" which means that the concurrency aspects are expected by the structure of programs [Agh86]. Since the introduction of the actor model, many languages have been proposed for programming concurrent computation using actors.

ABCL/1 (An object-based Concurrent Language 1) is intended to serve as an experimental programming language to construct software in the framework of object-based concurrent programming. It is also intended to serve as an executable language for modeling and designing various parallel and /or real-time systems. Thus **ABCL/1** also serves as a language for rapid prototyping [Yon90]. Furthermore, the application domains include AI fields. This language is also an executable thought-tool for developing the paradigm for distributed problems solving. **ABCL/1** is designed for describing distributed algorithms and modelling various types of distributed systems. They are three different message passing types: ***past***, ***now***, and ***future***. The ***past*** and ***future*** types are similar to asynchronous message passing and the ***now*** is close to synchronous message passing. The computation model and an

overview of **ABCL/1** are found in [Yon86]. Like other actor-based languages, **ABCL/1** modifies the actor [Agh86] semantics to conform with the requirements of the application domain. The significant modifications are:

a. Messages between two actors are ordered.

b. "Express" messages allow preemption.

c. Both remote procedure call and future style message passing are provided.

Act-1 [Lie87] is a Lisp-based language developed at **MIT**. In this language, the primitive mechanisms of Lisp are presented as actors. Sharing of abstraction is supported through a delegation mechanism. Highly parallel and distributed artificial intelligence applications are supported by the use of the actor model. Concurrency in **Act-1** is generated by the use of futures and restricted by the use of serializers and guardians. The use of serializers and guardians achieves the effect of the mail queue mechanism of **Agha's** actor model. However, the reusability issue is not addressed by **Act-1**.

The **Actra** language [Bar87] is used at Carleton University. This language pioneered the application of object-oriented techniques in embedded, real-time systems. **Actra** is implemented as an extension of Smalltalk. A class is added to the Smalltalk hierarchy that implements the basic actor abstraction. The actor abstraction in **DOSL** and **Actra** are quite different. First, the **Actra's** actor interface is fixed. Second, **Actra's** message passing semantics are synchronous (unbuffered), while those of **DOSL's** message passing semantics are both synchronous and asynchronous. One final difference between **Actra** and **DOSL** is the intended architecture. **Actra** uses a shared memory multiprocessor while **DOSL** is intended for use in a non-shared memory distributed environment.

A related but independent language is **Actalk** [Bri89], a small language kernel built into Smalltalk-80. **Actalk** is intended for classifying and simulating actor languages using a single framework. The approach used in **Actalk** for creating "activeness" is very close to that of **DOSL-II**.

ACT++ [Kaf90][Kaf88][Kaf89] is a programming environment in which concurrent programs can be written in **C++**. The current **ACT++** design extends **C++** [Str86] with a class hierarchy which provides the abstraction of the actor model of concurrency. The primary design goal of **ACT++** is to support software reusability through the class inheritance of an object-oriented language. The objects interface in **ACT++** are replaceable. Like **DOSL**, **ACT++** is intended for use in non-shared memory distributed environment. **ACT++** will be discussed in more detail in Chapter 4.

2.2.2 Related Executable Specification Languages

Many formal specification language have been previously proposed, designed, or put into use [Aue86][Ber87][Gut78][Gut80][Geh85][Gom81][Hen86][Lee91]; they can be either formally based or informal to incorporate natural language and graphics. The formally based methods can be roughly divided into model-oriented , axiomatic, and property-oriented approaches [Gut78], although languages that combine the two methods have also been proposed [Gut85]. Model-oriented specification language stress the internal behavior of the system while the property-oriented language emphasize the specification of constraints of a system. **DOSL** primarily follows the model-oriented approach, but it also uses the property-oriented approach to specify the system constraints.

PLEASE [Rob89] is an executable specification language that supports program development by incremental refinement. **PLEASE** is a model-oriented approach; in other words, components are described in terms of predefined types and operations. **PLEASE** is part of the **ENCOMPASS** environment that provides automated support for all aspect of the software development process. A **PLEASE** specification is transformed into a prototype that uses **Prolog** to "execute" pre- and post condition. In contrast, a **DOSL** specification is transformed into a prototype system that uses **ACT++** to "execute" **DOSL**.

The formal specification languages which have been developed to specify the behavior of distributed systems include **CSP** [Hoa85], **CCS** [Mil80], **DOSL** [Lee91], and **Unity** in the model-oriented approach. **LOTOS** [Eij89] and Lamport's transition axiom [Lam89] are examples of the property-oriented approach. The formal specification languages for sequential systems include **VDM** [Jon80], **Z** [Spi88], **Larch**, and **OBJ**. With the exception of **OBJ** and **DOSL**, none of the above specifications are executable. Two representative specification languages, **CSP** for distributed and **Z** for sequential environments, are described below.

Communicating Sequential Processes (**CSP**) is a language framework for concurrent programming which is suitable for distributed environments [Hoa85]. The following concepts are central to the language.

a. A **CSP** program consists of a fixed number of sequential processes that are mutually disjoint in address space.

b. Communication and synchronization are accomplished through the input and output constructs.

c. The sequential control structure is based on Dijkstra's *guarded command* [Dij75].

The **Z** (pronounced Zed, not Zee) is a formal specification language developed at the University of Oxford [Spi89]. **Z** is based on typed set theory because sets are mathematical entities whose semantics are formally defined. However, it includes a number of constructs which specifically support formal system specification. The developers of **Z** recognized the importance of both presentation and specification reuse. **Z** allows specifications to be highlighted graphically and integrated with other specifications.

Formal specifications can be difficult and tedious to read, especially when they are presented as large mathematical formulae. **Z** specifications are normally presented in small, easy to read chunks (called schemas) which are distinguished from associated commentary using graphical highlighting.

2.2.3 Related Prototyping Languages

A number of different high-level languages have been used for prototyping. In this section, a number of prototype languages are presented along with a recommendation for the most appropriate application domain where these languages can be applied. However, the domains suggested are not exclusive and the languages may be used for prototyping other classes of application system. These prototype languages includes **LISP** (based on list structures), **Prolog** (based on logic), **Smalltalk** (based on objects), C++ (based on objects), **APL** (based on vectors), and **SETL** (based on sets). They are useful prototyping languages because their dynamic features mean that rapid system development is possible. We also include wide-spectrum languages which combines a number of paradigms.

One of the most powerful prototyping systems for user interfaces is the **Smalltalk** [Gol83]. **Smalltalk** is an object-oriented programming language which is tightly integrated with its environment. This environment includes a graphical user interface.

Most system interaction is via menus, where selections are made by pointing with a mouse. C++ is an object-oriented programming language with a rich library. An object-oriented programming languages such as C++ and **Smalltalk** are excellent prototyping languages for two reasons:

1. The object -oriented nature of the language means that systems developed in the language are resilient to change. Rapid modification of the system is possible without unforeseen effects on the rest of the system.

2. All of the objects defined perviously are available to the programmer. Thus, a large number of reusable components are available which may be incorporated in a prototype under development.

A class of programming languages which has been proposed as programming languages are so-called multiparadigm programming languages. Example of such languages are **Gist** [Bal82], **EPROL** [Hek88], and **LOOPS** [Ste86]. Most languages are based on a single paradigm. For example, **LISP** is based on functions and lists, **Prolog** is based on facts and logic. By contrast, a multiparadigm language is a programming language which combines a number of paradigms rather than a single paradigm. It may include objects, logic programming, and imperative constructs. Although there has been a good deal of interest in such languages, the practical problems of developing efficient implementations have meant that few commercial language products are available.

Gist and its commercial derivative **REFINE** [Smi85] are perhaps the most developed multiparadigm language. **Gist** is a non-deterministic language in which the user writes a formal, executable, specification of the system to be prototyped. This specification is refined by the user with automated assistance to produce an executable system prototype. **Gist** incorporates concepts from logic programming, functional programming and imperative programming languages. A **LISP** implementation of the system is generated by the **Gist** processor.

As an alternative to using a multiparadigm language, a mixed-language approach to prototype development may be adopted. Different parts of the system are programmed in different languages and a communication framework is established between the parts. Zave [Zav89] describes this approach to development in the prototyping of a telephone network system. Four different languages are used: **Prolog** for database prototyping, **Awk** [Aho88] for billing, **CSP** [Hoa85] for protocol specification, and **PAISLey** [Zav86] for performance simulation.

There is no single ideal language for prototyping large systems as different parts of the system are so diverse. The advantage of a mixed-language approach is that the most appropriate language for the logical part of the application can be chosen, thus speeding up prototype development. The disadvantage is that it is may be difficult to establish a communication framework which allow multiple languages to communicate.

The goal is to have a rapid, and correct development of complex system. In this Chapter, related works concerning actor-based concurrent languages, executable specification languages and prototype languages have been discussed. In Section 2.2.1 on actor-based concurrent languages, we presented the concept of an actor as a model of concurrency and placed **DOSL** and **DOSL-II** in that category. We then compare the features of **ABCL/1** with **Act-1, Actra, Actalk**, and **ACT++**. In Section 2.2.2 the executable specification **PLEASE** was compared with **DOSL** and other formal specification languages, namely, **CSP**, **LOTOS**, **Unity** and **Z**. Finally, in Section 2.2.3 we have summarized a number of different high-level languages that have been used for prototyping and suggest their application domain.

Chapter 3

DOSL Extensions

3.1 Introduction

The overall goal of this research is to extend **DOSL** from an object-based to an object-oriented specification language, and to expand the environment for software development of distributed systems by extending the support for **DOSL** to include a prototyping system.

In this Chapter we present the extensions of the **DOSL** specification language to support class, inheritance, concurrent I/O and communication constructs. Once a class has been defined, any number of objects of that class are easily created. Designers and programmers are thus encouraged to reuse code by defining general purpose classes and to use them in many different application. A new class can be derived from one or more existing classes and can inherit some or all of their properties. This further encourages code reuse because classes derived from a general purpose class can be customized as needed for each particular application. The benefits of inheritance include reusability, code sharing and consistency of interface. In **DOSL**, a detailed inheritance mechanism is not included. Also, the **DOSL** specification language does not provide facilities for input or output. In the following section we present an overview of **DOSL**, and we then define the formal definition of the class object of **DOSL-II**. We also show a sample problem in **DOSL-II** using class object in section 3.8 and inheritance in section 3.9. In sections 3.5 , we present new **I/O** constructs and discuss general problems associated with concurrent **I/O**. In section 3.5.5, concurrent **I/O** support for **DOSL-II** is presented and an example is given in section 3.11. In section 3.6, we introduce new communication constructs for

DOSL-II, and an example is given in section 3.7. Finally, we present the formal definition for all extended constructs in section 3.10.

3.2 An Overview of DOSL

The Distributed Object-based Specification Language (**DOSL**) is defined by Lee [Lee91]. The primary features of **DOSL** are message-passing constructs, data abstraction, concurrency, nondeterministic execution patterns, object constraints using temporal logic, and message priority.

The formal definition of syntax of **DOSL** is presented below in extended Backus-Naur Form.

```
<parallel-module>: :=<dist-module> | <dist-module><parallel-module>

<dist-module>::=<module> | <module> <par-op> <module>

<module> ::= ObjectModule :: <object>
                        <definition-section>
                        <constraint-section>
                        <body-section>
                        End

<definition-section> :: Definition is
                          type      : <type>
                          class     : <object-list>
                          visible   : <object-list>
                          variable  : <declaration-sequence>
                       method   <method-declaration>

<type> ::=active | passive

<object-list> ::=  <object> | <object>  <object-list>

<object> ::= [<identifier>]

<delcaration-sequence> ::= <delcaration>
                              | <delcaration> <delcaration-sequence>
<declaration> ::= <identifier> {:=<data-type>} ;

<data-type> ::=integer | real | string | boolean | <array-type>
```

<array-type> ::= array [integer..interger] of <data-types>

<method-declarations> ::= <method>
| <method> <method-declaration>

<method> ::=method <identifer> ( ) : <op-sequence> --> <return-value>

<op-sequence> ::= <identifer> {x <identifier> }

<return-value> ::= nil | <identifier>

<constraint-section> ::=constraints
<logic-exp-sequence> | O

<logic-exp-sequence> ::= <templogic-exp>
| <templogic-exp> <logic-exp-sequence>
<templogic-exp> ::= {<temp-op>} (<logic-expression>) ;

<temp-op> ::= _ (always) | O (next) | (eventually) | --> (until)

<par-op> ::= ||

<logic-expression> ::=<sexpression> <relational-operator> <sexpression>

<sexpression> ::=<term> | <signed-term> | <additive-expression>

<term> ::=<factor> | <multiplying-expression>

<factor> ::=<variable> | <string> | <number> | <bracked-expression>
| <not-expression>
<bracked-expression> ::= (<expression>)

<not-expression> ::= <factor>

<relational-operator> ::= | = | < | > | <= | >=

<string> ::= <letter> | <letter> <string>

<number> ::= <integer> | <real-num>

<signed-term> ::= <sign> <term>

<multiplying-expression> ::= <sexpression> <multiplying-ops> | <sexpression>

```
<multiplying-ops> ::= * | / |

<additive-expression> ::= <sexpression> <adding-ops> <sexpression>

<adding-ops> ::= + | - |

<sign> ::= + | -

<body-section> ::= Body is
                        <declaration-method>

<declaration-method> ::= <method-exp>
                         | <method-exp> <declaration-method>

<method-exp> ::= (=><n> [:<identifer> {<iden-list>)}]
                  {when <logic-exp-sequence>};
                      begin
                          <statement-sequence>
                          | <guardcommand-sequence>
                       end)
                    {;;}
<n> ::=<digit>

<iden-list> ::=<identifier> | <identifer>  <iden-list>

<statement-sequence> ::= <statement>
                                | <statement> <statement-sequence>

<statement> ::= skip | abort | stop
                  | <assignment-statement>
                  | <communication-statment>
                  | <if-statement>
                  | <while-statement>

<assignment-statement> ::= <identifier> := <expression>;

<communication-statement> :=
  {<temp-op>} ({<identifier> :=} (<object> <=[:<identifier> {(<iden-list>)}]));

<if-statment> ::=if <condition> then <statement-sequence>
              else <statement-sequence> fi

<while-statement> ::= while <condition> do <statement-sequence> od

<guardcommand-sequence> ::= <gcommand>[
                                  <guard-sequence>  ]
```

<gcommand> ::=select | repeat

<guard-sequence> ::= <guarded-command>
| <guarded-command> <guard-sequence>
<guarded-command> ::= <condition> --> <statement-sequence>
{[]}

<condition> ::= <logic-expression>
| <logic-expression> <condition>

<integer> ::= <digit> | <digit> | <integer>

<real-num> ::= <integer> [. {integer> }] [E [<sign>] <integer>]

<identifier-list> ::=<identifier> | <identifier> <identifier-list>

<identifier> ::= <letter> <ident>

<ident> ::= <letter> | <digit>
| <letter> <ident> | <digit> <ident>

<letter> ::= A | B | .... | Z | a | b | c | ... | z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

A **DOSL** specification language consists of object modules that communicate with one another via a message passing mechanism. Figure 3.1 shows an object module skeleton.

In **DOSL** each object module has its own process. The **visible** part of the definition identifies those object names that have access rights to the operations of this object. The **variable** section is used to define private variable(s) of an object .

The **Methods** section specifies how the visible object can call this object and the **Constraints** section enforces the existing constraints on the object and its operation. The implementation details of each operation provided by an object are done in the section preceded by **Body is**.

```
ObjectModule :: [objectname]
 Definition is

    type    :-- passive or active
    class   :-- this slot is open, DOSL does not support class or inheritance
    visible :-- list the object(s) name that are visible to this object module
   variable :-- define private variable(s) of the object

 Methods
          :-- an object operation is defined in terms of signatures.

 Constraints

          :-- specifies the constraints on the object in form of temporal logic.

 Body is
            (=>P[:method-name]
       begin
              statement;
              statement;
              ...
              statement
        end)
;;
:-- more method-name
.....

End.
```

Figure 3.1 An Object Module Skeleton.

Temporal operators and temporal logic expressions are used to specify communication patterns among objects as well as constraints on objects of **DOSL**. The temporal operator symbols and their meaning are as follows:

_ : always true in future

O : the next state is true

^ : sometimes or eventually true in future

When the above temporal operator precedes the communication statement, it indicates the type of message passing methods. The operators _ and ^ are used to specify on asynchronous message passing method, where O is used to represent a synchronous message passing method. For example, suppose an object X wants to send a message to an object Y. It can be done in three different ways:

1. An object X sends a message to an object Y and continues its execution without expectation of any response from object Y. In this case, the operator _ is used by object X, and the method is asynchronous.

2. An object X sends a message to an object Y and continues its operation with an expectation that eventually, in the future, it will receive a response from an object Y. This method is also called asynchronous, but the operator ^ is used by an object X.

3. When the operator O is used by an object X to send a message to an object Y, the execution of an object X is suspended until an acknowledgement is received from an object Y. This method is called synchronous message passing.

3.3 Definition of Object-Oriented Extensions to DOSL

DOSL is extended to support **class** object and **inheritance** as follows:

<definition-section> :: Definition is
type : <type>
class : <object-list>
visible : <object-list>
variable : <declaration-sequence>
method <method-declaration>

<object-list> :: <object> | <object><object-list>
<object-list>:: [<identifier>] | <class-specifier>
<class-specifier> :: <class-head > { <member-list> }
<class-head> :: *class* <identifier> | *class* <base-spec>
<base-spec> :: :*public* <identifier> | :*private*<identifier>
<member-list> :: *protected* : <declaration-sequence> <object-operation>
| *public* : <declaration-sequence> <object-operation>

<object-operation> :: <identifier> (<identifier-list>)
|<identifier> (<identifier-list>);<object-operation>
<declaration-sequence>:: <declaration>
|<declaration> , <declaration-sequence>
<identifier-list> :: <identifier> | <identifier> , <identifier-list>

3.4 Semantics of the Class and Inheritance Features

3.4.1 Class

A *class* is the direct extension of the notion of an abstract data type; it is a template from which objects can be created. Every object is an instance of some class. Objects that have the same set of operations and the same state representations are considered to be of the same class.

The keyword **class** is used to declare a class. The keywords **protected** and **public** are used to declare some members protected and others public. All declarations following one of these keyword are protected or public, respectively, until another such keyword is encountered. For example a class account is declared as follows:

```
class account {
    protected :  balance : real;
                 rate    : real;
    public:
        account ( bal, pcnt);
        deposit ( amt);
        withdraw ( amt);
        compound();
        getbalance();
};
```

With this declaration, the instance variable **balance** and **rate** are protected; attempts to manipulate their values directly are disallowed. Member functions, on the other hand, are public, so they can be called. The ***constructor*** is a member function that returns an initialized object and has the same name as the class name. To allow

initialization while preserving data hiding a ***constructor*** to the class declaration of **account** is added. For example the following statement:

account acctone (1000.0, 0.6);

declares **acctone** as an account object with a balance of 1000.0 and a periodic percentage rate of 0.6.

To allow inheritance, a new class can be defined by extending or modifying an existing class. In this case, the new class is called a ***derived*** class and the parent class is called a ***base*** class. For example, in Figure 3.2 the derived class for savings accounts is declared as follows:

```
class savacct : public account {
    protected :
                    rate      : real;
    public:
        savacct ( bal, pcnt);
        withdraw ( amt);
        compound();
      };
```

In declaring a class, inheritance is specified by following the class name by a colon and a list of base classes; for single inheritance, this list names only one base class (account).

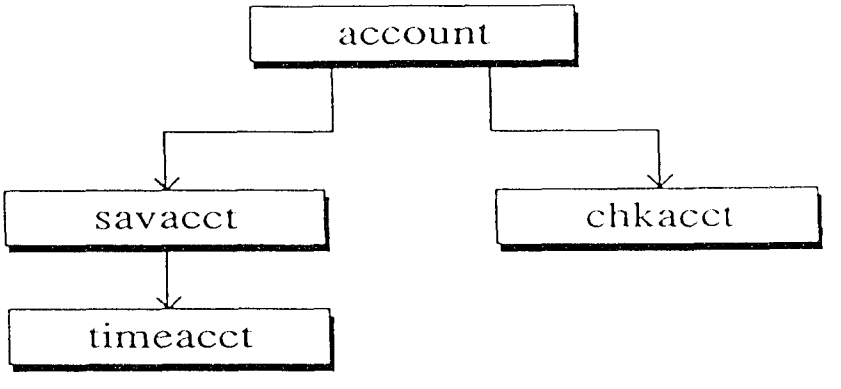

Figure 3.2 The Class Hierarchy.

Each base class is specified as ***private*** or ***public*** by preceding its name in the base-class list with the appropriate keyword.

If a base class is public, the public members of the base class are also public members of the derived class. Thus users of the derived class can refer to public members defined in the base class. If the base class is private, however, the public members of the base class become private members of the derived class. They can be accessed by member functions of the derived class, but they are not accessible to users of the derived class.

Generally, a public base class is used if we want the public member functions of the base class to also be available to users of the derived class. A private base class is used if we need to provide a different set of functions for users of the derived class and want to block access to the functions defined in the base class. In class **savacct**, class **account** is designated as a public base class so that the public member functions **deposit ( )** and **getbalance ( )**, defined for account, are also be available to users of **savacct**.

The accessibility of members inherited by a derived class depends on their accessibility in the base class and on whether the base class is private or public. Protected and public members of public class are inherited as, respectively, protected and public members of the derived class. Protected and public members of a private base class are inherited as private members of the derived class.

Note that a member inherited from a public base class has the same protected or public status in the derived class that it had in the base class. If all base classes in the hierarchy are public, then a class member declared as protected or public will retain that status in any class that inherits it; this is true regardless of whether the member is inherited directly from the class that defined it or indirectly from another class that also inherited it.

3.4.2 Inheritance

Inheritance allows a new class to be defined by extending or modifying one or more existing classes; the new class is called a ***derived*** class and the parent classes are its ***base classes***. A derived class can itself serve as a base class for other derived classes, enabling us to build hierarchies of classes related by inheritance . These four classes of account can be arranged in a ***class hierarchy*** as shown in Figure 3.2. At the top of the hierarchy is account, from which the other three classes are derived either directly or indirectly. Class **account** is not useful by itself, but is intended only as a starting point (base class) for deriving more specialize account classes. Both **savacct** and chkacct are extensions of account and so are derived directly from it. Class **timeacct** defines modified form of savings account and so is derived from **savacct**.

In Figure 3.2, each arrow connects a base class to a derived class; the direction of the arrow is the opposite direction of inheritance. Note that **account** serves as a base class for both **savacct** and **chkacct**, and **savacct** is both a derived class of **account** and the base class of **timeacct**.

Inheritance encourages reusability by allowing the use of existing classes as foundations on which new classes can be build. *Single inheritance* occurs when a derived class has only one base class and *multiple inheritance* occurs when a drived class has several base classes.

3.5 Input/Output Extensions

Designing and implementing a standard input/output facility for a language is difficult. It is even more difficult to provide an input/output (**I/O**) facility for a nontrivial language such as **DOSL**, that requires concurrent **I/O** and has many user-defined types and classes. The **DOSL** specification language does not provide facilities for input or output. As a part of this research, we have added **I/O** constructs to the **DOSL-II** specification language. These extensions support both standard **I/O** and concurrent **I/O**. In the following sections, the syntax and semantics of each construct are given along with an example. Also, problems related to concurrent input and output are discussed.

3.5.1 Standard Input/Output

The **doslin** and **doslout** are implemented for **DOSL-II** as standard input and output statements respectively with the following syntax:

doslin(argument-list);

doslout(argument-list);

The aurgument list is a quoted string followed by list of variable separated by comma. The quoted string contains a field descriptor, such as, **%d, %s, %f, %c** that will read/write integer, string, floating point, and character respectively. In addition, the control character "\n" causes the printer to skips to a new line. For example the statement,

doslout("line.....1 \n line.....2 \n line.....3\n"):

will print

line.....1
line.....2
line.....3

where doslout("line...1line...2line...3"); prints line....1line.....2line.....3

3.5.2 Stream Input/Output

The stream **I/O** facilities provided for **DOSL-II** are exclusively concerned with the process of converting typed objects into sequences of characters, and vice versa. There are other models for **I/O**, but this one is fundamental; and many forms of binary **I/O** can be handled by considering a character as simply a bit pattern and ignoring its conventional correspondence with the alphabet. The key problem for the programmer is then to specify a correspondence between a type object and an essentially untyped string. We have provided **DOSL-II** with three standard stream **I/O** constructs. They are **din**, **dout**, and **derr** used for input, output, and standard error output stream, respectively. We also adopt two operators, << and >>, from **C++**. The input operator >> means "get from" and the output operator << means "put to". The following examples shows how each construct works.

The standard input stream din and an extraction operator, >>, are used for extracting values from the stream and storing them in variables. If **din** is not explicitly redirected, the input will come from the user's keyboard. The type of a variable in **DOSL-II** determines the type of the input value. Thus

```
n : real;
din >> n;
```

read a real value into variable **n** and

```
x : integer;
din >> x;
```

read an integer value into variable **x** and

```
s : string;
dout << "password: ";
```

din >> s;

prints "password: " without the quotation marks and reads the string value into variable s from the same line. Note that the extraction and insertion operators >> and << each point in the direction of dataflow, either away from or toward the stream. The stream I/O allows the programmer to put out a sequence of objects in a single statement, for example,

derr << "x = " << x << '\n';

where derr is the standard error output stream. So, if x is an integer with the value 130, this statement would print

x = 130

and a newline onto the standard error output stream. The precedence of << is lower than arithmetic operators. This allows using arithmetic expressions without parentheses. For example: **dout << "a*b+c" << a*b+c << '\n';**

prints a*b+c followed by the value of an expression.

3.5.3 Problems With Concurrent Input/Output

The implementation and performance of concurrent I/O on a system encounters the following problems:

1. The interference between concurrent sequences of I/O operation directed at the same file;
2. The complexity of avoiding the blocking effects of low-level I/O system calls; and
3. The consistency of the interpretation of I/O commands executed in different process contexts.

The first problem is inherent to concurrent computation and is simply another instance of the general problem of interference among concurrent activities over their access to shared resources. **UNIX** ensures **I/O** calls are non-preemptive, that is, if **I/O** is initiated by a process on a file descriptor, the corresponding file table entry will remain locked until the **I/O** is complete. But in between system calls, there is no such locking available. As a result, different interleaved executions of **I/O** calls might lead to different results in different executions of the same program. The solution suggested to this problem is to do **I/O** from a ***critical section*** or to use an **I/O** ***server*** to do all **I/O**. Neither solution reduces the burden on the user. In the case of the server, the user must define the server and make sure the server does not become a performance bottleneck. In the case of using critical sections, there is no concept of encapsulation. The user is in charge of all locks and unlocks of the critical sections, which may easily introduces an errors to the system. Using an **I/O** server is somewhat more attractive because a server encapsulates all low level operations and provides high level abstractions for the user.

The second problem is the complexity of avoiding the blocking effects of low-level **I/O** system calls. The **I/O** features available in **UNIX** are a set of system calls like, ***read***, ***write***, ***open***, and ***close***, which perform **I/O** using a unique identifier called the file descriptor. The read and write system calls are blocking calls. This means that if the **I/O** is not possible immediately when the system call is made, the process making the **I/O** call will be interrupted and placed on hold until **I/O** is possible. As a result of the blocking nature of the **I/O** calls, applications doing real-time monitoring of the external world might miss important external events if all the processes that were running the application block on **I/O**.

UNIX also provides non-blocking **asynchronous I/O** facilities for *terminals* and *sockets*. But there is no construct that hides the details for doing asynchronous **I/O**. That is, to perform asynchronous **I/O** the user must do the following:

1. Write a signal handler for the **SIGIO** signal that the operating system will deliver to the process when the **I/O** is ready,

2. Set up the file descriptor for asynchronous **I/O** by using a special option of the *fcntl* system call, and

3. Identify the process or process group to which the **SIGIO** signal will be delivered.

The third problem is the consistency of file descriptors across processes. The file descriptors are reference to a process specific table called a ***file descriptor*** table. Thus, a file descriptor is meaningful only in the context of the single process in which it was created. In the case of multiprocess the run time system must ensure that a thread that executes for example an open call on a particular process is scheduled to run on the same process throughout its lifetime.

3.5.4 DOSL-II, Concurrent Input/Output

The following concurrent **I/O** constructs are added to **DOSL-II** based on the prototype system (see Chapter 4):

readsc (fname, actorname, buffername);
readasc(fname, actorname, buffername);
writesc(fname, actorname, buffername);
writeasc(fname, actorname, buffername);

The *readsc* and *readasc* construct are provided for the concurrent Read operation synchronously or asynchronously . It takes three arguments, fname for the

device name, actorname and buffername to name the actor and the buffer for the Read operation. Similar constructs are offered for the Write operation. They system prototype hides the details of these operations and thus provides **DOSL-II** with a higher **I/O** abstraction. Making the abstractions executable offers a powerful form of software prototyping(see Section 3.11).

3.6 DOSL-II, Communication Constructs

In this section, we include new communication constructs in **DOSL-II**. A request message is used for sending a request to another actor. A request message consists of the name of the method to be executed by the receiving actor and arguments for invoking the method. A request message is send by the following *send* construct:

send(argument1, argument2,...);

The send construct needs the Mbox (the mail queue) of the receiver and the message to be sent. Request messages are buffered in the mail queue (Mbox) of the receiving actor. An actor can refer to its own Mbox using the pseudo variable *self* . Each actor can process only one request message in the Mbox.

If the sender of a request wants to receive the result of the method invocation, it may provide a Cbox (the repository of reply messages is called a Cbox) name (see Figure 3.10) in the request message. The following *reply* construct is used to transmit a reply message containing the result:

reply(aurgument-list);

The name of a Cbox specified in a request message is called the reply destination [Yon87]. Since an actor knows the reply destination when it reads a request message, the reply destination needs not be explicitly provided by the programmer in the reply construct. If the sender, A, does not provide its own Cbox in a request message, a *reply forwarding* occurs. The reply is not delivered to the actor A. It is delivered to the actor who sent the current request message being processed by the actor A .

An actor can read from Cbox using the following receive construct:

receive(aurgument-list);

If a reply is available in the Cbox, it is immediately delivered to the actor. Otherwise, the receive operation blockes the caller until a reply arrives.

3.7 Definition of New Constructs

In this section we present a formal definition of DOSL extensions for the Standard Input/Output, Stream Input/Output, Concurrent Input/Output, and Communication constructs in extended Backus-Naur Form as follows:

<statement-sequence> ::= <statement>
| <statement> <statement-sequence>
<statement> ::= skip | abort | stop
| <assignment-statement>
| <if-statement>
| <while-statement>
| <inputoutput-statement>
| <communication-const>

<inputoutput-statement> ::= <standardio> | <streamio> | <concurrentio>
<standardio> ::= doslin (<identifier-list>) ; | doslout (<identifier-list>);
<identifier-list> ::=<identifier> | <identifier> , <identifier-list>
<identifier> ::= <letter> <ident> | <qstring >

<ident> ::= <letter> | <digit>
<qstring> ::= " <string> "
<string> ::= <letter> | <letter> <string>
<streamio> ::= din <argument-list> ; | dout <argument-list> ; | derr <argument-list>;
<argument-list> ::= <argument> | <argument> <argument-list>
<argument> ::= << <identifier-list>

<concurrentio-statement> ::= readsc (<identifier-list>);
| readasc (<identifier-list>);
| writesc (<identifier-list>);
| writeasc (<identifier>);

<communication-const> ::= send (<identifier-list>); | receive (<identifier-list>);
| reply (<identifier-list>) ;

3.8 Example of Class Feature

Figure 3.3 shows a program written in **DOSL-II** for computing the amount in a bank account after a given number of months. Interest is compounded monthly and a fixed amount is deposited at the beginning of each month. Given are starting balance (initial value of **balance**), the monthly deposit (**deposit**), the interest rate (**anpcntrate**), and number of months (months) for which deposits will be made and interest compounded. The monthly decimal rate (**rate**), which is equal to annual percentage rate divided by 1200 is used in the calculation.

The following program in **DOSL-II** defines a class of objects to represent bank accounts. To put the module account to work we need a driver module, called module main (see Figure 3.5). In the main module we first create a bank-account object with a certain initial balance and interest rate. A message is repeatedly send to the module account to accept deposits and compute interest. Finally a message asking the object to reply with its current balance is sent.

```
:-- ********************************************
:-- The following DOSL-II program defines a class of
:--  account to represent bank accounts with account, deposit
:--  , withdraw, compound, deposit and getbalance operations
:--  available to the public.
:--  ********************************************
ObjectModule ::[account]
  Definition is
     type : passive;
     class : class account {
              protected :    balance : real;
                             rate      : real;
      public:
             account ( bal,  pcnt);
             deposit (  amt);
              withdraw (  amt);
             compound();
             getbalance();
     };
      visible  : [main];
   Methods
        method  account ();
        method  deposit ();
        method withdraw ();
        method compound();
        method getbalance();
   Body is
:--  ***************************************
:--   Open account with starting balance bal and
:--   priodic percentage rate pcnt.
:-- ***************************************
     (=> [:account::account ( bal,   pcnt)]
      begin
             balance = bal;
             rate = pcnt /100.0;
      end;
     ) ;;
:--***************************************
:-- Deposit amount amt
:--***************************************
    (=>[: account::deposit( amt) ]
      begin
             balance = balance + amt;
      end;
    ) ;;
```

```
:--*****************************************
:-- Attempt to withdraw amount amt
:--*****************************************
    (=>[: account::withdraw( amt)]
      begin
            if ( amt <= balance ) then
             balance = balance - amt;
              return amt;
             else
                return 0.0;
            fi
       end;
     ) ;;
:--*********************************************
:-- Compute interest for current period and add to balance
:--*********************************************
     (=>[:account::compound()]
       begin
             interest = balance * rate;
            balance = balance + interest;
       end;
     ) ;;
     (=>[:account::getbalance()]
        begin
            return balance;
         end;
     ) end.
```

Figure 3.3 The ObjectModule Account.

Every instance of a bank account must keep track of two values: the current current balance in the account and the interest rate. These values are will be stored in the two variables, **balance** and **rate**. These variables are called ***instance variables*** because a separate set of them is needed for each instance of a bank-account (see Figure 3.4). For example we can create two account, acctone and accttwo as follows:

```
account acctone;
account accttwo;
```

As illustrated in Figure 3.4, each instance of account is composed of two instance variables, **balance**, and **rate**, which were declared as member variables of class **account**. The class members associated with a class object are referred to by using operator, . (dot or period). Thus **acctone.balance** refers to instance variable **balance** of **acctone**, **accttwo.rate** refers to the instance variable **rate** of **accttwo**, and so on.

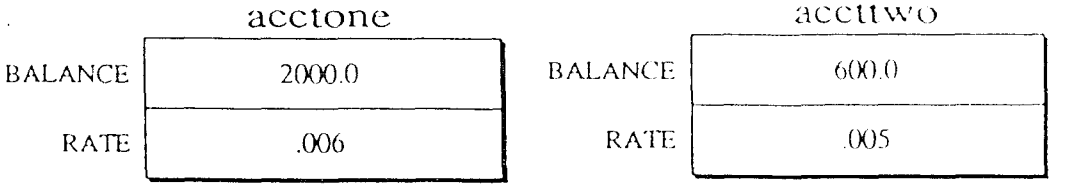

Figure 3.4 Two Instances of Class Account.

The **DOSL-II** program in Figure3.5 is a driver to make use of Module account. The **main** module uses variables named **balance** and **deposit**; however, the Module account also has members named **balance** and **deposit.** No conflict between the variable names and the member names can occur because they are separate module. This program begins by obtaining input from the user. It then uses the starting balance and interest rate to create an instance of **account** object:

account acct (balance, monper);

Because interest will be compounded monthly, the annual percentage rate is divided by 12.0 to get the monthly percentage that is required by above ***constructor***.

A **while** statement calls **acct.deposit()** to make each month's deposit and **acct.compound()** to compound the interest each month. After the deposit have been made and interest has been compounded for the required number of the months, **acct.getbalance()** is called to get the final balance, which is printed for the user.

```
:--*************************************
:-- Module driver
:--*************************************

ObjectModule ::[main]
Definition is

  type: passive;
  variable :
                balance : real;
                deposit  :real;
               anpctrate: real;
               monpcr   :real;
                months   : integer;
                       m  : integer;

  visible: [account];
  Body is
        begin
                doslout("Starting balance: ");
                doslin(balance);
                doslout("Monthly deposit");
                doslin(deposit);
                doslout("Annual percentage rate: ");
                doslin(anpctrate);
                doslout("Number of months: ");
                doslin(months);
:--     Compute new balance
                 m:= 0;
                 monpcr := anpctrate / 12.0;
                 account acct( balance, monpcr);
                 while (m < months) do
                     acct.deposit (deposit);
                     acct.compound();
                      m := m + 1;
                  od
:--  Print balance
                doslout("Balance after ", months,"months = $ ", acct.getbalance());
          end.
```

Figure 3.5 The ObjectModule Main.

3.9 Example of Inheritance Feature

For the first example of inheritance, we return to our example of class, a bank account module. Now, however several kinds of accounts, for example saving, checking and time account can be defined . A class **account** is defined as a generic account that can be opened with a given balance, accepts deposit, and return balance. Class **savacct** defines a traditional savings account that provides compound interest and withdraw privileges. Class **chkacct** defines simple checking account with no interest that allows check cashing and imposes a per-check charge if the balance falls below a given limit. Class **timeacct** defines a simplified form of time-deposit account that allows only accumulated interest to be withdrawn.

Figures 3.6, 3.7, 3.8, and 3.9 show four modules written in **DOSL-II** for account, savacct, timeacct, and chkacct which correspond to the hierarchy of Figure 3.2.

```
:-- Module account
ObjectModule ::[account]
  Definition is
      type : passive;
      class : class account {
       protected :     balance : real;
       public:
              account ( bal);
              deposit ( amt);
              getbalance();
      };
      visible  : [main];
   Methods
        method  account ();
        method  deposit ();
        method getbalance();
   Body is
:-- ****************************************
:--
:-- ****************************************
     (=> [:account::account ( bal)]
      begin
```

```
            balance = bal;
      end;
    ) ;;
:--*************************************
:--
:--*************************************
    (=>[: account::deposit( amt) ]
      begin
            balance = balance + amt;
      end;
    ) ;;
:-
     (=>[:account::getbalance()]
        begin
            return balance;
         end;
     )
     end.
```

Figure 3.6 The DOSL-II Account Module.

In module account, class account declares a variable balance as protected. This means the variable balance is accessible only to the member functions of classes derived from **account**.

```
ObjectModule ::[savacct]
  Definition is
     type : active;
     class : class savacct : public account {
     protected:          rate      : real;
     public:
            savacct ( bal, pcnt);
            withdraw ( amt);
            compound();
      };
     visible  : [account];
  Methods
       method savacct ();
       method withdraw ();
       method compound();
  Body is
```

```
(=> [:savacct::savacct ( bal,   pcnt)]
  begin
            rate = pcnt /100.0;
  end;
) ;;
(=>[:savacct:: withdraw( amt)]
   begin
         if ( amt <= balance ) then
          balance = balance - amt;
           return amt;
         else
          return 0.0;
         fi
   end;
 ) ;;
 (=>[: savacct::compound()]
    begin
         real interest = balance * rate;
         balance = balance + interest;
         return
          interest;
    end;
 ) end.
```

Figure 3.7 The DOSL-II Savacct Module.

In module **savacct**, inheritance is specified by following the class name by a colon and a base class. Each base class is denoted either ***private*** or ***public*** by preceding its name in the base class with the appropriate keywords.

If a base class is public, the public members of the base class are also public members of the derived class. Thus users of the derived class can refer to public members defined in the base class. If the base class is private, however, the public members of the base class become private members of the derived class. They can be accessed by member functions of the derived class, but they are not accessible to users of the derived class. In module **savacct**, class **account** is public base class so that the public member functions **deposite()** and **getbalance()**, defined for account, will be also available to users of **savacct** module.

An object of class **savacct** has two instance variables: **balance**, which is inherited from module account and **rate**, which is declared in **savacct**. Both **rate** and **balance** are protected members of **savacct**: **rate** because it is declared as protected in **savacct**, and **balance** because it is a protected member of public class.

```
ObjectModule ::[chkacct]
  Definition is
     type : active;
     class : class chkacct : public account {
     protected :          limit : real;
                            charge : real;
      public:
              chkacct ( bal, lim, chg);
              cashchk ( amt);
 };
     visible  : [main][savacct][account];
  Methods
        method cashchk();
        method chacct();
  Body is
    (=> [:chkacct :: chkacct( bal, lim, chg) : account(bal)]
     begin
            limit = bal;
            charge = chg;
     end;
    )
    ;;
   (=>[: chkacct :: cashchk( amt) ]
     begin
            if ((balance < limit) and ((amt + charge) <= balance)) then
                balance = balance - amt + charge;
                return amt;
            else
                return 0.0;
            fi
     end;
   )
    end.
```

Figure 3.8 The DOSL-II Chkacct Module.

The module **chkacct** specifies **account** as a public base class; thus **chkacct** inherits from **account** the protected instance variable **balance** and the public member function **deposit()** and **getbalance()**. Module **chkacct** also inherits function **deposite()** and **getbalance()** from **account**; therefore, the three functions **deposit()**, **getbalance()**, and **cashchk()** can all be applied to objects of class **chkacct**.

```
ObjectModule ::[timeacct]
  Definition is
     type : active;
     class : class  timeacct : public  account {
      protected :          fundsavail   : real;
      public:
            timeacct ( bal,  pcnt); withdraw ( amt); getavail(); compound(); };
     visible  : [main][account][savacct][chkacct];
  Methods
   method  timeacct (); method withdraw (); method compound(); method getavail();
  Body is
    (=> [:timeacct :: timeacct ( bal,  pcnt):savacct( bal, pcnt)]
     begin
            fundsavail = 0.0;
     end; ) ;;
   (=>[: timeacct :: withdraw(  amt)]
      begin
            if ( amt <= fundsavail ) then
             fundsavail = fundsavail - amt;
             balance = balance - amt; return amt;
            else   return 0.0;
            fi
       end; ) ;;
    (=>[: timeacct:: compound()]
       begin
             interest = savacct ::compound();
             fundsavail = fundsavail + interest;
             return interest;
       end; ) ;;
    (=>[: timeacct::getavail()]
        begin
            return fundsavail;
        end;)  end.
```

Figure 3.9 The DOSL-II Timeacct Module.

The module **timeacct** illustrates another important principle of inheritance: functions inherited from a base class can be redefined in the derived class. Note that the code for new definition can invoke the inherited function.

Module **timeacct** is defined as a derived class of **savacct** and inherits function **compound()** and **withdraw()**. However these functions must be redefined: **compound()** to update **fundsavail** and **withdraw()** to make withdrawals only from available funds.

The module **timeacct** have three instance variables: **balance** and rate inherited from **savacct** (which inherited **balance** from **account**) and **fundsavail** declared in module **timeacct**. Five functions can be applied to object of class **timeacct**: **deposite()** and **getbalance()**, which are inherited from **savacct** (which inherited them from **account**); **compound()** and **withdraw()**, which, although inherited from **savacct**, are redefined in **timeacct**; **getavail()**, which is defined in **timeacct**. The version of **compound()** inherited from **savacct** is used in defining the version for **timeacct**.

3.10 Example of Communication Constructs

The concurrent factorial program is written in **DOSL-II** to show the use of communication constructs **send**, **receive** and **reply**. This program consists of three separate object modules: **main**, **ConcFact**, and **RangeProduct,** which are presented in Figures 3.10, 3.11, and 3.12, respectively. The **main** module is the initiator of the whole process of computing 20!. It creates an instance of an object **ConcFact** using the **create** operation and assigns its **Mbox** address to a factorial variable. The **main** module sends a message to the **ConcFact** object using the **send** operation. The message consists of the address of **ConcFact**, method name to be called, the reply

destination, **myCbox**, and an integer n. The main module is blocked until the **myCbox** receives the result. Finally **Doslout** prints the result (see Figure 3.10).

The **ConcFact** module becomes active when it is called by the main object. The receive operation on **self** causes the **ConcFact** module to read the requested message.

ObjectModule :: **[main]**

```
Definition is
type          : active;
visible       : [ConcFact];
variable      : k: integer;
                n: integer;
              myCbox : Cbox;
              factorial  : Mbox;
body is
 begin
        n := 20;
        factorial := create (ConcFact);
        send (factorial, &ConcFact::computefact, myCbox,n);
        receive (myCbox, k);
        doslout("The factorial of %d  is %d \n", n,k);
end.
```

Figure 3.10 The Main Module.

The **ConcFact** module creates an object called **RangeProduct** and sends a message to it (see Figure 3.11).

The **RangeProduct** module uses a divide-and-conquer algorithm to compute the factorial. It multiplies all numbers in the range specified by its two input arguments. The **RangeProduct** module reads its requested message using operation **receive** on **self**; it then determines if the range contains one number, then returns low. Otherwise it divides the range into two sub-ranges.

```
ObjectModule :: [ConcFact]
Definition is
type          : active;
visible       : [RangeProduct];
variable        : m: integer;
                  one:integer;
                rpone: Mbox;
methods
      method computefact : ( ) --> integer;
body is
      (=> [: computefact ( )]
 begin
        one := 1;
        receive(self,m);
        rpone := create(RangeProduct);
       send(rpone, &RangeProduct::computeprod,one,n);
 end.
```

Figure 3.11 The ObjectModule Concfact.

To compute the product of these two sub-ranges in parallel, two new instances of the **RangeProduct** module are created. Two different messages are sent to each newly created object along with subCbox. This process continues until the sub-range computed by **RangeProduct** module contains only one number. The **RangeProduct** module will eventually receive two sub-range products to its subCbox and multiplies the two sub-range products. The **reply** operation will send the result to the reply destination, **myCbox** of the main module (see Figure 3.10).

```
ObjectModule :: [RangeProduct]
Definition is
type          : active;
visible       : [RangeProduct];
variable      : low: integer;
                  mid: integer;
                 high: integer;
```

```
        subone: integer;
        subtwo : integer;
        subCbox : Cbox;
        rpone      : Mbox;
        rptwo      : Mbox;
methods
    method computeprod : ( ) --> integer;
body is
    (=> [: computeprod ( )]
 begin
      receive(self,low,high);
      if ( low >= high) then
        reply(low);
      else
        mid := ( low + high ) /2;
        rpone := create(RangeProduct);
        rptwo  := create(RangeProduct);
        send (rpone, &RangeProduct::computeprod,subCbox,low, mid);
        mid := mid + 1;
        send (rptwo, &RangeProduct::computeprod,subCbox,mid,high);
        receive(subCbox,subone,subtwo);
        reply (subone * subtwo);
 end.
```

Figure 3.12 The ObjectModule RangeProduct.

3.11 Example of Concurrent I/O

Let us consider the example where an arbitrary actor wants to read from terminal **A** asynchronously and write the read information to terminal **B** synchronously. Figure 3.13 shows a program written in **DOSL-II** using concurrent input/output statements that defined earlier.

```
:-- DOSL-II program for concurrent I/O

ObjectModule ::[main]
Definition is
variable : mayactone, myacttwo,
           fnamone ,fnametwo : array [1 .. 20] of char;
           rbuf, wbuf  : array [1 .. 200] of char;
body is
begin
  fnameone := "/dev/ttyp9";
  fnametwo := "/dev/ttyp8";
  readasc(mayactone,fnameone,rbuf);
  writesc(myacttwo,fnametwo,wbuf);
end.
```

Figure 3.13 The DOSL-II Concurrent Input/Output.

The program begins by assigning the terminal names to the variables **fnameone** and **fnametwo** respectively. The concurrent read statement (**readasc**) reads from **ttyp9** terminal into **rbuf** asynchronously by creating an actor **myactone** to process the read operation concurrently. Similarly the concurrent write statement writes to terminal **ttyp8** from **wbuf** by creating an actor **myacttwo** to process the write operation synchronously.

3.12 Summary

An overview of **DOSL** and its complete syntax in extended Backus-Naur Form has been presented. The definition of **DOSL** was extended to support class and inheritance. This extension makes **DOSL** an object-oriented specification language rather than object-based. Three categories of new **I/O** constructs standard, stream, and concurrent **I/O** were introduced and common problems associated with concurrent **I/O** were discussed. A new communication construct is presented along with the definition of new constructs. Examples of class, inheritance, concurrent I/O and communication constructs are also given.

Chapter 4

The DOSL Transformation Process

4.1 Introduction

There are two approaches for making a prototyping language executable, one based on meta-programming, and the other based on executable specifications[Ber91][Rob89][Jos82]. The meta-programming approach provides facilities for adapting and interconnecting available software components. The processor for a meta-programming language generates the skeleton of an implementation, with empty places for the available components. These components can be drawn from a library, simulated, or manually programmed as needed.

The executable specification approach uses the specifications of a module for direct execution (see Section 4.3), and can succeed only if the specification is executable or can be transformed to a semantically equivalent form that is executable. In this work we use the second approach.

4.2 An Overview of ACT++

ACT++ is a concurrent object-oriented language [Kaf88][Kaf89][Kaf90]. The primary design goal of ACT++ is to develop a language which supports the powerful actor concurrent computation model and provides software reusability through the class inheritance of an object-oriented language. ACT++ is intended for exploring the actor style of programming and object-oriented programming with class inheritance. The current ACT++ design extends C++ [Str86] with a class hierarchy which provides the abstraction of the actor model of concurrency.

The asynchronous message passing in **ACT++** is supported by two predefined objects: **Mbox** and **Cbox**. The **Mbox** models the mail queue of the actor while **Cbox** allows the sender of a message to receive the result of the method invocation.

The primary **I/O** abstraction introduced in **ACT++** is that of an **interface actor(IA)**. An **IA** is an **I/O** server which manages **I/O** to a single device - a standard file or terminal special file. **IAs** are capable of doing both **synchronous** and **asynchronous I/O**. In **ACT++**, there are two types of objects **active** and **passive**. The distinction between the two types is that when active objects process a message they create an independent thread of control to execute the requested operation, whereas passive objects process a message using the thread of control of the requestor. Thus, an **ACT++** program is a coherent collection of active and passive objects - active objects execute independently and concurrently with other active objects whereas the passive objects act as subordinates of the active objects. **ACT++** has been successfully implemented on the Sequent Symmetry multiprocessor. In the following section, we explain how concurrent input/output is done in **ACT++**.

4.2.1 Example of Concurrent I/O in ACT++

Let us consider the same example of section 3.11, where an arbitrary actor wants to read from terminal **A** asynchronously and write the read information to terminal **B** synchronously. In **ACT++** we must create two interface actors (**IA**), each responsible for **I/O** to one terminal. We first present the Read operation in Figure 4.1, note that the numbers are not part of the code.

Line 1 assigns the name of terminal **A** to the variable fname. The second line creates an interface actor (**IA**) actor my-act1 and associates it with the /dev/ttyp9 special file corresponding to the terminal **A.**

```
1.    char* fname1 = "/dev/ttyp9";
2.    IActor my-act1 = new IACTOR (fname1, TTYBEH);
3.    Rbox* rb-empty = new Rbox();
4.    Message* read-mess = new Message (TTYACT::Read, rb-empty, rb-
empty-->size());
5.    read-mess-->send(my-act1);
```

Figure 4.1 The Concurrent Read Operation in ACT++.

TTYBEH is a predefined macro which creates the object behavior. Line 3 creates an empty buffer that is used by my-act1 to read data. All **Rboxes** in **ACT++** are instantiations of the predefine class **Rbox**. Line 4 creates a message for **IA** , the **TTYACT** macro must be used to obtain the address of the Read/Write method of **TTYBEH** class. Line 5 sends a read message to an **IA** and my-act1 actor reads from terminal A. Figure 4.2 shows similar coding for the Write operation except for the last line. Line 6 is the *wait* method called on the Write operation. The *wait* method defined in the **Rbox** and **Wbox** classes is used to implement blocking on these boxes.

```
1.  char* fname2 = "/dev/ttyp8";
2.  IActor my-act2 = new IACTOR (fname1, TTYBEH);
3.  Wbox* wb-from-rb = new Wbox(rb-empty);
4.  Message* write-mess = new Message (TTYACT::Write, wb-from-rb, wb-
from-rb-->size());
5.  write-mess-->send(my-act2);
6.  wb-from-rb-->wait();
```

Figure 4.2 The Concurrent Write Operation in ACT++.

For example, the wait on **Wbox** is used in line 6 to determine whether the Write operation has completed. If the write has completed, the call returns immediately; otherwise, the operation blocks.

In Figure 4.2, line 3 creates a write buffer for an actor my-act2 and passes the address of the read buffer in Figure 4.1. This means my-act2 will write to terminal **B** directly from the read buffer.

4.3 Methodology for Executing DOSL-II

An overall general structure of the technique for executing a **DOSL-II** specification is depicted in Figure 4.3.

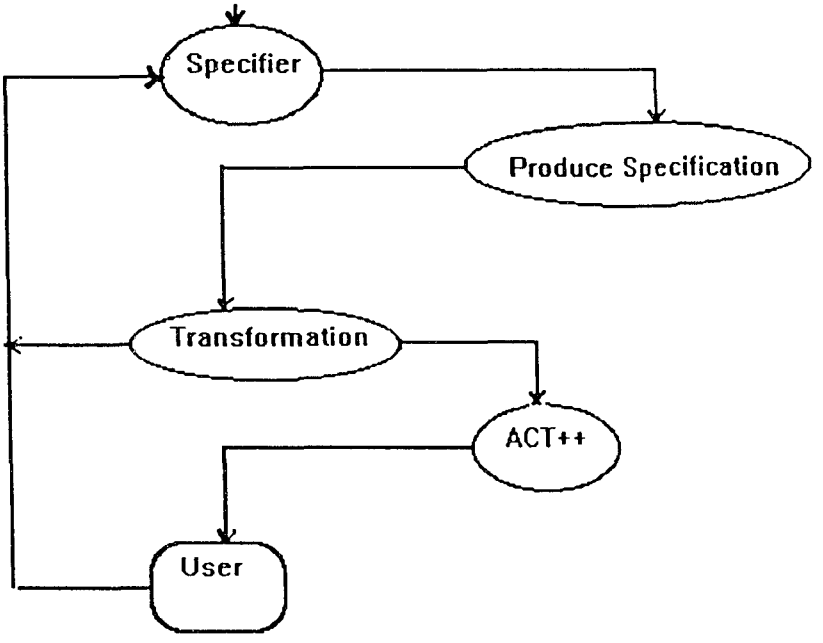

Figure 4.3 The Transformation Process

The specifier produces the specification using **DOSL-II** specification language. The **DOSL-II** specification is read by the transformation system, which checks its syntax and then transforms it into ACT++ code. The transformed output of the translator is then executable.

The transformation system must decide whether or not a given sentence is a correct sentence in the **DOSL** language. It does that by parsing the sentence using the language grammar.

4.3.1 Analysis of the DOSL-II Program Statement

The statements of the **DOSL-II** program are analyzed by the parsing phase. The basic function of the parsing phase is to build a unique parse tree from the sequence of tokens produced by the scanner. This parse tree is then traversed in an appropriate order by the code generation phase to produce the translation of the **DOSL-II** program into **ACT++** code.

There are two classes of parsing techniques: ***top-down*** and ***bottom-up***. Each class is characterized by the order in which the productions of the derivation tree are recognized [Aho79]. We have chosen the bottom-up parsing technique for the following reasons

1. With the bottom-up technique it is possible to take a grammar specified in **BNF** and generate tables automatically for a parser.

2. The changes to the syntax of the language can be accommodated quickly.

3. It also ensures that the language being parsed matches the language specified in written syntax.

In the bottom-up technique, the derivation tree is built from the terminal nodes up to the root node. As the parsing progresses, the input is scanned from left to right, and the input is converted into a list of subtrees from which the complete tree will be

constructed. At any stage, there are two alternatives from which the parsing algorithm must choose. It could ***shift*** a symbol from the head of the input over to the list of subtrees on a stack and form a new primitive subtree on the top of the stack. Alternatively, it could ***reduce*** one or more of the subtrees at the right end of the list of subtrees to a single subtree, using one of the production rules. A set of subtrees that can be reduced is known as a ***handle*** since they are grasped together to make the reduction.

In general, a parser will shift until the right end of the list of trees contains a handle and then reduce it. This technique, known as the shift-reduce principle, was introduced in [Flo61]. The parsing algorithm usually makes use of tables, which are constructed from the grammar by a special program, to base its decision on shifting or reduction. The bottom-up parser uses a technique known as **LR(k)** parsing; where **L** means scanning the input from left-to-right, and the **R** for constructing a rightmost derivation in reverse and **K** is referred to the number of input symbols of lookahead that are used in making parsing decisions. A modified algorithm from [Aho85] for **LR(1)** parsing is shown below.

LR parsing Algorithm:

Input. An input string w and an **LR(1)** parsing table for the **DOSL-II** grammar.

Output. If string w is in **DOSL-II** language, a bottom-up parse for w and mapping string w into **ACT++**; otherwise error indication

Method. The parser executes the algorithm in Figure 4.4 until an accept or error state is encountered.

When the scanner has converted a program text into a sequence of symbols, the parser performs a single scan of the symbols and checks whether they form a **DOSL-II** sentence. If the syntax is correct , then an equivalent sentence in **ACT++** is written to an output file (action code); otherwise, a syntax error is reported. The parser is

constructed directly from the **BNF** grammar of **DOSL-II** (See Chapter 3). To make the algorithm simple, we assume for every **BNF** rule :

N=E.

the parser defines a procedure of the same name:

Procedure N; **begin** a(**E**) **end**;

The procedure defines a parsing algorithm **a(E)**. When the algorithm is executed it examines one or more symbols and determines whether they form a sentence described by the syntax expression **E**. If they do, the algorithm calls procedure-action (procedure-action job is to write an equivalent sentence in **ACT++** or simulate one if there is no equivalent); otherwise, the algorithm reports an syntax error. We expressed the algorithm using a Pascal like notation:

```
program = "ObjectModule" "::" "[" programname "]" {action code}
          "Definition is" {action code}
           Declaration-part ";" {action code}
          "Body is " {action code}
           BlockBody; {action code}
           "end." {action code}
```

To recognize a program we need a procedure:

```
procedure Programx;
 begin
    a("ObjectModule" "::" "["ProgramName "]" {action code}
        "Definition is" {action code}
         Declaration-part ";" {action-code}
         "Body is" {action code}
         BlockBody ";" {action code}
        "end.")
 end;
```

that defines an algorithm:

```
a("ObjectModule" "::" "["ProgramName "]" {action code}

    "Definition is" {action code}
    DeclarationPart ";" {action-code}
   "Body is" {action code}
    BlockBody ";" {action code}

    "end."{ action code})
```

The algorithm scans a sentence consisting of the word ObjectModule followed a double colon, a left bracket, name, a right bracket, a newline, Definition is, newline, Declaration-part, semicolon, newline, Body is, newline, BlockBody, semicolon, end, and a period. The action codes are inserted when there is a need to translate a **DOSL-II** sentence into an **ACT++** code.

We construct this complicated algorithm out of following simpler algorithms:

a("ObjectModule" "::" "["ProgramName "]")	Recognizes a Module heading
a("Definition is")	Recognizes a Definition is
a(DeclarationPart)	Recognizes a DeclarationPart
a(";")	Recognizes a semicolon
a("Body is")	Recognizes a Body is
a(BlockBody)	Recognizes a BlockBody
a("end.")	Recognizes an end.

We program these algorithms, and construct the original algorithm as a sequence of the simpler algorithms. When the parser expects a single symbol s, it uses the following algorithm:

a(s) = **if** Symbol = s **then** NextSymbol

else SyntaxError

This algorithm is implemented as a procedure:

Procedure Expect(s: Symbol)

Now we can construct all but two of the algorithms above:

a("ObjectModule" "::" "["ProgramName "]") = Expect (ModuleHeading)

a("Definition is") = Expect (Definition is)

a(";") = Expect (Semicolon)

a("Body is") = Expect (Body is)

a("end.") = Expect(end.)

The DefinitionPart and BlockBody are not defined yet. They can also be recognized by a set of procedures. To make the algorithm simple, we did not include them here.

By combining the simpler algorithms, we obtain the procedure Programx (Figure 4.5)

```
procedure Programx;
begin
    Expect(ProgramHeading); {action code}
    Expect( Definition is); {action code}
    DefinitionPart; {action code}
    Expect(Body is); {action code}
    BlockBody; {action code}
    Expect(end.) {action code}
end;
```

Figure 4.4 The Procedure Programx

We need to develop two more procedures for DefinitionPart and BlockBody. A DefinitionPart is described as follows:

DefinitionPart = [TypeDefinitionPart] [ClassDefinitionPart]
[VisibleDefinitionPart] [VariableDefinitionPart]
[MethodDefinitionPart]

and can be recognized by the following algorithm (Figure 4.5):

```
procedure DefinitionPart;

begin
   if Symbol = Type then
        TypeDefinitionPart;
   if Symbol = Class then
        ClassDefinitionPart;
   if Symbol = Visible then
        VisibleDefinitionPart;
   if Symbol = Variable then
        VariableDefinitionPart;
   if Symbol = Method then
        MethodDefinitionPart;

end;
```

Figure 4.5 The Procedure DefinitionPart

The syntax factor

[TypeDefinitionPart]

shows that a DefinitionPart may or may not begin with TypeDefinitionPart.

A BlockBody is described as follows:

BlockBody = [MethodHeading][StatementDefinitionPart]

and can be recognized by the following algorithm:

```
procedure BlockBody;
begin
    if Symbol = MethodHeading then
      Heading;
    if Symbol = StatementDefinition then
      Statement;
end;
```

This process will continue until all the procedures are constructed. Figure 4.6 shows **DOSL-II** constructs and their corresponding codes in **ACT++**.

DOSL-II constructs	**Transformation code**
ObjectModule :: [sample]	main()
Definition is	none
variable : i:integer;	int i;
j:real;	double j;
table :array[0..6] of integer;	int table[6];
tabletwo: array [0..6] of char;	char tabletwo[6];
Body is	none
begin	{
end	}
(=>[:account::account (bal, pcnt)]	account::account(doublebal,double pcnt)
x := x + 1;	x = x + 1;
while expression do	while expression
...	{
od;	...
	}
if expression	if expression
then	{
statement1;	statement1;
statement2;	statement2;
.	.
.	.
statementn;	statementn;
fi	}

doslin(argument-list);	scanf(argument-list);
doslout(argument-list);	print(argument-list);
din << argument-list;	cin << argument-list;
dout << argument-list;	cout << argument-list;
readasc(myactone, fname, rbuffer);	IActor myactone = new IACTOR(fname,TTYBEH); Rbox* rbuffer = new Rbox(); Message* read-mess= new Massage(TTYACT::Read,rbuffer,rbuffer-->size()); read-mess--> send(myactone);

writesc(myactortwo,fnametwo,rbuffer);	IActor myacttwo = new
	IACTOR(fname,TTYBEH);
	Wbox* wbuffer = new Wbox(rbuffer);
	Message* write-mess= new
	Massage(TTYACT::Write,Wbuffer,Wbuffer-->size());
	write-mess--> send(myacttwo);
	wbuffer --> wait();

readsc(myactone, fname, rbuffer);	IActor myactone = new
	IACTOR(fname,TTYBEH);
	Rbox* rbuffer = new Rbox();
	Message* read-mess= new
	Message(TTYACT::Read,rbuffer,rbuffer-->size());
	read-mess--> send(myactone);
	rbuffer-->wait();

writesc(myactortwo,fnametwo,rbuffer);	IActor myacttwo = new IACTOR(fname,TTYBEH); Wbox* wbuffer = new Wbox(rbuffer); Message* write-mess= new Massage(TTYACT::Write,Wbuffer,Wbuffer-->size()); write-mess--> send(myacttwo);

Figure 4.6 DOSL-II to ACT++ Transformation

As an example, consider the loop construct in **DOSL-II**. Its action is based the grammar rule for the loop statement:

loop-statement ::= **while** expression **do**
statement1;
....
statement n;
od

Thus, the system performs the following steps:

1. Checks that the next token is a *while*-symbol, and maps this token to an equivalent token in ACT++, or starts simulating a loop construct.

2. Calls the procedure Expression , if accept then maps this expression into **ACT++** expression.

3. Checks that next token is a *do*-symbol, and does the mapping

4. Calls the procedure StatementDefinition.

5. Checks that next symbol is a *od*-symbol.

Note that the procedure StatementDefinition will be called recursively to recognize all the statement separated by semicolon in the body of the loop.

The mapping function is straightforward for those constructs of **DOSL-II** where there exists an equivalent construct in **ACT++**. However, there are many cases where there is no match for the **DOSL-II** construct in **ACT++**. In the cases of no match, the system must simulate the behavior of those constructs using a set of procedures written in **ACT++** language. As an example, consider a **DOSL-II** concurrent read statement that we have introduced in Chapter 3.

readasc (myactone, fname, rbuffer);

This statement reads from (terminal/file) fname into rbuffer asychronously by creating an actor myactone to process the read operation concurrently. The system prototype will simulate the above read statement in the environment of **ACT++** as follows and as shown in Figure 4.6:

```
IActor myactone = new IACTOR (fname, TTYBEH);
Rbox* rbuffer    = new Rbox();
Message* read-mess = new Message(TTYACT::Read, rbuffer, rbuffer-->size());
read-mess--> send (myactone);
```

As discussed in Chapter 3, an interface actor (**IA**) provided by **ACT++** encapsulates all low level details for performing **I/O** and relieves the user from managing all low level details explicitly. The system prototype makes it possible for the **DOSL-II** specification to do an **I/O** abstraction at higher level. This means to do a concurrent **I/O**, all the specifier has to do is to write only one statement.

Suppose that the specifier decides to write the read information concurrently into another terminal synchronously. Then he/she must issue the following statement in **DOSL-II**:

```
writesc (myactortwo,fnametwo,rbuffer);
```

The system prototype translates the above statement into the following sequence of statements for the environment of **ACT++** as also given in Figure 4.6:

```
IActor myacttwo = new IACTOR (fnametwo, TTYBEH);
Wbox*  wbuffer= new Wbox(rbuffer);
Message*  write-mess = new Message(TTYACT::Write, wbuffer,
wbuffer-->size());
write-mess--> send (myacttwo);
wbuffer --> wait();
```

In the following section, we present a prototype example of concurrent factorial program written in **DOSL-II** and its complete transformation into an **ACT++**.

4.4 Prototype System Example

The concurrent factorial program written in **DOSL-II** consists of three separate object modules: **main**, **ConcFact**, and **RangeProduct,** which are presented in Figure 4.7, 4.8, and 4.9, respectively. The **main** module is the initiator of the whole process of computing 20!. It creates an instance of an object **ConcFact** using the **create** operation and assign its **Mbox** address to a factorial variable. The **main** module sends a message to the **ConcFact** object using the **send** operation. The message consists of the address of **ConcFact**, method name to be called, the reply destination, **myCbox**, and an integer n. The main module is blocked until the **myCbox** receives the result. Finally **Doslout** prints the result (see Figure 4.7).

```
ObjectModule :: [main]

Definition is

type          : active;
visible       : [ConcFact];
variable      : k: integer;
                n: integer;
              myCbox : Cbox;
              factorial  : Mbox;
body is
 begin

        n := 20;

        factorial := create (ConcFact);

        send (factorial, &ConcFact::computefact, myCbox,n);

        receive (myCbox, k);

        doslout("The factorial of %d  is %d \n", n,k);

end.
```

Figure 4.7 The ObjectModule Main.

The **ConcFact** module becomes active when it is called by the main object. The receive operation on **self** causes the **ConcFact** module to read the requested message. The **ConcFact** module creates an object called **RangeProduct** and sends a message to it (see Figure 4.8).

ObjectModule :: **[ConcFact]**

```
Definition is
type          : active;
visible       : [RangeProduct];
variable        : m: integer;
                  one:integer;
                rpone: Mbox;
methods
      method computefact : ( ) --> integer;
body is
      (=> [: computefact ( )]
 begin  one:= 1;
        receive(self,m);
         rpone := create(RangeProduct);
        send(rpone, &RangeProduct::computeprod,one,n);
 end.
```

Figure 4.8 The ObjectModule ConcFact.

The **RangeProduct** module uses a divide-and-conquer algorithm to compute the factorial. It multiplies all numbers in the range specified by its two input arguments. The **RangeProduct** module reads its requested message using operation **receive** on **self**; it then determines if the range contains one number, then returns low. Otherwise it divides the range into two sub-ranges. To compute the product of these two sub-ranges in parallel, two new instances of the **RangeProduct** module are created. Two different messages are sent to each newly created object along with subCbox. This process continues until the sub-range computed by **RangeProduct** module contains only one number.

The **RangeProduct** module will eventually receive two sub-range products to its subCbox and multiplies the two sub-range products. The **reply** operation will send the result to the reply destination, **myCbox** of the main module (see Figure 4.9).

```
ObjectModule :: [RangeProduct]
Definition is
type          : active;
visible       : [RangeProduct];
variable      : low: integer;
                mid: integer;
               high: integer;
                subone: integer;
                subtwo : integer;
                subCbox : Cbox;
                rpone     : Mbox;
                rptwo      : Mbox;
methods
      method computeprod : ( ) --> integer;
body is
      (=> [: computeprod ( )]
 begin
        receive(self,low,high);
        if ( low >= high) then
          reply(low);
        else
           mid := ( low + high ) /2;
           rpone := create(RangeProduct);
           rptwo  := create(RangeProduct);
           send (rpone, &RangeProduct::computeprod,subCbox,low, mid);
           mid := mid + 1;
           send (rptwo, &RangeProduct::computeprod,subCbox,mid,high);
           receive(subCbox,subone,subtwo);
           reply (subone * subtwo);
 end.
```

Figure 4.9 The ObjectModule RangeProduct.

The ObjectModules main, ConcFact, and RangeProduct are transformed into ACT++ as presented in Figure 4.10. Figure 4.10 is factorial example found in [Kaf90] for ACT++. The code shown in Figure 4.10 agrees with the code in [Kaf90].

```
#include "act.h"                        // include the ACT++ kernel classes
class ConcFact: ACTOR {
public:
        void compute_factorial();};
class RangeProduct: ACTOR {
public:
       void compute_product();};
main()
{
     int k;
     int n=20;  /* compute 20! */
     Cbox myCbox;
     Mbox factorial = New(ConcFact);   // bind it to an instance of ConcFact;
     factorial << &ConcFact::compute_factorial << myCbox << n;
                                                  // send a request message
    myCbox >> k;                                 // receive a reply from factorial
    printf("%d\n",k);
}
void ConcFact::compute_factorial()    // a method of ConcFact
{
    int m;
    self >> m;                                   // read a request message
    Mbox rp1 = New(RangeProduct);
    rp1 <<&RangeProduct::compute_product <<1<<m;
}
void RangeProduct::compute_product()  // a method of RangeProduct
{
    int low, mid, high, sub1, sub2;
    self >> low >> high;
    if (low >= high)
         reply(low);
    else {
        mid = (low + high) /2;
        Mbox rp1 = New(RangeProduct);
        Mbox rp2 = New(RangeProduct);
        Cbox subCbox;
        rp1 <<&RangeProduct::compute_product << subCbox << low << mid;
        rp2 <<&RangeProduct::compute_product << subCbox <<  mid+1 <<high;
        subCbox >> sub1 >> sub2;
        reply(sub1 * sub2);
    }
}
```

Figure 4.10 The Factorial Example of ACT++.

4.5 Summary

Two approaches for making a prototyping language executable, one based on meta-programming, and other based on executable specifications are discussed. In section 4.2 and 4.3 an overview of **ACT++** along with an example of concurrent I/O in **ACT++** is presented. In section 4.3 we have introduced a methodology for executing **DOSL-II** specification languages. We also have discussed LR(1) parsing algorithm and introduced a series of procedures which recognizes the **DOSL-II** specification language according to its syntax and then transforms it into **ACT++** code. In Figure 4.6 we summarized **DOSL-II** constructs and its transformation code in **ACT++**. Finally, an example of concurrent factorial in **DOSL-II** along with its transformation into **ACT++** is explained.

Chapter 5

Validation of the Prototyping Environment

5.1 Introduction

The objective of this Chapter is to describe the testing of the transformation process for the prototyping system. The testing strategy ensures that all statements of **DOSL-II** are executed at least once. Since the system prototype must be able to translate an infinite number of possible **DOSL-II** programs, it is very unlikely that a few "typical" programs chosen at random will test the system prototype systematically. We did therefore carefully construct small programs for test purposes only. We are not presenting all test programs in this Chapter. In the following sections, we present those test programs that concern **I/O** , class and inheritance.

5.2 DOSL-II, Syntax Analysis

We begin by looking at the *test* program (Figure 5.1) for correct sentences of the **DOSL-II** program. This test program contains comment, declarations, input output statement, nested if-then-else statement, while-do statement, and assignment statements.

```
:-- test 1 : DOSL-II syntax analysis
:-- this is a comment in DOSL-II
ObjectModule ::[main]
Definition is
variable :  a:real;
            b:real;
       intarray    : array [1 .. 90] of integer;
       realarray   : array [1 .. 20] of real;
       chararray : array [2 .. 9] of char;
body is
begin
```

```
:-- testing output statement
 doslout(" please inter two  number\n");

:--  testing input statement

 doslin("%f%f", a,b);

:-- print the values of a and b

 doslout("the a value is %f\nb=%f\n", a,b);

:-- testing assignment statement with an expression

   a:=( a - b + (a + 8) * 2);

 doslout("the new value of a is %f",a);

:-- testing nested if-then-else statement

   if (a <> b) then
           a:= a + b;
           if ( a = b) then
             a:= 56;
           fi
             b:= 23 + 7;
   fi

   :-- testing a while do statement

   while (b < a) do
         b:= b + 1;
         doslout(" value of a= %f \n value of b= %f\n);
   od
 end.
```

Figure 5.1 The DOSL-II Syntax Analysis

As a result of running the above test program the system prototype indicates no syntax error. To show how the system prototype response to the syntax error, we modify the above test program by removing a semicolon from the first and second statement in the body of the program. Now, if we run the system using the modified test program the result will be printed as follows:

```
:-- test 2 : DOSL-II syntax analysis
:-- this is a comment in DOSL-II

ObjectModule ::[main]
Definition is
variable :  a:real;
            b:real;
        intarray    : array [1 .. 90] of integer;
        realarray   : array [1 .. 20] of real;
        chararray   : array [2 .. 9] of char;

body is
begin
:-- testing output statement
 doslout(" please inter two  number\n")$
Syntax error
```

Figure 5.2 The DOSL-II Syntax Analysis With an Error.

Figure 5.2 shows how the system prototype response to the syntax error. It places the symbol $ dollar sign in the position of first missing semicolon and writes the phrase "**Syntax error**" and stops. Note that the system will indicates the syntax errors one at a time.

In the case of no syntax error, the prototype system produces the **ACT++** version of the **DOSL-II** module. For example, in Figure 5.3 a syntactically correct bubble sort test program is written in **DOSL-II.**

```
:-- This the first executable program in DOSL-II
:--  ****************************************
:--  *   This program  sorts an array of integer          *
:--  ****************************************
ObjectModule :: [BubbleSort]
Definition is
 variable :          i : integer; j : integer;
                     size: integer; sizemin : integer;
                     save : integer;
                     table  :  array [0 .. 6] of integer;
body is
```

```
begin
        size := 5;
      sizemin:= size - 1;
            i := 0;  j := 0;
     doslout(" Unsorted Table\n");
     doslout("==============\n");
     while  ( i <= size ) do
        doslin("            %d\n", table[i]);
        i := i +1;
     od
      i := 0; j := 0;
      while ( i <= size )  do
        j := i + 1;
        while ( j <= sizemin ) do
           if ( table[i]  >= table[j] ) then
                save := table[i];
              table[i]:= table[j];
              table[j]:= save;
           fi
           j := j + 1;
        od
       i := i +1;
      od
     doslout("  Sorted Table  \n");
     doslout("============\n");
      i := 0;
      while ( i <= size) do
          doslout("        %d\n", table[i]);
          i  := i +1;
      od
end.
```

Figure 5.3 The BubbleSort ObjectModule.

The prototype system checks for syntax errors and converts the **DOSL-II** program into the following **ACT++** code (Figure 5.4). The code in Figure 5.4 is hidden from the user of the prototyping environment.

```
main()
{
   int i; int j;
   int size; int sizemin;
   int save; int table[6];
   size = 5; sizemin = size -1;
   i = 0;  j = 0;
   printf("Unsorted Table\n");
   printf("==============\n");
   while ( i <= size) {
      printf("          %d\n" , table[i]);
      i = i + 1;
    }
    i = 0; j = 0;
    while ( i <= size)  {
       j = i + 1;
         while ( j <= sizemin ) {
            if ( table[i] >= table[j]) {
                save = table [i];
                table[i] = table[j];
                table[j] = save;
             }
          j = j + 1;
          }
        i = i + 1;
      }

      printf("Sorted Table \n");
      printf("============\n");
      i = 0;
      while (i <= size) {
         printf("          %d\n", table[i]);
         i= i + 1;
       }
}
```

Figure 5.4 The ACT++ BubbleSort.

After the execution of the **ACT++** program the output is:

```
Unsorted  Table
==============
        5
        4
        3
        2
        1
        0

  Sorted  Table
==============
        0
        1
        2
        3
        4
        5
```

The prototype system is tested for numerous **DOSL-II** programs.

5.3 DOSL-II, I/O Syntax Analysis

In Chapter 3, we presented the extention of **DOSL** specification to support the following **I/O** constructs:

1. doslin (argument-list);
2. doslout(argument-list);
3. din
4. dout
5. readasc(actname,fname,bufname);
6. readsc(actname,fname,bufname);
7. writeasc(actname,fname,bufname);
8. writeasc(actname,fname,bufname);

Figure 5.5 The DOSL-II Input/Output.

Three different **I/O** commands are listed in Figure 5.5, the first two commands are referred to as a **simple I/O**, the third and fourth are **stream I/O**, and the fifth through eighth are **concurrent I/O** for synchronous and asynchronous. In the case of simple **I/O**, the user is responsible for specifying the correct field descriptor corresponding to the type of variable (s). Otherwise the system shows a syntax error. For example, to use the doslout construct, in Figure 5.6 user must specify the field descriptor **%d, %c, %f,** and **%s** for the type integer, character, real, and string respectively.

```
ObjectModule ::[sample]
Definition is
 variable: a:real;
             b:integer;
             c:char;
            string: array [ 1 .. 10 ] of char;
 body is
  begin
        b:=10;
        c:="T";
        a:= 1.34;
    string:= "testing";
    doslout("%d  %c %f %s",b,c,a,string);
 end.
```

Figure 5.6 The Output Sample.

In the case of stream **I/O**, the user is not responsible for specifying the field descriptor(s) as the system will automatically provide the corresponding field descriptor(s) based on the variable(s) type. For example in Figure 5.6, the statement **doslout("%d %c %f %s", b,c,a,string);** can be replaced by the stream **I/O** statement **dout <<b<<c<<a<<string;** without a syntax error.

5.4 DOSL-II, Concurrent I/O Syntax Analysis

The syntax analysis of the concurrent **I/O** is done using a test program. In the case of a syntax error, the system places the symbol "$" at the position of error. If there is no syntax error the system will produce executable code in the environment of **ACT++**. In this section, we refer to the example of Chapter 3 in section 3.11, where an arbitrary actor wants to read from terminal A asynchronously and write the read information to terminal **B** sychronously. Figure 5.7 shows a complete test program written in **DOSL-II** and saved under file name **testio**.

```
:-- test program for concurrent I/O
ObjectModule ::[main]
Definition is
variable : fnamone ,fnametwo : array [1 .. 20] of char;
           rbuf, wbuf  : array [1 .. 200] of char;
body is
begin
  fnameone := "/dev/ttyp9";
  fnametwo := "/dev/ttyp8";
  readasc(mayactone,fnameone,rbuf);
  writesc(myacttwo,fnametwo,wbuf);
end.
```

Figure 5.7 The Concurrent I/O Sample.

In order to execute the above program using the prototype system, the following steps were performed:

```
dosl < testio > acttestio.c
c++ acttestio.c
a.out
```

In the first command line, the system prototype (**dosl**) takes the file name *testio*(*testio* contains program in Figure 5.7) and check for syntax error. If there is no error,

it then maps the **DOSL-II** program into **ACT++** program and saves it in file name *acttestio*. The second command line compiles the *acttestio* using the **C++** compiler and generates executable code in *a.out*. The third command line executes the program, that is, the information is read from terminal **A** and written to terminal **B**. The Figure 5.8 shows the **ACT++** code produced by the system prototype.

```
#include "act++.h"
#include <iostream.h>

main ()
 {
 char fnamone[20];
 char fnametwo[20];
fnameone="/dev/ttyp9";
fnametwo="/dev/ttyp8";
IActor mayactone = new( fnameone,TTYBEH);
Rbox* rbuf = new Rbox();
Message* read-mess = new Message(TTYACT::Read,rbuf,rbuf->size());
read-mess ->send(mayactone);
IActor myacttwo = new( fnametwo,TTYBEH);
Wbox* wbuf = new Wbox(rbuf);
Message* write-mess = new Message(TTYACT::Write,wbuf,wbuf->size());
write-mess ->send(myacttwo);
wbuf -> wait();
}
```

Figure 5.8 The ACT++ Concurrent I/O.

Note that the translation of **DOSL-II** program into **ACT++** is transparent to the user.

5.5 DOSL-II, Class Syntax Analysis

In Chapter 3, we presented the formal syntax for class and class inheritance of **DOSL-II** specification. In this section, we recall the same example of bank account (Figure 5.9) and show how the system prototype detects syntax errors and produces an executable code.

Figure 5.9 presents the module account test program written in **DOSL-II.** This module includes all necessary declarations and definitions for class account and offers the following methods for manipulation of class account:

method

method account(bal,pct);

method deposit ();

method withdraw();

method compound();

method getbalance();

The instance variables **balance** and **rate** (see Figure 5.9) are declared protected. Attempts to manipulate their values directly, such as

acctone.balance := acctone.balance + 600;

are detected as a **syntax error** by the system. On the other hand, the methods can be called as follows:

acctone.deposit (600);

with no syntax error.

ObjectModule::[account]
Definition is
type : passive;
class : **class** account {
protected: balance : real;
rate : real;
public :
account (double bal, double pcnt);
void deposit (double amt);
double withdraw(double amt);
void compound();
double getbalance();
};
visible : [main];

```
method
      method account(bal,pct);
      method deposit ();
      method withdraw();
      method compound();
      method getbalance();

body is
(=>[:account::account(double bal,double pcnt)]
begin
    balance := bal;
    rate := pcnt/100;
end.
)
;;

body is
(=>[:void account::deposit(double amt)]
begin
   balance := balance + amt;
end.
)
;;

body is
(=>[:double account::withdraw(double amt)]
begin
   if( amt <= balance) then
   balance := balance -amt;
   return amt;
   else
   return 0;
   fi
end.
)
;;

body is
(=>[:void account::compound()]
begin
   balance := balance + rate * balance;
end.
)
;;
```

```
body is
(=>[:double account::getbalance()]
begin
  return balance;
end.
)
end.
```

Figure 5.9 The ObjectModule Account.

To execute the module account, the code in Figure 5.9 and Figure 5.10 is placed in source files **account** and **main** respectively. The following command produces an executable file **acct**:

```
dosl < accout > account.c

dosl < main > main.c

cat account >> acct.c

cat main.c >> acct.c

c++ -o acct acct.c
```

Figure 5.10 is a main module, the main module requests from the user to enter values for, Starting balance, Monthly deposit, Annual percentage, and Number of the months respectively. It then sends message to module account to create an instance of acct with Starting balance and Annual percentage rate (see Figure 5.10). The compound interest is computed based on monthly deposit and number of months and finally the balance after those months is printed.

```
ObjectModule::[main]
Definition is
type : passive;
variable : balance : real;
        deposit : real;
        anpctrate:real;
        monpcr : real;
        months : integer;
        m :integer;
```

```
visible : [account];
body is
begin
  dout <<"Starting balance: ";
  din >> balance;
  dout <<"Monthly deposit: ";
  din >>deposit;
  dout <<"Annual percentage rate: ";
  din >>anpctrate;
  dout <<"number of months: ";
  din >> months;
:-- Compute new balance
  m:=0;
  monpcr := anpctrate / 12;
  account acct(balance, monpcr);
  while (m < months) do
    acct.deposit(deposit);
    acct.compound();
    m:=m + 1;
  od
:-- Print balance
  dout<<"Blance after "<< months << " months = $";
  dout<<acct.getbalance();
end.
```

Figure 5.10 The ObjectModule Main.

Once again, the ACT++ code produced by system prototype is hidden from user.

5.6 Summary

In sections 5.1 though 5.6, we presented a series of test programs and used them to examine the correctness of **DOSL-II's** syntax and semantics. The system that does the transformation contains about 6,000 lines of a **C** code and system calls excluding the comments. It has been successfully implemented on the **AT&T 3b2** with the environment of the **UNIX** operating system.

Each component of the system prototype , for example, the lexical analyzer and parser, has been tested separately and then combined and tested with different test programs. Each time a new **DOSL-II** construct is added to the system prototype, a series of test programs has been designed and run to validate the system. Thus, incremental testing of the system prototype was conducted.

Chapter 6

Summary and Future Research

The advent of commercial parallel processing machines in the hardware area and the emergence of new programming paradigms such as object-oriented programming in the software area have a positive impact on the development of efficient and reliable software. As a result, integrated software environment that satisfy sufficiently the requirements for the parallel and distributed programming applications are needed. It is also necessary that this integrated environment support good software methodologies. The focus of this research was to provide an integrated software environment for distributed systems by extending an existing requirement specification language to support prototyping. Advances in rapid prototyping have increased the awareness of the software industry to the possible benefits to be gained from the use of prototyping. Rapid prototyping involves the fast construction of a prototype version of a system in order that it may be evaluated by a customer or end-user and subsequently refined in the light of the feedback generated during this evaluation.

The main advantage of prototyping is that it allows the system analyst to gather customer or end-user generated feedback [Hor84] earlier in the software development process than is otherwise possible using conventional software development methods. In this way, prototyping can be used to reduce the number of errors in requirements specifications. These are often the most difficult and expensive errors to correct because they are often discovered after the system is placed in operational use. However, prototyping is ineffective if it is not supported by a development environment that provides an easy derivation of prototypes from formal specifications and makes the implementation process partially automated.

6.1 Contributions of the Research

The goal of this research was to develop a prototyping environment for the formal distributed object-based specification language **DOSL**. Thus, a methodology for executing the **DOSL** specification language was defined and a prototype system was developed. The **DOSL** specification language was extended as a new formal distributed object-oriented specification language **DOSL-II**. **DOSL-II** is an object-oriented rather than object-based, and includes class, inheritance, simple **I/O**, stream **I/O**, concurrent **I/O**, and new constructs for object communication. The major contributions of this work are:

1. Definition of an enhancement for an object-oriented specification language that supports the modeling of synchronous and asynchronous communication, priority message passing, and inheritance. These features make the language a unique combination of features that are individually found in other specification language. The combined result is a versatile, multi-purposed specification language.
2. Expansion of the scope of use of the **DOSL** language by defining a prototyping methodology that takes a **DOSL** specification and provides an executable environment by transformation of the **DOSL** specification to the metalanguage **ACT++**. As a result, animation of a **DOSL** specification is possible with only minimal effort by the specifier.

We first a prototype system to verify the syntax of **DOSL**. We then designed **DOSL-II** a formal specification language with a run-time support. This new formal distributed object-oriented specification language supports class, inheritance, simple **I/O**, stream **I/O**, concurrent **I/O** and new constructs for an object communications.

Finally, we have provided an integrated software environment which combines, formalized methodology for identification of objects from multi-mode formats (data flow diagram, state transition, and Petri nets), a directly executable formal distributed

object-oriented specification language (**DOSL-II**) and system prototype. With this environment, one can directly observe the behavior of any system that can be specified in the **DOSL-II** formal specification language. Since the **DOSL-II** specification is very high-level and easy to work with, one can experiment with variations of the specification and fine-tune it until the desired behavior is obtained. These modules can then be reused.

When an **DOSL-II** formal specification is used for systems development, the inheritance in **DOSL-II** can be used to adapt components for reuse. For example, we can provide a base object class with minimal functionality. When additional or different functionality is required, a new version is created taking the base class as a starting point. The methods provided in the based object need not be re-implemented; they are reused in the new implementation.

6.2 Future Research

This research provides direction for future research. Each part of this integrated environment could be further improved. For example, to improve the system prototype, feedback from the user is essential. Improvement of the feedback will result in better service from the system prototype.

The integrated software environment could be improved by adding a front-end user interface. This front-end user interface would support visual/graphical representation. Visual and graphical representations provide a mechanism to the designers and the users to understand the intended system and to enhance communication.

In the future, a clear, complete, concise, correct, and consistent definition of large distributed systems will be crucial. It is crucial in reducing the cost of software development, testing, and maintenance. Prototyping can be a powerful approach to achieve this goal. Better and more effective tools are needed. The advancement in

graphical prototype tools, specification validation tools, system modeling tools will help standardize prototype-based software methodologies and make them more accessible.

References

[Age79] T. Agerwala, "Putting Petri Nets to Work." *IEEE Computer*, Dec. 1979, pp. 85-94.

[Agh86] G.A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[Aho79] A.V. Aho and J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley Publishing Company, 1979.

[Aho88] A.V. Aho, B.W. Kernighan and P.J. Weinberger, *The Awk Programming Language*, Englewood Cliffs NJ: Prentice-Hall, 1988.

[Asu93] S. Asur and S. Hufnagel, "Taxonomy of Rapid-Prototyping Methods and Tools." *Proceedings 4th International IEEE Workshop on Rapid System Prototyping*, June 1993.

[Aue86] B. Auernheimer and R. A. Kemmerer, "RT-ASLAN : A Specification Language for Real-Time Systems." *IEEE Trans. Software Eng.*, SE-12, No. 9, June 1986, pp. 879-889.

[Bal82] R.M. Balzer, N.M. Goldman, and D.S. Wile, "Operational Specification as the Basis for Rapid Prototyping." *ACM Software Eng.*, No. 7, Sept., 1982, pp. 3-16.

[Bal89] R.M. Balzer and R.P. Gabriel, "Draft Report on Requirements for a Common Prototyping System." *SIGPLAN Notices*, Vol. 24, No. 3 , March 1989.

[Bar87] B.M. Barry, J. Altoft, D.A. Thomas, and M. Wilson, "Using Objects to Design and build Radar ESM Systems." *OOPSLA'87 Conference Proceedings, SIGPLAN*, ACM, 1987.

[Ber82] H.K. Berg, W.E. Boebert, W.R. Franta, and T.G. Moher, *Formal Methods of Program Verification and Specification*, Englewood Cliffs, NJ: Prentice-Hall., 1982.

[Ber87] D. M. Berry, "Towards a Formal Basis for the Formal Development Method and the Ina Jo Specification Language." *IEEE Trans. Software Eng.* SE-13, No. 2, 1987, pp. 184-201.

[Ber90] V. Berzins and V. Luqi, *Software Engineering with Abstractions*, Addison-Wesley Publishing Company, Inc., 1990.

[Bis89] W. Bischofberger and R. Keller, "Enhancing the Software Lifecycle by Prototyping." *Structured Programing*, Vol. 10, No. 1, Jan 1989, pp. 47-59.

[Bjo82] D. Bjorner, and C.B. Jones, *Formal Specification and Software Development*, London: Prentice-Hall.

[Bri89] J.P. Briot, "Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment." *Proceedings of European Conference on Object-Oriented Programming (ECOOP'89)*, July 1989.

[Cor90] A. Corradi and L. Leonardi, "Parallelism in Object-Oriented Programming Languages." *IEEE Int'l Conference on Computer Languages.*, 1990.

[Deh93] A. Dehkhoda and D.L. Carver., "Prototyping Distributed Systems Using an Object-Oriented Specification Language." *Proceedings 4th International IEEE Workshop on Rapid System Prototyping*, June 1993.

[Did93] M.M. Didic and M. Wojciech, "A Reusable Rapid Prototyping Environment for Monitoring in a Discret Part Manufacturing and Semi Process Industry." *Proceedings 4th International IEEE Workshop on Rapid System Prototyping*, June 1993.

[Dij75] E.W. Dijstra, "Guarded Commands, Nondeterminancy and Formal Derivation of Programs." *Comm. of ACM*, Vol. 18, No. 8, Aug. 1975, pp. 453-457.

[Dol90] A. Dollas and V. Chi, "Rapid System Prototyping in Academic Laboratories of the 1990's." *1st International Workshop on Rapid System Prototyping*, Research Triangle Park, North Carolina, June 1990.

[Eij89] P.H.J. van Eijk and M. Diaz (ed), *The Formal Description Technique LOTOS*, North-Holland, 1989.

[Geh85] N. Gehani and A.D. McGettrick, *Software Specification Technique*, Addison-Wesley, 1985.

[Gol83] A. Goldberg and D. Robson, *Smalltalk-80. The Language and its Implementation*, Reading MA: Addison-Wesley, 1983.

[Gom81] H. Gomaa and T. Scott, "Prototyping as a Tool in the Specification of User Requirements." *Proceedings of the Fifth International Conference on Software Engineering*, IEEE, March, 1981, pp. 333-339.

[Gut75] J.V. Guttag, *The Specification and Application to Programming of Abstract Data Types*, Ph.D. Dissertation, Report No. CRSG-59, Computational Science Group, Univ. of Toronto, 1975.

[Gut78] J.V. Guttag and J.J. Horning, "The Algebraic Specification of Abstract Data Types." *Acta Informatica*,Oct. 1978, pp. 27-52.

[Gut80] J.V. Guttag and J.J. Horning, "Formal Specification as a Design Tool." *Proceedings of the 7th ACM Symposium on the Principles of Programming Languages*, 1985, pp. 251-261.

[Gut85] J.V. Guttag, J.J. Horning, and J.M. Wing, "The Larch Family of Specification Languages." *IEEE, Software Eng.*, No. 2, 1985, pp. 24-36.

[Hai85] B. Hailpern and M.R. Laff, "SW 2--An Object-based Programming Environment." *Proceedings of the ACM SIGPLAN, Symposium on Language Issues in Programming Environments*, June 1985, pp. 1-11.

[Hek88] S. Hekmatpour and D. Ince, *Software Prototyping, Formal Methods and VDM*, Addison-Wesley Publishing Company, 1988.

[Hen86] P. Henderson, "Functional Programming, Formal Specification, and Rapid Prototyping." *IEEE Trans. Software Eng.*, SE-12, No. 2, 1986, pp. 241-250.

[Hoa69] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming." *Comm. ACM*, Vol. 12, No. 10, 1969, pp. 576-83.

[Hoa85] C.A.R. Hoare, *Communication Sequential Processes*, Prentice Hall Int'l, 1985.

[Jon80] C.B. Jones, *Software Development: A Rigorous Approach*, Prentice Hall Int'l, 1980.

[Jos82] G. Joseph and M. Joes, "Rapid Prototyping in the OBJ Executable Specification Language." *ACM SIGSOFT Software Eng. Notes* Vol. 7, No. 5, Dec. 1982, pp. 75-83.

[Kaf88] D. Kafura, "Concurrent Object-Oriented Real-time System Research", Technical Report, TR 88-47, Department of Computer Science, Virginia Polytechnic Institute and State University, 1988.

[Kaf89] D. Kafura, "Concurrent Object-Oriented Real-time System Research." *SIGPLAN Notices*, ACM, Feb. 1989.

[Kaf90] D. Kafura, and K.H. Lee, "ACT++ : Building a Concurrent C++ with Actors." *JOOP*, June 1990, pp. 25-37.

[Ker78a] B.W. Kernighan and M.D. McIlroy, *UNIX Programmer's Manual, Bell Laboratories*, Seventh Edition1978.

[Kla82] A. Klausner and T.E. Konchan, "Rapid Prototyping and Requirements Specification Using PDS." *ACM SIGSOFT Software Eng. Notes* Vol. 7, No. 5, Dec. 1982, pp. 96-105.

[Kor92] F. Kordon, "On the Use of Structuration Rules for Rapid Prototyping." *Proceeding of the Workshop "Concurrency, Specification & Programming"*, Berlin, Nov. 1992.

[Kor93] F. Kordon, "A Generic Prototype Model for Distributed System Based on High Level Object Oriented Specification." *Proceedings 4th International IEEE Workshop on Rapid System Prototyping*, June 1993.

[Lam89] L. Lamport, "A Simple Approach to Specifying Concurrent Systems." *Comm. ACM*, Vol. 32, No. 1, Jan. 1989, pp. 32-45.

[Lee91] S. Lee and D. Carver, "Object-Oriented analysis Specification: A Knowledge Base Approach." *Journal of Object-Oriented Programming*, Jan. 1991, pp. 35-43.

[Lew89] T.G. Lewis, F. Bose, and S. Yang, "Prototypes from Standard User Interface Management System." *IEEE Computer*, May 1989.

[Lie87] H. Lieberman, "Concurrent Object-Oriented Programming in Act 1." *In: Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro (eds.), MIT Press, Cambridge, MA, 1987, pp. 9-36.

[Luq88] V. Luqi and Berzins, "Rapidly Prototyping Real-Time System." *IEEE Software*, Sept. 1988, pp. 25-36.

[Luq89] V. Luqi, "Software Evolution via Rapid Prototying." *IEEE Computer*, Vol. 22, No. 5, May 1989, pp. 13-25.

[Luq92] V. Luqi, "Computer-Aided Prototyping for Command-and-Control System Using Caps." *IEEE Software*, Jan., 1992, pp. 56-67.

[Luq93] V. Luqi, M. Shing, and J. Brockett, "Real-Time Scheduling for Software Prototyping." *Proceedings 4th International IEEE Workshop on Rapid System Prototyping*, June 1993.

[Mey92] B.A. Myers, D.A.Giuse, and B. Vander-Zanden, "Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods." *Proceedings of OOPSLA'92*, 1992.

[Mil80] H.D. Mills, D. O'Neill, R.C. Linger, M. Dyer, and R.E. Quinnan, "The Management of Software Engineering." *IBM Sys. J.*, Vol. 24, No. 2, 1980, pp. 414-77.

[Pag81] F.G. Page, *Formal Specification of Programming Language -- A Panoramic Primer*, Englewood Cliffs, NJ: Prentice-Hall, 1981.

[Red81] S.T. Redwine and E. D. Berglass, *Candidate R&D Thrusts for the Software Technology Initiative*, DoD Report, Department of Defense, USA, May, 1981.

[Rob89] B. Robert, V. Terwillinger, and R.H. Campbell, "PLEASE: Executable Specifications for Incremental Software Development." *The Jounal of Systems and Software*, Vol. 10, 1989, pp. 97-112.

[Rta84] R.D. Tavendale, "A Description and Assessment of FDL." *MTR* 84/67, Marconi Research Center, Chelmsford, UK., 1984.

[Rta85] R.D. Tavendale, "A Technique for Prototyping Directly From a Specification." GEC Research Ltd., Chelmsford, CM2 8HN, UK. *IEEE Computer*, 1985, pp. 224-229.

[Smi85] D.R. Smith, G.B. Kotik, and S.J. Westfold, "Research on Knowledge-based Software Environments at Kertrel Institute." *IEEE Trans. Software Eng.*, Vol. 11, No. 11, 1985, pp. 1278-95.

[Spi88] J.M. Spivey, *Introducing Z: A Specification Language and its Formal Semantics*, Cambridge Univ. Press, 1988.

[Spi89] J.M. Spivey, *The Z Notation: A Reference Manual*, London: Prentice-Hall, 1989.

[Sta84] T.A. Standish and R.N. Taylor, "Arcturus: A Prototype Advanced ADA Programming Environment." *Proceedings of the ACM SIGSOFT/SIGPLN Software Engineering Symposium on Practical Software Development Environments*, April 1984, pp. 57-64.

[Ste86] M.J. Stefik, D.G. Bobrow, and K.M. Kahn, "Integrating Access-oriented Programming into a Multiparadigm Environment." *IEEE Software*, Vol.3, No. 1, 1986, pp. 10-18.

[Sto77] J.E. Stoy, *Denotational Semantics: The Scott Strachey Approach to Programming Languages*, Cambridge, Mass: MIT Press, 1977.

[Str76] C. Strachey and R. Milne, *A Theory of Programming Language Semantics*, London: Chapman and Hall, 1976.

[Str86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Menlo Park CA, 1986.

[Wan90] Z. Wanlei, "PM: A System For Prototyping and Monitoring Remote Procedure Call Programs." *ACM SIGSOFT Software Eng. Notes* Vol. 15, No. 1, Jan. 1990, pp. 59-63.

[Win90] J.M. Wing, "A Specifier's Introduction to Formal Methods." *IEEE Computer*, Sept. 1990, pp. 8-22.

[Wir73] N. Wirth and C.A.R. Hoare, "An Axiomatic Definition of Pascal." *Acta Informatica*, Vol. 2, No. 3, 1973, pp. 335-355.

[Yon86] A. Yonezawa, A.E. Shibayama, Y. Honda, and T. Takada, Modelling and Programming in a Concurrent Object-Oriented Language ABCL/1, *in: Object Oriented Concurrent Programming*, edited by A. Yonezawa and M. Tokoro, MIT Press, 1986.

[Yon90] A. Yonezawa, *ABCL: An Object-Oriented Concurrent System*, The MIT Press, Cambridge, MA, 1990.

[Zav86] P. Zave and W. Schell, "Salient Featurs of an Executable Specification Language and its Environment." *IEEE Trans. Software Eng.*, Vol. 12, No. 2, 1986, pp. 312-25.

[Zav89] P. Zave, "A Compositional Approach to Mutiparadigm Programming." *IEEE Software*, Vol. 6, No. 5, 1989, pp. 15-27.

[Zel80] M.V. Zelkowitz, "A Case Study in Rapid Prototyping." *Software - Practice and Exprience*, Vol. 10, No. 12, Dec. 1980, pp. 1037-1042.

Vita

Abbas Dehkhoda was born in the Isfahan city of Iran, on November 23, 1948. He received his B.A. degree in Economics in 1972 from National University of Iran in Tehran. He joined the department of Computer Science at Southern Methodist University in 1979 and received the Master's degree in Computer Science in Spring of 1980. Since August of 1981 he has been working as an assistant professor in the Computer Science Department at Xavier University in New Orleans, Louisiana. His current research interests include prototyping of distributed systems using an object-oriented model, formal specification languages, and object-oriented systems development.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Abbas Dehkhoda

Major Field: Computer Science

Title of Dissertation: A Specification Environment That Supports the Prototyping of Distributed Systems Using an Object Oriented Model

Approved:

Doris L. Carver
Major Professor and Chairman

Daniel Fogel
Dean of the Graduate School

EXAMINING COMMITTEE:

A. Elamawy

V. V. [illegible]

David A. Huffman

A. Bart Jones

Dan B. Rinks

Date of Examination:

December 17, 1993